MODELING HUMAN DECISION POINTS IN COMPLEX SYSTEMS

BY

DOUGLAS C. ESKINS

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Professor William H. Sanders, Chair
Professor Alex Kirlik
Assistant Professor Sayan Mitra
Professor David Malcolm Nicol

ABSTRACT

The actions of human participants in complex systems can greatly affect system outcomes. In particular, the security of cyber-human systems (CHSs) can be highly influenced by the actions, especially the decisions, of human participants within the system. To provide insight into CHS security and aid decision-makers, we propose the Human-Influenced Task-Oriented Process (HITOP) modeling formalism and the Multiple-Asymmetric-Utility System experimental framework (MAUS) as tools to quantitatively evaluate the influence of human actions, especially human decisions, on CHS security. We present both a modeling definition for the CHS and an ontology for identifying the relationships between CHS elements and human task performance. We introduce the Human Decision Point (HDP) as an explicit construct for modeling human decisions and explain how the HDP may be represented within a CHS model. We provide a formal definition of HITOP, its notions of state, and its execution algorithms. We introduce the MAUS experimental framework and formally define several HDP probability solution methods using it. We solve an example CHS modeling problem, and present a set of example results and decision tools. We develop an executable HITOP modeling formalism and use it to analyze a real-world case study.

To my father, for setting the example, and to my daughter, Kourtney, for providing the incentive to do the same

ACKNOWLEDGMENTS

The work described in this dissertation was performed, in part, with funding from the Hewlett-Packard Labs Innovation Research Program. I thank the staff and management team at HP Labs Bristol, UK, for their support and guidance during the early phases of this work. I would like to also thank the UK members of the HP project team, Professor Aad van Moorsel, Dr. Simon Parkin, Robert Cain, and James Turland, for their insights and collaboration.

I thank the staff and management of the U.S. Nuclear Regulatory Commission Office of Nuclear Regulatory Research for their financial support of my work, and I would like to especially thank Russell Sydnor for his support and advocacy on my behalf.

I thank the staff at the Coordinated Science Lab, the Information Trust Institute, and the Department of Electrical and Computer Engineering for their administrative and technical support. I thank Tim Yardley for helping me get the right tools and equipment and Jeremy Jones for helping me wire it up. I thank Laurie Fisher for keeping me on top of all the paperwork.

I thank Professor Carl Elks at the University of Virginia for his generous support and donation of equipment in support of this work.

I thank the members of the Perform research group, past and present, for their invaluable support, insights, and friendship. Through the many late nights and weekends in the lab, there was always someone available for a whiteboard session. The development of HITOP was a team effort. I thank Dr. Robin Berthier for his advice and technical assistance, especially for his help with the case study definition and data collection efforts. I thank Professor Eric Rozier for his invaluable guidance on optimization. I thank Craig Buchanan for his work debugging the HITOP code and HITOP models. I thank Edmond Rogers and

Graeme Neilson for their in insights into real-world cyber attacks. I especially thank Ken Keefe for his insights, advice, and the tremendous labor he put forth to make the theoretical concrete. Ken was substantially responsible for the implementation of the HITOP atomic formalism with the Möbius tool and provided unending support to ensure the project was completed under a very tight deadline. I am also grateful to Jenny Applequist for her valuable support, comments, and long hours of technical editing. She helped me find the right books and the right words.

Finally, I thank my advisor, Professor William H. Sanders, for his technical advice and unending support over the years. His guidance and belief in the value of this research was an essential part of its success. I would also like to thank my other committee members, Professor David M. Nicol, Professor Alex Kirlik, and Professor Sayan Mitra, for their questions, insights, and technical guidance throughout the final phases of this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

A	Availability
C	Confidentiality
CHS	Cyber-Human System
Det-F model	Deterministic Dynamic Flow Model
Det-P model	Deterministic Dynamic Performance Model
DF Model	Dynamic Flow Model
DP Model	Dynamic Performance Model
DPO Model	Dynamic Performance Outcome Model
DPP Model	Dynamic Performance Probability Model
DPT Model	Dynamic Performance Timing Model
DS	Direct Search
FU	Flow Utility Variable
HDP	Human Decision Point
HITOP	Human-Influenced Task-Oriented Process
LMLU	Linear Most Likely Utility
MAUS	Multiple-Asymmetric-Utility System
MU	Markov Utility Variable
OWC	Opportunity-Willingness-Capability
P_W	Willingness Probability
PF Graph	Process Flow Graph

PI	Process Instance
PIT	Process Instance Tree
PU	Participant Utility
PUV	Performance Utility Variable
PUE	Participant Utility Element
SC	System Configuration
SF Model	Static Flow Model
SD Task	State-Dependent Task
SI Task	State-Independent Task
SP Model	Static Performance Model
SPO Model	Static Performance Outcome Model
SPP Model	Static Performance Probability Model
SPT Model	Static Performance Timing Model
SSF Graph	State Space Flow Graph
Sto-F model	Stochastic Dynamic Flow Model
Sto-P model	Stochastic Dynamic Performance Model
SU	Security Utility Value
SU_{Ideal}	Ideal Security Utility Value
SU_{Eff}	Effective Security Utility Value
SUE	Security Utility Element
SUD	Security Utility Divergence
SUDR	Security Utility Divergence Ratio
UE	Utility Element

CHAPTER 1

INTRODUCTION

1.1 Humans and Complex Systems

Since the earliest days of humanity, people have developed technologies to serve their needs. A technology and the people whom it serves form a system to accomplish tasks with increased efficiency or to enable processes that were previously impossible or impractical. A simple example of an early technological system is the plow. The plow-human system enabled ancient farmers to more efficiently produce agricultural goods.

Fast forward to today, where techno-human systems abound. In fact, nearly all human activity is part of a system in which humans interact with technology. Consider the simple act of driving. We routinely rely on the automobile-human system to transport us quickly and safely, but a failure of the automobile can lead to delays and hazards. It is obvious that the state of our technologies can greatly influence the *outcomes* of our systems. Thus, understandably, there has been much focus on making our technologies more powerful and more reliable.

Of course, technology is only half of the story. The other half is the human side of the techno-human system. Humans, like the technologies they use, have capabilities that influence the outcomes of the tasks they perform. For example, if the driver of the automobile-human system does not see well at night, the "performance outcomes" of driving may be worsened despite a perfectly working automobile.

Perhaps more importantly, unlike most technologies, humans *make decisions*, and decisions may be the dominant influences on task outcomes [1]. For example, a driver may decide to use excessive speeds on an icy road despite training and legal requirements to the contrary. The driver is capable of following the rules of safe driving, but is unwilling to do so. In other

words, with respect to task performance, it is important for a person to be *willing* as well as *capable*.

Furthermore, it is interesting to consider that the many influences on human decisions or human willingness are outside what is normally considered "the system," i.e., the purely technological system. By explicitly considering the influences on and from human "system elements," we can enrich our ideas about what constitutes a system and increase our understanding of how to improve the system.

1.2 A Modeling Viewpoint that Includes Human Decisions

It is with the above insight that we hope to better understand how human decisions influence task outcomes in areas for which research typically focuses, on the technological aspects of a system. We will do so by constructing a modeling framework in which human decisions and capabilities are explicitly represented along with the technological elements of a system. We will develop a modeling formalism within that framework and use it to analyze and predict human decisions in complex technological systems. In particular, we will apply this methodology to the domain of cyber security by creating and analyzing models of cyber-human systems and measuring cyber security outcomes.

1.3 Background

First, we will quickly review some pertinent research in related areas. To model how human decisions affect complex systems, with a special focus on cyber security, we require knowledge from many fields of research. Some of those fields are human performance, decision theory, utility theory, economics, education and training, cyber security, human computer interaction, human factors, process models, and discrete-event models. Obviously, an extensive review of those areas is beyond the scope of this thesis, and it is not our intention to add a new theoretical basis for a well-established field such as decision theory. Instead, we intend to present a modeling framework that is congruent with a great deal of existing practical and theoretical work. For example, rather than present a new measure of human

utility, we instead provide a modeling and experimental framework in which a variety of existing utility frameworks may be applied and/or interfaced.

Next, we will briefly summarize work that is relevant to our research. Those areas of work are grouped into four categories: human performance and decision-making, human-machine systems, process models, and cyber security models.

1.3.1 Human Performance and Decision-Making

To build a model of how humans act and react within a complex system, we must model many aspects of human activities, including ones related, for example, to education, training, decision-making, and human performance. We will group all those areas under the single subject heading of "human performance and decision-making." Perhaps obviously, human performance and decision-making have been studied in a variety of fields because of their importance to the outcomes of many systems. In particular, we will focus on how human performance and/or decision-making have been studied in the fields of psychology and economics.

The field of psychology has a long history of research on human performance. In general psychological theories focus on behavioral or cognitive explanations for human behaviors and decisions. Behavioral theories focus on how the environment affects human performance [2]. For example, people seek rewards and avoid punishments. The rewards and punishments can be physical, social, or even abstract concepts. The basis of many behavioral theories is the idea that, given enough information about the state of the world, humans will make decisions that are deterministic [3], [4].

Cognitive theories, on the other hand, focus on internal mental states that affect human performance [5]. That is, people make decisions based on who they are and what they feel. Many models of cognitive states are used to explain human behavior. The theories that have been advanced over the years are numerous and complex [1], [6], [7], [8], [9], [10].

Economics, especially microeconomics, uses a brand of behavioral theory, generally applicable to groups of people or organizations, known as rational choice theory [11], [12]. Theories of rational choice hold that decisions among several choices are based on the cur-

rent state of the world (as understood by the decision maker) and the potential outcomes of each choice given that state. The values of different outcomes are often quantified by utility functions [13], [14], [15]. Utility functions capture important aspects of the world "state" and translate them into measurable quantities. Utility functions thus enable comparison and ordering of decision outcomes. Because they are rational, decision-makers are assumed to make a choice that maximizes their utility in some defined sense.

A ideal rational decision-maker is a person who will make the same decision every time given the same data. Experimental data show that actual decisions can be variable given the same world "state," so several explanations have been advanced to explain this and still maintain a rational decision-maker framework.

One such explanation is that people's ability to recognize the state of the world is limited, and thus uncertainties in the perceived state of the world are reflected in decision-making uncertainties [16]. That is especially true if decisions are based in part on the actions of other people. The decision-maker must then "guess" or make a subjective evaluation of the state of the world in order to make a decision.

A related explanation known as *random utility theory* suggests that known (by the decision-maker) uncertainty in the outcomes of external events leads to uncertainty in decisions [17]. For example, if a person must choose between two bus lines to get someplace with the goal of minimizing the travel time, there is an uncertainty, i.e., a random variable measuring the travel times, associated with each bus line. Even if the decision-maker knows the distributions of each bus line's random variable, e.g., X and Y, the choice of bus line may vary given the same circumstances. The explanation per random utility theory is that the choice is not made based on the lowest expected value, i.e., choose Y if $E[X] > E[Y]$. Instead, the choice is based on the probability that, in this case, bus line X is faster than bus line Y, i.e., $Pr[X < Y]$. Random utility theory offers an explanation for many experimental data sets in which rational choice does not appear to hold [18], [19].

Another explanation is that uncertainty in a decision-maker's internal "state" (a more cognitive explanation) leads to uncertainty in the decision [20]. That explanation still leads to uncertainty, as any model of internal cognitive state will be incomplete at best and have many unknowns when applied to situations in the world at large.

While it remains the subject of academic debate [12], [21], we adopt initially for our work the idea that human beings are rational decision-makers, or, at the very least, bounded-rational or reasonable decision-makers [12], [22]. That allows us to avoid the complexity of an additional cognitive model for decision-makers, though it should be noted that our framework would support such a model. A bounded rational or reasonable decision-maker is not "all-knowing," but makes a rational decision based on the information that is known, and the decision is in some sense "good enough," i.e., the decision improves utility but is not guaranteed to maximize it.

We also will adopt the idea that important aspects of the world "state" may be represented and quantified with utility functions, as is widely accepted in the field of economics [23], [24], [25].

We will add to this work the idea that if reasonable decision-making is translated to a modeling framework, it can be interpreted as making decisions based on a limited set of model states. That is, we do not (and probably cannot) model everything in the world that might affect a decision. However, we will model those aspects that are most important, i.e., that reflect the bounds of actual human perceptions. In addition, we will be somewhat flexible in our solution method with respect to maximum utility values, meaning that we will assume, knowing that humans are reasonable decision-makers, that any one of a set of "good" decisions may be "good enough" for our decision-maker. That means that our solution does not always have to be the mathematically perfect maximum utility value down to the last significant digit in order to be a useful approximation of actual human behavior. In fact, we propose that in many cases, just knowing the "average" decision can provide useful information when characterizing the average behavior of many people in a system.

We will use those ideas to define, in part, how humans act and interact within a human-machine system.

1.3.2 Human-Machine Systems

Human-machine systems, as mentioned above, are the basis for many types of technology, and their development is often impelled by the desire to make human tasks easier. Rele-

vant fields of study involving human-machine systems include human-computer interaction, human factors, and various branches of engineering.

Human-computer interaction (HCI) and human factors (HF) both involve the study of how people and computers interact, with a special focus on design problems [26], [27]. Much of the research in those areas relates to how people perform tasks in an operational environment. For example, analyzing how best to enable operators at a nuclear power plant to perceive and respond to plant emergency conditions is an HF problem (and increasingly an HCI problem as digital safety systems are introduced into nuclear power plants).

Cognitive engineering takes a high-level systems viewpoint and seeks to design systems that match the cognitive requirements of system users [1].

The industrial and systems engineering fields have also tackled the problems of human performance within systems. Rasmussen provides both a historical perspective on humans and computers in socio-technical systems and a framework for viewing such systems, called *cognitive work analysis* [7]. He discusses the need for consistent models for routine tasks involving information technology and proposes different modeling levels based on whether the type of performance skill needed to perform a task is skill-, rule-, or knowledge-based [8].

Some useful tools from these fields include task analysis, GOMS, and the HCI/HF viewpoint that humans are an important part of a system.

Task analysis is a method, used in many fields, by which human activities can be decomposed into sets of tasks [28]. Tasks are associated with sets of internal characteristics and external conditions that affect performance. Task analysis provides an important tool for breaking a general system down into a set of important tasks and documenting the conditions related to the performance of the tasks.

Goals, operators, methods, and selection rules (GOMS) is a modeling methodology by which human actions are related to sets of goals and possible actions [26]. Selection rules are provided if more than one action is possible. GOMS is used to deterministically model how humans interact with computer systems.

The HCI/HF viewpoint in general is that human users of a system are an important topic of study and that a system's performance must be characterized relative to that human performance [29], [30].

We adopt from the above discussed work the ideas that human activities can be represented and modeled as a set of tasks within a process; that task performance can be modeled relative to a goal and using a set of possible task operations; and that humans within a complex system must be explicitly modeled in order to adequately characterize system performance. We add to the earlier work our approach of defining a cyber-human system model and the relationships between model elements with the opportunity-willingness-capability ontology. We use all of those ideas to construct process models for our solution method.

1.3.3 Process Models

As we will discuss in later chapters, we have chosen to implement our model as a process model. Process models are formalized descriptions of activities within a process. Process models are often used in process engineering to model physical and/or industrial processes, e.g., the production of steel. Process models are also used to model informational and/or business processes; such models are typically referred to as *business process models*. While there are many types of process model formalisms and methods, we will focus primarily on informational or business process models.

A process model can be any representation of a process. For example, a simple flow chart is a process model. The Program Evaluation and Review Technique used by the U.S. Navy [31] to manage project performance is also an example of a process model.

More modern methods make use of computer-based process models. Two good surveys of business process models are provided by van der Aalst [32] and White [33]. Many formalisms exist for process models. There are commercial products such as COSA (based on Petri nets [32]) and the MQSeries/workflow tool provided by IBM [34]. There are open-source projects such as Business Process Model and Notation (BPMN) [35] and UML activity diagrams [36]. Also, there are a few models whose purpose is research. YAWL [37] is one such modeling formalism and it proved to be the most useful of the formalisms we reviewed. However, as noted by van der Aalst, many of the reviewed models lack a formal foundation and a consistent technical definition, and thus are not always useful in a formal modeling environment.

We adopt from the earlier work the general modeling notion of process models and some of the formal definitions of workflow patterns and their implementations, especially as provided by YAWL [37], [38]. We add to the body of work our approach which is to build a formally defined process model specifically designed to model human activities within complex systems and to be solved via discrete-event simulation. We will use our formalism to analyze cyber security problems.

1.3.4 Cyber Security Models

Cyber security models are increasingly being used as a means to understand, measure, and predict how a system will perform relative to various cyber security metrics. Most cyber security models and tools fall into two categories: system-based models and attacker-based models.

System-based models focus on system characteristics that influence system vulnerabilities to attack. An example is the Topological Vulnerability Analysis (TVA) tool [39], which produces attack graphs, i.e., possible paths through the system that exploit vulnerabilities, based on scans of the analyzed network. Attack graphs have as their basis a system evaluation tool known as an *attack tree* [40]. An *attack tree* is a tree with nodes joined by ANDs or ORs that produces a Boolean statement for each possible attack path through the tree.

Attacker-based models focus primarily on the attacker of the system. An example of such an approach is the Mission Oriented Risk and Design Analysis (MORDA) model [41] used to characterize the attack preferences of various types of attackers and predict the impact of attacks on the system being analyzed.

An example of an integrated system-attacker-based modeling approach is the ADversary-VIew Security Evaluation (ADVISE) modeling formalism [42]. It uses a system model to build an attack execution graph and an adversary profile to build a state-based attacker model. The model is then solved to determine the likelihood of a given attacker achieving a given goal for that system.

We adopt from the above discussed work the ideas that model-based evaluation of cyber security is a useful approach and that integrated system models of humans and computer

systems can provide valuable tools for system analysis with respect to cyber security. We add to this body of work our approach, which is to build an integrated model of human users and computer systems, i.e., an approach that uses user-system hybrid models.

1.4 Thesis Statement

There is a need for to perform system level cyber security assessments that can take into account the actions and decisions of human users within the system.

It is our thesis that a model formalism and experimental framework, as well as associated execution algorithms and solution methods can be created such that quantitative, security-relevant information about cyber-human systems can be analyzed to evaluate the influence of human behaviors, especially human decisions, on system-level security.

The contributions of this dissertation are as follows:

- We present a modeling definition of the cyber-human system (CHS). It defines the elements of a CHS and describes the relationships between elements.
- We introduce the Opportunity-Willingness-Capability (OWC) ontology as a method for associating CHS model elements relative to task performance.
- We introduce the concept of the Human Decision Point (HDP) as an explicit modeling element and relates it to the CHS modeling definition.
- We introduce the Human-Influenced Task-Oriented Process (HITOP) modeling formalism and presents a formal definition of its elements and states.
- We describe the two viewpoints of HITOP execution and presents algorithms for implementing them.
- We introduce the Multiple-Asymmetric-Utility System (MAUS) experimental framework and presents an example solution method.
- We formally define the MAUS solution problem, develop several solution methods, and present a performance analysis of these methods.

- We implement the HITOP and MAUS mathematical models by building the HITOP atomic model formalism.
- We apply the HITOP/ MAUS methodology to a real-world case study and presents the results.

1.5 Dissertation Organization

In this dissertation we present a novel method for accounting for the effects of human decisions within cyber-human system models. Our technique involves the explicit modeling of human decisions as first-class elements, and construction of models that include both human and technological elements. In Chapter 2, we introduce the cyber-human system (CHS) and present a modeling definition for it; we introduce the Opportunity-Willingness-Capability (OWC) ontology and use it to describe the relationship between CHS model elements and task performance; and we introduce the Human Decision Point and define it as an explicit element within a CHS model. In Chapter 3, we introduce the Human-Influenced Task-Oriented Process (HITOP) modeling formalism and formally define its structure and notions of state. In Chapter 4, we describe the two viewpoints of execution within the HITOP model and provide execution algorithms for each. In Chapter 5, we present the Multiple-Asymmetric-Utility System (MAUS) experimental framework and describe how it may be used in combination with a modeling formalism to analyze and predict the consequences of human decisions. We also provide an example application of the MAUS framework to a sample problem in CHS security. In Chapter 6, we formally specify the MAUS solution problem statement; present several applicable solution methods; and discuss performance of these methods. In Chapter 7, we discuss the implementation of the mathematical models with the HITOP atomic formalism and we use this executable tool to apply the MAUS/HITOP methodology to a real-world case study. Finally, we conclude with Chapter 8.

CHAPTER 2

SPECIFYING THE CONCEPTUAL MODEL: THE CYBER-HUMAN SYSTEM MODEL

2.1 Introduction

In the previous chapter, we provided a motivation for the modeling of human decisions as part of complex systems, and some background on related work. In this chapter, we will describe our approach for building a *conceptual model* of a cyber-human system (CHS). This approach has three phases. In the first phase, we represent a system of interest in appropriate modeling terms by decomposing it into elements using our set of CHS definitions. In the next phase, we define the relationships between those model elements using the opportunity-willingness-capability ontology. In the final phase, we define special points within the model called *human-decision points* and then link these points to the human decisions we wish to study. The conceptual CHS model has been specified once those phases are complete.

2.2 The Cyber-Human System

To describe how we model the effects of human behaviors, especially human decisions, on a system, we will first define our conceptual model of the cyber-human system. That is, we will describe how a cyber-human system may be decomposed into, or represented by, sets of model element types specific to our aim. A running example will be presented in this and the following chapters to add clarity.

This chapter contains previously published material by D. Eskins and W. H. Sanders. It is reused by permission of IEEE and was published in the *Proceedings of the 8th International Conference on Quantitative Evaluation of SysTems* (QEST 2011).

2.2.1 Cyber-Human System Definition

Definition 2.2.1. A *Cyber-Human System* (CHS) is any system in which both computers and humans are key elements.

In its most basic sense, a CHS is a system in which the decisions and actions of humans in relation to computers affect measurable and important system outcomes.

Examples of such systems include typical business or academic settings in which system users must interact with an IT infrastructure, industrial settings in which workers must interact with computer-based process control systems, or something as simple as a cell phone and its user. For each of these examples, a CHS model may be created to analyze how humans perform tasks, achieve goals, and affect outcomes within the overall system.

2.2.2 Cyber-Human System Element Types

The first step in constructing a conceptual model is the definition of our CHS of interest. It is important to have clear system boundaries when defining what is "in" a model. Thus, it is essential to know beforehand what measures will be made with the model so that the relevant elements to be measured can be included within the system boundaries. This is especially true of CHS models, as elements not typically included in physical-only system models must be included. For example, if a modeler wants to measure human satisfaction, he or she must determine the model elements that are relevant to human satisfaction and be sure to include them within the conceptual model boundaries. Once the system boundaries have been defined, the CHS can be decomposed into modeling elements according to type.

Our conceptual model divides CHS elements into four types: components, participants, processes, and tasks. We assume that all relevant CHSs, plus their desired properties, can be represented using these element types. All of the element types are described next.

Definition 2.2.2. *Components* are the physical objects that make up a system.

All real-world systems must be constructed from sets of physical objects. Even so-called *informational* components, e.g., data or electronic communications, must be stored on, displayed by, or interpreted by physical system components, and are themselves manifestations

of the physical world, e.g., charged states in semiconductors. Thus, informational components are represented as properties of the physical components on which they are stored, displayed, or interpreted.

For example, a system model of a business IT infrastructure might include the following components: system servers, a network, workstations, and network storage devices.

Definition 2.2.3. *Participants* are the entities that initiate and perform actions within a system.

In most cases, participants are humans, but a nonhuman entity, such as an automatic security routine, may also be modeled as a participant. The key characteristic of a participant is that a participant provides and controls the "action" when a task is performed. A component without a participant cannot perform a task.

For example, a system model of a business IT infrastructure might include the following participants: system users, a system administrator, and an automated data backup routine.

Definition 2.2.4. *Processes* are ordered arrangements of activities that specify how the system is used and are usually associated with some purpose or goal.

Processes are sometimes called *workflows* [43], and represent a "road map" of what can happen while a set of actions is being performed. One may imagine a process as a kind of flowchart for which all possible alternatives for a set of actions, including sequences and conditions for performance, have been detailed.

For example, a process in a business IT system might define how Internet sales are credited to accounts, with a goal of 100% transaction accuracy. In Section 5.4 we will describe how utility functions are used to measure the achievement of goals. A process defines a flow or ordering of smaller elements called *tasks*, which are defined next.

Definition 2.2.5. A *task*, using terminology similar to that of [43], is an atomic unit of work that is carried out by a *resource*, where a resource is one or more participants using one or more components.

Tasks are the units of action within a system. In modeling terms, the performance of a task is an event, i.e., it causes the model to change state.

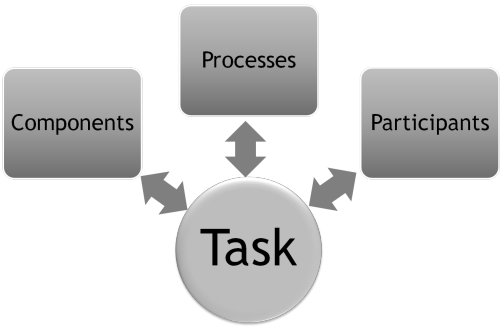

Figure 2.1: Cyber-Human System Element Types

For example, a task within an Internet sales process might be the step "validate customer credit information" or "select invoice." The choice of what constitutes a task for a given system depends on the level of granularity required to achieve the modeling goals. While the Internet sales process could be decomposed into a set of keystroke-level tasks, this is required only if the modeler needs to measure events at that level of detail. A good rule of thumb is that tasks should be defined only to the level of detail required to capture a predefined set of important model state changes.

A compact statement showing the relationship among CHS elements is: A *process* is a defined flow of *tasks* performed by one or more *participants* using one or more system *components*. In modeling terms, this means that state changes can occur only as the result of task performance, and model elements can affect each other only via some task. Figure 2.1 illustrates this relationship. For example, a participant can change the state of a process only via a task that is associated with both the participant and the process.

Running Example: USB Stick Usage as a CHS. Based on a study of the security aspects of USB stick usage [44], we use as our running example the *USB Stick Usage* CHS (Figure 2.2). Here, the CHS is a business environment in which company representatives use USB sticks to store sensitive data. USB sticks are used in various locations. These locations include the employee workstation, company meetings, visits to business clients, the employee's home, and transit between locations. Each location has a set of allowed tasks and potential risks associated with using the USB stick.

Company security policy states that sensitive information stored on USB sticks must be

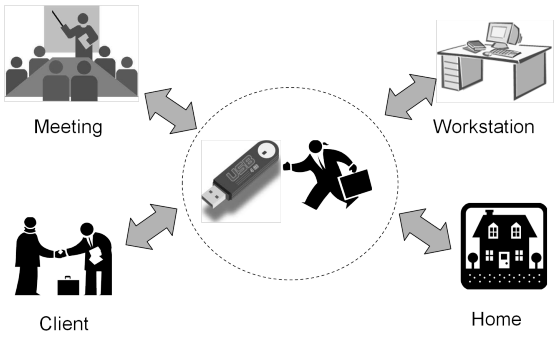

Figure 2.2: Example Cyber-Human System

encrypted. However, employees can choose whether to follow this policy or not. Periodically, the company performs a security scan of USB sticks for unencrypted sensitive data and punishes employees who have violated the security policy. Note that our example illustrates one of the important keys to building a CHS model: the definition of a relevant human decision. In this case, a user of the system can decide to encrypt USB data or not, and that decision is presumably relevant to measured security outcomes for data stored on that USB device.

The company also provides IT support to employees who use USB sticks. The IT department has a fixed budget that is allocated between IT support and security scans.

Company IT security goals include measures of USB data availability and confidentiality. Successful transfers of USB data positively affect availability. Loss or compromise of USB data negatively affects confidentiality.

Additionally, participants have their own goals. They value the availability of USB data but seek to avoid embarrassment when they cannot access data, frustration with IT support failures, and management sanctions for violating the security policy. Note that defining what users value is another key to a CHS model. Building a measure of what users value, what we will later call a *participant utility function*, allows the modeler to evaluate the effects of human decisions from the standpoint of the user. Contrast this with a typical cyber-security viewpoint that considers only a set of security metrics. This different viewpoint is one of the reasons why a CHS model may provide new and important insights into human performance as part of a system.

Below we list some representative CHS elements, grouped by their element types, for the

example system.

- **Components:** USB stick, home computer, and business IT system.
- **Participants:** business representative and a software routine to perform an automated security scan.
- **Processes:** writing (reading) company data to (from) the USB stick, conducting client visits, checking for compliance with security policies, and moving between physical locations.
- **Tasks:** encrypting USB data, reading encrypted USB data, and utilizing IT support.

We will use this example throughout this thesis to illustrate and clarify concepts as we introduce and discuss them.

2.2.3 Section Summary

In this section, we introduced the Cyber-Human System and defined its basic elements as the first step of building a conceptual model. This modeling approach provides guidance for representing a physical system as a set of related model elements. We will show in the following section how this simple classification of objects within a CHS leads to a formal modeling approach that allows us to answer questions about CHS performance outcomes. Now that the basic elements of the CHS model have been defined, we will next describe how the opportunity-willingness-capability ontology can be used to group CHS elements relative to CHS tasks.

2.3 The Opportunity-Willingness-Capability Ontology

In Section 2.2 we described how a CHS could be decomposed into components, participants, processes, and tasks. In this chapter, we will show how these elements may be related to task performance. This is important because all actions, e.g., state changes, within a CHS model occur in terms of tasks, and it is therefore necessary to specify what system states can lead

to task performance and how the outcomes of task performance may be affected by these states. We do so by using a formal naming system, the *opportunity-willingness-capability* (OWC) ontology, to associate sets of CHS elements with each task. Simply put, the OWC ontology classifies a set of CHS elements conditioned on how these sets of elements affect task performance. In order to understand this classification, we must first describe task performance.

2.3.1 Task Performance

Task performance is the "action" within a process. Generally, task performance may lead to several outcomes. However, for simplicity, we shall at first restrict task performance to the binary outcomes of "proper" or "not proper" performance. Here *proper* performance means the achievement of some desirable end state defined relative to the task. Tasks with only two possible outcomes are called *binary tasks*. It is reasonable to assume that all tasks in a system may be represented with binary tasks, as any task that features multiple outcomes can be decomposed into sets of binary tasks.

Running Example: Task. Continuing our running example from Section 2.2, we shall select the task *Encrypt USB Data* (EUD) to illustrate the concept of task performance outcomes. EUD is a task contained in the process for writing new USB data. EUD is performed properly if the data written to the USB drive are encrypted, and not performed properly otherwise.

As shown in Figure 2.3, for a task to have the *potential* to be properly performed, the current model state must be the intersection of possible model states for which OWC (as described next) can be evaluated as true.

2.3.2 Opportunity

In this section, we will clarify the terminology used to describe the concept of opportunity, which is always in reference to a given task. A model may be in a *potential opportunity state* relative to a task and a task itself may be in an *opportunity state* as defined next.

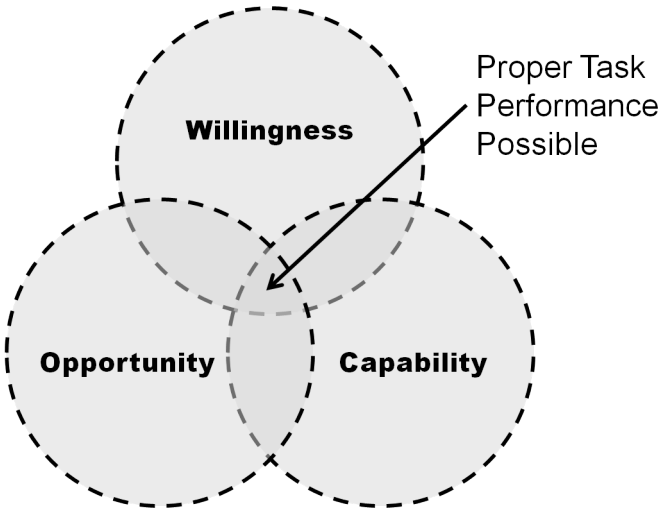

Figure 2.3: OWC Ontology

Definition 2.3.1. A task is in an *opportunity state*, or *opportunity exists* for a task, if the necessary and sufficient conditions are available to attempt task performance. If a task is not in an opportunity state, it is in a *no-opportunity state*.

Definition 2.3.2. A model *potential opportunity state* is a model state in which the probability of opportunity existing for a referenced task is nonzero.

From the above definitions, it can be seen that opportunity captures the idea of prerequisites for task performance. In terms of our system model, the opportunity to perform a task, i.e., task opportunity state, is determined by the states of a set of model elements, i.e., model potential opportunity states. A model element can be thought of as a collection of related state variables that take on different values during model execution. Some of these variable values imply a nonzero probability of task opportunity state and thus determine the set of potential opportunity states for a model. The group of model elements whose variables influence task opportunity is known as the *task-specific set of opportunity elements*.

Definition 2.3.3. *Opportunity elements (OEs)* are the task-specific set of model elements whose values determine the set of model potential opportunity states for the referenced task.

OEs may be drawn in any combination from CHS elements (see Figure 2.4) and may form overlapping sets with other task-specific model element groupings. We shall use the term *OE*

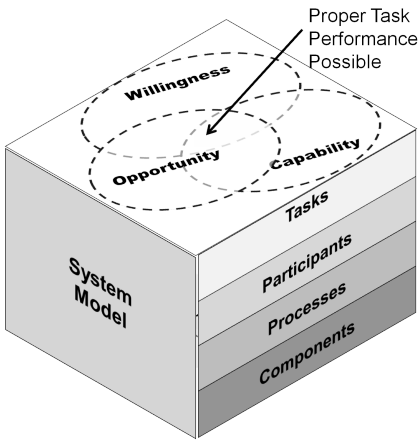

Figure 2.4: A Task's OWC Defines Sets of System Elements

state vice the more general term *model state* when discussing opportunity, as, by definition, only the state of the OEs can affect opportunity. For each task, an opportunity function is defined that maps OE state to a task opportunity state.

Definition 2.3.4. A task *opportunity function (f_O)* is a task-specific function that maps OE states to $\{0, 1\}$. A 1 indicates that the task is in an *opportunity* state and a 0 indicates that the task is in a *no-opportunity* state.

Note that f_O can be mapped to 1 only if the model is in a potential opportunity state (as determined by the set of OEs). The probability of a given potential opportunity state's being mapped to 1 is the opportunity probability (defined next).

Definition 2.3.5. *Opportunity probability (P_O)* is a task-specific probability that a given OE state will result in a task opportunity state. The set of potential opportunity states relative to that task consists of the model states that have a nonzero P_O.

In general, f_O maps each OE state to an arbitrary random variable, and P_O is determined by that random variable, i.e., $f_O : S_{OE} \rightarrow (\{0, 1\} \rightarrow [0, 1])$, where S_{OE} is the set of OE states, and P_O is the probability that the random variable equals 1. However, for the remainder of this thesis, we will simplify the meaning of the opportunity function f_O by restricting it to a purely deterministic function. That is, P_O for a given OE state is either 0 or 1, and a model potential opportunity state always implies an opportunity state for the referenced task, i.e., if state $s \in \{\text{set of potential opportunity states}\}$, $f_O(s) = 1$, and otherwise $f_O(s) = 0$.

Running Example: Opportunity Elements (OE). Per our running example, we will use the following notation to list the OEs for the task EUD: $Element.variable = Value$.

The opportunity elements for EUD are contained within the set $OE = \{U, R\}$, where U is a component element modeling the USB stick, and R is a participant element modeling the business representative.

The OE states relevant to this task are:

- $R.USB\_issued \in \{0, 1\}$, where a 1 means the USB stick has been issued to participant R.
- $U.ESW\_installed \in \{0, 1\}$, where a 1 means that encryption software has been installed on the USB stick.
- $U.location \in \{$workstation (W), transit (T), personal computer $(P)\}$ represents the possible locations for the USB stick.

Running Example: Opportunity Function (f_O). f_O is a Boolean equation that uses as inputs the states of the task-specific OEs above and evaluates to 1 or 0 (see Equation 2.1).

$$f_O = (R.USB\_issued == 1)(U.ESW\_installed == 1)(U.location == W). \tag{2.1}$$

If the model is in a potential opportunity state, which means (per our simplifying assumption) that f_O is evaluated as 1, then we often refer to this situation as on in which *opportunity exists* for task performance, i.e., the task is in an opportunity state. Of course, for a task to be properly performed, there must be not only opportunity, but also willingness and capability. Willingness is discussed next.

2.3.3 Willingness

In this section, we will clarify the terminology and the concepts of willingness, which, like opportunity, is always referenced relative to a given task. A model may be in a *potential willingness state*, and a task may be in a *willingness state*.

Definition 2.3.6. A task is in a *willingness state* or *willingness exists* for a task if a participant has the desire to perform a task. A willingness state is only applicable to tasks performed by humans. If a task is not in a *willingness* state it is in a *no-willingness* state.

Definition 2.3.7. A model *potential willingness state* is a model state in which the probability of willingness existing for a referenced task is nonzero.

Willingness is distinct from the participant's opportunity or capability to perform a task and essentially captures the notion that performance of a task depends on whether a participant wants to perform the task, as well as other factors relevant to task performance.

In terms of our system model, the probability of a task's being in a willingness state is related to the set of model elements that may influence a participant's desire to perform a task. These elements are called the *task-specific set of willingness elements*.

Definition 2.3.8. *Willingness elements (WEs)* are the task-specific set of model elements whose values determine the set of model potential willingness states for the referenced task.

Like OEs, WEs may be drawn in any combination from CHS elements (see Figure 2.4) and may form overlapping sets with other task-specific model element groupings. As with opportunity, a willingness function is defined that maps WE state to a task willingness state.

Definition 2.3.9. A task *willingness function (f_W)* is a task-specific function that maps WE states to $\{0, 1\}$. A 1 indicates that the task is in a *willingness* state and a 0 indicates the task in in a *no-willingness* state.

Note that f_W can be mapped only to 1 if the model is in a potential willingness state (as determined by the set of WEs). The probability of a given potential willingness state's being mapped to 1 is the willingness probability (defined next).

Definition 2.3.10. *Willingness probability (P_W)* is a task-specific probability that a given WE state will result in a task willingness state. The set of potential willingness states relative to that task consists of the model states that have a nonzero P_W.

In general, f_W maps each WE state to an arbitrary random variable, and P_W is determined by that random variable, i.e., $f_W : S_{WE} \rightarrow (\{0, 1\} \rightarrow [0, 1])$, where S_{WE} is the set of WE

states, and P_W is the probability that the random variable equals 1. However, for the remainder of this thesis, we will simplify the willingness function f_W by restricting it to a Bernoulli random variable with a fixed P_W (see Equation 2.2). As discussed further in Chapter 5, use of a state independent random variable facilitates the basic solution method for models built using the OWC ontology.

$$f_W = Bernoulli(P_W). \tag{2.2}$$

Running Example: Willingness Elements (WEs). As discussed above, our running example generates a P_W that is independent of model state. Thus, there are no specified WEs. See Chapter 5 for more details.

Running Example: Willingness Function (f_W). The f_W used in our example is a simple Bernoulli random variable in which the probability of success is simply the willingness probability (P_W). (See Equation 2.2.) Success, i.e., a result of 1, means that the participant is willing to attempt the task.

We have discussed opportunity and willingness, two of the three task states that are required for proper task performance. Next, we will discuss the final task state, capability.

2.3.4 Capability

In this section, we will clarify the terminology and concepts of capability, which, like opportunity and willingness, is always referenced to a given task. A model may be in a *potential capability state* and a task may be in one of several mutually exclusive *capability states*.

Definition 2.3.11. A task is in a *capability state* or *capability exists* for a task if a participant can, i.e., has the ability, to perform a task such that one of a set of mutually exclusive outcomes can be achieved. If a task is not in a *capability* state, it is in a *no-capability* state.

Definition 2.3.12. A model *potential capability state* is one of a set of model states in which the probability of capability existing for a referenced task is nonzero.

Capability captures the idea that a participant, given opportunity and willingness, can properly perform a task, i.e., achieve one of a set of possible task outcomes.

In terms of our system model, the probability of a task's having a capability state is related to the set of model elements that may influence a participant's ability to perform a task. These elements are called the *task-specific set of capability elements*.

Definition 2.3.13. *Capability elements (CEs)* are the task-specific set of model elements whose values determine the set of model potential capability states for the referenced task.

Like OEs and WEs, CEs may be drawn in any combination from CHS elements (see Figure 2.4) and may form overlapping sets with other task-specific model element groupings. As with opportunity and willingness, a capability function is defined that maps CE state to a task capability state. The difference is that a task may in general have more than one capability state, i.e., its associated set of performance outcomes.

Definition 2.3.14. A task *outcome* is a member of a mutually exclusive, task-specific set of events that result from proper task performance.

For example, a binary task has a single proper performance outcome and a no-OWC outcome. Here, the no-OWC outcome can result from a task's having any combination of no-opportunity, no-willingness, and no-capability states.

Definition 2.3.15. A task *capability function (f_C)* is a task-specific function that maps each CE state and task outcome pair to $\{0, 1\}$. A 1 indicates that the task is in a *capability* state with respect to that outcome, and a 0 indicates that the task is in the *no-capability* state.

Note that f_C can map a model state to 1 only if the model is in a potential capability state (as determined by the set of CEs). The probability of a given potential capability state's being mapped to 1 is the capability probability (defined next).

Definition 2.3.16. *Capability probability (P_C)* is a task-specific probability that a given CE state will result in a task capability state for a given outcome. The set of potential capability states relative to that task consists of the model states that have a nonzero P_C.

In general, f_C maps each CE state to an arbitrary random variable, and P_C for that outcome is determined by that random variable, i.e., $f_C : S_{CE} \rightarrow (O \rightarrow [0, 1])$, where S_{WE} is

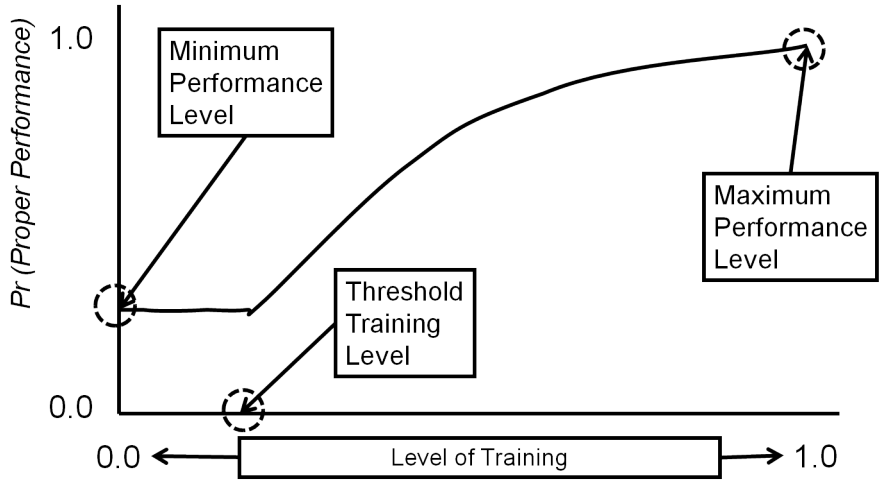

Figure 2.5: Example of a CTF That Maps Training to the Probability of Proper Performance

the set of WE states, O is the set of outcomes, and $P_C(o_i)$ is the probability that the random variable equals 1 for outcome o_i. Note that because the outcomes are mutually exclusive, $\sum_{i=1}^{n} P_C(o_i) = 1$ for a task with n outcomes.

One way of interpreting the capability probability is that some CE states produce a higher probability of capability for a given outcome i, e.g., evaluation of $f_C(o_i)$ as 1, than other states do.

For example, suppose the capability function f_C for an outcome is defined as a state-dependent Bernoulli random variable in which the capability probability (P_C) is the product of n independent *capability transfer functions* (CTFs) (see Equation 2.3).

Each CTF maps a subset of CEs to an independent Bernoulli probability of success in performing some aspect of a task outcome. The CTF shown in Figure 2.5 is an example of the "training" aspect of task performance. This function maps the CE state *level of training* to a probability of successful task performance for that aspect. For the outcome to be achieved, all the associated CTFs must evaluate to 1. The CTF product, P_C, represents the overall probability that the capability for that outcome exists. Note that we assume that the set of CEs can be divided into independent (relative to that task) subsets and that the probabilities are distributed for a set of mutually exclusive outcomes, so that the sum of probabilities for all the outcomes is equal to 1.

$$P_C = CTF_1 \times CTF_2 \times ... \times CTF_n. \tag{2.3}$$

Running Example: Capability Elements (CEs). In our running example, CEs are those model elements that affect the likelihood that the task *Encrypt USB Data* will be properly performed.

The CEs for that task form the set $CE = \{R, H\}$, where R is the participant element modeling the business representative and H is a component element modeling IT help desk support.

In general, CE states can be drawn from a set of discrete values or even a continuous interval. The CE states relevant to this task EUD are:

- $R.training \in \{0.2, 0.8\}$ is the training level of the business representative, and
- $H.available \in \{0, 1\}$ is the availability of IT help desk support where a 1 means support is available.

Running Example: Capability Function (f_C). For our running example, the capability probability is $P_C = CTF_{R.training} \times CTF_{H.available}$, where $CTF_{j.i}$ represents the CTF mapped to state variable i of $CE\ j$. The resulting f_C maps $\{R.training \times H.available\}$ to $\{0, 1\}$ (see Equation 2.4). If f_C can be evaluated as 1, the model is in a potential capability state. If the model is in a potential capability state and f_C is evaluated as 1 relative to a given task outcome, that task EUD is in a capability state for the outcome "USB data encrypted."

$$f_C = Bernoulli(CP). \tag{2.4}$$

2.3.5 Proper Task Performance

Given the above definitions of opportunity, willingness, and capability, it is clear that for a task to have the potential to be properly performed, a model must be in an appropriate potential opportunity state, potential willingness state, and potential capability state, i.e.,

a potential OWC state. Those model states are defined by the states of the OEs, WEs, and CEs that are relevant to the task. It is also clear that because task willingness and capability states are determined by random variables (per our simplifying assumptions), the existence of an appropriate model potential OWC state does not guarantee proper task performance. Rather, the existence of a task-specific potential OWC state defines a *probability* that a task will be properly performed. Equation 2.5 illustrates the relationship between OWC functions

$$f_P = f_O \times f_W \times f_C, \tag{2.5}$$

where f_P is a Boolean function that equals 1 if the task is properly performed; f_O is the opportunity function for a task; f_W is the willingness function for a task; and f_C is the capability function for a task.

The probability that a task is properly performed, $Pr(f_P = 1)$, is equal to the probability that each of the associated OWC functions will be evaluated as 1 (see Equation 2.6). In other words, a task will be properly performed if opportunity, willingness, and capability exist for that task. Note that Equation 2.6 implicitly assumes an independence of probabilities for opportunity, willingness, and capability. In practice, the probability of proper performance is determined in aggregate within the model, thus accounting for any dependencies between OWC elements.

$$Pr(f_P = 1) = Pr(f_O = 1) \times Pr(f_W = 1) \times Pr(f_C = 1). \tag{2.6}$$

2.3.6 Benefits of Using OWC to Define a Model

The OWC ontology is useful in at least two ways. First, during the model definition phase, the OWC ontology provides a structured way of defining and grouping model elements for each task. For example, to model a process, we typically first decompose the process into an ordered arrangement of tasks [45], [46]. While defining each task from the process decomposition, the analyst can directly define other related model elements and element states by listing the task's OWC elements. That provides a "ground-up" approach to defining

a model that is intuitive and also guarantees the minimal state definition for each task.

Second, the OWC ontology speeds up model execution. Only those elements defined by the OWC ontology as related to a task are evaluated for changes when each task is evaluated for execution. That implies a smaller state space to search relative to each task and contributes to simulation efficiency.

2.3.7 Section Summary

In this section, we presented the OWC ontology. First, we defined task performance. Next, we described how a subset of model elements and their associated states could be mapped to three prerequisite task states for task performance: opportunity, willingness, and capability. Each OWC state was defined as a function of a set of model element states, and the probability of proper task performance was related to each model potential OWC state via the OWC functions. In the next section, we will use the OWC ontology to define a special type of task, the human decision point.

2.4 Human Decision Points

Now that we have defined a CHS model as described in Section 2.2 and described how the OWC ontology may be used to associate each task in the CHS model with a set of model elements as described in Section 2.3, we are ready to introduce a special type of task, the *human decision point* (HDP). As we discussed in Chapter 1, one of the goals of this research is to formalize in modeling terms the influence of human decisions on system outcomes. To that end, we now define a unique model element that represents not only the ability of a human to perform a task, but also the *decision* by that same human to *attempt* the task. Using that novel approach to distinguish between performance factors and decision factors, we provide new insight into CHS security outcomes and introduce an quantitative modeling technique that has never been applied to problems of CHS security in this way.

2.4.1 Task Performance

Let us first reframe the problem of modeling human decisions by considering the abstract view of a system model as a set of states and a set of events that define transitions between states [47]. From that viewpoint, a task outcome is an event. Per our simplifying assumption of binary tasks, a task can result in two possible events, proper or not-proper task performance.

The proper performance event occurs when the task is attempted and the opportunity, willingness, and capability for proper performance exist, i.e., the task is in an opportunity, willingness, and capability state. Not-proper performance will occur if opportunity, willingness, or capability does not exist for that task. A task's willingness state depends on a participant's decision to perform the task. Thus, from a modeling point of view, a *human decision point* (HDP) represents a point in the execution of the model in which the next state depends, at least in part, on a human decision.

Using the OWC ontology, we can group task outcomes into two mutually exclusive sets, the *willing* outcomes and the *not-willing* outcomes. Thus, the set of possible outcomes for a task is determined by a decision made by the participant. That is, given that a participant has the opportunity and capability to properly perform a task, the participant must also decide, i.e., be *willing*, to attempt task performance. Since the willingness of a participant is in general determined by the willingness function f_W, and f_W is determined by the willingness element (WE) state, the set of possible task outcomes is therefore influenced by the current states of the task-specific WEs (as described in Section 2.3) and determined directly by f_W. That is, f_W provides a probabilistic measure of how likely the person is to decide to perform the task.

2.4.2 Defining the Human Decision Point as a Special Kind of Task

Given the discussion above of the relation of willingness to task performance, it is straightforward to define an HDP using the OWC ontology as follows.

Definition 2.4.1. A *human decision point (HDP)* is a task performed by a human partici-

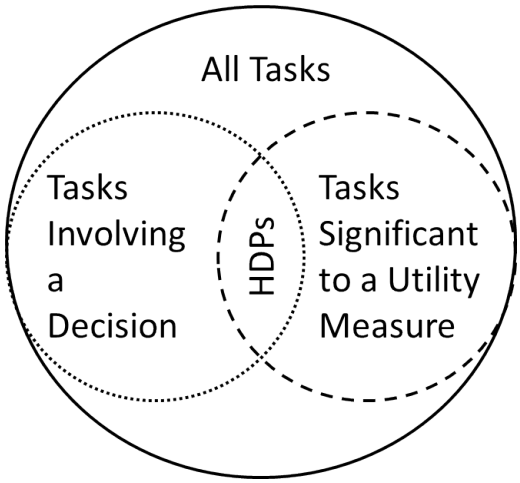

Figure 2.6: The HDP is a Special Type of Task

pant for which willingness is not always true, i.e., f_W is not always 1, and for which the set of outcomes is significant to some utility measure.

Definition 2.4.1 requires that an HDP be a task that requires both a human decision and a significant influence on some measure of interest. The second criterion for an HDP, i.e., significant influence on some measure, is a necessary requirement when characterizing a CHS model as described in Chapter 5. We often refer to the second criterion by saying that the task is "significant." Obviously, what is considered significant varies with the model analyzed and the measure used. Thus, it is important to define what is significant during the conceptual model definition phase. The concept of HDP selection is illustrated with a Venn diagram in Figure 2.6, which shows that the set of HDPs is a subset of all tasks in a model. Furthermore, the set of HDPs is the intersection of the set of tasks that involve a decision and the set of tasks that are significant to system outcomes.

Running Example: HDPs. Continuing our running example from the previous chapter, we now wish to select for special evaluation a task for which human decisions are both relevant (i.e., the participant is allowed to make a decision) and significant (i.e., the decision affects a utility value in an important way) to the USB Stick Usage CHS.

Previously, we defined the task EUD. Recall that the system security policy states that sensitive USB data should always be encrypted; however, the participant may choose whether or not to comply with the stated policy. Because the performance of this task is dependent

upon the participant's willingness to encrypt data, we shall also select the task EUD to be our HDP of interest.

It is not obvious that the EUD task will have a significant effect on a utility function value. However, because the task directly affects both the security and participant utility functions via data availability, it seems to be a good candidate for analysis. By looking at the results in Section 5.8, we shall see that task EUD does indeed have a significant effect on the participant and security utility function values and thus also meets the second criterion for an HDP.

2.4.3 Section Summary

In this section, we defined the HDP in terms of the conceptual CHS model and then used the OWC ontology to define the HDP as a special kind of task. We also presented two criteria for determining whether a task is an HDP. First, the task outcomes must be dependent on a decision by the task performer, and second, the outcomes of the task must be significant to at least one measure of system value. We then provided an example HDP.

2.5 Conclusion

In this chapter, we described our approach for building a conceptual model of a cyber-human system (CHS). We discussed the three phases of the approach, including how to represent a system of interest using the CHS definition set, how to define the relationships between model elements using the OWC ontology, and, finally, how to define special points within the model, called *human decision points*, that are linked to the human decisions of interest. In the next chapter, we will build upon our conceptual CHS model by detailing our approach to building a formal mathematical model of a CHS using the Human-Influenced Task-Oriented Process (HITOP) modeling formalism.

CHAPTER 3

SPECIFYING THE MATHEMATICAL MODEL: THE HITOP MODEL FORMALISM

3.1 Human-Influenced Task-Oriented Process Formalism

In Chapter 2 we discussed the basis and construction of a conceptual cyber-human system (CHS) model. In Section 2.2 we defined what a CHS is. In Section 2.3 we introduced the opportunity-willingness-capability (OWC) ontology to relate CHS elements to task performance. In Section 2.4 we defined the human decision point (HDP) as a special kind of task. With the conceptual model specified, in this chapter we will introduce a mathematical modeling formalism that can be used with the conceptual model to rigorously define a CHS mathematical model. This mathematical model is a further step in building an executable model applicable to CHS security analysis.

To recap our reasoning, building a cyber-security model that can account for human behavior is an important goal because many cyber systems, by design, require interactions with human users. Current research indicates that the security state of cyber systems can be highly influenced by the actions of users [48], [49], [50], [51]. For example, a user's decision not to follow security policy can introduce serious system vulnerabilities.

Nowadays modeling is an important method for designing and assessing cyber systems, and a good model can detect system vulnerabilities before they are evidenced by system performance. A good model of CHS security must be capable of representing the system, the system users, and the interactions between the two.

Traditional approaches to modeling cyber security do not explicitly consider system users. Many models focus on issues with the system hardware and software, such as firewall settings and network configurations, but lack the capability to analyze important aspects outside the set of physical system components.

We propose a mathematical modeling formalism that addresses these issues by explicitly including the actions and decisions of human users within the overall system model. Our goals are to produce a modeling formalism that 1) reflects the effects of human users, especially the effects of human decisions, on CHS security, 2) enables quantitative measures and executable models, 3) captures variable levels of granularity for process and technical system details, 4) is powerful enough to represent most CHSs, 5) is simple and intuitive enough to enable domain experts to construct useful domain models without extensive modeling expertise, and 6) is technically capable of leveraging existing analytic and simulation methods.

Specifically, we present the Human-Influenced Task-Oriented Process (HITOP) formalism, a process modeling formalism that enables the explicit representation of user decisions, as well as other aspects of human performance, within an integrated system model.

We begin in Section 3.2 by reviewing CHSs and process models generally, and discuss why process models are well-suited to modeling of CHSs. In Section 3.4 we introduce the HITOP formalism and detail its application to a CHS. In Section 3.5 we formally define the HITOP set notation, and in Section 3.6 we define how state is represented in a HITOP model. We summarize the chapter in Section 3.7.

3.2 Process Models and Cyber-Human Systems

In this section, we will discuss process models and how they may be applied to CHSs. A *process model* represents the possible flows of actions related to some specified activity. It can be thought of as a map of all the things that could happen as part of that process. A *process instance* (PI) represents the actual performance of that activity and can be thought of as one possible path through the associated process model.

Process models are sometimes also referred to as *workflow models* [43] or *business process models* (when referencing business activities). Much work has been done on applying workflow models to business process management and Web services [43], [52], [53], [54], [55], [56], [57]. Such models are often used to define a business process for the purposes of documentation, analysis, or development.

We have chosen to implement HITOP in terms of a process model because 1) there is a general familiarity with the concepts of process models among our target users in the business world; 2) we believe that the process model viewpoint is a natural and intuitive way to view the actions of humans within a system; 3) a process is a basic element type in the CHS definition set used to build our CHS conceptual model [58]; 4) a wide variety of model granularities are allowed by a general process model that is decomposable into many more detailed subprocesses; and 5) we can leverage analytical and simulation methods already available for process models in our formalism.

In the context of a CHS, we will by necessity use a broader definition of a process model so that it represents not just a business process like "customer billing," but also *any* relevant activity involving people or computers. For example, if a user's job satisfaction is part of the model, the activities that affect job satisfaction can be modeled as a process, even though such activities are normally not included within a cyber-security model.

3.3 Defining the CHS

The first step of constructing a HITOP model is to define the system to be modeled. Recalling the CHS definitions developed in Section 2.2, we will represent a CHS as a collection of four types of elements: components, participants, processes, and tasks. These element types are defined such that:

- *Components* are the physical objects that make up the system,
- *Participants* are the entities that perform actions and make decisions within a system,
- *Processes* are maps for all possible sequences of tasks within the system, and
- *Tasks* are the units of action within a system.

Recall from Section 2.2 that a compact statement of the relationship among CHS elements is "a *process* is a defined flow of *tasks* performed by one or more *participants* using one or more system *components*."

Thus, a domain expert begins construction of a HITOP model by first decomposing the system of interest into sets of tasks, processes, participants, and components. Once the CHS has been represented as sets of CHS elements, it is straightforward to construct a process model. Each CHS element will be represented by a corresponding type of HITOP element defined within the HITOP model. As shown in the next section, CHS processes and tasks will be represented by HITOP processes with embedded tasks, CHS participants will be represented by HITOP participants, and CHS components will be represented by HITOP components.

3.4 Human-Influenced Task-Oriented Process Model Formalism Description

In this section, we will introduce our process model formalism, known as the *Human-Influenced Task-Oriented Process* or *HITOP* Model. This formalism represents human behavior, especially human decisions, directly as first-class model elements. We will first cover some basic process modeling background.

3.4.1 Why a New Process Modeling Formalism?

In designing this formalism, we reviewed numerous existing process modeling tools and formalisms. None of them turned out to be a good match for our purposes, but one formalism in particular, Yet Another Workflow Language (YAWL) [37], provided many of the formal definitions and structures we required, such as an ability to implement a large variety of control flow patterns. Additionally, the research associated with YAWL on workflow patterns [43] was useful in understanding the generic properties of an all-purpose process model. However, YAWL did not meet our research objectives for several reasons. First, the formal YAWL specification, as detailed in [37], was insufficient for implementing an executable model per our requirements. Second, YAWL, while far simpler to use than other process modeling formalisms, like BPMN [35], still required a feature set too extensive for our goals. Third, YAWL was intended to perform workflow analysis with respect to control, information, and

resource flow and not discrete event simulation, so YAWL did not feature a required temporal aspect. Thus, while YAWL has served as the inspiration for HITOP, and where possible we have tried to preserve terminology and symbolism congruent with YAWL, HITOP is in many ways different from YAWL in purpose and implementation. Section 3.4.11 describes some of the differences between HITOP and YAWL.

3.4.2 Flow Perspectives within HITOP

In general, flow within a process model is grouped into three categories: control, information, and resource [54], [55], [56]. Control flow describes the sequence in which tasks are performed. Information flow describes how data are produced and distributed throughout the process. Resource flow describes who and/or what is assigned/needed to perform tasks. Another way to describe flows within a process model is to answer the following questions for each task:

- Who is performing the task?
- What are the information and resource requirements to perform the task?
- What are the task outcomes, i.e., what happens when the task is performed?
- How long does it take to perform a task?
- When should the task be performed? and
- Where (e.g., in what context) can the task be performed?

HITOP answers these questions using a primarily *control* flow-focused perspective (see Section 3.4.3). HITOP was designed using this perspective because we are particularly interested in explicitly representing and measuring the order and outcomes of task performance, vice information or resource routing, within an organization. Information and resource flows are handled within HITOP via shared global state variables and task definitions. Effectively, shared state variables directly connect all model elements, and, if the information and resource flow were explicitly drawn, this relationship would form a complete (thus not very interesting in terms of flow pattern analysis) graph with the model elements as nodes.

3.4.3 Control Flow Patterns

Control flow describes the order or sequence in which a set of tasks are executed. A pattern-based analysis of existing process models [54] describes a set of control flow patterns that encompass the majority of possible control flows. To meet our goal of a simple model, we designed HITOP to implement only a basic subset of these patterns, but with sufficient variety that most systems of interest could be represented.

In HITOP a control flow pattern is implemented as a set of tasks connected by flow arcs. Control moves from one task to another along the flow arcs, providing a sequence or order of task performances. Often this control flow is imagined as a "token" moving from task to task along the flow arcs. In HITOP, we formalize this idea by referring to the location of process instance (PI) tokens (sometimes shortened to just *tokens*) within the process model to describe the states of tasks. Each token is unique, but may also be related to other tokens (see Section 3.6). HITOP uses colored tokens, and each token is assigned three colors distinguishing it in terms of three qualities. Specifically it has a family color that indicates the root token that began the process, a generation color that identifies siblings, and an individual color that makes each token unique. We will discuss the possible state of tasks in more detail in the next section and use the term *type* to describe the set of family and generation colors associated with a token. For example, sibling tokens entering an AND-type join must all be of the same type, i.e., the same family and generation color.

3.4.4 Task States

There are two viewpoints or levels of HITOP state: the process level and the task level. The *process-level* viewpoint of HITOP state is useful for understanding the state of a process, e.g., what tasks are currently being performed, and the *task-level* viewpoint is useful for understanding what is happening inside each task as it executes. The state of a task can be considered at either level, with the task-level viewpoint being the more detailed of the two. We will confine our discussion in this chapter to the process-level viewpoint and save the more detailed discussion of the task-level viewpoint for Chapter 4.

At the process level, each task can have two possible states: active and inactive. An active

Table 3.1: Task State Changes

Event	Inactive State	Active State
tokens entering task	Active State	*
tokens exiting task	*	Inactive State

task represents the execution of some activity. An inactive task represents no activity, or idleness. Two events can occur with respect to tasks and tokens: *enter tokens* and *exit tokens*. When tokens enter an inactive task, i.e., the enter tokens event occurs, the task state changes to active. When an active task changes to the inactive state, e.g., completes whatever job it was performing, the exit tokens event occurs. For clarity, we focus on token movement, not the task state change, when naming events.

The possible task state transitions are summarized in Table 3.1, where the intersection of an event row and a current state column indicates the next task state. It should also be noted that the event-state combinations in which an "$*$" appears cannot occur because of the definition of task state at this level. A more detailed definition of task state will be provided in Section 3.6.

3.4.5 Connectors and Flow Arcs

Each task has a set of entering and exiting flow arcs. Each arc is connected from only one task to only one task. Entering flow arcs connect to a task via a *join* connector and are used to enter tokens. Exiting flow arcs connect to a task via a *split* connector and are used to exit tokens. A task's join connector represents the function of tokens or the set of rules followed to enter the task, e.g., the number and type of tokens that must be present on the arcs (also called an *entering set*). A task's split connector represents a function describing how tokens should exit a task, e.g., the number and type of tokens to be placed on the exiting arcs (also known as the *exiting set*). See Figure 3.1 for an illustration. Note that graphically, join connectors are always attached to the left of a task, and split connectors are always attached to the right of a task.

Figure 3.1: Task and Connectors

3.4.6 Types of Connectors

Each split or join connector may be of one of three types: AND, XOR, or OR. AND-type joins require an entering set consisting of tokens of the same type, i.e., the same process instance family and generation color (described in Section 3.6), on all of the entering arcs. AND-type splits produce an exiting set consisting of one token of the same type on each of the exiting arcs.

XOR-type joins require an entering set consisting of a single token on any one of the task's entering arcs. XOR-type splits produce an exiting set consisting of a single token on a single exiting arc. Conceptually, XOR-type connectors are considered "pass through" connectors in that they pass a token through a task without splitting it or joining it with other tokens. One consequence, is that tokens that reach an XOR-type join are immediately entered into the task. That is, the tokens do not have to wait to synchronize with other tokens.

OR-type joins (splits) are more complicated. They can require (produce) tokens on a subset of the entering (exiting) arcs as defined by some model-state-dependent function. There are several ways to define the functioning of an OR-type join or split. We have chosen one of the more simple definitions, and believe that even though it is relatively simple, it will still allow us to build a wide variety of useful process structures.

The OR-connector is defined using $OR - x$ where $x \in \mathbb{N}$ represents the number of tokens of a color in an entering or exiting set. If the $-x$ is not specified, the default size of the entering (exiting) set is the number of entering (exiting) arcs connected to the OR. Just as with AND and XOR connectors, a task will not be entered until an entering set defined by the OR-type join is present on the entering arcs. An OR-type split will produce an exiting

set that meets that requirement using a state-dependent function that maps tokens to exiting arcs. See Figure 3.3 for a graphical representation of HITOP connectors.

3.4.7 Control Flow Branching and Merging

We will illustrate control flow patterns with a set of examples. For an exhaustive review of control flow patterns, see [54]. Here we will confine our discussion to process control flow branching and merging.

Control flow branching involves activation of one or more paths (connected via the exiting arcs) from a starting task. Whenever a token is placed in an exiting arc, that path is activated. An XOR split places an exiting token on only one arc and thus implements a "choose one out of many" branching function. An AND split places a token on each exiting arc and thus implements a "choose all" branching function. An OR split places a token on some of the exiting arcs and thus implements a "some of many" branching choice. When more than one path is activated at the same time, that represents the starting of parallel activities within the process.

In a similar way, control flow merging involves merging of two or more active paths into a single active path. It occurs when tokens arriving from different active paths enter a single task and are combined into a single active path. An AND join will merge active paths from all entering arcs, and an OR-x join will merge x active paths among its entering arcs. An XOR join requires only a single token to enter a task, and thus does not merge multiple active paths, but passes through a token from one path to another.

In HITOP, control flow is represented using solid, directed arcs pointing from one task to another task.

As an example, consider the simple flow in Figure 3.2. Assume that task T_1 has an AND type split connector. When T_1 exits, tokens (represented by P's in the figure) are placed on both exiting arcs. That corresponds to placing of tokens on the input arcs of tasks T_2 and T_3, and indicates that the next step in the process is the parallel activation of process flow paths begun by T_2 and T_3.

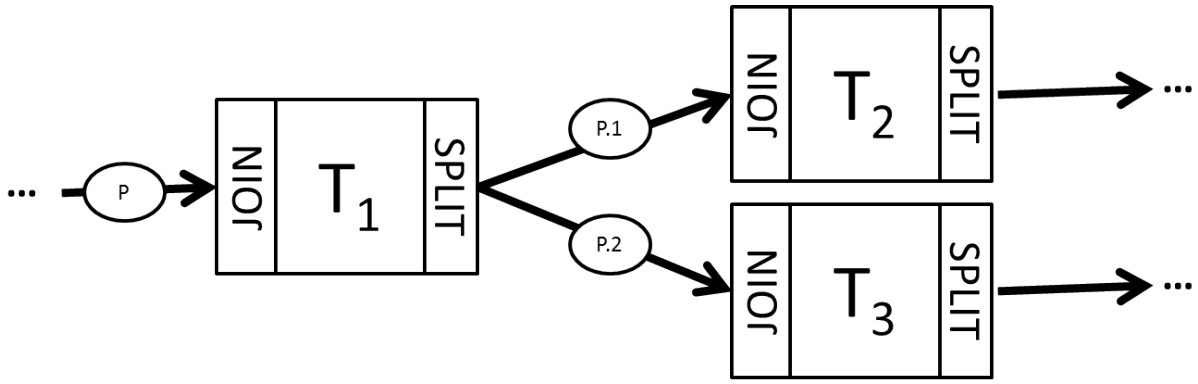

Figure 3.2: Parallel Process Flow

3.4.8 Tasks

A HITOP task represents the performance of some discrete action or set of actions.

Task Types

HITOP has two basic types of tasks: atomic and composite. An *atomic* task represents the lowest level of granularity in a process and is a complete "unit" of work. The actual level of detail at which an atomic task is defined is model-dependent. A task must be defined with sufficient detail to capture important system properties and to enable adequate measures of the system, e.g., utility functions (discussed in Chapter 5).

A *composite* task (also called a *parent* task) links to another process (called a *child* process). A composite task can be thought of as a pointer to and from its associated child process. When a token enters a composite task, that task passes that token to start the child process. When the child process ends, it returns that token to the parent task, and the parent task then exits the token. Composite tasks are useful in representing portions of a process model compactly for readability or to divide a process model functionally.

HITOP represents a task as a box with connectors attached to either side (see Figure 3.3). Atomic tasks have single borders and composite tasks have double borders. If no connector is shown on a side of a task, the task is treated as if an AND connector were attached in that position.

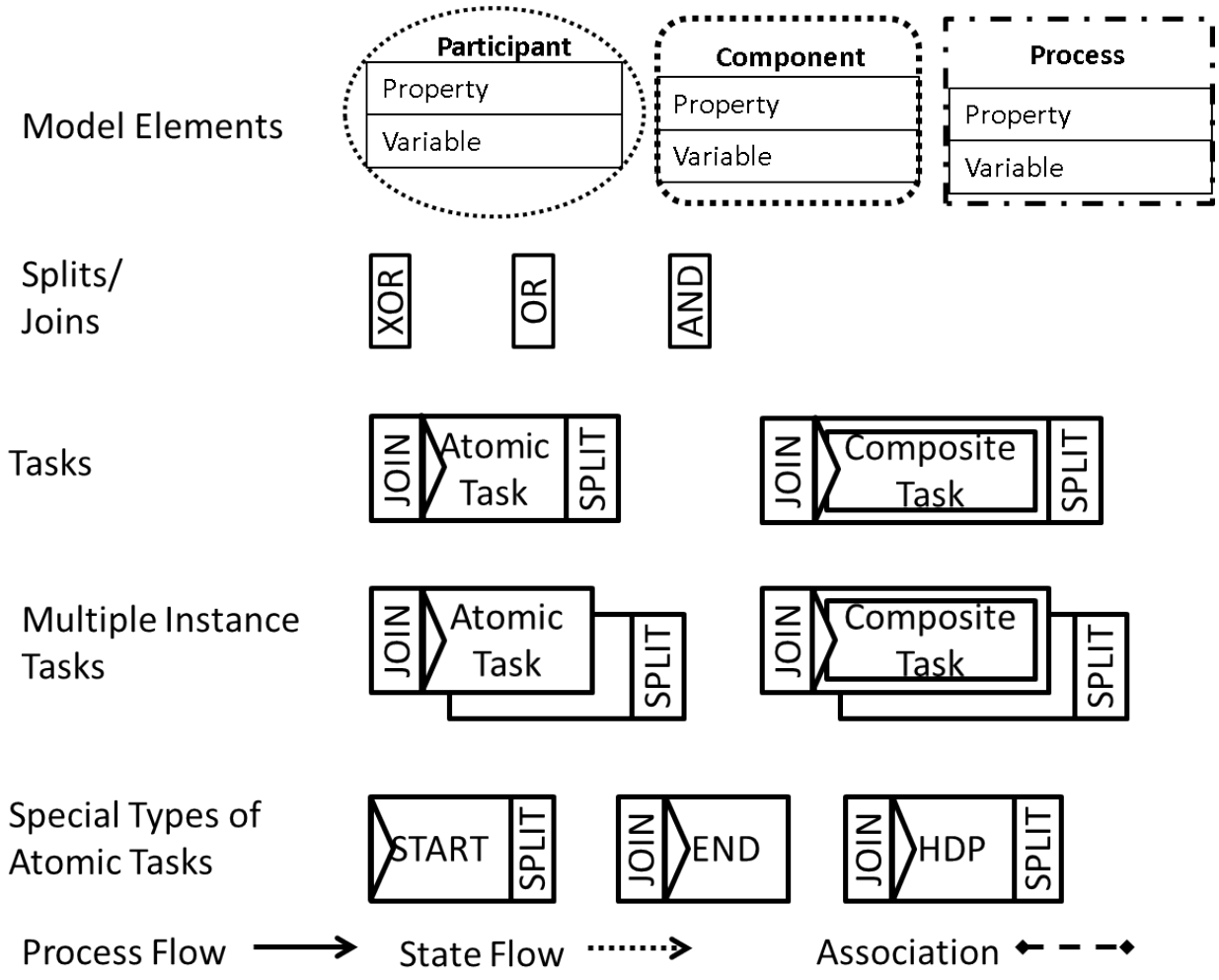

Figure 3.3: HITOP Elements

Special Kinds of Atomic Tasks

There are three special kinds of atomic tasks: start, end, and HDP. A *start task* is used to start a process instance by exiting a token. An *end task* is used to end a process instance by entering a token. Each process has a single start task and a single end task (similar to the single start and end conditions of YAWL [37]). Tokens flow from the start task to the end task via activated paths within the process.

An *HDP task* represents a human decision point (as introduced in Section 2.4). The HDP task differs from a general atomic task in that its outcomes depend in part on the decisions made by the human participant performing the task. This means that the willingness function, f_W (defined in Section 3.5.3), associated with this task will not always evaluate to 1.0 during model execution.

HITOP uses labels to differentiate these special types of atomic tasks. Start tasks are labeled with "START"; end tasks are labeled with "END"; and HDP tasks are labeled with "HDP".

Categories of Tasks

There are two basic instance categories of tasks: single and multiple. A *single-instance* task, as the name implies, represents a single task in the process model. A *multiple-instance task* can represent two or more copies of that task. Activating a multiple-instance task of x instances is functionally equivalent to activating x copies of a single-instance task. Instances may be activated in series or parallel depending upon the type of multiple-instance task. It should also be noted that the start, end, and HDP tasks cannot be multiple-instance tasks. However, in general, atomic tasks and composite tasks may also be multiple-instance tasks. That is, a task is described by both its task type, e.g., atomic or composite, and its instance category, e.g., single or multiple.

HITOP represents a single-instance task with the default symbols and a multiple-instance task as two slightly offset boxes (see Figure 3.3).

3.4.9 Participants

A HITOP participant is a model element that consists of a set of related state variables. It represents the aspects of the associated CHS participant that are pertinent to the tasks being performed and the system utility functions.

All HITOP participants must have three state variables, called *properties*. One property, MULTITASK, represents the ability of the participant to multi-task, e.g., to perform more than one task at a time. A second property, RESOURCES, represents the amount of resources a participant has to apply to a given task. A third property, BUSY, indicates whether the participant is busy and unable to perform a new task. For example, if a task requires x units of abstract work effort, a participant can only be assigned to this task if the participant resource property is at least equal to x.

HITOP participants may also have additional state variables defined as required for a particular CHS model. Those types of state variables are called simply *variables*. For example, a participant may use a variable to represent the amount of training received by a person to perform a certain task.

When a task is activated, a set of participants is associated with the task. This association can be thought of as the selection of a participant (or group of participants) that meets certain predefined task requirements, and the assignment of the participant(s) to the task. Additionally, the resources required to perform the task are removed from the participant resource property. In order for a task to be properly performed, it must have at least one associated participant.

When a task returns to an inactive state, the participant association is removed, and the resources that were required by the task are returned to the participant resource property.

HITOP represents a participant using a hashed oval. Associations of participants with tasks are indicated by a hashed line (see Figure 3.3).

3.4.10 Components

Much like a participant, a HITOP component is a model element that consists of a set of related state variables. A HITOP component represents the aspects of the associated CHS

component that are pertinent to the modeled system and the system's utility functions.

All HITOP components must have three state variables, called *properties*. One property, SHARED, represents the capacity of the component to be shared, e.g., to be used to perform more than one task at a time. A second property, RESOURCES, represents the amount of resources a component has to apply to a given task. A third property, INUSE, indicates whether the component is already in use and unavailable for use in a new task. For example, if a task requires x units of abstract resources, a component can only be assigned to this task if the component resource property is at least equal to x.

HITOP components may also have additional state variables defined as required for a particular CHS model. Those types of state variables are called *variables*. For example, if a component represents a hard drive, it may use a variable to represent the amount of memory available to store new data.

When a task is activated, a set of components is associated with the task. This association can be thought of as the selection of a component (or group of components) that meets certain predefined task requirements, and assignment of the component(s) to the task. Additionally, the resources required to perform the task are removed from the component resource property. To be performed, a task must have at least one associated component.

When a task returns to an inactive state, the component association is removed, and the resources that were required by the task are returned to the component resource property.

HITOP uses a hashed rectangle to represent a component. Associations of components with tasks are indicated by hashed lines (see Figure 3.3).

3.4.11 Processes

A HITOP process is a structured flow of tasks grouped according to some unifying purpose (see Figure 3.4). It consists of sets of tasks and associated connectors that are joined together by directed arcs. Each arc connects a single pair of tasks. Each process starts from a single start task that only has exiting arcs, and ends with a single end task that only has entering arcs. Tokens are exited by the start task and flow along the directed arcs from task to task until they reach the end task. Tokens that reach the end task are entered. When all tokens

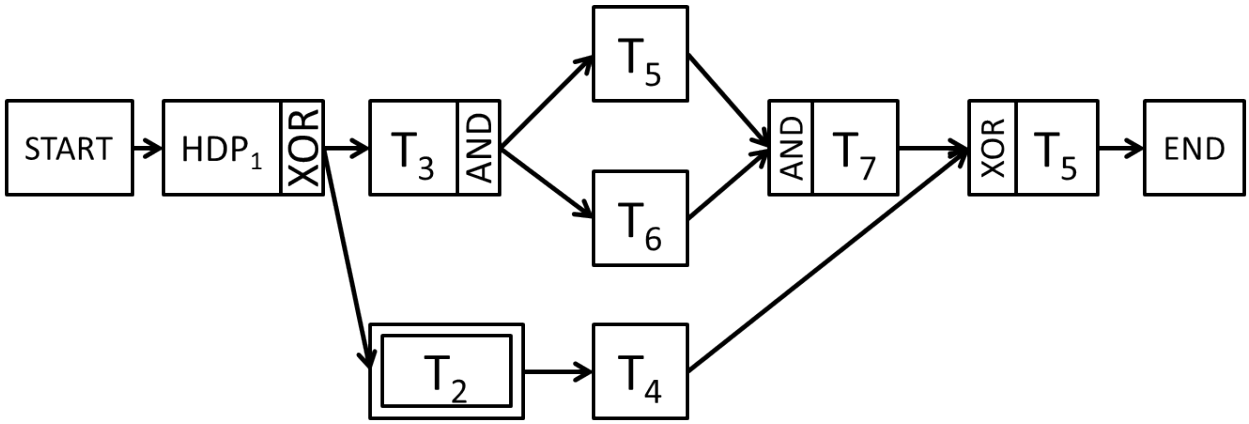

Figure 3.4: Example HITOP Process

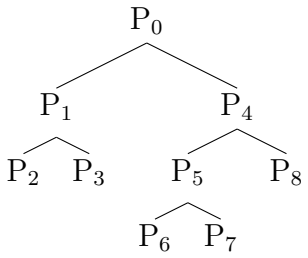

Figure 3.5: Process Tree

have been entered into the end task, the process is complete.

All HITOP processes have a property, PARENT, that points to the parent task of the process. All processes except the *root* process have a parent task. If a set of processes in a HITOP model, for example $\{P_0, P_1, \cdots, P_8\}$, were represented as a tree (as in Figure 3.5), the process P_0 would be the root process or the root of the tree, and the other child processes would form the branches. Each link between a parent and a child node in the tree is a composite task.

HITOP represents a process as a directed graph of tasks, connectors, and arc symbols (see Figure 3.4).

Relationship to YAWL

As discussed in the introduction to this section, HITOP implements a subset of YAWL features in its basic structures where possible. For those familiar with YAWL and other

popular process-modeling formalisms, many of the concepts and structures described above should seem familiar. However, HITOP differs in some significant ways from YAWL. The reasons include differences between the basic goals of YAWL and HITOP, HITOP's need for a feature set more basic than a full YAWL implementation, and HITOP's need to add additional structures that YAWL, as defined, did not support. Some differences include:

- YAWL uses both explicit and implicit conditions. HITOP uses only implicit conditions.
- HITOP uses a restricted definition of OR-type functionality whereas YAWL uses a more general OR functionality.
- HITOP does not have the capability to instantiate additional instances of a multiple-instance task once it is active. YAWL does.
- HITOP does not currently implement a specified removal set feature associated with tasks, as does YAWL, but performs implicit cancellation of sibling tokens to implement OR-type joins and complete multi-instance tasks.
- HITOP is designed specifically with discrete-event simulation in mind, leading to different internal task structures and interpretations of the basic process different than those in YAWL.
- YAWL, as defined, did not have a concept of execution time such as we implemented in HITOP.
- Significant additional structure was added to the HITOP task definition to support the OWC ontology and the HDP implementation including the "no opportunity," "no willingness," and "no capability" bags and the use of *parallel activities* within a task.
- In the original YAWL paper [37], data and resource flow were not defined. HITOP implements those flows using state variables.

We have now provided a basic description of the parts of a HITOP model. In the next section, we specify each of the parts using set notation.

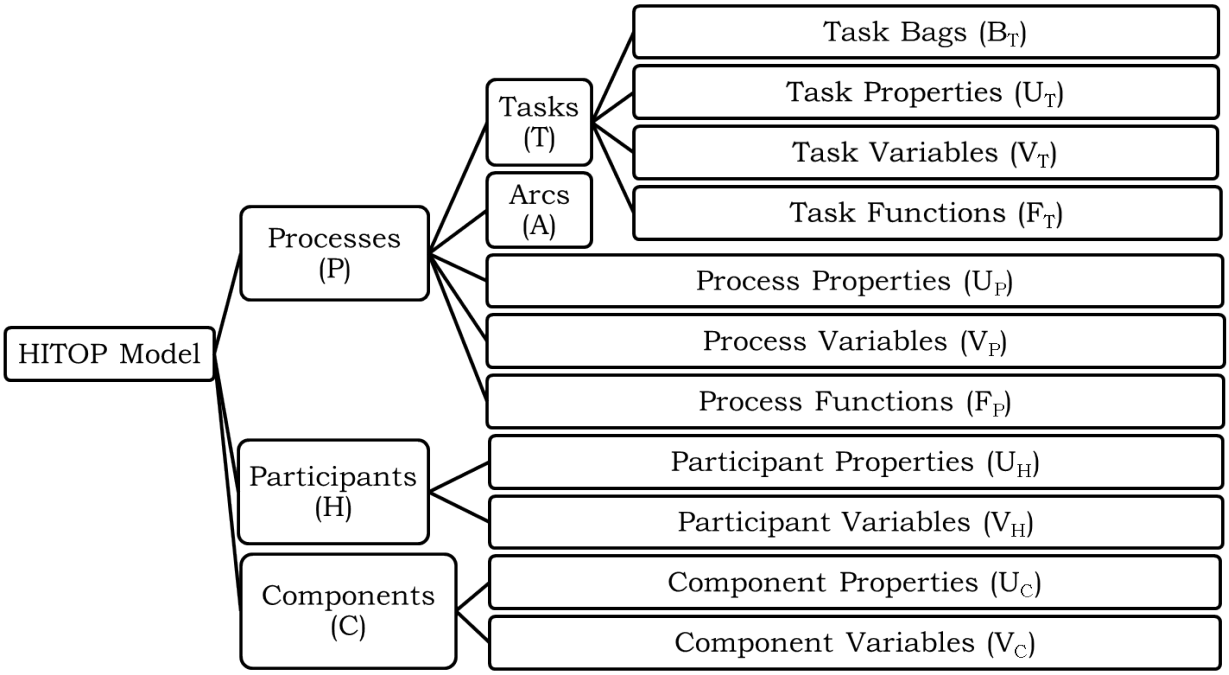

Figure 3.6: HITOP Structure

3.5 HITOP Set Notation

In this section, we will define the HITOP modeling formalism precisely using set notation. We will start with high-level model details and then become increasingly more detailed in describing individual model elements. Each element described in Section 3.4 will be formally defined using set notation in Section 3.5.

3.5.1 HITOP Model

A HITOP model, M, is defined by the tuple of model elements

$$M = (P, H, C), \tag{3.1}$$

where $P = \{P_1, P_2, \ldots, P_n\}$ is the set of n *processes* in the model, $H = \{H_1, H_2, \ldots, H_q\}$ is the set of q *participants* in the model, and $C = \{C_1, C_2, \ldots, C_r\}$ is the set of r *components* in the model. Figure 3.6 illustrates the general structure of a HITOP model.

3.5.2 Process Notation

A process represents an ordering of tasks as described in Section 3.4.11. Each *process* $P_i \in P$ in the model is defined by the tuple:

$$P_i = (T_{P_i}, A_{P_i}, PARENT_{P_i}, V_{P_i}, F_{P_i}). \tag{3.2}$$

$T_{P_i} = \{T_1^{P_i}, T_2^{P_i}, \ldots, T_m^{P_i}\}$ is the set of m process-specific *tasks* associated with process i. When the process context is clear, we may omit the superscript "P_i" for this and the following notations. Tasks are described in Section 3.4.8 and defined in greater detail in Section 3.5.3.

$A_{P_i} = \{a_1, a_2, \ldots, a_s\}$ is the set of s process-specific directed *arcs* associated with process i. Each arc points from a single task to a single task and can "hold" process instance tokens (as discussed later in this section).

$PARENT_{P_i} \in \{\{\bigcup_{P_i \in P} T_{P_i} | T_{P_i}.TYPE = Composite\} \bigcup \emptyset\}$ is a property indicating the parent task of process i. Each process has a parent task that is in the set of composite tasks or is empty. As will be described later, a *composite task* is considered the parent task of the child process that it references. Note also the use of the term *property*, which is one of the three basic types of state variables. A property takes on a single value (from a set of predefined possible values) for each model state and must be specified for every HITOP model. In this case, the value of the PARENT property is a pointer to a composite task. As discussed in Section 3.4.8, each set of processes has a single root process that begins model execution. The root process may be identified as the only process without a parent task, i.e., a PARENT property equal to $\emptyset$.

$V_{P_i} = \{v_1, v_2, \ldots, v_e\}$ is the set of e *variables* associated with process i. Here we use the term *variable* to specify the second type of basic state variables. A variable, like a property, takes on a single value (from a set of possible values) for each model state. However, the set of variables used is specified for each individual model at design time.

$F_{P_i} = \{f_{IN}, f_{OUT}, f_{JOIN\_TYPE}, f_{SPLIT\_TYPE},$
$f_{JOIN}, f_{SPLIT}, f_{ORJOIN\_NUM}, f_{ORSPLIT\_NUM}\}$ is the set of process-specific functions associated with process i, where:

- $f_{IN} : T_{P_i} \rightarrow 2^{A_{P_i}}$ maps each task to a set of entering arcs,
- $f_{OUT} : T_{P_i} \rightarrow 2^{A_{P_i}}$ maps each task to a set of exiting arcs,
- $f_{JOIN\_TYPE} : T_{P_i} \rightarrow \{AND, OR, XOR\}$ maps each task to a join type,
- $f_{SPLIT\_TYPE} : T_{P_i} \rightarrow \{AND, OR, XOR\}$ maps each task to a split type,
- $f_{JOIN} : T_{P_i} \times S \rightarrow \{0, 1\}$ maps each task, $T_j \in T_{P_i}$, and model state, $s \in S$, to 0 or 1, representing the entering function for task T_j,
- $f_{SPLIT} : T_{P_i} \times A \times S \rightarrow \{0, 1\}$ maps each task, $T_j \in T_{P_i}$; model state, $s \in S$; and arc, $A \in A_{P_i}$ to a Boolean value, where a 1 indicates that an exiting token should be placed on that arc,
- $f_{ORJOIN\_NUM} : T_{P_i} \rightarrow \mathbb{N}$ maps each task T_j to a natural number designating the minimum number of tokens needed to activate an OR join, and
- $f_{ORSPLIT\_NUM} : T_{P_i} \rightarrow \mathbb{N}$ maps each task T_j to a natural number designating the number of tokens output from an OR split.

Process-specific functions describe aspects of process instance token flow within the process. Unlike the set of task-specific functions described next, process-specific functions are independent of the set of tasks underlying the process. Process-specific functions support the process-level viewpoint.

3.5.3 Task Notation

Tasks represent units of activity. They are described in Section 3.4.8. Each *task* $T_j^{P_i} \in T_{P_i}$ is defined by the tuple:

$$T_j^{P_i} = (B_{T_j}, V_{T_j}, U_{T_j}, F_{T_j}). \tag{3.3}$$

The elements of each task tuple are define below.

$B_{T_j} = \{b_{ENT}, b_{EXE}, b_{TO}, b_{COMP}, b_{NoO}, b_{NoW}, b_{NoC}\}$ is the set of task-specific *bags* associated with task j. Each $b \in B_{T_j}$ is a bag-type state variable which holds PI tokens as they

move through a task. Note that the superscript denoting the associated process and task has been omitted in the interest of clarity, but may be added if disambiguation is required, e.g., $B_{T_j}^{P_i}$ would reference the set of bags in task j in process i or $b_{ENT}^{T_j P_i}$ the "entered" bag in task j in process i. When the process and task context is clear, we may omit the superscripts T_j and P_i as in the following notations. The task-level state of a task is determined by the set of tokens in each of its bags. For example, a token in the no-willingness bag, b_{NoW}, indicates a no-willingness task state for that PI token.

$V_{T_j} = \{v_1, v_2, \ldots, v_g\}$ is the set of g variables associated with task j. It should be noted that these variables take on values that are tied to the task and not the particular PI executing the task. For example, a task variable may hold a value that counts the number of times the task has been attempted by all PIs executing the process.

$U_{T_j} = \{TYPE, INST, MIN, MAX, THRES, CHILD, O, TIMEOUT\}$ is the set of properties associated with task j where:

- $TYPE \in$ {Atomic, Composite, Start, End, HDP} defines the task type,
- $INST \in$ {Single, Multiple} defines the task instance category,
- $MIN \in \mathbb{N}$ represents the minimum number of instances created for a multiple-instance task,
- $MAX \in \mathbb{N}$ represents the maximum number of instances that can be created for a multiple-instance task,
- $THRESH \in \mathbb{N}$ represents the threshold for task instances that must be completed to complete a multiple-instance task,
- $CHILD \in P$ indicates the child process associated with a composite task,
- O is the set of possible task outcomes, and
- $TIMEOUT \in \mathbb{R}$ is the timeout setting for a task.

$F_{T_j} = \{f_{PRED\_ENTER}, f_{PRED\_EXEC}, f_{PRED\_COMPLT}, f_{PRED\_TIMER},$
$f_{PRED\_OCHECK}, f_{PRED\_EXIT}, f_{ENTER}, f_{EXEC}, f_{COMPLT}, f_{EXIT},$

$f_{TIMER}, f_{OCHECK}, f_{OPP}, f_{WILL}, f_{CAP}, f_{DIST}, f_{OUTCOME}$,
$f_{EligH}, f_{EligC}, f_{Hexec}, f_{Cexec}, f_{Hcomplt}, f_{Ccomplt}, f_{NUMINST}\}$ is the set of task-specific functions associated with task j, where:

- $f_{PRED\_ENTER} : S \rightarrow \{0, 1\}$ represents an enabling predicate for the enter activity in each task,
- $f_{PRED\_EXEC} : S \rightarrow \{0, 1\}$ represents an enabling predicate for the execute activity in each task,
- $f_{PRED\_COMPLT} : S \rightarrow \{0, 1\}$ represents an enabling predicate for the complete activity in each task,
- $f_{PRED\_TIMER} : S \rightarrow \{0, 1\}$ represents an enabling predicate for the timer activity in each task,
- $f_{PRED\_OCHECK} : S \rightarrow \{0, 1\}$ represents an enabling predicate for the opportunity check activity in each task,
- $f_{PRED\_EXIT} : S \rightarrow \{0, 1\}$ represents an enabling predicate for the exit activity in each task,
- $f_{ENTER} : S\times \rightarrow S$ is the state transition function for the enter activity in each task,
- $f_{EXEC} : S \rightarrow S$ is the state transition function for the execute activity in each task,
- $f_{COMPLT} : S \rightarrow S$ is the state transition function for the complete activity in each task,
- $f_{EXIT} : S \rightarrow S$ is the state transition function for the exit activity in each task,
- $f_{TIMER} : S \rightarrow S$ is the state transition function for the timer activity in each task,
- $f_{OCHECK} : S \rightarrow S$ is the state transition function for the opportunity check activity in each task,

- $f_{OPP} : S \times I \rightarrow \{0, 1\}$ maps a model state and process instance to a 0 or 1 describing the opportunity function in each task,

- $f_{WILL} : S \times I \rightarrow (\{0, 1\} \rightarrow [0, 1])$ maps model state and process instance to a distribution function of a Bernoulli random variable that describes the willingness function in each task,

- $f_{CAP} : S \times I \rightarrow (O \rightarrow [0, 1])$ represents the capability function in each task, which maps a model state and process instance to a random variable that maps each outcome $o \in O$ to a probability for that outcome $Pr(o)$,

- $f_{DIST} : S \rightarrow (\mathbb{R} \rightarrow [0, 1])$ maps model state to a distribution function of a continuous random variable that describes the time to complete a task,

- $f_{OUTCOME} : S \times O \rightarrow S$ maps a model state and task outcome to a new state,

- $f_{EligH} : M_{\hat{H}} \rightarrow \{0, 1\}$ maps each possible association of a subset of participants, $\hat{H} \subseteq H$, to a binary value where a 1 indicates that the subset of participants $\hat{H}$ is eligible to perform the task,

- $f_{EligC} : M_{\hat{C}} \rightarrow \{0, 1\}$ maps each possible association of a subset of components, $\hat{C} \subseteq C$, to a binary value where a 1 indicates that the subset of components $\hat{C}$ can be used to perform the task,

- $f_{Hexec} : M_{\hat{H}} \rightarrow M_{\hat{H}}$ maps each possible state of a subset of participants, $\hat{H}$, to a new state as a result of executing a task,

- $f_{Cexec} : M_{\hat{C}} \rightarrow M_{\hat{C}}$ maps each possible state of a subset of components, $\hat{C}$, to a new state as a result of executing a task,

- $f_{Hcomplt} : M_{\hat{H}} \rightarrow M_{\hat{H}}$ maps each possible state of a subset of participants, $\hat{H}$, to a new state as a result of completing a task,

- $f_{Ccomplt} : M_{\hat{C}} \rightarrow M_{\hat{C}}$ maps each possible state of a subset of components, $\hat{C}$, to a new state as a result of completing a task, and

- $f_{NUMINST} : S \rightarrow \mathbb{N}$ maps a model state to the number of instances to create for a multiple-instance task.

Task-specific functions define the inner workings of each task, including how model state changes during the execution of the task. Task-specific functions support the task-level execution viewpoint. In general, each task activity must have an enabling predicate to specify the conditions that enable the activity and a transition function to specify what happens when the activity fires. Also, before a task can be executed it must be associated with an eligible set of participants and components. Thus, each task must have an eligibility function to indicate which participants and components can perform the task. The state of associated participants and components is changed when executing and when completing the task. Thus, a execution function and a completion function must be specified that change participant and component states during task execution and after task completion.

3.5.4 Participant Notation

Participants are the initiators and performers of actions within a HITOP model. Participants are described in Section 3.4.9. Each *participant* $H_i \in H$ in the model is defined by the tuple:

$$H_i = (MT_{H_i}, R_{H_i}, BUSY_{H_i}, V_{H_i}). \tag{3.4}$$

The elements of the participant tuple are defined below.

$MT_{H_i} \in \{0, 1\}$ is a participant property indicating whether multitasking is allowed. If $MT_{H_i} = 1$ then multitasking is allowed, and the participant may perform as many tasks simultaneously as allowed by available participant resources.

$R_{H_i} \rightarrow \mathbb{N} \bigcup \infty$ is a participant property indicating the amount of resources a participant has to perform tasks. We use the term *resources* here in a very general sense. The specific interpretation of resources is dependent upon the task to which the participant is assigned; however, in most cases, the resources can be related to the time needed for a participant to perform a task.

$BUSY_{H_i} \in \{0, 1\}$ is a participant property indicating whether the participant is currently

busy performing a task. If $\text{BUSY}_{H_i} = 1$, then the participant is busy and cannot be assigned to perform another task.

$V_{H_i} = \{v_1, v_2, \ldots, v_s\}$ is the set of s participant-specific variables associated with participant i. The set and possible values of variables are defined as required for each specific model.

3.5.5 Component Notation

Components represent the physical objects used to perform a task. Components are described in Section 3.4.10. Each *component* $C_i \in C$ in the model is defined by the tuple:

$$C_i = (SHARED_{Ci}, R_{Ci}, INUSE_{Ci}, V_{Ci}). \tag{3.5}$$

The elements of the component tuple are defined below.

$SHARED_{Ci} \in \{0, 1\}$ is a component property indicating whether the component may be shared among tasks. If $SHARED_{Ci} = 1$, then a component may be shared among several tasks as the resources of the component allow.

$R_{Ci} \rightarrow \mathbb{N} \bigcup \infty$ is a component property indicating the resources available to a component. Just as with participants, components have a potentially infinite set of resources that may be applied toward the performance of a task. For example, if a component models a server, then resources might be interpreted as the amount of processing power that can be distributed among a set of processes.

$INUSE_{Ci} \in \{0, 1\}$ is a component property indicating whether the component is currently in use by a participant performing a task. If $INUSE_{Ci} = 1$ then a component is in use and may not be used to perform a task.

$V_{Ci} = \{v_1, v_2, \ldots, v_s\}$ is the set of s *variables* associated with component i. As with participants, the set and possible values of variables are defined as required for each specific model.

We have now covered the basic HITOP set notation. In the next section, we will reference this notation to explain how state is defined within a HITOP model.

3.6 HITOP State-Value Functions

HITOP is designed to be an executable discrete-event model. Thus, HITOP must be designed to support the notions of state and state changes. In this section, we will use the set notation developed in the previous section to define how HITOP stores and changes state. In 3.5.2, we briefly discussed *properties* and *variables*, which are two of the three basic types of state variables in HITOP. A third more complex type is the *process instances tree (PIT)*. We will define all three types of state variables next as a prerequisite for understanding how HITOP state is defined.

Properties represent values that must be assigned in every HITOP model.

Definition 3.6.1. A *property* is a single-valued state variable associated with a given model element whose value is assigned from a fixed set. A property must be assigned a value for every HITOP model.

For example, every task in a HITOP model will have a task property, TYPE, that is mapped to an element of the set {Atomic, Composite, Start, End, HDP}.

Variables, on the other hand, are like properties in that each one must be mapped to a single value from a set of possible values, but both the set of variables and the corresponding sets of possible values are model-dependent.

Definition 3.6.2. A *variable* is a single-valued state variable associated with a given model element whose value is assigned from a model-dependent set. Variable definitions are dependent on the specific HITOP model implementing them.

For example, suppose two components, a workstation and a router, are being modeled. Both components have the property R_C to describe the resources available to perform a task. However, the workstation and router may require different variables to capture important component aspects. For example, the router may require a variable to capture the number of active connections made to the router, while the workstation may require a variable to represent the amount of e-mail stored on the workstation. Variables provide the modeler with the flexibility to "add" details as required to accurately capture system behavior. Thus, the set of variables used by any particular HITOP model may be unique to that model.

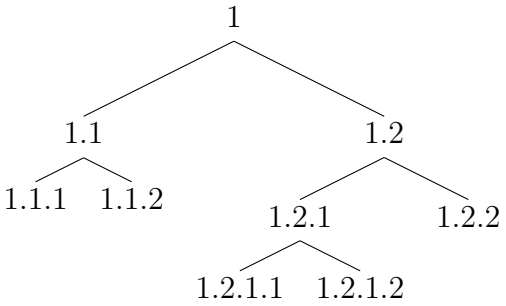

Figure 3.7: Process Instance Tree

Process instance trees, unlike properties or variables that map to a single value, represent a more complex notion of state. See Figure 3.7 for an example of a process instance tree. A PIT stores each token state as basic information about the token, such as where the token is in the process and what associations it has with participants and components.

Definition 3.6.3. A *process instance tree (PIT)* is a connected graph without cycles that specifies family relationships among PI tokens, and maps each leaf node token to a location, a set of participants, a set of components, a set of colors, and a set of status information.

Each active PI token whose value is stored in the tree is mapped to five types of values. The values are pointers to a location, a set of associated participants, a set of associated components, a color set, and a set of token status values. The location value indicates where the token is within the process, e.g., it identifies an arc or a bag. The associated participant and component values point to the participants that are performing the task and the components that are being used to perform the task. The color set describes the family, generation, and individual color of the token. The root node of a PIT specifies the token family color. The node level within the tree and parent token specifies the token generation color. Each token is assigned a unique identifier specifying the individual token color. The status set describes token status information such as whether the token is active, i.e., it is a leaf node on the PIT, or whether it has enabled a given action.

For example, if a particular PI token is located in the executing bag of task T and is associated with participant Bob and component Tool, this token state can be interpreted to mean that for this particular process instance, Bob is using Tool to perform task T.

When a token passes through a split connector, it may be split into child tokens. On the

PIT this is represented by splitting the current node into a set of child nodes, with each child node storing the state for each child token. The child tokens are "active" status tokens, i.e., the parent token has been replaced in the process by its set of children. The status of the parent node is "dormant" until all the child process instances have completed. The state of active tokens is stored on the PIT in the leaf nodes.

When child tokens are combined by a join connector, the representative child nodes on the PIT are collapsed back into the parent node, and the parent node once again becomes an active leaf node, i.e., it stores the state for an active token.

Next, we will describe how values are assigned to each type of state variable using a state-value function.

3.6.1 Basic State-Value Functions

A basic state-value function is used to assign a value to a state variables. In combination over all state variables, basic state-value functions define the state of the model. There are three basic types of state-value functions (one for each type of state variable): property, variable, and PIT. More complex state-value functions are built by combining these basic functions into sets that represent the elements of state present in composite model elements. For example, a component contains both property-type and variable-type state variables. Thus, a component state-value function must contain sets of property and variable state-value functions. Each state-value function will be represented with a μ and a subscript/superscript defining what state variable or set of state variables is being mapped by the function.

Property State-Value Functions

Property state-value functions map a fixed set of properties associated with each model element type to a fixed set of values (represented by natural numbers). The property state-value functions for tasks, processes, participants, and components are listed below.

- The task property state-value function $\mu_U^{T_j P_i} : U_{T_j}^{P_i} \rightarrow \mathbb{N}$ maps the set of properties associated with task j in process i to the natural numbers.

- The process property state-value function $\mu_U^{P_i} : U_{P_i} \rightarrow \mathbb{N}$ maps the set of properties associated with process i to the natural numbers.
- The participant property state-value function $\mu_U^{H_i} : U_{H_i} \rightarrow \mathbb{N}$ maps the set of properties associated with participant i to the natural numbers.
- The component property state-value function $\mu_U^{Ci} : U_{Ci} \rightarrow \mathbb{N}$ maps the set of properties associated with component i to the natural numbers.

Next, we will define the variable state-value functions for each model element type.

Variable State-Value Functions

Variable state-value functions map a model-dependent set of variables to a model-dependent set of values for each variable (represented here by the set of natural numbers). The variable state-value functions for tasks, processes, participants, and components are listed below.

- The task variable state-value function $\mu_V^{T_j P_i} : V_{T_j}^{P_i} \rightarrow \mathbb{N}$ maps the set of variables associated with task j in process i to the natural numbers.
- The process variable state-value function $\mu_V^{P_i} : V_{P_i} \rightarrow \mathbb{N}$ maps the set of variables associated with process i to the natural numbers.
- The participant variable state-value function $\mu_V^{H_i} : V_{H_i} \rightarrow \mathbb{N}$ maps the set of variables associated with participant i to the natural numbers.
- The component variable state-value function $\mu_V^{Ci} : V_{Ci} \rightarrow \mathbb{N}$ maps the set of variables associated with component i to the natural numbers.

Next, we will define the state-value functions for the PIT state variables.

Process Instance Tree State-Value Functions

When a process is executed, it can be thought of as a set of unique process instance (PI) tokens moving through the graph of the process from the start task to the end task. The

state of a process is thus determined by the set of PI tokens within the process graph and the states of each of the individual tokens. The state of each token has five parts: a set of associated participants, a set of associated components, a location within the process, a set of colors, and a set of status information. We shall use a *Process Instance Tree* (PIT) value function to define a unique set of PI tokens, a set of relationships among these tokens, and the state of each token. A PIT value function is a tuple defined in Equation 3.6.

$$\mu_{PIT}^{P_i} = (\mu_I, \mu_{Relation}, \mu_{Active}). \tag{3.6}$$

The elements of the PIT tuple are defined below.

The *identity-value function*, $\mu_I : \mathbb{N}^* \rightarrow \{0, 1\}$, maps the set of all possible strings of natural numbers (separated by periods for clarity) to membership in a set of unique identities or nodes, $I = \{i | \mu_I(i) = 1\}$ (Figure 3.7). The state-value function μ_I is defined so that the following relationships hold true for every $i \in I$: $i \in \mathbb{N}^*$, and if $i.n \in I$ and $n \in \mathbb{N}$, then $i \in I$ and $i.j \in I \; \forall \; 0 \leq j < i$. That mapping assigns a set of colors to each token where the first digit of the identifier signifies the family color, the second to last digit signifies the generation color, and the last digit signifies the individual token color.

The *relation-value function*,$\mu_{Relation} = \{gen, parent, children, siblings, family\}$, maps the set of relationships among nodes in I, where:

- $gen : I \rightarrow \mathbb{N}$ maps each node to a generation (corresponds to the length of the string used to represent that node, e.g., $gen(1) = 1$ and $gen(3.2.4) = 3$),
- $parent : I \rightarrow I$ maps each node to its parent node, where $parent(i.n) = i \; \forall \; i.n \in I, n \in \mathbb{N}$ and $parent(i) = i$ if $gen(i) = 1$,
- $children : I \rightarrow 2^I$ maps each node to a set of its children where $children(i) = \{i.n | i.n \in I, n \in \mathbb{N}\}$ and $children(i) = \{\}$ if i is a leaf node (i.e., i has no children),
- $siblings : I \rightarrow 2^I$ maps each node to a set of its siblings where $siblings(i.n) = children(i) \; \forall \; i.n \in I, n \in \mathbb{N}$, and

- $family : I \rightarrow 2^I$ maps each node to a set of all its descendants $family(i) = \{i.n | i.n \in I, n \in \mathbb{N}^*\}$.

Those relationships are encoded into the structure of the tree and are shown clearly in Figure 3.7.

The *node-value function*, $\mu_{Node} = \{\mu_{IH}^{P_i}, \mu_{IC}^{P_i}, \mu_{IL}^{P_i}, \mu_{ST}^{P_i}\}$, maps each node in the PIT to individual PI state information. Note that the state information stored for active PIs, $I_A = \{ i | i \in I, children(i) = \{\} \}$, i.e., leaf nodes in the PIT, is different than state information stored for dormant PIs, $I_D = I \setminus I_A$ i.e., non-leaf nodes in the PIT. The state-value function μ_{Node} maps the state of each active node in I_A using a set of four value functions:

- $\mu_{IH}^{P_i} : I_A \rightarrow 2^H$ maps each active node in the PIT for process i to a set of participants,
- $\mu_{IC}^{P_i} : I_A \rightarrow 2^C$ maps each active node in the PIT for process i to a set of components,
- $\mu_{IL}^{P_i} : I_A \rightarrow B_{P_i} \bigcup A_{P_i}$ maps each active node in the PIT for process i to a location where $B_{P_i} = \{\bigcup_{T \in T_{P_i}} B_T\}$ is the set of all task bags in process i (see Section 3.5.3), and
- $\mu_{ST}^{P_i} : I_A \rightarrow \mathbb{N}$ maps each active node to a token status.

The state-value function μ_{Node} maps the state of each dormant node in I_D using the same set of four value functions, but a different set of values for location and status:

- $\mu_{IH}^{P_i} : I_A \rightarrow 2^H$ maps each dormant node in the PIT for process i to a set of participants,
- $\mu_{IC}^{P_i} : I_A \rightarrow 2^C$ maps each dormant node in the PIT for process i to a set of components,
- $\mu_{IL}^{P_i} : I_A \rightarrow \{\}$ maps each dormant node to the empty set (as a dormant node does not have a location within the process), and
- $\mu_{ST}^{P_i} : I_A \rightarrow \mathbb{N}$ maps each dormant node to a set of dormant status states.

This concludes the set of basic value functions. In the next section we will use these basic state-value functions to construct the state-value functions for model elements.

3.6.2 Element State-Value Functions

Using the basic state-value functions defined in Section 3.6.1, we can now represent the state-value functions for model elements as sets of those basic state-value functions.

Task State-Value Functions

Task state is represented by a tuple of property and variable state-value functions associated with the task. The state-value function for task j in process i is:

$$\mu_{T_j}^{P_i} = (\mu_U^{T_j P_i}, \mu_V^{T_j P_i}), \tag{3.7}$$

and the set of state-value functions for the m tasks in process i is:

$$\mu_T^{P_i} = \{\mu_{T1}^{P_i}, \mu_{T2}^{P_i}, \ldots, \mu_{Tm}^{P_i}\}. \tag{3.8}$$

Process State-Value Functions

Process state is represented by a tuple of task, property, variable, and process instance tree state-value functions associated with the process. The state-value function for process i is:

$$\mu_{P_i} = (\mu_T^{P_i}, \mu_U^{P_i}, \mu_V^{P_i}, \mu_{PIT}^{P_i}), \tag{3.9}$$

and the set of state-value functions for the n processes in M is:

$$\mu_P = \{\mu_{P1}, \mu_{P2}, \ldots, \mu_{Pn}\}. \tag{3.10}$$

Participant State-Value Functions

Participant state is represented by a tuple of property and variable state-value functions associated with the participant. The state-value function for the participant i is:

$$\mu_{H_i} = (\mu_U^{H_i}, \mu_V^{H_i}), \tag{3.11}$$

and the set of state-value functions for the q participants in M is:

$$\mu_H = \{\mu_{H1}, \mu_{H2}, \ldots, \mu_{Hq}\}. \tag{3.12}$$

Component State-Value Functions

Component state is represented by a tuple of property and variable state-value functions associated with the component. The state-value function for component i is:

$$\mu_{Ci} = (\mu_U^{Ci}, \mu_V^{Ci}), \tag{3.13}$$

and the set of state-value functions for the r components in M is:

$$\mu_C = \{\mu_{C1}, \mu_{C2}, \ldots, \mu_{Cr}\}. \tag{3.14}$$

Model State-Value Function

Having defined the state-value functions for processes, participants, and components, it is now quite straightforward to define the model state-value function as a tuple of those functions:

$$\mu_M = (\mu_P, \mu_H, \mu_C). \tag{3.15}$$

3.6.3 Sets of State-Value Functions

Above we have defined the types of state-value functions used in HITOP. Similarly, if E is a model element and μ is the state-value function for E, as in [59], we shall define the set of all *possible values* of state for E as the set of functions $M_E = \{\mu | \mu$ is a value function for $E\}$. As an example, if μ_{T_j} is the state-value function for task j, then M_{T_j} is the set of all possible state values for task j. If M is a HITOP model, then M_M is the state space for that model.

3.6.4 Changes of State

Now that we have defined the notion of HITOP state, we will briefly discuss how state changes, i.e., events, occur in a HITOP model. As we discussed in Section 2.2.2, the only way that model state can change is via the actions of a task. Token state changes via a token entering or exiting event and process, participant, and component states can be changed only by a task function. Those functions were described in detail in Section 3.5 as process-specific and task-specific functions. Thus, the task functions define the possible state transition functions of a HITOP model.

State transition within a HITOP model can then be defined as a transition from one model-value function to another, with the set of possible transitions defined by each task function. In the next chapter, we will precisely describe the execution algorithms that can be used to change the state of a HITOP model, and in Section 7.2, we will use the idea to construct an executable HITOP model to perform discrete-event simulation.

3.7 Conclusion

In this chapter, we discussed why process models are well-suited to modeling of cyber-human systems and then introduced the Human-Influenced Task-Oriented Process (HITOP) modeling formalism. We first provided a high-level description of HITOP elements, and then provided a precise set of mathematical definitions of all HITOP elements. Finally, we defined the HITOP state-value functions and described how they can be related to state transition functions contained within each task.

CHAPTER 4

SPECIFYING THE EXECUTABLE MODEL: HITOP EXECUTION ALGORITHMS

4.1 Introduction

In Chapter 3, we introduced and formally defined the HITOP formalism and its associated definition of state. In this chapter, we will define how HITOP is executed as part of a discrete-event simulation. We will do so by first describing the two viewpoints of HITOP execution, and then providing a set of algorithms that can be used to implement them.

By design, the execution of a HITOP model can be viewed at two levels: the process level and the task level. This idea was described in Section 3.4.4. Recall that the process-level execution view is designed to be intuitive to users who are not modeling experts. It simplifies the notion of task state into just two possible states: active and inactive.

The task-level execution view, on the other hand, reveals the details of internal task state changes and is necessary for a full understanding of how a HITOP model is executed.

In Section 4.2, we will describe HITOP execution from a process-level viewpoint. In Section 4.3, we will describe the execution of HITOP from a task-level viewpoint. In Section 4.4, we will relate the two execution viewpoints. We summarize in Section 4.5.

4.2 Process-Level Execution

In this section, we will discuss the process-level viewpoint for HITOP execution. A HITOP *process-level execution viewpoint* is a view of HITOP state that focuses on process flow. It combines all possible individual task-level task states into active and inactive process-level task states. Tasks are viewed as "black box" elements, i.e., the internal task execution details are hidden, and process flows are specified as deterministic functions of model state.

At the process level, model execution is viewed as distinct process instance (PI) tokens moving between tasks and arcs within a process model. Recalling our discussion of process-level state in Section 3.4.4, when a token enters a task, the task becomes active, i.e., the task is attempted. When a token exits a task, the task becomes inactive, i.e., the task is not being attempted. The term *attempted* is used here purposefully. At the process level, no internal task performance details are viewed, and only the attempt at task performance, i.e., the transition from the inactive to the active state, can be observed. Thus, the results of task performance are only evident from the process view if 1) the performance of the task causes a change in one or more HITOP model state variables, and 2) the execution of a process instance, i.e., a token flow through the process model, is dependent on the changed state variable.

For example, suppose there is a task that encrypts data on a USB stick. The task can result in the data on the USB stick either being encrypted or not. Recall that the USB stick would be modeled using a HITOP component and a set of associated variables. If the initial state of the data is unencrypted, the process model at the process-level viewpoint can specify that branching to task "write encrypted data" is allowed only if the state of the USB data is "encrypted;" otherwise the process will branch to task "write unencrypted data." That is, flow branching is dependent on the state of the USB stick, not on the outcome of the task. Thus it should be clear that the process-level viewpoint removes itself from the details of task execution and focuses on process instance flow through the process model.

Next we will describe how token relationships are viewed at the process level.

4.2.1 Token Flow and Token Relationships

As a token flows through a process, it may be split, joined, or passed through by a task it enters and exits. This token flow can change relationships among tokens within the process. Recall from Section 3.6 that a process instance (PI) has as part of its state a set of relationships with other PIs. The relationships include parent, child, and sibling relationships.

Recall also from Section 3.4.6 that split and join connectors control token flow entering

and exiting tasks. Further recall that connectors come in three types: AND, XOR, and OR. When a token is split by an AND or OR split connector, it is split into a set of tokens, each with a child relationship relative to the parent token. We refer to those tokens as the *children* of the referenced parent token. The children also share a sibling relation with each other.

When a set of siblings is joined into a single token by an AND or OR join connector, the new token has a parent relationship to each member of the set of joined children.

Recall also from Section 3.4.6 that the set of tokens (plus their entering arc locations) that are eligible to enter a task is called the *entering set* and that the set of tokens (plus their exiting arc locations) that are produced when exiting a task are called the *exiting set*.

At the process-level execution viewpoint, each of those connector types has a very specific meaning that will be discussed next.

4.2.2 AND Connectors

Using the terminology developed above, when a token passes through an AND split connector, it produces an exiting set of that token's children with one child placed on every exiting arc. That is called an *AND exiting set*. Similarly, an *AND entering set* is a set in which there is a token on every entering arc, and all the tokens are siblings. When an AND entering set passes through an AND join connector, it enters a single parent token of the siblings into the task.

4.2.3 XOR Connectors

When a token passes through an XOR split connector, the result is an exiting set consisting of a token of the same type placed on a single exiting arc. Recall from Section 3.6 that the term *token type* refers to the token's color set, i.e., colors that represent a token's family, generation, and individual color. The exiting set is called an *XOR exiting set*. Similarly, an *XOR entering set* is formed by a single token on a single entering arc. When an XOR entering set passes through an XOR join connector, it enters a single token of the same type

into the task.

4.2.4 OR Connectors

When a token passes through an OR-x split connector, the result is an exiting set of an x number of that token's children, with one child placed on each of the x exiting arcs. The set is called an *OR-x exiting set*. Similarly, an *OR-x entering set* is formed by a set in which there are tokens on x of the entering arcs, and all the tokens are siblings. When an OR-x entering set passes through an OR-x join connector, it enters a single parent token of the siblings into the task.

4.2.5 Join and Split Execution

Given the above descriptions of token flow, the execution of a join connector may be formalized using Algorithm 1, where the function *entering_set_present* evaluates to true if a TYPE-type entering set is present on the set of entering arcs, the function *choose_set* chooses in a uniform way one of the TYPE-type entering sets present on the entering arcs, and the function *enter* enters the chosen entering set via a TYPE-type join connector, i.e., it removes the designated tokens from the entering arcs and places them within the task.

Algorithm 1 HITOP Join Execution Algorithm

```
1: TYPE ← join connector type
2: A_ENT ← set of entering arcs
3: while entering_set_present(TYPE, A_ENT) do
4:    Entering_Set ← choose_set(TYPE, A_ENT)
5:    Token_Entered ← enter(TYPE, Entering_Set)
6: end while
```

Recall that a join type is from the set {AND, XOR, OR}, each with a specified type of entering set. The state changes required to implement the functions *entering_set_present*, *choose_set*, and *enter* correspond to the process-specific functions detailed in the HITOP set notation outlined in Section 3.5.2.

In a similar fashion, the execution of a split connector may be formalized using Algorithm 2, where the function *exiting_set_ready?* evaluates to true if the task is ready to exit

one or more TYPE-type exiting sets, and *exit* is a function that chooses in a uniform way one of the sets to be exited and exits the task by generating a TYPE-type exiting set, i.e., it removes the specified token from the task and places an appropriate token on one or more exiting arcs, as required.

Algorithm 2 HITOP Split Execution Algorithm

```
1: TYPE ← split connector type
2: A_EXT ← set of exiting arcs
3: while exiting_set_ready(TYPE) do
4:    Exiting_Set ← exit(TYPE)
5:    A_EXT ← Exiting_Set
6: end while
```

Recall that a split type is from the set {AND, XOR, OR}, each with a specified type of exiting set. The state changes required to implement the functions *exiting_set_ready?* and *exit* correspond to the process-specific functions detailed in the HITOP set notation described in Section 3.5.2.

4.2.6 Process Execution

Now that the process-level split and join execution algorithms have been defined, we can formalize how a process model is executed from the process-level viewpoint using Algorithm 3, where 1) the function *choose* picks a task T in a uniform way from the set of tasks with an entering set present, T_J, or the set of tasks with tokens to be exited, T_S; 2) the function *join* performs the join execution algorithm, Algorithm 1, on task T, and 3) the function *split* performs the split execution algorithm, Algorithm 2, on task T.

Algorithm 3 continues as long as any of the set of tasks with entering sets, the set of tasks with exiting sets, or the set of active tasks, T_A, has at least one element. In other words, process-level execution will continue as long as continued token flow is possible.

Algorithm 1, Algorithm 2, and Algorithm 3 are all that is required to specify HITOP execution on the process level. Of course, to fully specify the execution of a HITOP model at the level at which discrete-event simulation is possible, a more detailed explanation of execution at the task level is required.

Algorithm 3 HITOP Process-Level Execution Algorithm

```
1: T_J ← set of tasks with entering sets
2: T_S ← set of tasks with tokens to be exited
3: T_A ← set of tasks in the active state
4: while (|T_J| + |T_S| + |T_A|) > 0 do
5:    while (|T_J| > 0 do
6:       T ← choose(T_J)
7:       Join(T)
8:    end while
9:    while (|T_S| > 0 do
10:      T ← choose(T_S)
11:      Split(T)
12:   end while
13: end while
```

4.3 Task-Level Execution

In the previous section, we described the execution of HITOP as viewed from the process level. In this section, we present a more detailed view of HITOP by describing the execution of HITOP as viewed from the task level. A HITOP *task-level execution viewpoint* is a view of HITOP state that focuses on the details of internal task token flow and explicitly represents the task-level task state using task variables, properties, and token locations. Recall from Section 3.6 that a task's internal state is defined by its task state-value function, and that the location of a token is defined by its associated process instance tree. We use the term *task-level task state* to refer to a task's internal state and the identify and location of tokens within the task.

Once execution at the task level has been defined and combined with the process-level execution algorithms, the overall execution of HITOP will be defined. To enable the task-level description, we first informally describe a shorthand notation for primitives within a task.

4.3.1 Actions and Bags

There are two basic primitives within a task: the action and the bag. The action moves tokens between bags and changes model state. The bag stores tokens. We describe each of

these primitives in more detail next.

Bag

A *bag* is a storage location for tokens. Because HITOP tokens have color, a bag stores a multiset of colored tokens. In general, a bag may provide tokens to one or more actions, i.e., it is an *enabling bag* for those actions, and a bag may receive tokens from one or more actions, i.e., it is a *firing bag* for those actions. Note that a process-level arc is a bag that is restricted such that it is an enabling bag for a single task's *enter* action and a firing bag for a single task's *exit* action.

We will represent bags informally using ovals. Enabling bags are represented by directed arcs from a bag to an action. Firing bags are represented by directed arcs from an action to a bag.

Action

An *action* performs a state change and the state change may be one of a set of mutually exclusive model state changes. An action consists of an enabling predicate, a set of firing functions, and a timing distribution, and it has two possible states: enabled and not-enabled. Note that an action's state is always relative to some token. That is, an action may be enabled relative to one token and not-enabled relative to another token. An action may *fire*, i.e., perform a state change, only from the enabled state. When an action fires, its state will change to not-enabled. An action is enabled when its enabling predicate evaluates to 1.

An action's *enabling predicate* consists of two Boolean functions: the *enabling_set_present function* and the *enabling function*. Both functions must evaluate to 1 for the enabling predicate to evaluate to 1.

The *enabling_set_present function* is a Boolean function of token/ token location pairs (where the set of token/ token location pairs that exist in a given task-level task state is defined by the process instance tree). The *enabling_set_present* function will evaluate to 1 if an *enabling set* for the action is *present* in the current task-level task state (defined next).

Definition 4.3.1. An *enabling set* is the set of token/token location pairs that may enable an action. An enabling set is *present* if tokens are stored among the enabling bags such that the tokens and the token locations form an enabling set. An action may have multiple enabling sets present simultaneously.

In general, the token/ token location pairs necessary to form an action's enabling set depend both upon the action and the task-level task state. For example, if a multiple instance category task's THRESH property is set to x, then an enabling set for the *exit* action would require at least x tokens in the *COMP* bag.

An action's *enabling function* is a Boolean function of model state. Thus, in general, an action's enabling predicate depends on both the task-level task state and the model state overall.

An action's *set of firing functions* is a state-dependent set of one or more firing functions. An action's *firing function* is a function that maps a current model state to a new model state, i.e., it maps the current model state-value function to a new model state-value function. The effects of a firing function may be grouped into two categories: *token change* and *model state change*. The *token change* effect transforms the token(s) that form the enabling set in the enabling bag(s) into the token(s) in the firing bag(s) needed to form the firing set. The transformation can be thought of as "moving" tokens from the enabling bag(s) to the firing bag(s).

Definition 4.3.2. A *firing set* is the set of token/token location pairs that is the result of firing an action. A firing set is *produced* by placing one or more tokens in the action's firing bags such that the tokens and the token locations form a firing set.

Here it is important to note that because of the firing rules of actions, the number of tokens may not be conserved; however, the family relationship of the involved tokens is conserved, e.g., each enabling set results in a corresponding firing set. This concept will become clearer as we discuss the algorithms for task-level execution.

Multiple firing functions are mutually exclusive and each firing function has a state-dependent probability of occurrence. Each of the probabilities is constructed such that the sum of the probabilities for all firing functions is 1.0. For example, the *complete* action

has a set of state-dependent outcomes, one of which will occur, and each outcome has a state-dependent probability of occurrence. Each outcome corresponds to a firing function in the *complete* action's set of firing functions.

An action's *timing distribution* represents the time required to fire an action once it is enabled, i.e., its firing time. In general, a timing distribution is the probability distribution of an arbitrary random variable. An action's timing distribution is a function of the type of action and the model state.

Types of Actions

There are two types of actions: instant actions and parallel actions. *Instant actions* (IAs) have a firing time of 0 and fire in the same instant of time in which they are enabled. Thus, their timing distribution is the probability density function of a discrete random variable with all the probability mass at 0. Instant actions are functionally similar to the instantaneous activities used in SANs [60].

Definition 4.3.3. An *instant action* is a type of task-level primitive action that immediately fires, i.e., produces an event, for each enabling set present while the enabling predicate is 1.

The other type of action is the *parallel action* (PA). PAs, in general, have timing distributions that are the probability distributions of arbitrary random variables. PAs may be enabled in parallel by multiple enabling sets (as long as the action's enabling predicate evaluates to 1). A PA enabled by n enabling sets is functionally equivalent to n "single" actions enabled at the same time in parallel.

Definition 4.3.4. A *parallel action (PA)* is a type of task-level primitive action that enables a separate and parallel action for each enabling set present while the enabling predicate is 1. Theoretically, the number of actions that may be enabled is unlimited.

We represent an IA using a black triangle and a PA using a white triangle.

Now that we have defined the basic task-level primitives, we will informally describe the internal task structure using these primitives.

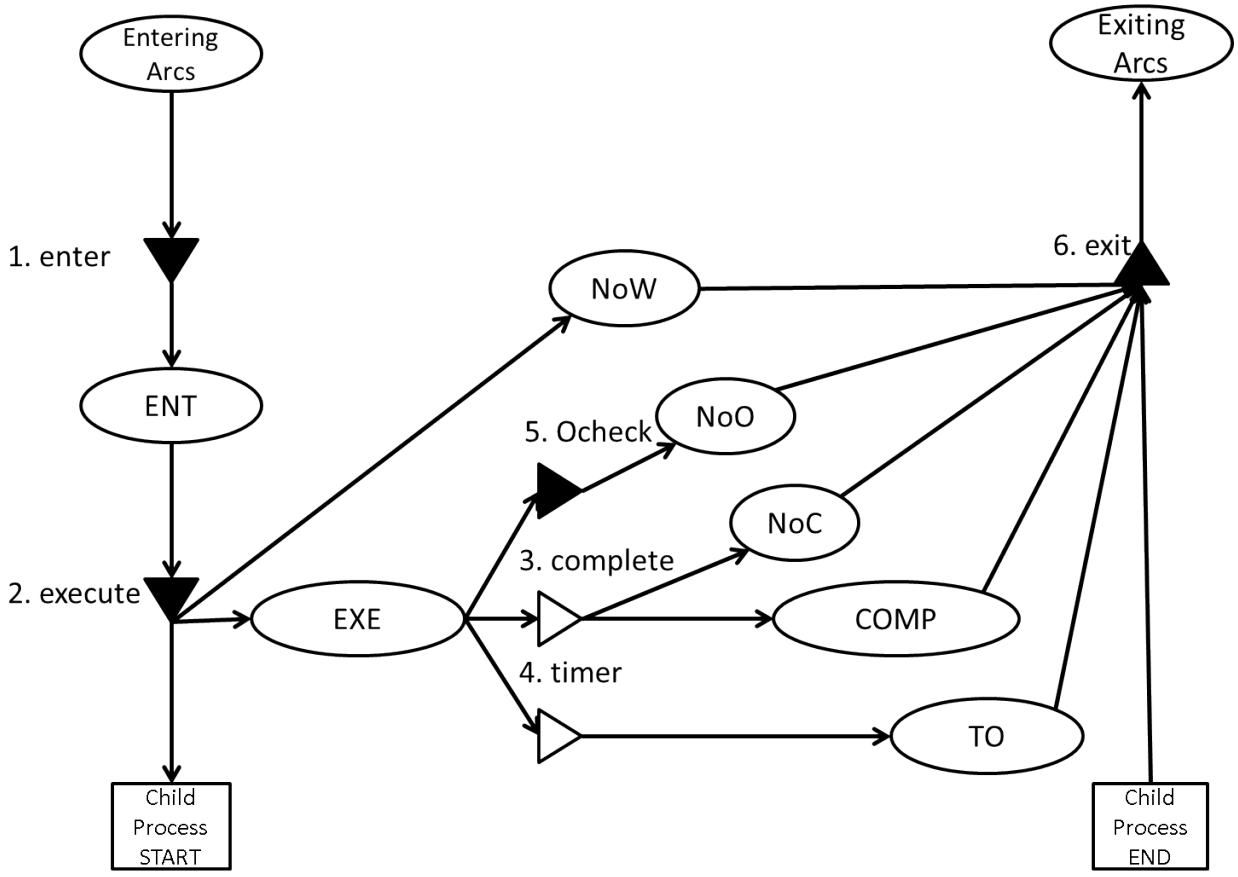

Figure 4.1: Task Internal Structure

4.3.2 Task Internal Structure

The task is the most complex of the HITOP elements. That is by design, as one of our goals was to create a process model that was both simple to use and powerful. To do so, we "hid" the internal details of HITOP task execution from process-level modelers. Thus, a domain-expert analyst can work with a simple interface that treats tasks as "black boxes" at the process-level viewpoint. However, the power of HITOP is contained at the task level, and a true understanding of HITOP execution can only be gained by examining what is inside a task.

We begin by examining the internal structure of a task (see Figure 4.1). Recall from Section 3.5.3 that a task is represented in set notation by using a bags, properties, variables, and functions. Those set notation elements are represented informally using our task primitives as a set of bags and actions. Task-level task state changes are represented by movement of tokens from one bag to another. To make token flow clear graphically, we have informally represented the internal task structure in Figure 4.1 as a set of actions and bags connected by directed arcs. Tokens flow from the entering arcs through the task internal bags and actions until they exit the task via the exiting arcs.

A task consists of six actions and nine bags (see Figure 4.1). The *enter*, *execute*, *Ocheck*, and *exit* actions are IAs represented by black triangles. The *complete* and *timer* actions are PAs represented by white triangles. The *Entering Arcs*, *ENT*, *EXE*, *NoW*, *NoO*, *NoC*, *COMP*, *TO*, and *Exiting Arcs* bags are represented by a white ovals. The task-level task state (with reference to a particular token) is determined by the bag in which the token is located. For example, if a token is stored in the no-willingness bag, NoW, the task-level task state for that token is the no-willingness state.

A high-level description of the actions in a task follows. Note that we sometimes use the terminology "a token is moved from bag A to bag B" as a shorthand for "the enabling set stored in bag A is transformed into the firing set stored in bag B."

- The *enter* action transforms the enabling set stored in the *entering arcs* bag into the firing set that has one token for each task instance in the *ENT* bag. Firing this action represents the activation of a task at the process level. That is, when a token is stored

in the *ENT* bag, i.e., the task-level task state is "entered," it means that the process associated with the token is attempting to perform the task.

- The *execute* action associates a token with a participant set and a component set, and evaluates if willingness exists (relative to the token, set of participants, and set of components) to perform the task. If willingness does not exist, the token is moved from the *ENT* bag to the *NoW* bag, i.e., the task-level task state is changed to "no-willingness." If willingness exists, the token is moved to the *EXE* bag, i.e., the task-level task state is "executing." Firing this action represents either the failure to properly perform the task because the participant is not willing, or the attempt to properly perform the task because the participant is willing.

- The *complete* action determines a firing time using a continuous random variable and fires at the firing time if the token remains in the *EXE* bag. When the action fires, a token is moved from the *EXE* bag to either the *NoC* bag, i.e., the task-level task state is changed to "no-capability," or the *COMP* bag, i.e., the task-level task state is changed to "completed." The *NoC* bag or the *COMP* bag is selected based on the stochastically-determined task outcome. The no-capability outcome results in the selection of the *NoC* bag. A proper performance outcome results in the selection of the *COMP* bag. Note that this action can only fire if the associated token has not been removed from the *EXE* bag by the firing of the *Ocheck* action or the *timer* action.

- The *timer* action determines a firing time using a deterministic function of state and fires at the firing time if the token remains in the *EXE* bag. When the action fires, the token is moved from the *EXE* bag to the *TO* bag, i.e., the task-level task state is changed to "timed out." Firing this action represents the timing out of an action that has not completed by the specified time. A timed out task cannot be properly performed.

- The *Ocheck* action moves a token from the *EXE* bag to the *NoO* bag, i.e., the task-level task state is changed to"no-opportunity," when the task's opportunity function evaluates to 0. Firing this action represents the failure to properly perform the task

due to no opportunity.

- The *exit* action moves a token from the task outcome bags, i.e., Now, NoO, NoC, COMP, and TO, to the Exiting Arcs bag. Firing this action exits tokens from the task and represents the transition of the task from the active state to the inactive state.

A high-level description of the bags in a task follows.

- The *Entering Arcs* bags are the entering arcs for the task. A set of tokens in the Entering Arcs bags represents the task-level task state in which the task is waiting to be attempted.
- The *ENT* bag holds tokens that have been entered into the task. A token in this bag represents the task-level task state in which the process is waiting to be executed.
- The *EXE* bag holds tokens that are being executed. A token in this bag represents the task-level task state in which a process is being executed, i.e., proper performance is being attempted.
- The *NoW* bag holds tokens for which willingness does not exist. A token in this bag represents the task-level task state of no-willingness.
- The *NoO* bag holds tokens for which opportunity does not exist. A token in this bag represents the task-level task state of no-opportunity.
- The *NoC* bag holds tokens for which capability does not exist. A token in this bag represents the task-level task state of no-capability.
- The *COMP* bag holds tokens for which a task has been properly performed. A token in this bag represents the task-level task state of proper performance.
- The *TO* bag holds tokens which have timed out. A token in this bag represents the task-level task state of timed out.
- The *Exiting Arcs* bags are the exiting arcs for the task. A set of tokens in these bags represents the task-level task state in which the task attempt has been finished.

Note that Figure 4.1 indicates flow from the *execute* action to a block labeled "Child Process START" and from a block labeled "Child Process END" to the *exit* action. These portions of our informal diagram indicate internal task flows that occur for the composite type task. For this type of task, tokens are directed to the start task of a child process and received from the end task of the child process.

4.3.3 Reusing Model Structure

A HITOP model is able to *reuse* structure, i.e., many tokens may independently and in parallel use the same model structure, through two design features. First, tasks are built to be stateless with respect to the process-level viewpoint. This means that all tokens entering a task "see" the same task state and thus the flows of those tokens are not affected by tokens that have already entered the task. Second, tasks use PAs for timed actions. That means that all tokens in the *EXE* bag can enable a separate parallel event. Together, these two design features allow many different tokens to simultaneously use the same task structure. The benefit of this is that the number of processes being simulated, i.e., the set of tokens in the process model, may change dynamically during run-time, and each process can be individually simulated without the need to add modeling structure for each new token.

For example, a typical Petri net model would require a separate defined process structure for each token that would be executed in parallel. Each of the structures would have to be built and included in the model prior to run-time. Obviously, the total number of possible parallel process instances would have to be specified beforehand. HITOP, on the other hand, is designed to reuse one existing modeling structure for any number of tokens, and the number of these tokens may be determined during model execution. That allows HITOP to dynamically change the number of parallel discrete-event simulations it is running. Such a construct is useful, for example, in situations where the number of parallel processes being simulated may change during a simulation.

4.3.4 Task-Level Action Execution Order

Now that we have informally defined the internal task structure, we will specify how a task is executed at the task level by using execution algorithms. Algorithm 4 specifies the order in which actions within the task are executed where *predicate_true?* is a function that evaluates to 1 if any action's predicate in the task evaluates to 1 and *execute_action(action)* calls the execution algorithm for the specified action.

Algorithm 4 Task Action Execution Order Algorithm

```
1: TASK ← the task {Point to a task}
2: while predicate_true?(TASK) {While any action is enabled} do
3:    execute_action(TASK.enter) {execute enter action, Algorithm 5}
4:    execute_action(TASK.execute) {execute execute action, Algorithm 6}
5:    execute_action(TASK.complete) {execute complete action, Algorithm 7}
6:    execute_action(TASK.timer) {execute timer action, Algorithm 8}
7:    execute_action(TASK.opportunity check) {execute Ocheck action, Algorithm 9}
8:    execute_action(TASK.exit) {execute exit action, Algorithm 10}
9: end while
```

It can be seen from Algorithm 4 that tokens are handled "left to right" within the task, i.e., first from the entering activity and last from the exit activity. Next, we will define each action's execution algorithm.

4.3.5 Task-Level Action Execution Algorithms

Given the previous discussion of overall task action execution order, we are now ready to discuss the internal task-level action execution algorithms. Figure 4.1 numbers each of the six actions, and we shall define task-level execution by defining the execution algorithm for each of these actions.

1: The Enter Action

The *enter action* moves tokens from the entering arcs to the *ENT* bag. It is an instantaneous action. At the process level, firing of this action transitions the task from an inactive state to an active state. The enabling bags of the enter action at the task level are the same as the

entering arcs for the task at the process level. The enabling predicate for this action consists only of the *enabling_set_present* function and is the same as the process-level execution algorithm function *entering_set_present* used in Algorithm 1, except that the entering set is now considered an enabling set, and the entering arcs are now considered the enabling bags.

The firing function for this action has the following effects: 1) an enabling set is uniformly selected from among those enabling sets present in the *entering arcs* bag ; 2) that enabling set is transformed into the firing set in the *ENT* bag appropriate to the task instance category; and 3) any leftover siblings of the enabling set as appropriate to the join type are canceled, i.e., removed from the model as a whole.

The execution of the enter action is given in detail in Algorithm 5, where B_{EN} is the set of enabling bags, B_{FR} is the set of firing bags, *enabling_set_present* is a Boolean function indicating whether a TYPE-type enabling set is present in the enabling bags; *choose_set* is a function that chooses, in a uniform way, a TYPE-type enabling set from enabling sets present in the enabling bags and removes the tokens associated with that enabling set from the enabling bags; *transform_set* is a function that transforms the enabling set into a firing set appropriate to the join type, task instance category, and task number of instances; *fire_action* is a function that performs the state change associated with firing the action for a given firing set; and *cancel_siblings* is a function that cancels any siblings of the enabling set as required by the task join type.

Algorithm 5 Enter Action Execution Algorithm

```
1: JOINTYPE ← join connector type
2: INST ← task instance category
3: NUMINST ← task number of instances
4: B_EN ← the set of entering arcs
5: B_FR ← task ENT bag
6: while enabling_set_present(JOINTYPE, B_EN) {while there are enabling sets} do
7:    Enabling_Set ← choose_set(JOINTYPE, B_EN)
8:    Firing_Set ← transform_set(JOINTYPE, INST, INSTNUM, Enabling_Set)
9:    B_FR ← Firing_Set
10:   fire_action(Firing_Set, enter) {fire the action}
11:   cancel_siblings(JOINTYPE, Enabling_Set)
12: end while
```

Algorithm 5 details how the enter action will move every enabling set from the entering arcs

to the *ENT* bag sequentially during the same instant, i.e., from the process-level viewpoint it enters all of the entering sets present.

2: The Execute Action

The *execute action* moves tokens from the *ENT* bag to the *EXE* bag. It is an instantaneous action. The enabling predicate evaluates to true whenever there is a token in the *ENT* bag. An enabling set for this action is a single token.

The firing function for this action has the following effects: 1) a token is uniformly selected from the *ENT* bag; 2) if the task type is "atomic," participants and components are assigned to the token; 3) if the willingness function evaluates to 1 for that token and the task type is "atomic," the token is moved to the *EXE* bag; 4) if the willingness function evaluates to 0 for that token and the task type is "atomic," the token is moved to the *NoW* bag; and 5) if task type is "composite," the firing set is moved to the *ENT* bag of the start task of the associated child process. Note that if an appropriate participant or component set cannot be assigned, the token will wait in the *ENT* bag until such a set can be assigned.

The firing of the execute action is given in detail in Algorithm 6, where B_{EN} is the set of enabling bags, B_{FR} is the set of firing bags, *choose_token* is a function that chooses in a uniform way a token from the enabling bag; *choose_participants* is a function that chooses in a uniform way a set of participants to perform this task and associates the set with the token; *choose_components* is a function that chooses in a uniform way a set of components to perform the task and associates this set with the token; *fire_action* is a function that performs the state change associated with firing the action for a given firing set; and f_W is the willingness function that evaluates the willingness of the participants assigned to attempt task performance.

3: The Complete Action

The *complete action* moves tokens from the *EXE* bag of a task to the *COMP* bag. It is a PA. It has two phases: "enable actions" and "fire actions." Each phase can be thought of as having a separate predicate and firing function.

Algorithm 6 Execute Action Execution Algorithm

```
1: TASKTYPE ← task type
2: CHILD ← the ENT bag in the start task of the child process
3: B_EN ← task ENT bag
4: B_FR ← task EXE bag
5: B_NoW ← task NoW bag
6: while |B_EN| > 0 {while there are tokens in ENT bag} do
7:    Firing_Set ← choose_token(B_EN) {choose a token}
8:    if TASKTYPE == 'atomic' then
9:       Participant_Set ← choose_participants(Firing_Set)
10:      Component_Set ← choose_components(Firing_Set)
11:      if Participant_Set == ØOR Component_Set == Ø then
12:         exit {if no component or participant match, wait}
13:      end if
14:      if f_W(Firing_Set) == 1 then
15:         B_FR ← Firing_Set {if willing, begin execution}
16:         fire_action(Firing_Set, execute) {fire the action}
17:      end if
18:      if f_W(Firing_Set) == 0 then
19:         B_NoW ← Firing_Set {if not willing, move to NoW bag}
20:         fire_action(Firing_Set, NoW) {fire the action}
21:      end if
22:   end if
23:   if TASKTYPE == 'composite' then
24:      CHILD ← Firing_Set {if composite task, start child process}
25:   end if
26: end while
```

For the "enable actions" phase, the enabling predicate evaluates to true whenever there is an unmarked token in the *EXE* bag. An enabling set for this action is a single unmarked token. The firing function for this phase has the following effects: 1) a token is uniformly selected from the *EXE* bag; 2) a firing time for the action, is determined from the action's state-dependent timing distribution; 3) an event noting the token, firing time, and action identifier, i.e., the *complete* action, is enabled and placed on the event list; and 4) the token is "marked" as enabled. Note that we use the term *event list* to signify a storage location for events that is sorted by firing time.

For the "fire actions" phase, the enabling predicate evaluates to true whenever there is an enabled event associated with this action with a firing time equal to the current time. An enabling set for this action is the token associated with the current event.

The firing function for this phase has the following effects: 1) an event associated with the current time is uniformly selected from the event list; 2) the token associated with the event is moved from the *EXE* bag to the *COMP* bag; 3) any events on the event list that are associated with this token are canceled; and 4) any associations with participants or components the token has are removed.

The execution of the complete action is detailed in Algorithm 7, where B_{EN} is the set of enabling bags, B_{FR} is the set of firing bags, *unmarked* is a function that evaluates as true if there is an unmarked token in the enabling bags; *mark_token* is a function that chooses in a uniform way an unmarked token from the enabling bags and marks that token as entered on the event list; *choose_time* is a function that chooses a firing time from the state-dependent distribution of the complete action; *make_event* is a function that builds an event marking the token identity, firing time, and action that is the source of the event; *next_event* is a function that evaluates to true whenever there is an event corresponding to the current time on the event list associated with the *complete* action; *pop_event* is a function that selects in a uniform way an event scheduled for the current time and removes it from the event list; *remove_event* is a function that removes any events that are associated with the firing set from the event list; *fire_action* is a function that performs the state change associated with firing the action for a given firing set and returns the outcome; *release_participants* is a function that removes the association of the firing set with its set of participants; and

release_components is a function that removes the association of the firing set with its set of components.

Algorithm 7 Complete Action Execution Algorithm

```
1: B_EN ← the task EXE bag
2: EL ← the event list
3: while unmarked(B_EN) {while there are tokens to place on event list} do
4:    Token ← mark_token(B_EN)
5:    FiringTime ← choose_time(token)
6:    Event ← make_event(Token, FiringTime)
7:    EL ← Event {enter event on event list}
8: end while
9: B_FR ← the task COMP bag
10: B_NoC ← the task NoC bag
11: EL ← the event list
12: t ← the current time
13: while next_event(EL, t) {while there are events to fire} do
14:    Event ← pop_event(EL, t)
15:    Firing_Set ← Event.token
16:    remove_event(Firing_Set)
17:    Outcome ← fire_action(Firing_Set, complete) {fire the action}
18:    release_participants(Firing_Set)
19:    release_components(Firing_Set)
20:    if Outcome == no-capability then
21:       B_NoC ← Firing_Set
22:    else
23:       B_FR ← Firing_Set
24:    end if
25: end while
```

4: The Timer Action

The *timer action* moves tokens from the *EXE* bag of a task to the *TO* bag. It is a PA. It has two phases: "enable actions" and "fire actions." Each phase can be thought of as having a separate predicate and firing function.

For the "enable actions" phase, the enabling predicate evaluates to true whenever there is an unmarked token in the *EXE* bag. An enabling set for this action is a single unmarked token. The firing function for this action has the following effects: 1) a token is uniformly selected from the *EXE* bag; 2) a completion time, i.e., a firing time for the action, is

determined from the action's state-dependent timing distribution; 3) an event noting the token, firing time, and action identifier, e.g., the *timer* action, is enabled; and 4) the token is "marked" as entered on the event list.

For the "fire actions" phase, the enabling predicate evaluates to true whenever there is an enabled action associated with this action with a firing time equal to the current time. An enabling set for this action is the token associated with the current event.

The firing function for this action has the following effects: 1) an event associated with the current time is uniformly selected from the event list; 2) the token associated with the event is moved from the *EXE* bag to the *TO* bag; 3) any events on the event list that are associated with this token are canceled; and 4) the token's associations with participants and components are removed.

The execution of the *timer* action is detailed in Algorithm 8, where B_{EN} is the set of enabling bags, B_{FR} is the set of firing bags, *unmarked* is a function that evaluates as true if there is an unmarked token in the enabling bag; *mark_token* is a function that chooses, in a uniform way, an unmarked token from the enabling bags and marks that token as entered on the event list; *choose_time* is a function that chooses a firing time from the state-dependent distribution of the *timer* action; *make_event* is a function that builds an event marking the token identity, firing time, and action that is the source of the event; *next_event* is a function that evaluates to true whenever there is an event corresponding to the current time on the event list associated with this *complete* action; *pop_event* is a function that selects in a uniform way an event scheduled for the current time and removes it from the event list; *remove_event* is a function that removes any events that are associated with the firing set from the event list; *fire_action* is a function that performs the state change associated with firing the action for a given firing set; *release_participants* is a function that removes the association of the firing set with its set of participants; and *release_components* is a function that removes the association of the firing set with its set of components.

Algorithm 8 Timer Action Execution Algorithm

1: $B_{EN} \leftarrow$ the task *EXE* bag
2: $EL \leftarrow$ the event list
3: **while** unmarked(B_{EN}) {while there are tokens to place on the event list} **do**
4: Token $\leftarrow$ mark_token(B_{EN})
5: FiringTime $\leftarrow$ choose_time(token)
6: Event $\leftarrow$ make_event(Token, FiringTime)
7: $EL \leftarrow$ Event
8: **end while**
9: $B_{FR} \leftarrow$ the task *TO* bag
10: $EL \leftarrow$ the event list
11: $t \leftarrow$ the current time
12: **while** next_event(EL, t) {while there are events to fire} **do**
13: Event $\leftarrow$ pop_event(EL, t)
14: Firing_Set $\leftarrow$ Event.token
15: remove_event(Firing_Set)
16: fire_action(Firing_Set, timer) {fire the action}
17: release_participants(Firing_Set)
18: release_components(Firing_Set)
19: $B_{FR} \leftarrow$ Firing_Set
20: **end while**

5: The Opportunity Check Action

The *opportunity check (Ocheck) action* moves tokens from the *EXE* bag of a task to the *NoO* bag. It is an instantaneous action. The enabling predicate evaluates to true whenever there is a token in the *EXE* bag and the opportunity function evaluates to 0 for at least one token in the *EXE* bag. An enabling set for this action is a single token.

The firing function for this action has the following effects: 1) a token from the set of tokens with no opportunity is randomly selected in a uniform way from the *EXE* bag; 2) any events on the event list that are associated with this token are canceled; 3) any associations with participants or components that the token has are removed; and 4) the token is moved to the *NoW* bag.

The execution of the opportunity check action is detailed in Algorithm 9, where B_{EN} is the set of enabling bags, B_{FR} is the set of firing bags, the *choose_NoO_token* function chooses in a uniform way a token from the set of tokens with no opportunity in the enabling bag; *fire_action* is a function that performs the state change associated with firing the action for

Algorithm 9 Opportunity Check Action Execution Algorithm

1: $B_{EN} \leftarrow$ the task *EXE* bag
2: $B_{FR} \leftarrow$ the task no opportunity bag
3: **while** $(|B_{EN}| > 0)(f_O(B_{EN}) == 0)$ {while there are tokens with no opportunity} **do**
4: Firing_Set $\leftarrow$ choose_NoO_token(B_{EN})
5: fire_action(Firing_Set, NoO) {fire the action}
6: remove_event(Firing_Set)
7: release_participants(Firing_Set)
8: release_components(Firing_Set)
9: $B_{FR} \leftarrow$ Firing_Set
10: **end while**

a given firing set; *release_participants* is a function that removes the association of the firing set with its set of participants; *release_components* is a function that removes the association of the firing set with its set of components; and f_O is the opportunity function that evaluates to 0 if the opportunity for task performance does not exist for any token in the enabling bag.

6: The Exit Action

The *exit action* moves tokens from the set of task outcome bags to the exiting arcs of the task. It is the same as the "exiting tokens" action performed by a split connector, except that the term *firing set* is used instead of *exiting set* and the term *firing bags* is used in place of *exiting arcs*. Firing of this action equates to a task state change on the process-level viewpoint from active to inactive. The exit action is an instantaneous action. The enabling predicate evaluates to true whenever there is an enabling set present in the set of outcome bags. The set of outcome bags is the set of the following bags: *NoO*, *NoC*, *NoW*, *TO*, *COMP* and the *COMP* bag of the end task of the associated child process. An enabling set is determined by four cases, depending upon the task type and the task instance category.

Case 1: An enabling set for an atomic, single-instance task is a single token in the task outcome bags.

Case 2: An enabling set for a composite, single-instance task is a single token in the *COMP* bag of the end task of the child process.

Case 3: An enabling set for a composite, multiple-instance task is either the task threshold

number of siblings or all the siblings (whichever is less) in the *COMP* bag of the end task of the child process.

Case 4: An enabling set for an atomic, multiple-instance task is the task threshold number of siblings or all the siblings (whichever is less) in the *COMP* bag, or it is a set of siblings (not in the *COMP* bag) elsewhere that indicate that the task threshold number of sibling tokens cannot be achieved (at any future time) in the *COMP* bag.

The firing function for this action has the following effects: 1) an enabling set is selected in a uniform way from the set of enabling sets present in the outcome bags; 2) the enabling set is transformed into the firing set appropriate to the task split type; and 3) the firing set is placed on the exiting arcs per the functions defined by the task split.

Algorithm 10 Exit Action Execution Algorithm

```
1: SPLITTYPE ← task split connector type
2: TASKTYPE ← task type
3: CHILD ← the ENT bag in the start task of the child process
4: INST ← task instance category
5: THRESH ← task threshold for completion
6: B_EN ← the task outcome set of bags
7: B_FR ← task exiting arcs
8: while enabling_set_present(INST, THRESH, TASKTYPE,B_EN) {while there are tokens
   to exit} do
9:    Enabling_Set ← choose_set(INST, THRESH, TASKTYPE,B_EN)
10:   Firing_Set ← transform_set(SPLITTYPE, Enabling_Set)
11:   fire_action(Firing_Set, exit) {fire the action}
12:   B_FR ← Firing_Set
13: end while
```

The execution of the exit action is detailed in Algorithm 10, where B_{EN} is the set of enabling bags, B_{FR} is the set of firing bags, *enabling_set_present* is a function that evaluates to 1 when there is an enabling set contained within the set of outcome bags appropriate to the task type and category; *choose_set* is a function that chooses, in a uniform way, an enabling set from the set of all enabling sets contained within the set of outcome bags appropriate to the task type and category; *fire_action* is a function that performs the state change associated with firing the action for a given firing set; and *transform_set* is a function that transforms the enabling set to the firing set appropriate to the task connector split

Table 4.1: Relation Between Process and Task States

Process-Level Event	Task-Level Event	Process State after Event	Task State after Event
tokens entering task	enter fires	Active	Entered
*	execute fires, willingness is 1	Active	Executing
*	execute fires, willingness is 0	Active	No-Willingness
*	execute fires, composite task	Active	Executing Child Process
*	complete fires, capability exists	Active	Completed
*	complete fires, no capability	Active	No-Capability
*	timer fires	Active	Timed-Out
*	opportunity check fires	Active	No-Opportunity
tokens exiting task	exit fires	Inactive	Not Enabled

type.

These algorithms fully specify the internal execution of a task and, when related to the process-level execution algorithms, will fully specify the execution of a HITOP model.

4.4 Relating the Process-Level and Task-Level

In Section 4.2 we defined the process-level execution algorithms, and in Section 4.3 we defined the task-level execution algorithms. It is possible to completely specify the execution of a HITOP model by linking the execution algorithms between the two levels, as the process level explains the order in which tasks are activated within a process and the task-level explains the order in which actions are fired within a task.

The linkage between the two viewpoints is as follows. The "tokens entering task" event at the process-level is equivalent to the firing of the enter activity at the task-level viewpoint. Similarly, the "tokens exiting task" event at the process level is equivalent to the firing of the exit activity at the task level. All internal states after tokens enter a task and before tokens exit a task at the task-level are combined into the single process-level state "active."

Table 4.1 summarizes the relationship.

Note that the "$*$" symbol in the process-level event column means that there is not an equivalent event at the process-level viewpoint.

Using the linkage between process-level and task-level execution algorithms, we have fully specified the execution of a HITOP model.

4.5 Conclusion

In this chapter, we introduced the two viewpoints of HITOP execution: process-level and task-level. Next, we defined a set of execution algorithms for the process-level viewpoint and a set of execution algorithms for the task-level viewpoint. Finally, we completely specified the execution of a HITOP model by relating the two viewpoints though equivalent states and events.

CHAPTER 5

SPECIFYING THE SOLUTION MODEL: THE MULTIPLE-ASYMMETRIC-UTILITY SYSTEM EXPERIMENTAL FRAMEWORK

5.1 Introduction

In Chapters 3 and 4, we defined the HITOP process modeling formalism and execution algorithms, which gave us sufficient information to build an executable HITOP model. However, to reach our goal of building an experimental framework, we need more than an executable process model. We need a way to apply this model to the solution of problems of interest. In this chapter, we will discuss how to apply a process modeling formalism such as HITOP to an experimental framework, i.e., a solution model, and use it to answer questions about CHSs via discrete-event simulation.

Traditional approaches to cyber security evaluation either do not explicitly consider human participants "within" the system or view participants as a set of static properties that are independent of the system. In this chapter, we discuss a new approach that explicitly considers human participants as integral system elements. Here, *cyber security evaluation* means the assessment of security properties related to the prevention of, detection of, and response to attacks on any complex system in which computers play a key role.

Many cyber systems, by design or implication, involve human participants. Recall that we refer to systems of this kind as *cyber-human systems* (CHSs) (see Section 2.2) and propose that any model of a CHS is incomplete without due consideration of its human participants. CHSs are increasingly important within organizations, and thus, the security concerns sur-

This chapter contains previously published material by D. Eskins and W. H. Sanders. It is reused by permission of IEEE and was published in the *Proceedings of the 8th International Conference on Quantitative Evaluation of SysTems* (QEST 2011).

rounding them, i.e., *CHS security*, are also increasingly important.

Typically, CHS security measures focus on the evaluation of non-human system elements. For example, a CHS security measure might evaluate physical devices such as firewalls, software such as anti-virus programs, or administrative controls (ACs) such as security policies and procedures. Many ACs, however, depend upon the compliance of system participants, and it is common when evaluating such controls to assume that participants will always comply with the rules.

We believe that this assumption does not reflect how people behave in real environments and that the willingness of participants to comply with ACs can and will vary, depending on factors such as the design and configuration of the system, the state of the system, and the incentives for participants.

The failure of participants to comply with ACs can lead to serious security vulnerabilities that are not accounted for in traditional CHS security evaluation methods. At the core of this issue is the difference in goals between individuals and the organization [44], [61].

We propose to study the following problems. 1) How can modeling the difference between participant goals and other goals provide insights into human and system performance? 2) What can models of this type reveal about unexpected system performance or a persistent failure to meet goals? 3) Finally, how can explicit modeling of human decisions provide better analysis and configuration management tools than traditional CHS security evaluation approaches do?

In this chapter, we introduce the multiple-asymmetric-utility system experimental framework (MAUS) as a modeling tool for analyzing cyber-human systems. MAUS is a modeling approach for quantifying how humans "within" a CHS can affect the system security state. Our approach is to explicitly model human behaviors as integral system elements and measure how human decisions affect system state and vice versa.

We begin in Section 5.2 by introducing MAUS. We will discuss the construction of the MAUS system model in Section 5.3, and the definition of MAUS utility functions in Section 5.4. Next, we will we define MAUS system configurations in Section 5.5 and the use of willingness probabilities in MAUS experimental runs in Section 5.6. We will discuss the MAUS solution method and present a solution algorithm in Section 5.7. In Section 5.8,

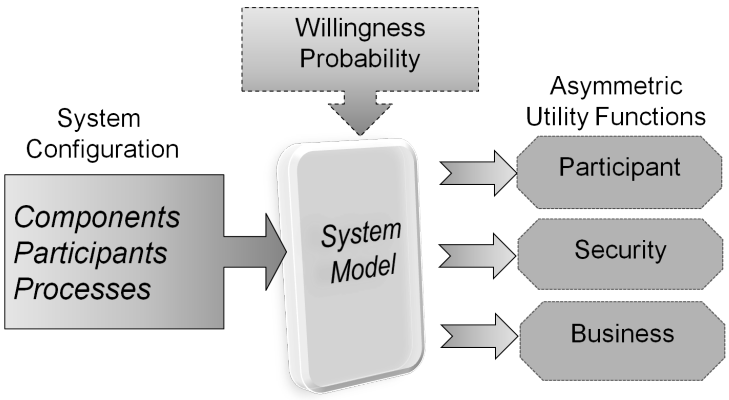

Figure 5.1: MAUS Experimental Framework

we apply MAUS to an example CHS and show how MAUS may be used 1) to validate human decision points (HDPs) within a given CHS; 2) to quantify the significance of human decisions relative to security goals; 3) to quantify the divergence between expected human behavior and ideal human behavior; and 4) to provide decision tools for system design and configuration. We summarize the chapter in Section 5.9.

5.2 The Multiple-Asymmetric-Utility System Experimental Framework (MAUS)

Given the modeling concepts and goals introduced in Section 5.2, we now introduce an experimental modeling structure for applying those concepts and goals, called the *multiple-asymmetric-utility system experimental framework (MAUS)* (shown in Figure 5.1). MAUS includes not only the system model, but also a structured way of running many experiments and consolidating results. It has four basic parts: 1) a system model, 2) a set of utility functions, 3) a set of system configurations, and 4) a range of willingness probabilities (WPs) for each HDP in the system model.

MAUS can be viewed in some sense as a relabeling of a process model's elements (such as defined for HITOP) and the setting of the values of those elements with an overarching experimental framework. The setting of experimental values is done for each MAUS solution. For example, suppose the task-specific outcome functions within a HITOP model (see

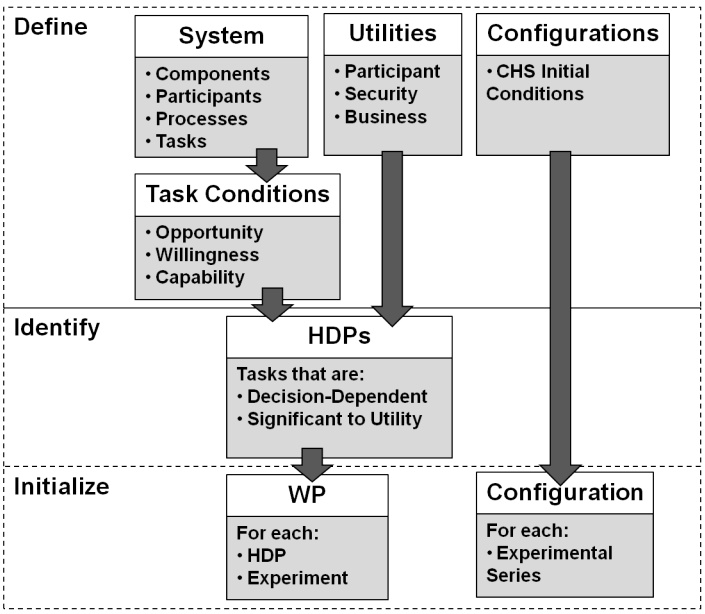

Figure 5.2: Building the MAUS Framework

Section 3.5) depend on the values of HITOP state variables. The values of the state variables may be set or varied to specify a given experimental configuration. Thus, within MAUS, the HITOP state variables which affect task-specific outcome functions would be relabeled as experimental configuration variables.

We set the stage for describing MAUS with the following problem. Given a CHS, a set of CHS configurations, and a set of CHS measures of value, what is the best configuration? MAUS will help us answer such questions. Figure 5.2 summarizes the basic process of building a MAUS framework as described in the following sections.

5.3 The System Model

The first step to building a MAUS model is to define and construct a system model as we have described in Chapters 2 and 3. We will briefly summarize this process as follows.

Recall that the system of interest is initially defined in terms of CHS elements as described in Section 2.2. It should be noted that when the CHS is defined, it must include measures of system value called *utility functions* (see Section 5.4), and that it is important to fully define the set of utility functions during the conceptual model definition phase, because these

utility values must be translated into HITOP state variables during the executable model construction phase. Next, we use the OWC ontology described in Section 2.3 to define the task conditions by associating sets of model elements, i.e., components and participants, with the functions of each task. Then we make an initial identification of potential human decision points (HDPs) by identifying tasks within the model whose outcomes depend on human decisions (as described in Section 2.4). Note that the second HDP criterion, significance to some utility function, cannot be determined until a working model has been constructed. Next, the CHS description and the OWC relationships are combined to define and build a HITOP model of the CHS as described in Chapters 3 and 4. Next, by using the HITOP model and the set of utility functions, we can identify a set of HDPs from the set of potential HDPs by determining which potential HDP is significant with respect to a given utility function. The set of HDPs along with the executable HITOP model is the MAUS system model. At this point, let us clarify how the system model is just a "relabeling" of the HITOP model.

5.3.1 Relabeling HITOP Elements for Use in MAUS

To be used as a MAUS model, a HITOP model must have certain parts "relabeled" for use in experiments. The willingness probability P_W for each HDP is "relabeled" as an experimental input. A subset of other HITOP state variables are also relabeled as configuration inputs. That is, HITOP state variables that reflect system properties to be evaluated, e.g., possible system configurations, are assigned values per the experimental framework. Additionally, we implement the system measures of value defined in the conceptual model definition phase by relabeling a subset of HITOP elements as utility elements. MAUS utility functions measure the values of the utility elements to represent the "value" of model state. Overall, this relabeling of the HITOP model results in the MAUS system model, as shown in Figure 5.1.

Note that we have assumed that the process model formalism used is HITOP. However, any appropriate discrete event modeling formalism may be used within MAUS, although they can be used with varying degrees of difficulty. As an example, the sample results presented in this chapter were obtained using a Stochastic Activity Network (SAN) [60] formalism (as HITOP had not yet been developed as an executable model when the results

were computed). Note that it was the need for a specific modeling tool to represent such systems simply that inspired HITOP.

In the next section, we shall discuss how utility elements are used by utility functions to measure system value.

5.4 Utility Functions

Utility functions define measures of value for a given system state during simulation. Much like the OWC functions described in Chapter 3, utility functions use a collection of state variables, called *utility elements* (UEs), to calculate domain-specific values. *Domain-specific* here references a given viewpoint, e.g., the viewpoint of a system participant or the IT security manager.

In modeling terms, UEs may represent additional states beyond those needed to simply represent the physical system accurately. As already discussed, these additional elements are added in the conceptual model definition stage as required to represent important system properties to be measured. For example, it may not be essential to model the cost of installing anti-virus software with respect to the operation of a network system model, but cost information is vital to the business utility metric. Because MAUS itself may be considered a larger model containing the system model, UEs can be defined as part of the state of that larger model, or as additional elements within the HITOP model.

MAUS uses *multiple asymmetric* utility function groups, one for each domain of interest, including a utility group for system participants. Though these domains may not be mutually exclusive, they can still be very different, and this difference is key to our solution method. That is, the decisions of participants, which influence the system state, are made by considering participant utility (not security utility). Thus, to accurately evaluate a system, multiple utility measures are used both to predict the decisions of participants (what we call in Chapter 6 *characterizing* the HITOP model) and to assess the value of the system state, i.e., configuration comparison. For now, let us consider two basic utility function groups: participant and security.

The *participant utility* (PU) function group measures the values of state variables called

participant utility elements (PUEs) defined relative to a participant. Many different types of measures are possible, including measures on instantaneous and time-averaged values of state. As implemented in the Möbius abstract functional interface (see Chapter 7), these measures are called *reward functions* and can be almost arbitrarily complex. See [47] and [60] for greater detail on reward functions. Example participant utility measures might include measures of job satisfaction or embarrassment.

Similarly, the *security utility* (SU) function group measures the values of state variables called *security utility elements* (SUEs) defined relative to a set of security metrics or goals. Example security utility measures might include measures of data integrity, confidentiality, and availability.

Because MAUS can define utility functions separately from the HITOP model, the addition of new utility function groups is straightforward if sufficient utility variables exist in the HITOP model, i.e., the required measurable state, to support the new utility functions. If not, the underlying process model must be modified. As an example of possible additional utility functions, Figure 5.1 includes a business utility function group that might be added to measure a business metric such as the cost of providing IT support.

Running Example: Utility Functions. We previously defined an example system in which business representatives used a USB stick to perform various functions. Company policy required that data on the USB stick be encrypted; however, each business representative could choose to follow that policy or not.

In order to evaluate the consequences of that decision, we must now define utility functions to assess the value of the resultant model state both to the participant, i.e., the business representative, and to the cyber security of the system. First, we shall define the utility function for the participant (PU). The PU function (f_{PU}) is a weighted linear combination of four PUEs (see Equation 5.1): successful data transfers, S; embarrassments, E; management dings, M; and negative support experiences, N.

$$f_{PU}(S, E, M, N) = \gamma_1 S - (\gamma_2 E + \gamma_3 M + \gamma_4 N). \tag{5.1}$$

S reflects the number of times that data were successfully transferred by the participant. It

is considered a reward and adds positive value to f_{PU}.

E represents the number of times a business representative was unable to retrieve data and was embarrassed. M represents the number of times a security scan discovered unencrypted sensitive USB data and a "black mark" was placed on the employee's record. N represents negative IT support experiences. E, M, and N are considered penalties and have a negative effect on f_{PU}.

Each PUE has an associated weight (γ_i) reflecting its importance to the overall PU value. The set of PUEs and their weights are determined by system stakeholders when the utility functions are defined as part of the model definition phase.

Next, we will define the security utility function (SU). The SU function (f_{SU}) for our example is a weighted linear combination (see Equation 5.2) of confidentiality, C, and availability, A.

$$f_{SU}(C, A) = \alpha(A - \beta C). \tag{5.2}$$

Availability is defined as the average number of successful data transfers performed by a participant each year. *Confidentiality* is defined as a weighted linear combination of data exposure events and amount of data exposed. A data exposure event occurs when unencrypted sensitive USB data are exposed to an unauthorized person. The weights α and β represent the relative importance of C and A with respect to f_{SU}.

It should be noted that the utility functions defined in this dissertation were selected during a study of the security aspects of USB stick usage [44]. The intent of MAUS is to provide a rationale and method for using utility functions to study CHSs. It is not a means of validation for utility functions, and any utility function applied to MAUS should be subject to domain-specific validation prior to use.

Next, given a defined system model and a set of utility functions, a set of initial conditions must be defined via the system configuration.

5.5 System Configurations

A *system configuration (SC)* is the set of initial conditions for which the system model will be solved. Typically, when referring to a given SC, we will describe only those items that will be varied for the purposes of the experiment. It should be noted that the initial conditions for most CHS elements are fixed, as they either do not frequently change (like hardware used by the system) or are physical properties over which the system manager has no control. As we discussed in Section 5.3, the SC assigns values to HITOP elements relabeled as configuration elements.

As an example, let us suppose that a given CHS model requires that initial values be defined for the set of n elements, $E = \{e_1, e_2, ..., e_n\}$, where each element, e_i, is itself a set of possible element state values. We define the set of configuration elements, $E_C \subset E$, as the subset of elements in E that will be varied between experimental series. If $E_C = \{e_1, e_2, ..., e_m\}$, we define C, the set of all possible SCs, as the set of all possible states of the configuration elements E_C or $C = \{e_1 \times e_2 \times ... \times e_m\}$. A specific SC, c_i, is thus an element in C.

Each c_i defines a corresponding *experimental series* i, ES_i. Each *experiment* in an experimental series is a system solution obtained using a fixed SC and a fixed willingness probability P_W. See Section 5.7 for more details.

Running Example: System Configuration. In our example, the system manager can control the portion of the IT budget spent on IT support, e.g., the help desk. The remaining budget is spent on enforcement of IT security policies. Since the portion of the IT budget spent on IT support or enforcement is the only aspect of SC that we are studying, i.e., it is the only initial SC value we are varying, each SC is characterized solely by the portion of the IT budget spent on IT support. Thus,

$$C = \{c_i \in [0, 1]\}, \tag{5.3}$$

where C is the set of all possible SCs and each c_i defines an experimental series with a fixed portion of the IT budget spent on support.

Because the underlying discrete-event system model is usually stochastic, an individual experiment must be repeated many times to build sets of results that have the desired confidence intervals (at least with respect to simulation results, as discussed in Section 5.3). Each repetition of an experiment is known as an *experimental run*, and the number of experimental runs required to achieve the specified accuracy for a given experiment will vary based upon the complexity of the model and the type of measures collected. In MAUS, each experiment is associated with a fixed P_W value as described in the next section.

5.6 Willingness Probability

For each experiment, a willingness probability P_W, as discussed in Section 2.3.3, is set for each HDP within the system. As stated in Section 2.4, P_W is in general one of many state-based inputs to the willingness function f_W. However, continuing the problem simplifications of Section 2.4, we will represent f_W as a Bernoulli random variable, with P_W representing the probability of a participant's deciding to properly perform an HDP. For example, if P_W is set to 0.30, each time the task is performed, there is a 30% probability that the participant will actually choose to attempt proper task performance.

For each HDP, a set of P_W values must be selected from the interval $[0, 1]$ to represent possible expected values for that decision. It is the goal of MAUS to determine over an experimental series which value of P_W maximizes f_{PU}, as described in the next section. Note that the problem setup can be viewed as a classical optimization problem; we will discuss solution methods of this sort in more detail in Chapter 6.

5.7 An Example MAUS Model Solution Method

Once all the relabeled system elements have been defined and linked to MAUS as described in the previous section, the MAUS solution model must be solved. To illustrate MAUS, we will describe an example MAUS solution method that is in effect a limited type of optimization. It solves each experimental configuration (as described in Section 5.5) over a range of P_W values for each HDP (described in Section 5.6). Solving the system model for each experiment

in most cases requires simulation, as models' complexity or structural characteristics do not often support an analytical solution. Thus, simulation was used to obtain the example MAUS solution results presented in Section 5.8. During model simulation, utility functions were applied to measure and record system values, as described in the next section.

To use our MAUS solution method, each experimental result is obtained using an iterative simulation method paired with an assumption about human decision-making, as described next.

5.7.1 Example Solution Description and Assumptions

The MAUS example solution method uses the following procedure for solving a set of experimental series: for each fixed SC, an experimental series (as discussed in Section 5.5) is solved in which P_W for each HDP is varied from 0.0 to 1.0. Each experiment (within an experimental series) consists of solving the system model for a fixed SC and a fixed P_W. Utility measures are extracted during each experiment and stored.

When all experiments in the series are complete, the process is repeated for the next SC until all SCs have been solved.

It is our explicit assumption here that the expected value of a given human decision, i.e., the *expected willingness probability (EWP)* that a participant will attempt proper task performance, can be estimated by choosing the experimental run with the highest participant utility value. That is, on average, a participant will make the decision that provides the most value to him or her based on the provided system state. This is our assumption that humans are reasonable decision-makers and will attempt to maximize their utility value. However, the decision made to maximize the participant's utility may not maximize the utility of other system measures. Thus, because human decisions are part of the system model, i.e., they in part characterize system performance, other utility measures must be experimentally derived using the expected value of P_W for each HDP within the system.

Note that other models of human decision-making are possible, and while this example solution method makes use of the idea that decisions are made to maximize utility, the MAUS modeling framework itself can be used with any decision-making model.

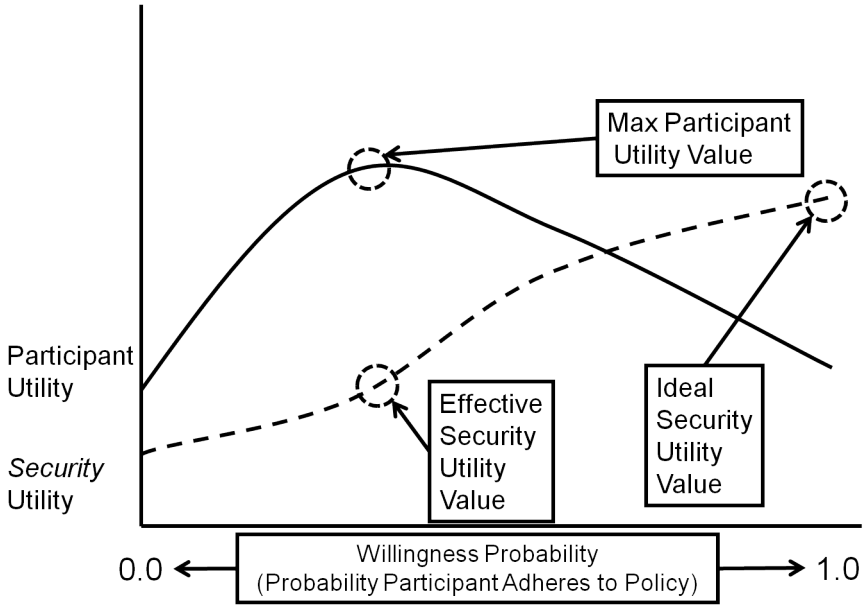

Figure 5.3: Single HDP Example

Our example solution method is illustrated for a single experimental series in Figure 5.3. Here we consider a simple case in which MAUS is used to solve for the EWP of a single HDP using a single PU function and a single SU function. Note that the highest participant utility is achieved at approximately a 40% P_W value. That value is considered the *EWP* for this configuration, i.e., the human decision probability that characterizes actual system performance. The *effective SU* value can be determined by computing the SU value that corresponds to a system solution using the EWP. The effective SU value is the SU value expected during actual system operation in the given configuration. We also note that many current security metrics assume 100% participant compliance with administrative security policy, i.e., an EWP of 1.0, and we shall refer to the SU value corresponding to 100% compliance as the *ideal SU* value, SU_{Ideal}.

We assert that the effective SU value is a more accurate representation of security performance than the ideal SU value is because the effective SU value accounts for the effects of human decisions.

Simple decision-making tools can be developed in light of that realization. For example, the difference between the ideal and effective SU values can be defined as the *SU divergence (SUD)*. System performance metrics utilizing concepts such as the SUD, when calculated

over a set of possible SCs, can provide security managers with intuitive tools for managing SCs. For example, in Section 5.8.3 we use SUD to provide a simple graphical tool for selecting a configuration with security performance closest to the "ideal," i.e., a SUD closest to zero.

5.7.2 Exhaustive Search Solution Algorithm

Given the assumptions and descriptions provided in the previous section, we can now write precisely the steps needed to solve MAUS. Algorithm 11 provides such an example MAUS solution method that is an exhaustive search method. That is, this example solution method will generate every possible solution and identify the highest value among them. Note that for each fixed SC c_i, the system model (M) is iteratively solved for a range of P_W's from 0.0 to 1.0. A fixed P_W value (i.e., $PU(P_W)$) is used to record a participant utility (PU) value for each solution. The P_W that results in the highest PU value, i.e., PU_{MAX}=max{PU(P_W), $P_W \in [0,1]$}, determines the EWP, i.e., EWP = max arg $\{PU(P_W), P_W \in [0,1]\}$. The EWP is then used to solve the system model for the effective SU value, i.e., SU_{EFF}.

Algorithm 11 MAUS Solution Method

```
1: c_i ← SC for Experiment i
2: EWP ← 0 {Initialize EWP to zero}
3: PU_MAX ← 0 {Initialize PU_MAX to zero}
4: for P_W = 0.0 to 1.0 do
5:    PU(P_W) ← M(c_i, P_W) {Solve model for SC i and P_W}
6:    if PU(P_W) ≥ PU_MAX {Find maximum PU value} then
7:       EWP ← P_W {Set EWP to corresponding P_W}
8:       PU_MAX ← PU(P_W) {Record maximum PU value}
9:    end if
10: end for
11: SU_EFF ← M(c_i, EWP) {Solve model for SC i and EWP}
```

It should be noted that this method, like any exhaustive search method, is not easily scalable. The solution space that must be explored becomes prohibitively large as the number of HDPs grows.

5.8 Running Example: Application of MAUS Example Solution Method to USB Usage Example

In this section, we will provide some actual experimental results from using our example MAUS solution method to analyze our running example. As stated previously, the example is based upon the USB costs and benefits study undertaken in [44] (introduced in Chapter 2.2) and examines the effects of IT security investment decisions on administrative security policy compliance and SU values.

Recall that spending in the IT security budget can take two forms: IT support or IT security policy enforcement. The annual security investment is fixed and can be spent to improve the IT support desk or to increase participant monitoring. Spending on IT support increases the probability that IT support will be successful, perhaps as the result of hiring more support staff or improved customer service.

Monitoring is used to check USB devices periodically for unencrypted sensitive data (a violation of the security policy that is punished with a management ding, M). Increasing the investment in monitoring results in more frequent checks of USB devices.

The portion of the total IT budget spent on either the IT support desk or monitoring is varied for each configuration. For example, a particular configuration might spend 30 percent of the budget on monitoring and the remaining 70 percent on IT support.

To produce the results described below, we examined a range of SCs, i.e., the portion of the IT security budget that was spent on support, and we calculated the EWP, i.e., the expected probability the business representative will make the decision to encrypt USB data per the security policy, for each configuration.

Results are presented first as raw participant utility values for each configuration, and then as plots of the EWP for each configuration.

5.8.1 Finding the Expected Willingness Probability

We examined six SCs (see Figure 5.4) by varying the portion of the annual security investment spent on IT support, using values between 0.0 and 1.0. For each configuration, the

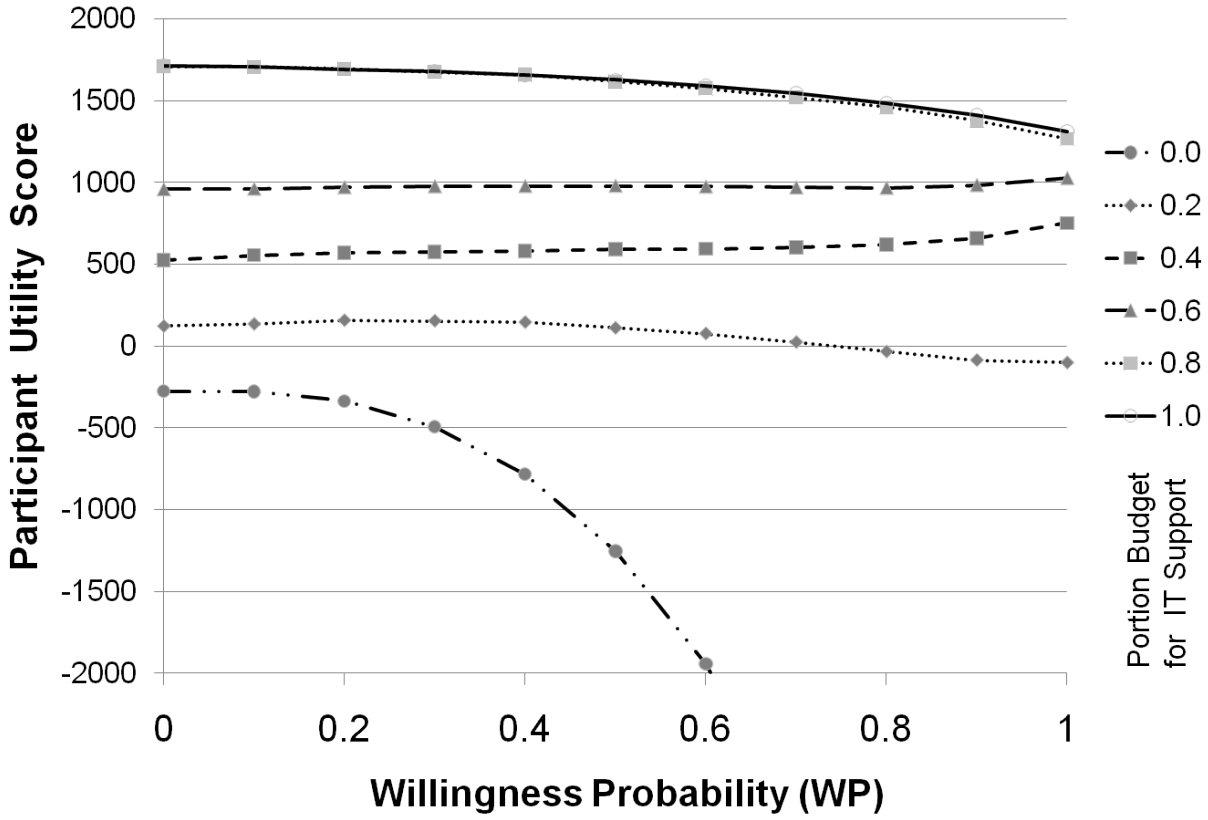

Figure 5.4: IT Support Budget Portion Study

probability that the participant would comply with the security policy and encrypt sensitive data, namely the P_W for the *Encrypt USB Data* HDP, was varied from 0.0 to 1.0. A corresponding PU value was calculated for each P_W value. By the assumptions of the example MAUS solution method, the P_W corresponding to the highest PU value represents the EWP for that SC.

We should note here that some experts believe that human beings do not strictly make utility-maximizing decisions [62]. We address that concern with two points. First, MAUS is used to find the expected value of a decision over time, not calculate an individual decision. We believe this can provide at the very least the general tendency of human behavior relative to the system, or the average decision value of many people over many task attempts. Second, on can use the current MAUS framework to evaluate more complex human decision models by modifying the participant utility functions and changing the basic assumption of the solution method, i.e., that humans make decisions to maximize utility. Literally any human decision or human motivation theory that relies on measurable quantities [10], [63], [64], be it behavioral, cognitive, or economic, can be implemented using MAUS as long as appropriate model state is defined.

5.8.2 Interesting Observations about the Results

Several interesting observations can be made from the results. First, note in Figure 5.4 that when all of the security investment was spent on monitoring, i.e., a completely punitive strategy corresponding to an SC with an IT support budget portion of 0.0, the highest PU value occurred at an EWP of 0.0. This indicates that participants find it more rewarding not to follow the security policy even when punishment is highest. Why? The results indicate that without good IT support, encryption becomes so difficult that it is actually more beneficial for participants to transfer unencrypted data and risk punishment than to attempt encryption. The value of increased successful data transfers (S), fewer embarrassments in front of clients (E), and fewer negative support experiences (N), outweighs the increased risk of management dings (M).

The results from the opposite case, i.e., the case in which all the security investment was

spent on IT support (corresponding to an SC with an IT support budget portion of 1.0), can be understood in a similar way. Here, the EWP, i.e., the expected compliance with the security policy, is also zero, because the risk of management dings (M) is low (because of no monitoring) compared to increases in successful data transfers (S) and decreases in embarrassments (E) and negative support experiences (N) that result from not using encryption.

5.8.3 Derived Functions

Above, we described how a given SC may be solved for an EWP. Now, we will describe two example functions of EWP that may be used for configuration decisions: the willingness transfer function and the SU divergence ratio.

Willingness Transfer Function

Sometimes it may be desirable for a security manager to understand both how much users of the system will comply with security policy and what effect their compliance will have on the system cyber-security measures. We propose an example tool called the *willingness transfer function* (see Figure 5.5) that helps security managers understand such relationships. It is a decision-making tool that allows security managers to take human behavior into account when selecting the best SC. Figure 5.5 plots the EWP and SU_{EFF} values for each SC in our running example system. The graph in the figure shows how participant behavior, i.e., the EWP for complying with the administrative security policy, varies with SC, i.e., the portion of the IT security budget that is spent on IT support. The graph also shows the net effect of business representative behavior on the resultant system SU value, i.e., SU_{EFF}.

Note that the highest levels of compliance were actually achieved when spending was approximately evenly divided between IT support and monitoring. That seemingly provides the right mix of rewards and punishments for this particular system.

Of course, achieving the highest compliance with security policy is not always the same as achieving the highest SU_{EFF} value. Figure 5.5 indicates that the configurations with the highest participant compliance (configurations with an IT support budget of 0.4 to 0.6) actually correspond to the lowest SU_{EFF} values. Effects of this type demonstrate the kind of counterintuitive result that a MAUS quantitative analysis can reveal. In this case,

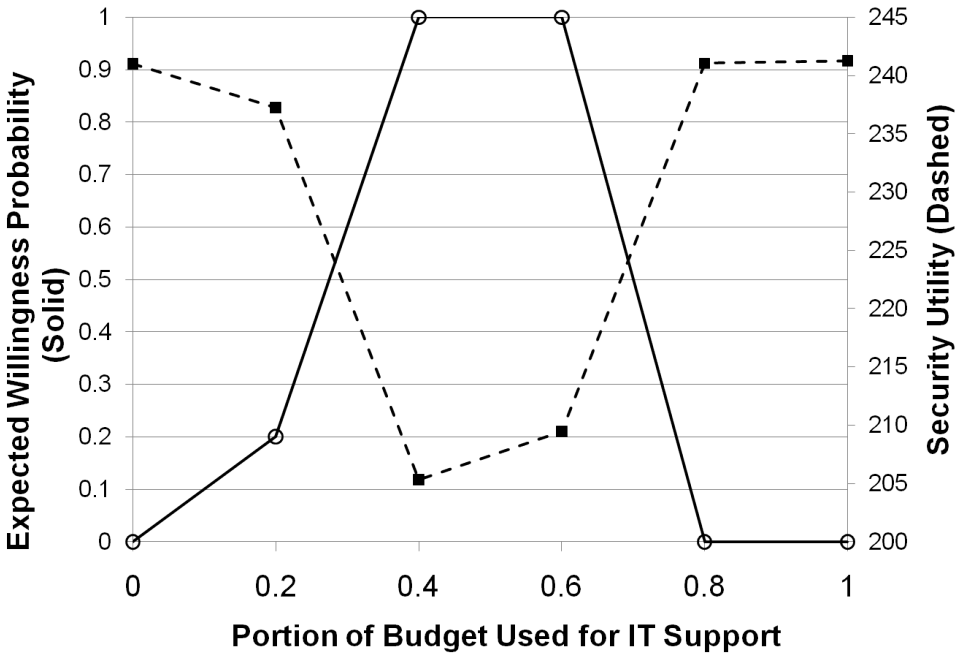

Figure 5.5: USB Usage Willingness Transfer Function

compliance with the security policy actually has a negative effect on the company IT security metric.

In fact, the highest SU values were achieved by allocating the security investment budget all on monitoring or all on IT support. Those configurations, as discussed before, also happen to be those that result in the lowest participant compliance with administrative security policy. Why is this the case? Another advantage of quantitative models is that one can "look" into the model structure to examine why certain results are found. By looking at the basis model for our running example, we can see that the results most likely reflect the importance of USB data availability to both the participant utility and security utility values. Increased availability results in higher participant and security utility values. Thus, we observe the counterintuitive result that lower participant compliance with security policy actually results in "better" security performance. Since the system security value is measured by the SU function, a system manager would presumably select a configuration to maximize the value of SU, not participant compliance. Understanding how system participants are expected to behave and how that behavior may affect overall system SU values becomes one more tool system managers can use in making SC decisions.

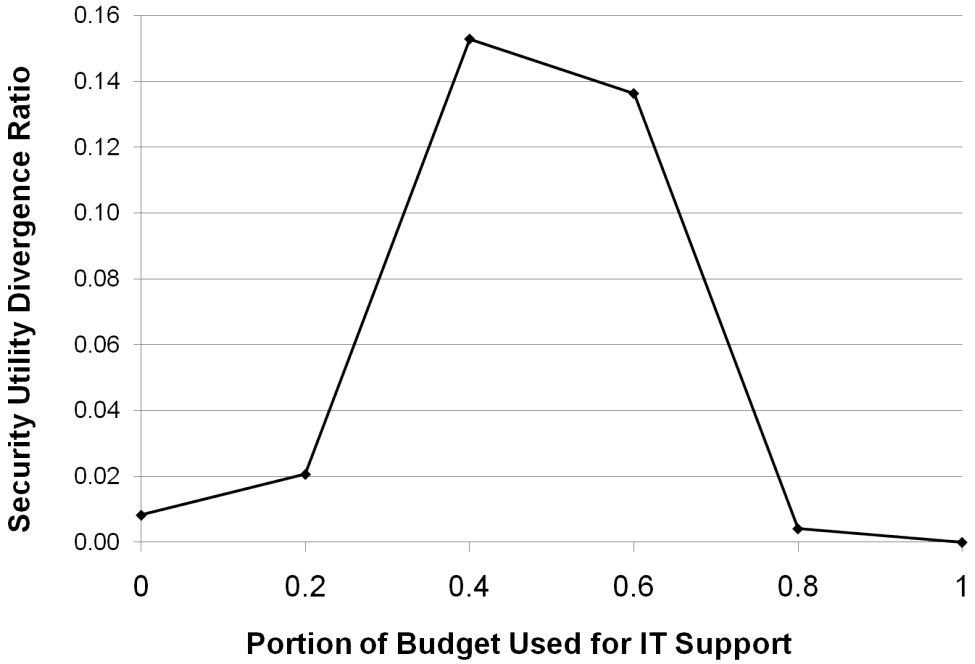

Figure 5.6: Security Utility Divergence Ratio

Security Utility Divergence Ratio

A security manager may also wish to quantify how the SU value for a given SC varies from its ideal value, i.e., the value achieved at 100% participant compliance with security policy. Using the idea of SU divergence developed in Section 5.7, we can create a simple metric called the *SU divergence ratio* (SUDR) calculated relative to the highest SU value (see Equation 5.4), and plot the SUDR value for each configuration as shown in Figure 5.6.

$$SUDR = (SU_{Ideal} - SU_{Eff})/SU_{Ideal}, \tag{5.4}$$

where SU_{Ideal} is the ideal SU value, and SU_{Eff} is the effective SU value for a given configuration. Let us assume for the sake of simplicity that the SU_{Ideal} for each SC is approximately 240. It can easily be seen from Figure 5.6 that the worst security performance, i.e., the largest SU divergence, is approximately 15% and occurs at a 40% investment in IT support.

Per our discussion in Section 2.4, an HDP must both involve a human decision and have a significant effect on a system utility function. For our purposes, we will consider the 15% maximum SU divergence to be significant and thus claim that the task, EUD, is a valid HDP per our criteria.

5.8.4 MAUS Insights

In this section, we examined the results of applying MAUS to an example CHS and showed that several useful insights into system performance could be gained. We showed how the effects of human decisions could be separately quantified using asymmetric utility functions (Figures 5.4 and 5.5). We presented the willingness transfer function as a tool for security configuration decision-making (Figure 5.5), and showed how a divergence between effective and ideal human behaviors could be detected and quantified (Figure 5.6). Finally, we showed how MAUS can be used to identify the significance of an HDP.

5.9 Conclusion

In this chapter, we introduced the multiple-asymmetric-utility system experimental framework (MAUS). We described how a system model can be constructed using a process model and a set of human decision points. We defined system utility functions and provided examples of how utility functions might be constructed. We defined system configurations and related them to a MAUS experimental series. We defined how willingness probability is related to a MAUS experiment and provided the basic algorithm for the MAUS experimental solution method. Finally, we provided an example MAUS solution set to demonstrate how MAUS may be used 1) to validate human decision points (HDPs) within a given CHS; 2) to quantify the significance of human decisions relative to security goals; 3) to quantify the divergence between expected human behavior and ideal human behavior; and 4) to provide decision tools for system design and configuration.

In general, we showed how MAUS can be used to develop tools to quantify the difference between human goals and other organizational goals and to provide quantitative insights into overall system performance. We also provided an example in which changing of security policies resulted in counterintuitive results, and we used MAUS to examine the underlying reasons for these results. Additionally, we demonstrated how MAUS could be used to provide quantitative decision-making tools that are useful for system configuration and management decisions.

CHAPTER 6

SOLUTION METHODS

6.1 Introduction

In the previous chapter, we outlined an experimental framework called MAUS and provided an example solution method for MAUS. However, in order to discuss solution methods and solution efficiencies for MAUS in general, we must first formally define the problems we are using MAUS to solve.

In this chapter we will formally state the MAUS HDP solution problem, provide some general background on optimization methods, describe in detail the Linear-Most-Likely-Utility (LMLU) HDP solution method (including applicable model classes, algorithms, and theorems), and discuss the general efficiency of the LMLU algorithm and how insights gained from the LMLU approach may be applied to make general optimization approaches more efficient for certain classes of models.

6.2 The MAUS Problem

A MAUS problem consists of a system model, e.g., a HITOP model, a set of system configurations to be studied, a set of human decision points (HDPs), and a set of utility functions.

A MAUS solution has two basic steps. Step one is to *characterize* the set of HDPs for each system configuration and participant utility function. Characterizing an HDP means assigning a probability value to the decision represented by the HDP, i.e., the willingness probability P_W. HDPs may be characterized using various methods such as expert knowledge, experimental data, or an HDP solution method. A characterized system model represents the behavior of a cyber-human system (CHS) in the presence of human decisions that has

been characterized relative to a given set of participants, utility functions, configurations, and HDPs. The measured behavior of a characterized model is known as the *expected system performance*.

Step two of a MAUS solution is to make configuration decisions based on the characterized system model. The expected system performance relative to some utility measure for each configuration is evaluated and a configuration is selected based on some desired outcome. For example, the security manager of a CHS may evaluate various configurations in order to select the one that will maximize some cyber-security utility function. It is assumed here that once the effects of human decisions have been determined via system characterization, standard configuration optimization methods [65] can be applied to the performance of step two.

Thus, we shall focus generally in this chapter on step one, characterization of a system model, and we will focus specifically on the various HDP solution methods that are applicable to a HITOP model.

6.2.1 General HDP Solution Problem

In general, an HDP solution requires solution of an optimization problem of the form:

$$\underset{x \in \mathbb{A}}{\operatorname{argmax}} \; f(x) : \mathbb{R}^n \rightarrow \mathbb{R}, \tag{6.1}$$

where x is a vector of n parameter values; $\mathbb{A} \subset \mathbb{R}^n$ is the n-dimensional parameter space known as the *feasible region* or *feasible set*; and f is the objective function. The function f maps each set of n parameter values to a single value. To solve Equation 6.1, it is necessary to find the set of x's for which f is maximized subject to the constraint of $x \in \mathbb{A}$.

6.2.2 The MAUS HDP Solution Problem

For the optimization problem associated with an HDP solution, $\mathbb{A}$ represents the set of all possible P_W's associated with n HDPs, and the objective function is associated with a participant utility function (u) for the CHS under study. Each optimization problem must be

solved for a given system configuration and experimental series, where each experiment in an experimental series is a set of P_W values in $\mathbb{A}$. Note that if there are multiple participant utility functions, this optimization problem may be solved for each participant utility function or, if desired, by using an approach for optimizing all utility functions. Such multi-utility function approaches include constructing a "meta-utility function" or multiple-factor utility function [14], [66], i.e., a function built using the individual utility functions as parameters, and solving it, or building a set of "best possible" solutions using a Pareto optimization approach [65].

We also note that in solving that optimization problem, it is assumed that all HDPs have been identified as tasks that 1) involve human decisions, and 2) are "significant" to the values of the objective function being optimized. All tasks that involve decisions are easily identified by construction of the model; however, a screening method like sequential bifurcation [67] can be used to identify which of the tasks are significant to the objective function, e.g., the utility function of interest.

Structure and Constraints of the HDP Solution Problem

The structure of the CHS we are solving provides us with several important constraints. One constraint is that the feasible region for each willingness probability parameter is between 0 and 1, i.e., $P_W \in [0, 1]$, as willingness is a probability measure. Also, while P_W is theoretically continuous over the interval $[0, 1]$, it may be reasonable in certain cases to solve HDPs for only a finite set of discrete values of P_W. That may be justified for problems in which the overall uncertainty involved in other parts of the system model dominates utility measures, and in which, P_W precision beyond a certain point will not contribute to uncertainty reduction. Limiting the precision that is required for our P_W solutions is an example of a so-called "good enough" solution in which differences between elements of $\mathbb{A}$ must exceed some threshold in order for those differences to be considered significant. The threshold may reflect such things as measurement limitations or the general structure of the problem (as discussed above with respect to P_W precision).

Additionally, our knowledge of human behavior and the process structure may allow us

to place further constraints on $\mathbb{A}$. For example, we may know that certain decisions are always made with probabilities in the range $[a_1, a_2] \subset [0, 1]$. We may also know from system structure that the probability of one decision is highly dependent on the outcome of an earlier (within the process) decision. Thus, if the probability of decision D_1 is within the range $[a_1, a_2]$, that may imply that the probability of decision D_2 is within the range $[a_3, a_4]$.

Another important constraint is on the output set $\mathbb{B}$. A simulation such as HITOP results in probabilistic estimates of system behaviors. A utility function may be viewed as a random variable that maps a set of probabilistic system outcomes to a real number value. A simulation output can estimate a utility function in a variety of ways, perhaps as an expected value, a variance, or even a probability distribution. For the purposes of this discussion, we will measure the *expected value* of a utility function estimated within a given confidence interval. That is, the objective function being optimized will be $J(x) = E[u(x)]$, where u is the utility function of interest.

Because $J(x)$ is an estimate with an associated confidence interval, the question of comparison between two values of the objective function, e.g., $J(x_1) = j_1$ and $J(x_2) = j_2$, must be considered. We must be able to order solutions if we are to select the maximum value per our optimization problem. The optimization literature provides many, sometimes computationally expensive, methods for providing a total order among solutions of stochastic optimization problems, such as statistical hypothesis testing or many-to-one comparison tests [68]. Usually such comparisons come with a confidence level, i.e., $j_1 \leq j_2$ with a probability of p. In certain cases we can approximate the comparison problem by choosing a positive value ϵ for each utility function and defining the approximately equal relation between solutions as follows: $j_1 \approx j_2$ if $j_2 \in [j_1 - \epsilon, j_1 + \epsilon]$ $\forall$ $j_1, j_2 \in \mathbb{B}$. Furthermore, we choose ϵ to be larger than either of the associated confidence intervals of j_1 and j_2 so that values with overlapping confidence intervals will be considered approximately equal. We may justify such a simplification for certain problems for which the structure of the utility function does not require a precision beyond ϵ. For example, a company's business utility, measuring company expenditures in the millions of dollars, may consider differences of 100 dollars or less as unimportant and will thus present utility values only in terms of larger monetary increments. Dollar amounts whose difference is less than 100 dollars would thus

be considered equal for the purposes of this measurement.

Statement of MAUS HDP Solution Problem

Equation 6.2 specifies the MAUS HDP solution problem for each participant utility function.

$$\underset{x \in \mathbb{A}, J(x) \in \mathbb{B}}{\operatorname{argmax}} \; J(x) : \mathbb{A} \rightarrow \mathbb{R}, \tag{6.2}$$

where $x = \{\theta_1, \theta_2, \cdots, \theta_n\}$ is a vector of n P_W values; $\theta_i \in \Theta_i$ where $\Theta_i = \{\theta_i | \theta_i \in [0, 1]$ and θ_i is the P_W for HDP $i\}$ is the finite set of all possible willingness probabilities for HDP i; $\mathbb{A}$ $= \{\Theta_1 \times \Theta_2 \times \cdots \times \Theta_n\}$ is the set of all possible P_W configurations for the system; $J(x)$ $= E[u(x)]$ is the expected value of the utility function u for the set of parameter values x; and $\mathbb{B}$ is the set of all possible unique values for $J(x)$. The solution to Equation 6.2 is the set of willingness probabilities that maximizes the objective function J.

6.3 Types of Optimization Methods

In this section, we will briefly discuss the various classes of optimization methods as background for solving Equation 6.2. Here *optimization method* means any heuristic or analytical means used to determine a local or global maximum value for the objective function. Optimization methods can be broadly divided into two types: those that involve *solving* an analytical expression for the system, and those that involve *searching* the feasible set. Because the objective function we are optimizing is the output of a complex system simulation, it cannot usually be represented with an analytical expression. Thus, our focus in this section will be on optimization methods that search the feasible set for an x that produces a maximum value of the objective function.

Search optimization methods may be divided into two basic types: *derivative* and *derivative-free* methods [69], [70], [71]. Derivative methods rely on derivative information of either the objective function or some approximation of the derivative of the objective function to identify both the direction of steepest ascent and the location of maximum solution points.

In most cases, there is no direct analytical method for calculating the derivatives of a complex simulation objective function, and because of uncertainty in the resulting objective function values, a local derivative approximation method may be unreliable. If derivative-based search methods are used on problem classes such as HITOP process models, solutions usually require a significant number of additional simulation runs and greater solution complexity, as in RSM [72].

Derivative-free search methods rely directly on the values of the objective function, and not on derivative information, for their search heuristics. Derivative-free methods are sometimes better-suited than derivative-based methods to problems in which derivative information is unavailable or difficult to approximate [70]. Additionally, derivative-free methods can offer convergence guarantees similar to those of derivative-based models [71]. Thus, our focus will be on search methods that are derivative-free. An example of such a derivative-free optimization method is a technique known as *direct search* [71].

6.3.1 Direct Search

Hooke describes direct search as follows [73]:

> We use the phrase "direct search" to describe sequential examination of trial solutions involving comparison of each trial solution with the "best" obtained up to that time together with a strategy for determining (as a function of earlier results) what the next trial solution will be. The phrase implies our preference, based on experience, for straightforward search strategies which employ no techniques of classical analysis except where there is a demonstrable advantage in doing so.

Kolda sums up the requirements of direct search mathematically in [71]:

1. There is assumed to be an order relation $\prec$ between any two points x and y. For instance, in unconstrained minimization, points x and y may be compared as follows: $x \prec y$ if $f(x) < f(y)$. That is, x is "better" than y because it yields a lower objective function value.

2. At any iteration, only a finite number of possible new iterates exists and the possibilities can be enumerated in advance.

Direct search (DS) has several advantages when applied to the class of optimization problems exemplified by the HDP solution problem:

1. Derivative information is not required, so DS may be applied to problems for which derivatives are not available or are difficult to compute.

2. Objective function values are used directly, so DS does not require the construction of local or global models of the objective function, as RSM does [72].

3. DS optimization methods can be implemented in a straightforward manner that offers good performance for many classes of problems, and overall solution effort may be minimized compared to other solution methods.

4. DS can be provably convergent for certain classes of objective functions, and such convergence guarantees are comparable to those of derivative-based methods.

5. Because they do not rely on objective function structure, DS methods are applicable to many general classes of objective functions.

6. Because the number of iterates must be finite, a finite step size must be defined. That is a natural solution structure for problems in which a "good enough" solution may be specified.

7. DS is better-suited to handle cases for which objective function evaluations are "noisy" as is the case for stochastic model outputs [74].

DS also has certain disadvantages:

1. Convergence can be slower than for derivative-based methods.

2. DS may not be as efficient as other search methods for certain classes of problems when evaluated with respect to the total number of iterates needed to find a solution.

However, this disadvantage may not be as certain when "total" solution effort is considered, such as, the effort needed to calculate derivative information or implement more complex algorithms.

Next we will discuss several examples of direct search methods.

Exhaustive Search

The first and most basic type of direct search method is an *exhaustive search (ES)*, sometimes referred to as the *combinatorial* method in deterministic optimization. An ES method solves the optimization problem by calculating f for all possible combinations of parameters and choosing the largest objective function value(s). Thus, if $\{\theta_1, \theta_2, \cdots, \theta_n\}$ is a vector of n P_W parameters for which f must be optimized, and $|\theta_i|$ is the number of possible values for parameter i, the total number of solutions for f required for the ES solution method is $\Pi|\theta_i|$, $i = \{1, \ldots, n\}$. As an example, if this method were used to solve an HDP solution problem with just three HDPs, and only ten values were evaluated for each W_P, $10 \times 10 \times 10 = 1000$ individual solutions for f would be required, and each solution, i.e., an experimental run, might require $100,000$ or more iterations to achieve the desired confidence interval. The method is guaranteed, given enough time, to identify the element of $\mathbb{A}$ for which the objective function is maximized; however, the method is obviously not very scalable in terms of the number of HDPs and the precision of solutions.

6.3.2 Random Search

Another simple-to-implement direct search method is random search. In random search, points within the feasible set are randomly selected for each iteration, and the point with the maximum value is retained. Instead of searching all possible points, this method searches a random subset of $\mathbb{A}$ and produces a maximum value that has a given probability of being the true maximum of the objective function. If one assumes that there is a satisfactory region $S(\theta^*)$ within the search space for which all solutions are considered equally acceptable, and that all iterations in the search algorithm are independent, the probability of the maximum

of n iterates $\hat{\theta}_n$ being within $S(\theta^*)$ can be represented using Equation 6.3 [68].

$$P(\hat{\theta}_n \in S(\theta^*)) = 1 - \prod_{k=1}^{n}[1 - P(\theta_k \in S(\theta^*))], \tag{6.3}$$

where $1 - P(\theta_k \in S(\Theta^*))$ is the probability that iterate k does not fall within the satisfactory region. If an acceptable region can be defined, then n may be calculated to achieve any desired probability that the estimated maximum value will be the actual maximum value. However, n grows rapidly as the number of search dimensions increases, and thus random search becomes impractical if used to search solution spaces with a large number of search parameters.

6.3.3 Compass Search

Direct search (DS) methods solve the optimization problem by exploring a subset of the exhaustive search space. A DS method uses information about the relationships between different values of f to "guide" the search to the maximum value without necessarily having to explore all possible values of x. In [71], Kolda proposes a basic compass search algorithm in which

- θ_i is the ith iterate.
- Δ_i is the step-size parameter.
- $\Delta_{tol} > 0$ is the minimum step size to be used for convergence testing.
- $D_{\oplus}$ is the set of unit coordinate vectors in the n-dimensional search space.

Algorithm 12 implements that type of compass search for each iteration of the search. It should be noted that each iteration looks for a search direction d_k that increases the value of the objective function for the current step size. If care is taken when implementing this algorithm to ensure an appropriate 1) set of possible search directions $D_{\oplus}$, 2) iterate comparison method, and 3) step-size change method, then compass search can be proved convergent for certain classes of objective functions [71].

Algorithm 12 Direct Search Algorithm

1: **if** $\exists\ d_k \in D_{\oplus}$ such that $f(x_k + \Delta_k d_k) > f(x_k)$ **then**
2: set $x_{k+1} = x_k + \Delta_k d_k$ {Update current iterate value}
3: set $\Delta_{k+1} = \Delta_k$ {Do not change step size}
4: **else**
5: set $x_{k+1} = x_k$ {Do not change current iterate value}
6: set $\Delta_{k+1} = \frac{1}{2}\ \Delta_k$ {Decrease step size by half}
7: **if** $\Delta_{k+1} < \Delta_{tol}$ **then**
8: terminate search
9: **end if**
10: **end if**

The optimization literature can provide additional, sometimes more complex methods for solving the HDP solution problem. However, we shall end our discussion here by noting that while many traditional stochastic optimization approaches could be used to solve for HDP P_W values, they all suffer from issues with computational complexity and/or scalability. Thus, in the next section, we propose a solution method that exploits our structural knowledge of the HITOP system model to gain solution efficiency. It is called the *Linear Most Likely Utility (LMLU)* solution method.

6.4 Linear Most Likely Utility Solution Method

In this section, we will discuss the linear most likely utility (LMLU) method for solving a MAUS HDP solution model. The LMLU method exploits both the process structure and the nature of human decisions to solve the optimization problem using a method entirely different from direct search. The LMLU method uses the structure of HITOP models to solve for HDP probabilities in the reverse linear order of the HDPs' appearances within the process model. The LMLU method assumes that human decisions are made to maximize a utility value as perceived by the human participant in the system. In order to describe the LMLU solution, we must first describe the LMLU solution model state.

6.4.1 LMLU Model State Space Description

In Chapter 3, we described the state variables and state space used for the construction of a general HITOP model. In Chapter 5, we described how the state variables and state space of a HITOP model may be relabeled to better describe the workings of the MAUS solution model. In this section, we will again relabel a subset of the HITOP state variables and state space that are used by the LMLU method to solve for HDP P_W values. The set of relabeled state variables will be known as the *LMLU model* and the set of relabeled states as the *LMLU model state*.

LMLU State Variables and Model Types

Before we can describe the LMLU state space, we must first define the set of LMLU state variables used to hold state values.

Definition 6.4.1. The set of *LMLU state variables* $\mathcal{V} = \{\mathcal{L}, \mathcal{F}, \mathcal{E}, \mathcal{U}\}$ is a regrouping of a subset of the HITOP state variables defined in Section 3.6 for the purpose of characterizing process and state space flow. The set of token location variables $\mathcal{L}$ marks token locations within the model. The set of process flow variables $\mathcal{F}$ controls process flow within the model, i.e., these variables determine token exit and entry states for each task. The set of task performance variables $\mathcal{E}$ controls task execution properties like timing, performance outcomes, and performance outcome probabilities. The set of utility variables $\mathcal{U}$ records utility function values on model state. $\mathcal{L}$ is a subset of the HITOP process instance tree state variables (Figure 6.1), which in turn are a subset of the process state variables. $\mathcal{F}$, $\mathcal{E}$, and $\mathcal{U}$ may be drawn from subsets of any HITOP state variables.

Figure 6.2 provides a representation of the LMLU state variable classes and their purposes. Each class of LMLU state variable is represented by a circle; process-level task features are represented within boxes; and arrows indicate influences of one object on another. Note that the sets $\mathcal{F}$, $\mathcal{E}$, and $\mathcal{U}$ can overlap, as a HITOP model variable may belong to several classes of LMLU state variables. Note also that the set of flow variables $\mathcal{F}$ along with the set of location variables $\mathcal{L}$ determines how tokens enter or exit a task and that task performance is

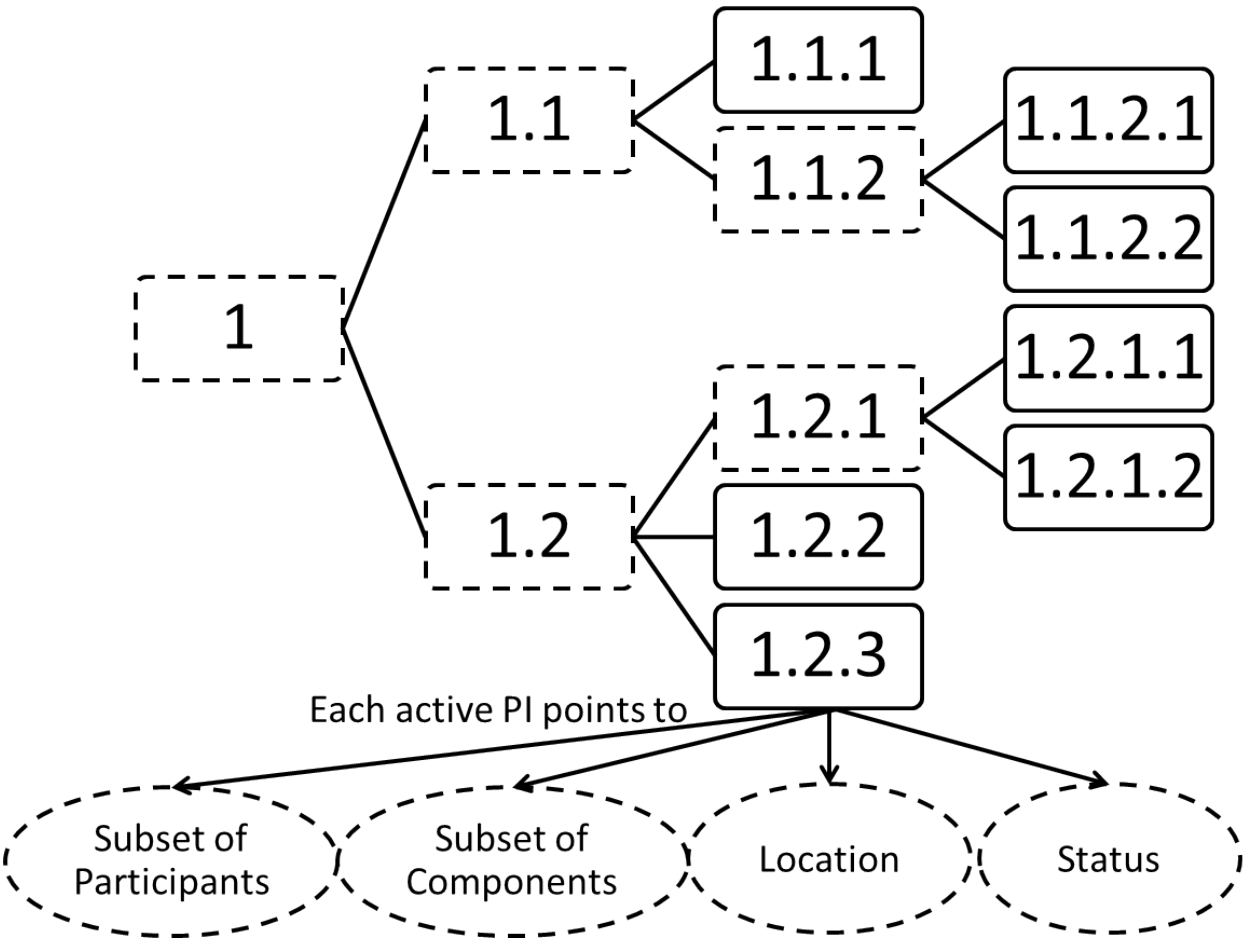

Figure 6.1: Process Instance Tree

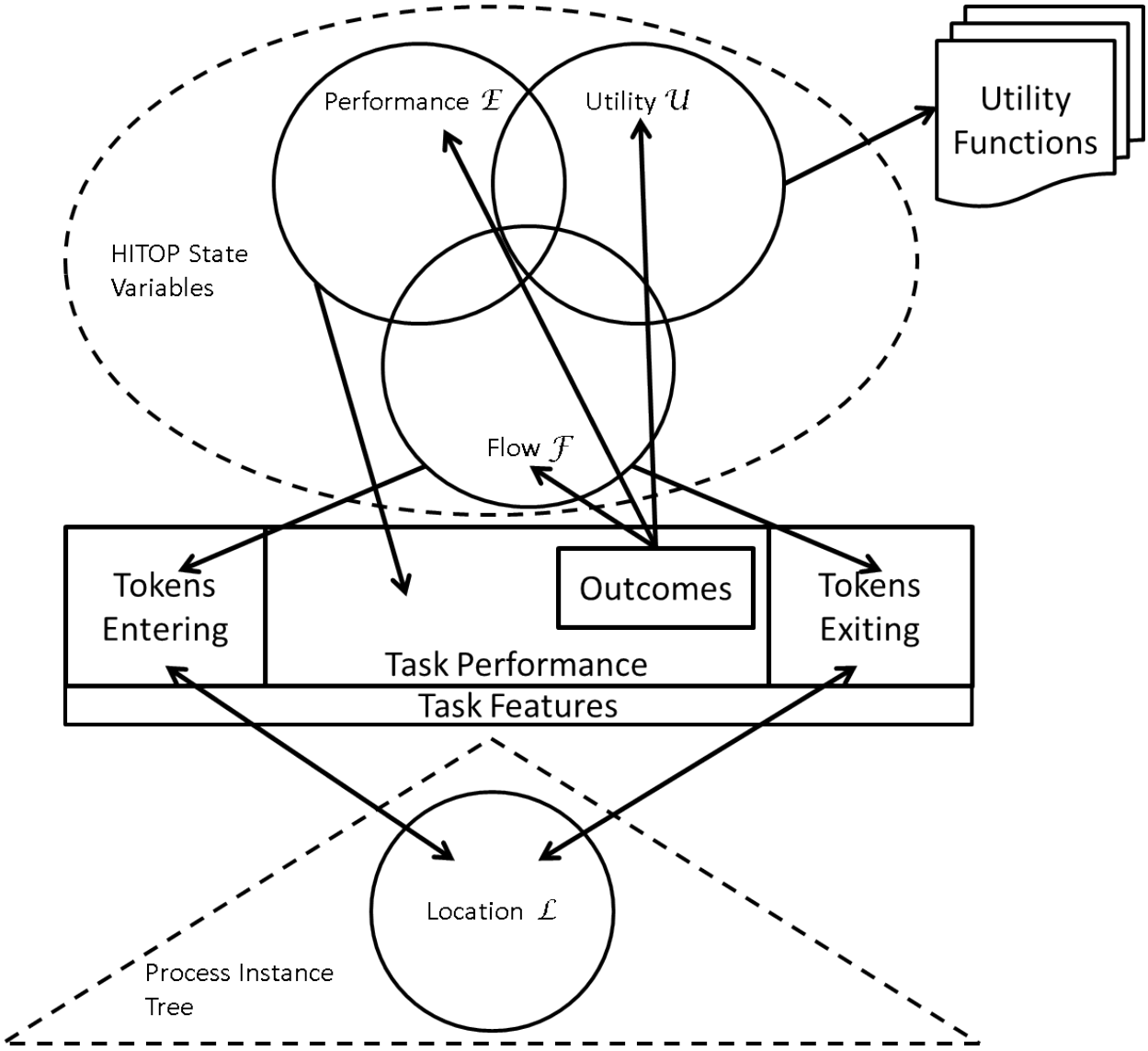

Figure 6.2: LMLU State Variables

determined solely by the set of performance variables $\mathcal{E}$, but that task performance outcomes can affect the values of flow, performance, and utility variables.

For an example of LMLU state variables, suppose that the HITOP model is constructed such that participant Bob has a variable x, component Tool has a variable y, and process Root has a variable u. Suppose that outcome 1 of task T increments y by the value of x, and if the value of y is greater than some value z, T will exit a token to exiting arc 1; otherwise T will exit a token to exiting arc 2. Suppose also that $u = y/x$ stores the value of the model state. In this example, x is a performance variable in $\mathcal{E}$ because the value of x affects the result of a task outcome; y is a flow variable in $\mathcal{F}$ because y determines which arc receives an exiting token; and the variables $\{x, y, u\}$ are all utility variables in $\mathcal{U}$, as their values are used to measure model state.

In the next sections, we will discuss the LMLU state variable classes in greater detail.

Location Variables

The set of location variables $\mathcal{L}$ captures token location information. In HITOP, token location state is stored by the process instance tree (PIT) (see Figure 6.1). The PIT indicates for each token its location, associations with components or participants, and other token status information. In the PIT, token location can be within a task or on an arc between tasks. The LMLU location state variable stores only the name of a token's associated task location. For reasons that shall become apparent later, no state changes of relevance to the LMLU solution method occur while a token is on an arc. Thus, for purposes of the LMLU model, a token's associated task location consists of a task and all entering arcs to that task. For example, if the LMLU location state of a token is task T, the token may be on an entering arc for task T or within task T.

Flow Variables

The set of *flow variables* $\mathcal{F}$ consists of the state variables that affect the token flow entering and exiting a task. LMLU token location state is determined, in part, by the values of $\mathcal{F}$, and is indicated over time by a process flow (PF) path within the process model. For example,

an XOR split will exit a single token to one of the task's exiting arcs. The arc that receives the token is determined by the state of $\mathcal{F}$. Because the underlying process flow equations for a HITOP model are deterministic functions of $\mathcal{F}$, token flow through a process model is deterministic if the values of $\mathcal{F}$ are static. We shall call models in which the values of $\mathcal{F}$ do not change *static* $\mathcal{F}$ or *SF* models. Process flow through an SF model follows a single path.

A model in which the values of $\mathcal{F}$ change during execution is called a *dynamic* $\mathcal{F}$ or *DF* model. DF models can be broken into two types: deterministic and stochastic. A deterministic DF model allows $\mathcal{F}$ values to change during execution in a deterministic way. Thus, a deterministic DF model, like an SF model, is characterized by a single PF path through the process model. We shall refer to both deterministic DF and SF models simply as *det-F* models to indicate that token flow through these types of models follows a single path.

Stochastic DF models or simply *sto-F* models change $\mathcal{F}$ values during model execution according to some random process. That results in a stochastically determined set of PF paths through a process model. In other words, a stochastic PF model $M = \{r_1, r_2, \ldots, r_n\}$ can be represented as a set n of pairs $r_i =< \omega_i, P_i >$, where n is the number of flow paths, ω_i is path i, and P_i is the probability of path i.

Performance Variables

The set of *performance variables* $\mathcal{E}$ consists of those variables whose values specify task performance time, outcomes, and the probability of outcomes. For example, if the probability of a given outcome, e.g., no opportunity or proper performance, is dependent upon $\mathcal{E}$ and the values of $\mathcal{E}$ are static, the probabilities of events within the model will not change with time or model state. We shall call a model of the type in the example a *static performance variable* or *SP* model. An SP model may further be subclassified into model types according to what task properties are static. A model in which the elements of $\mathcal{E}$ that affect task timing are static is known as a *static performance timing* or *SPT* model. A model in which the elements of $\mathcal{E}$ that affect task outcomes are static is known as a *static performance outcome* or *SPO* model. A model in which the elements of $\mathcal{E}$ that affect task outcome probability are

static is known as a *static performance probability* or *SPP* model. An SP model is also an SPT, SPO, and SPP model.

A model in which the values of $\mathcal{E}$ can change is called a *dynamic performance variable* or *DP* model. A DP model is DPT if the variables that affect performance timing can change. A DP model is DPO if the variables that affect performance outcomes can change. A DP model is DPP if the variables that affect performance outcome probability can change.

Just as with $\mathcal{F}$, the values of $\mathcal{E}$ can change in a deterministic or stochastic manner. In a *deterministic DP* or *det-P* model, at least one member of $\mathcal{E}$ is a det-P variable, i.e., a variable that can change values in a deterministic manner during model execution. In a *stochastic DP* or *sto-P* model, at least one member of $\mathcal{E}$ is a sto-P variable, i.e., a variable that can change values in a stochastic manner during model execution. For example, suppose $\lambda \in \mathcal{E}$ sets the rate of a task performance. If λ is reduced by half every time a certain task is performed (and that task is necessarily part of a det-F model), λ is a det-P variable. On the other hand, if the value of λ is set by stochastically determined outcomes, λ is a sto-P variable. DP classifications are not mutually exclusive, as a DP model may be both a det-P and a sto-P model.

Sometimes it is important to reference the performance properties of an individual task instead of an entire model. While flow entering and exiting a task is always dependent on $\mathcal{F}$, task performance, i.e., timing, outcome, and outcome probabilities, may be independent of $\mathcal{E}$ for a given task. A task with outcomes that are independent of model state is called a *state-independent* or *SI* task. If a task's timing, outcome, or outcome probabilities are individually independent of $\mathcal{E}$, we call them, respectively, *SI timing* or *SIT* tasks, *SI outcome* or *SIO* tasks, and *SI probability* or *SIP* tasks. If all tasks within a model are SI tasks, the model must by definition be a SP model.

Utility Variables

The set of *utility variables* $\mathcal{U}$ consists of the variables that measure various model properties. For example, an element of $\mathcal{U}$ might, in the simplest case, measure the number of times an event occurs, or, in a more complex case, store the value mapped by an arbitrary function

of model states. An element of $\mathcal{U}$ that measures properties related to task flow, e.g., the number of times a task is performed, is called a *flow utility variable* or *FU*. Each flow path in a process model results in a set of FU values. Thus, a deterministic PF model, i.e., a model with a single flow path, results in FUs with fixed values for each step in the flow path, and a dynamic PF model results in FUs that are random variables for each step in the flow path. A model with one or more FUs is a FU model.

An element of $\mathcal{U}$ that measures model properties related to task performance is called a *performance utility variable* or *PU*. For example, suppose a task produces outcome o_1 with a probability $Pr\{o_1\}$ determined by $\mathcal{E}$. A PU would be used to measure the number of times outcome o_1 occurs. The value of that PU would depend in this case on both $\mathcal{F}$, e.g., the number of times the task is performed, and on $\mathcal{E}$, e.g., the probability of outcome o_1 on each task attempt. Thus, the PU would represent a random process for each PF path through the process model, i.e., a random variable for each step in the flow path. A model with one or more PUs is a PU model.

As discussed at the beginning of this section, $\mathcal{U}$ can specify an arbitrary function on model states over time. For our solution method, it is important to identify a special class of utility variables known as *Markov utility variable* or *MU*. While in general $\mathcal{U}$ may make measurements dependent on both the previous model states and the time spent in those states, an MU measurement depends only on the current model state.

The most general type of LMLU model is a sto-F, sto-P, and FU/PU model. As we shall discuss further in Section 6.4.4, HDP solution methods are more tractable when the LMLU model type is restricted to a model class less complex than the general LMLU model type. For example, it is much easier to solve a det-F, SP, MU model.

The values assigned to $\mathcal{V}$, i.e., the marking of $\mathcal{V}$, represent the state of the LMLU model, as described in the next section.

LMLU State Space

LMLU state is defined by the values of the LMLU state variables. Just as described in Section 3.6.1 for basic HITOP variables, marking functions can be defined to assign values

to each LMLU state variable. LMLU state is defined as a tuple of LMLU marking functions, as defined below.

Definition 6.4.2. An *LMLU process-level state* $s = \mu_{\mathcal{V}} = (\mu_{\mathcal{L}}, \mu_{\mathcal{F}}, \mu_{\mathcal{E}}, \mu_{\mathcal{U}})$ is a tuple of marking functions for LMLU state variables, where $\mu_{\mathcal{L}}$ is a marking of token location variables; $\mu_{\mathcal{F}}$ is a marking of process flow variables; $\mu_{\mathcal{E}}$ is a marking of the performance variables; and $\mu_{\mathcal{U}}$ is a marking of the utility variables.

The state space of an LMLU model is the set of possible values that may be assigned to the LMLU state variables. The sequence in which these states may occur is discussed in Section 6.4.2.

Process-level LMLU state, i.e., state observed at the process level at which task-level details are hidden, is usually described in terms of token location and overall model state. For the LMLU model, it is often important to consider the model state from the viewpoint of a single token. Thus, we will next define the task-specific process-level state and its notation.

Definition 6.4.3. A *task-specific process-level state* $T(s)$ represents a token in task T with the model in state s. $T(s)$ is used to describe token flow through a PF graph or an SSF graph. The notation $T(*)$ represents a task-specific process-level state in which a token is in task T and the model is in some arbitrary state.

Task-specific token state is the viewpoint that will be used to describe flow through a process and the state space graphs discussed in the next section.

6.4.2 Flow Graphs

In this section, we will describe the flow graphs used for calculating the LMLU solution. They describe possible paths through the process model and its resulting state space. Thus, we must first define the process model on which the flow graphs are based and the terminology used to describe them.

LMLU Terminology

In this section, we will define the terminology used to build and describe LMLU flow graphs. First, we will define the HITOP model on which each LMLU model is based known as the *basis process model*.

Definition 6.4.4. A *basis process model* P is a HITOP process model used to define sets of possible process flow (PF) paths within a PF graph and the state space flow (SSF) paths within the corresponding SSF graph. For our purposes, in this chapter, we will specify that P is constructed so that the probability of transitioning from one state to another is dependent only on the current state and is independent of time spent within a state; e.g., one way to achieve that would be to base all event timing on exponential distributions. P is also constructed so that a single token is generated by each task performance outcome. For example, we could construct P in that way by building a flow graph that uses only XOR splits.

We use a basis MAUS utility function, defined next, to use the LMLU solution method to assess the value of a given LMLU model.

Definition 6.4.5. A *basis utility function* u is a function of model state used to assess the value of that state. We will specify that for the LMLU method, a basis utility function is constructed to be a Markov utility function, i.e., changes in its value depend only on the transition from the current state to the next state, and are independent of the time spent in the current state.

Next, because we will use the LMLU method to solve for the value of flow graphs, we must define an LMLU graph. In general, a graph is a collection of paths, so we must first define what we mean by a path in terms of process flow and state space flow.

Definition 6.4.6. A *process flow (PF) path* $F_{PF}(T, P, n)$ is an n-step (task) or less sequence of tasks beginning with a task T and ending with a task E that represents a single process iteration, i.e., a token flow path, within a portion of a basis process model. $F_{PF}(T, P, n)$ is written as $F_{PF}(T)$ if the process model and maximum number of steps are implied. If

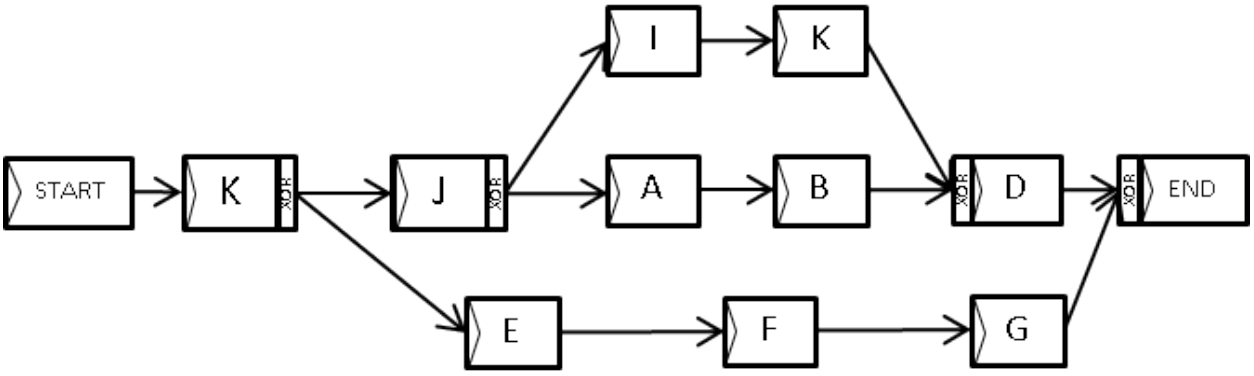

Figure 6.3: Process Model

multiple PF paths are possible through a basis process model, $F_{PFi}(T)$ is used to represent the ith path.

Definition 6.4.7. A *state space flow (SF) path* $F_{SF}(T, s, P, n)$ is an n-step (state) or less sequence of states beginning with a state $T(s)$ and ending with a task $E(*)$ that represents a single process iteration, i.e., a token flow path, within a portion of a basis process model. $F_{SF}(T, s, P, n)$ is written as $F_{SF}(T, s)$ if the process model and maximum number of steps are implied. If multiple SSF paths are possible through a basis process model, $F_{SFi}(T, s)$ is used to represent the ith path.

Definition 6.4.8. A path F' is a *subpath* of path F if each step in F' consecutively matches a step in F and both F and F' share a common end step.

Now that we have defined the basis for LMLU graph construction, we can describe the two types of graphs used by LMLU: process flow graphs and state space flow graphs.

Process Flow Graphs

Process flow (PF) graphs represent possible flow paths through the basis process model which begin from a single task and end at the end task. Consider the process model in Figure 6.3.

The PF graph $A \rightarrow B \rightarrow D \rightarrow End$ can be represented by the notation $G_{PF}(A)$. Here, we are referencing a known basis process model, so we may specify a given PF graph using only the amount of information needed to designate it uniquely. In the above example, A is

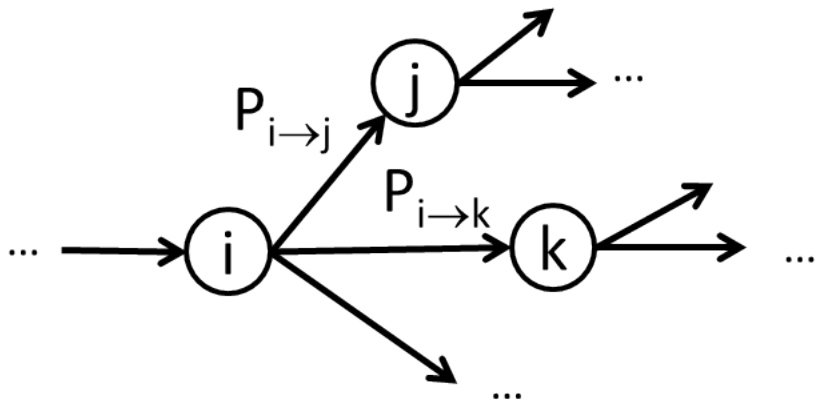

Figure 6.4: State Space Flow Graph

the only information needed to specify PF graph $A \rightarrow B \rightarrow D \rightarrow End$ uniquely, as only one PF graph exists that starts with A and ends with End. Note: by definition, all PF graphs terminate with an End task.

PF graphs are formally defined below:

Definition 6.4.9. A *process flow (PF) graph* $G_{PF}(T, P, n)$ is an acyclic directed graph containing a set of possible, n-step-maximum process flow paths within a basis process model P that begin from a single task T and end with a single end task E. $G_{PF}(T, P, n)$ is written as $G_{PF}(T)$ if the process model and maximum number of steps are implied.

The process flow graph is a tool for visualizing how a token moves through a basis process model. Note that if the basis process model has loops within the process flow, a task may appear multiple times in the PF graph. However, the n-step max restriction ensures that all PF graphs will terminate in a finite number of steps.

State Space Flow Graphs

State space flow (SSF) graphs represent possible state flow paths through the basis process model beginning with a single task in a single state, and ending with the end task in one of potentially several possible states. Because a HITOP basis process model is a discrete-event system model, the state of an LMLU model over time can be viewed as a set of discrete

states connected by directed arcs (see Figure 6.4). This figure is an example of an SSF graph. Each arc represents an event transitioning from a current state to a succeeding state. Each PF graph has a corresponding SSF graph that represents the flow of states that results as a token moves through a process. Thus, an SSF graph is a more detailed, i.e., state-level, viewpoint of a PF graph. An SSF graph is formally defined in Definition 6.4.10.

Definition 6.4.10. A *state space flow (SSF) graph* $G_{SF}(T, s, P, n)$ is an acyclic directed graph containing a set of possible, n-step-maximum state space flow paths within a basis process model P that begins with a single state $T(s)$, i.e., task T in state s, and ends with one of the possible end states for task E, $E(*)$. The set of possible end states is determined by the possible ways a token may traverse the SSF graph. $G_{SF}(T, s, P, n)$ is written as $G_{SF}(T, s)$ if the process model and maximum number of steps n are implied and $G_{SF}(T)$ if all tasks in the graph are state-independent.

Note that an SSF graph $G_{SF}(T)$ constructed from a basis process model with state-independent tasks mirrors the flow paths of a PF graph $G_{PF}(T)$ with the same basis process model.

By the assumptions made in the construction of a basis process model, we may associate a probability of transitioning from one state to a succeeding state with each arc. For example, in Figure 6.4, the probability of transitioning from state i to state j is $P_{i \to j}$. Each LMLU state is defined by the variable state $\mathcal{V}$. There may be multiple variable states for a given token state. Thus, as described above, each PF graph is associated with an SSF graph. Figure 6.5 illustrates that concept by indicating possible variable states with circles beneath the associated token locations, i.e., tasks. For example, note that while a token in the end task has only one possible location state, the associated model variables V may be in one of four states.

For understanding the LMLU solution method, it is also important to understand the concept of the subgraph. A subgraph is basically a graph contained within another graph. Subgraphs are formally defined below.

Definition 6.4.11. A graph G' is a *subgraph* of a graph G if every path in G' is a subpath of a path in G.

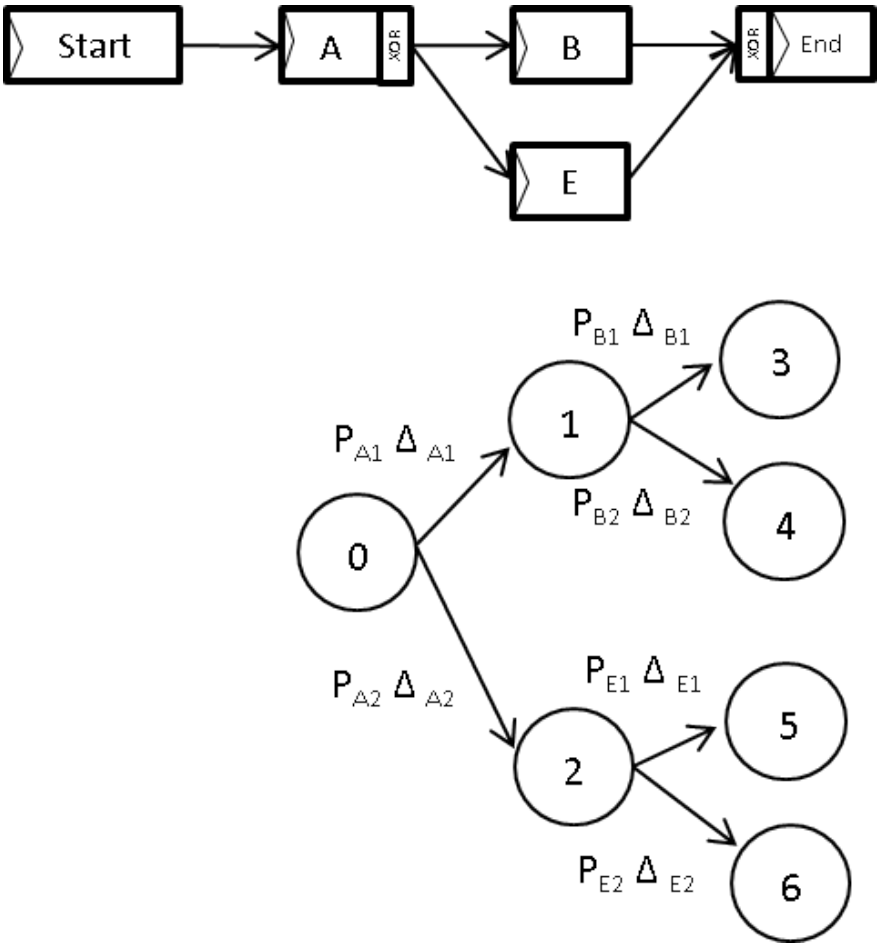

Figure 6.5: PF Graph and Associated SSF Graph

Definition 6.4.12. A graph G' is a *pf subgraph* of a pf graph G if G' is a pf graph and G' is a subgraph of G.

Definition 6.4.13. A graph G' is an *ssf subgraph* of an ssf graph G if G' is an ssf graph and G' is a subgraph of G.

6.4.3 Expected Utility Change

The LMLU solution method requires that each PF and associated SSF graph be assigned some value. One such measure of value is the *expected utility change (EUC)*. The EUC of an SSF graph is the expected change to a utility variable $u \in \mathcal{U}$ due to the traversal of a token though the corresponding SSF graph. Because the values of u at each step in the SSF graph may be represented as a random variable, we use the term "expected" to signify that we are measuring the expected value of the random variable that represents the changes to u that result from traversal of the SSF graph. The EUC to u from a token traversing $G_{SF}(T)$ is represented by $E(T)$.

Definition 6.4.14. The *expected utility change of an SSF graph* $E(T, s, u)$ is the expected change of value relative to a basis utility function u that is the result of a token traversing an SSF graph $G_{SF}(T, s)$. $E(T, s, u)$ is written as $E(T, s)$ if the utility function is implied and as $E(T)$ if the tasks in the basis process model are state-independent.

Let us illustrate the EUC with a further example. The PF graph in Figure 6.5 features two possible flow paths from task A: $A \rightarrow B \rightarrow End$ and $A \rightarrow E \rightarrow End$. $G_{PF}(A)$ represents that PF graph and $E(A)$ represents the EUC from a token traversing this graph.

Calculation of the EUC for a graph depends upon the type of task that begins the graph. An HDP graph begins with an HDP task, and a non-HDP graph begins with a task that is not an HDP. Next, we will discuss calculation of the EUC for each type of graph.

EUC of a Non-HDP Task SSF graph

In this section, we will describe how to calculate $E(T)$ for a *non-HDP task SSF graph* (or simply a non-HDP graph) $G_{SF}(T)$, which is an SSF graph that begins with a non-HDP task.

Recall that each non-HDP task by definition has one or more mutually exclusive outcomes. That is, only one of the outcomes may occur for each task performance attempt. Each outcome may change the values of the system state variables $\mathcal{V}$, and each outcome has a probability of occurrence based on the performance variables, $\mathcal{E}$. We shall use the notations δ_i to represent the change to a utility variable that is a result of outcome i and p_i to represent the probability of outcome i. We assume here that the utility variable referenced by the EUC value is an MU, i.e., the change in its value is dependent only on the current state transition. Thus, each δ_i represents the change to utility variable u that results from the state transition caused by outcome i. For an MU, the EUC of any SSF path is just the sum of the individual changes to u caused by each state transition along the path.

Also recall that process flow exiting a task is deterministic based on the values of the flow variables $\mathcal{F}$. Thus, because a token must be placed on one or more exiting arcs, each set of exiting arcs must necessarily be associated with one or more outcomes. In our example using $G_{SF}(A)$, some set of task A outcomes is associated with path 1, i.e., $A \rightarrow B \rightarrow End$, and the remaining set of outcomes must be associated with path 2, i.e., $A \rightarrow E \rightarrow End$. Because each outcome has a probability, it is obvious that each exiting arc and its associated path has a probability that is the sum of the probabilities of the outcomes associated with that path. Thus, the probability of taking path 1 is the sum of the probabilities of all the mutually exclusive outcomes that would result in a transition to task B. The idea is formally presented in Theorem 6.4.1.

Theorem 6.4.1. *The expected utility change of a non-HDP SSF graph $G_{SF}(T, s)$ is $E(T, s) = \sum_{i=1}^{n} p_i(\delta_i + E(T_i, s_i))$, where $T(s)$ has n outcomes; $P(T, s) = \{p_1, p_2, \ldots, p_n\}$ is the associated set of outcome probabilities; $\Delta(T, s) = \{\delta_1, \delta_2, \ldots, \delta_n\}$ is the associated set of outcome-related changes to the reference utility function; and $T_i(s_i)$ is the state that results from outcome i.*

Proof. Let $G_{SF}(T, s)$ be an SSF graph that begins with the state $T(s)$. Let $\Delta(T, s) = \{\delta_1, \delta_2, \ldots, \delta_n\}$ be the set of changes to the associated utility value for each of the n mutually exclusive outcomes for state $T(s)$, and let $P(T, s) = \{p_1, p_2, \ldots, p_n\}$ be the set of n probabilities, each associated with a corresponding outcome for $T(s)$.

By definition of a task, one of the possible outcomes in $\Delta(T, s)$ will occur, and a token-

arc pair will be generated that is associated with that outcome. Each token-arc pair is associated with a token location state, i.e., a marking of $\mathcal{L}$, and each outcome is associated with a variable state, i.e., a marking of $\mathcal{V}$. Thus, each outcome i is associated with a PF path from task T to task T_i and an SSF path from state s to state s_i, where the probability of moving from state $T(s)$ to state $T_i(s_i)$ is p_i. Because the outcomes in $\Delta(T, s)$ are mutually exclusive and one must occur, the sum of all p_i's is 1. Thus, the expected value of an atomic SSF graph is the sum over all outcomes of each outcome SSF path weighted by the probability of that path. Therefore, the expected value of an SSF graph that starts with $T(s)$ is $E[G_{SF}(T, s)] = E(T, s) = \sum_{i=1}^{n} p_i(\delta_i + E(T_i, s_i))$. □

EUC of an HDP Task SSF Graph

In the previous section, we discussed how to calculate the EUC for an SSF graph that begins with a non-HDP task. A similar method is used to calculate the EUC for a graph that begins with an HDP, i.e., an HDP task SSF graph or simply an HDP graph.

First, recall that in order to understand the effect of human decisions within a system, the value of the willingness probability P_W for each HDP must be calculated. The above non-HDP task EUC solution method can be modified to solve for P_W as follows. Instead of solving for the EUC that is the result of all possible outcomes associated with an HDP, we instead solve for the set of outcomes possible when P_W is 1.0 and when P_W is 0.0, and compare them to determine the preferred P_W. The preferred P_W for an HDP is the one that yields the highest EUC.

In general, to solve for the P_W of HDP T, we must set P_W to 1.0 or 0.0 and solve for $E(T)$ for each case. Whichever value of P_W results in the higher EUC is selected. The HDP P_W selection problem is stated formally in Equation 6.4.

$$\underset{P_W \in \{1,0\}}{\text{argmax}}\ E(T(P_W)) : G(T(P_W)) \rightarrow \mathbb{R}, \tag{6.4}$$

where $T(P_W)$ is an HDP with a willingness probability set to P_W.

Note that we assume that the human decisions are based on maximizing EUC and not on

some other basis. That assumption led us to the conclusion that the solution value for each HDP will always be a P_W of 1 or 0 as formalized in Theorem 6.4.2. Other assumptions, such as those made by random utility theory [17], can lead to values of $W_P \in [0, 1]$.

Given Theorem 6.4.1, it is now straightforward to solve for the maximum value for an HDP graph. Recall that for an HDP, P_W determines the probability that the task will produce the set of *willing outcomes*, i.e., the set of task outcomes that are possible if the participant is willing. We *solve* an HDP by finding the value of P_W that maximizes the EUC of the SSF graph that begins with that HDP. Note that from Definition 6.4.1, $\mathcal{E}$ is the set of performance variables for an SSF graph, and thus P_W is an element of $\mathcal{E}$.

Next, we will state our primary HDP solution theorem, which relates the maximum expected value of an HDP SSF graph to the value of P_W for that HDP.

Theorem 6.4.2. *The maximum expected value of an HDP task SSF graph $E_{MAX}(T, s)$ occurs when P_W for that HDP is set to* 0 *or* 1.

Proof. Let $G_{SF}(T, s)$ be an HDP task SSF graph that begins with the state $T(s)$. From Theorem 6.4.1, the expected value of $G_{SF}(T, s)$ is $E(T, s) = \sum_{i=1}^{n} p_i(\delta_i + E(T_i, s_i))$. The expected change to utility value for outcome i is $EUC_i = \delta_i + E(T_i, s_i)$.

Let $O(T, s) = \{o_1, o_2, \ldots, o_n\}$ be the set of n outcomes for $T(s)$ and $P(T, s) = \{p_1, p_2, \ldots, p_n\}$ be the set of n probabilities associated with the n possible outcomes for $T(s)$. One of these outcomes is the "no-willingness" outcome; call it outcome j. Let EUC_j be the expected change in utility associated with outcome j and $p_j = 1 - P_W$ be the associated probability of outcome j. Because the outcomes in a task are mutually exclusive, they may be partitioned into two sets of events: the willingness (W) and no-willingness (NW) events. The NW event includes just the outcome j, while the W event contains the remaining $n - 1$ possible outcomes. Each p_i in $P(T, s)$ may be rewritten in terms of the W and NW events as

$$p_i = Pr(o_i \mid W)Pr(W) + Pr(o_i \mid NW)Pr(NW), \tag{6.5}$$

where $Pr(o_i \mid W)$ is the probability that outcome i occurs given the W event occurs; $Pr(W)$ is the probability that event W occurs; $Pr(o_i \mid NW)$ is the probability that outcome i occurs

given the NW event occurs; and $Pr(NW)$ is the probability that event NW occurs. Because the outcomes are mutually exclusive, $Pr(o_i \mid NW) = 0$ for all outcomes $i \neq j$, $Pr(o_j \mid NW) = 1$, and $Pr(o_j \mid W) = 0$.

Thus, $p_i = p'_i \, P_W$ for all $i \neq j$ where $p'_i = Pr(o_i \mid W)$ and $p_j = 1\text{-} \, P_W$. The expected value of the HDP task SSF graph may be rewritten as

$$E(T,s) = \sum_{i \neq j} p'_i EUC_i(P_W) + EUC_j(1 - P_W) = E(T,s)_W P_W + E(T,s)_{NW}(1 - P_W), \quad (6.6)$$

where $E(T,s)_W$ is the expected value of the HDP task SSF graph if the participant is willing, and $E(T,s)_{NW}$ is the expected value of the HDP task SSF graph if the participant is not willing.

Mathematically, it is obvious that since either $E(T,s)_W$ or $E(T,s)_{NW}$ will be the larger value, $E_{MAX}(T,s)$ is achieved by setting P_W to either 1.0 or 0.0. □

Note that if $E(T,s)_W$ and $E(T,s)_{NW}$ are equal, the participant is assumed to have no preference as to the decision. Thus, a P_W value of either 1 or 0 may be used to maximize the expected value of the HDP SSF graph.

For example, if task A in Figure 6.5 is an HDP, the participant will choose an outcome and associated path that provide the highest EUC.

As before, we shall assume that outcome 1 occurs with probability p_1 and results in the selection of path 1, and that outcome 2 occurs with probability p_2 and results in the selection of path 2. Path 1 has a value of $\delta_1 + E(B)$, and path 2 has a value of $\delta_2 + E(E)$.

Since one path will always have the higher EUC (we assume no ties for now), the participant is assumed always to choose the path with the highest EUC. Thus, either P_1 or P_2 will be set to 1.0. If P_1 represents the P_W for HDP A, then the optimal value for P_W will be either 1.0 or 0.0. Thus, the problem of solving for P_W in this example is reduced to determining which path has the higher EUC.

Average-State HDP Willingness Probability

Up to this point, we have solved HDPs as either state-independent tasks, or as state-dependent tasks and an associated state. There is a third way to solve for HDP values known as solving for the *average state HDP willingness probability*.

Definition 6.4.15. The *average-state HDP willingness probability* P_{Wave} is the average value of willingness probability for an HDP state averaged over all possible HDP states, weighted by the likelihood of that state.

The average-state HDP willingness probability represents the probability that a participant will make a decision, averaged over all states in which the participant can make that decision. Another way to say this is that, if P_W is a random variable that maps a given HDP state $T(s)$ to a willingness probability, then $P_{Wave} = E[P_W]$ is the expected value of the random variable P_W.

Suppose an HDP occurs multiple times within a PF graph. Each time a token enters an occurrence of that HDP, the model may be in a different state. In the previous section, we handled such cases by assuming that each HDP/state pair should be solved individually. However, it is often a more natural viewpoint to consider the *average* decision represented by the set of HDP/state pairs (vice the set of state-specific decisions themselves).

For example, suppose a system user must decide many times a day whether to comply with a company's Web browsing policy. Rather than model the human decision-maker in a given model state for *each* decision opportunity, it may instead be advantageous in terms of solution efficiency to model the *average* human decision made for a set of decision opportunities over time. Suppose in our example that violating the company Web browsing policy x number of times gives the user a positive EUC, but that more numerous violations result in a negative EUC. If, over the course of time, the user has 100 opportunities to violate the policy, i.e., the HDP modeling the user's decision to comply with policy occurs 100 times in the PF graph, then (assuming $x < 100$) we would expect the user, on average, to violate policy x out of 100 times (all other things being equal). Those decisions can be modeled as 100 different HDP/state pairs (which leads to a very large solution state space as discussed in Section 6.4.5), or as a single HDP with an average P_W of $x/100$.

Using an average value of P_W is an example of an average-state HDP willingness probability solution. In practical terms, an average-state HDP willingness probability solution approach is often useful, because empirical data gathered to represent such decision probabilities are typically data aggregating the decisions of many different people made over many different times. That is, decision data for a single person and a single system state are not easily gathered. Thus, a model viewpoint in which average-state HDP willingness probability solutions are used may be a more practical method for using empirical data to characterize a system model or to validate the results of the LMLU solution method.

It is possible to solve for an average-state HDP willingness probability by solving for each HDP/state pair using the LMLU method and averaging the results according to their probability of occurrence, i.e., calculating an expected value for P_W, or by solving a lumped-state HDP model similar to those discussed in Section 6.4.5. Note that a lumped-state model can only be used to solve for HDP/state pairs that occur concurrently in the PF graph, i.e., the same task occurs in different states during the same step in the PF graph. If it is desirable to solve for the average-state HDP willingness probability when the HDP occurs during multiple steps in the PF graph, a method such as traditional optimization may be more efficient. Note that, unlike state-specific HDPs which have values in $\{0, 1\}$, average-state HDPs may have values in $[0, 1]$, as they represent an expected value based on many cases in which HDP solutions are 1 or 0.

In the next section, we apply the non-HDP and HDP graph solution methods to define a general SSF graph solution method.

EUC of a General Task Graph

A *general task SSF graph* or *general graph* is a graph that begins with a non-HDP or HDP task. Using Theorem 6.4.1 and Theorem 6.4.2, we can solve for the EUC of a general graph in a recursive fashion. As the graph is solved, each HDP P_W value must be stored for use in later system solutions. Solving for both the EUC of an SSF graph and its corresponding set of P_W values is known as *solving* an SSF graph. The procedure is formalized through the use of the below algorithms for both state-independent and state-dependent task graphs.

Theorem 6.4.3. *The maximum expected utility change of a general SSF graph $E_{MAX}(T, s)$ occurs when its HDP task SSF subgraphs are solved (i.e., a value of P_W is selected to maximize the expected value of each HDP task SSF subgraph), in the reverse of the order in which they appear in the graph.*

Proof. A general SSF graph may begin with either an HDP or a non-HDP task-specific state $T(s)$.

Case 1. Let $G_{SF}(T, s)$ be an SSF graph beginning with the non-HDP task state $T(s)$ and containing no HDP task states. Because the outcome probabilities of the task states in the graph are fixed, i.e., there are no HDP P_W values to change, the expected value of $G_{SF}(T, s)$ is fixed, i.e., $E_{MAX}(T, s) = E(T, s)$.

Case 2. Let $G_{SF}(T, s)$ be an SSF graph beginning with the non-HDP task state $T(s)$ and containing at least one HDP task state. From Theorem 6.4.1, the expected value of $G_{SF}(T, s)$ is $E(T, s) = \sum_{i=1}^{n} p_i(\delta_i + E(T_i, s_i))$. The expected change to the utility value for outcome i is $EUC_i = \delta_i + E(T_i, s_i)$.

Because the values of p_i are fixed for a given non-HDP task and state, we maximize the value of $E(T, s)$ by maximizing the value of EUC_i for each outcome i. Because the value of δ_i is fixed for each state $T(s)$, we maximize EUC_i by maximizing the expected value of each SSF graph associated with an outcome i, $E(T_i, s_i)$. Note that if T_i is the end task, then $E(T_i, s_i)$ is defined to be zero. Thus, we maximize the expected value of a non-HDP SSF graph by maximizing the expected values of all of its child SSF subgraphs, i.e., $E_{MAX}(T, s) = \sum_{i=1}^{n} p_i(\delta_i + E_{MAX}(T_i, s_i))$.

Case 3. Let $G_{SF}(T, s)$ be an SSF graph beginning with an HDP task state $T(s)$. From Theorem 6.4.3, the expected value of $G_{SF}(T, s)$ is $E(T, s) = E(T, s)_W \ P_W + E(T, s)_{NW}$ (1- P_W), and we maximize $E(T, s)$, i.e., we solve the HDP graph, by setting P_W such that the greater value of $E(T, s)_W$ and $E(T, s)_{NW}$ is selected. Thus, we solve $G_{SF}(T, s)$ by selecting between the maximized values of $E(T, s)_W$ and $E(T, s)_{NW}$, i.e., $E_{MAX}(T, s) = E_{MAX}(T, s)_W(P_W) + E_{MAX}(T, s)_{NW}$ (1- P_W). $E(T, s)_W$ and $E(T, s)_{NW}$ are both non-HDP SSF graphs and may be maximized, as in Case 2.

All SSF graphs may be grouped according to case 1, case 2, or case 3. Thus, maximization

of the EUC of a general task SSF graph always requires maximizing the EUC of its child SSF graphs, and, if a child graph is an HDP graph, the maximizing P_W value for the HDP graph cannot be selected until the EUCs for all the child graphs of the HDP graph have been maximized. Therefore, the EUC of an HDP graph cannot be maximized until all of its descendant HDP subgraphs have been maximized, i.e., solved. Therefore, since the first HDP subgraph that can be solved is the last HDP subgraph to appear in the graph, i.e., it is a graph that has no HDP subgraph descendants, the set of HDP subgraphs in an HDP graph must be solved in reverse order of their appearance in the graph.

Therefore, the maximum EUC of a general SSF graph is the value achieved by maximizing the HDP subgraphs in reverse order of their appearance in the graph. □

Now that we have proved the basis of the LMLU solution method, we will describe it in more detail in the next section as a series of algorithms.

6.4.4 LMLU Algorithms

In this section, we will specify the algorithms needed to solve both state-independent and state-dependent task SSF graphs.

State-Independent Task Graph LMLU Algorithms

State-independent or SI tasks have performance outcomes that do not depend on LMLU model state. The below set of functions, field definitions, and algorithms may be used to solve an SI task SSF graph, i.e., find the graph's maximum EUC value and the set of HDP P_W values which will maximum the graph's EUC as in Theorem 6.4.3.

First, we will list the following *functions* used by the LMLU algorithms:

- E(T) is described in Algorithm 13 which solves a general SSF graph $G_{SF}(T)$.
- EH(T) is described in Algorithm 14 which solves an HDP SSF graph $G_{SF}(T)$.
- ET(T) is described in Algorithm 15 which solves a non-HDP SSF graph $G_{SF}(T)$.
- $\delta(T, i)$ returns the expected utility change of outcome i of task T.

- NextTask(T, i) returns the next task in the PF following outcome i of task T.

Next, we shall use the following notation to identify select *fields* of task T abstracted from the HITOP set notation described in Chapter 3 where each field represents a value associated with a task These fields are referenced by the solution algorithms to access various task property values and solution values.

- $T.p$: willingness probability (WP) for an HDP task T.
- T.type: task T task type.
- $T.n$: number of outcomes for task T.
- $T.i$: outcome i of task T.
- $T.i.p$: probability of outcome i for task T.
- $T.v$: value of PF flow graph starting with task T.

Next, we define three basic algorithms that are used to solve a TI SSF graph recursively. Algorithm 13 solves for the EUC of a general graph, i.e., an SSF graph that begins with a general task. Algorithm 14 solves an HDP graph and Algorithm 15 solves a non-HDP graph. To solve an SSF graph, we execute Algorithm 13 using Start, the start task of the basis process model, as input, and recursively call either Algorithm 14 or Algorithm 15 based on the type of task encountered while the graph is being traversed.

Algorithm 13 Find E(T), the Expected Utility Change for a General SI Task SSF Graph

```
1: TYPE ← T.type
2: if TYPE == HDP then
3:    return EH(T) {Return EUC of HDP graph}
4: else if TYPE ≠ HDP AND TYPE ≠ End then
5:    return ET(T) {Return EUC of non-HDP graph}
6: else
7:    return 0 {if End task, return zero }
8: end if
```

Algorithm 14 Find EH(T), the Expected Utility Change for an HDP SI Task SSF Graph

$T.p \leftarrow 1$ {Set HDP P_W to 1}
$E_{TP1} \leftarrow ET(T)$ {EUC with $P_W = 1$}
$T.p \leftarrow 0$ {Set HDP P_W to 0}
$E_{TP0} \leftarrow ET(T)$ {EUC with $P_W = 0$}
if $E_{TP1} \leq E_{TP0}$ **then**
    $T.p \leftarrow 1$ {Set $P_W = 1$ if highest EUC}
    restore HDP SSF graph solution values for $P_W = 1$
    return E_{TP1} {Return $E(T)$ with $P_W = 1$ if highest EUC}
else
    return E_{TP0} {Else return $E(T)$ with $P_W = 0$}
end if

Algorithm 15 Find ET(T), the Expected Utility Change for a non-HDP SI Task SSF Graph

if T on Solved_Graph_List **then**
    $E_T \leftarrow T.v$ {If graph already solved, assign stored value}
else
    $n \leftarrow T.n$ {Number of outcomes}
    $E_T \leftarrow 0$ {Initialize EUC to zero}
    for $i = 1$ to n **do**
        $E_T \leftarrow E_T + T.i.p \times(\Delta(T,i) + \mathrm{E}(\mathrm{NextTask}(T, i))$ {Sum the EUCs for each outcome}
    end for
    Solved_Graph_List $\leftarrow T$ {Enter T in stored solution list}
    $T.v \leftarrow E_T$ {Store the solution value}
end if
return E_T

State-Dependent Task LMLU Algorithms

In the previous section, we made the simplifying assumption that all task outcomes are independent of state. We will now consider how LMLU algorithms may be extended to cases in which task outcomes are state-dependent.

As described in Section 6.4.1, a state-dependent task is a task within a DP model whose performance outcomes and outcome probabilities depend on the set of performance variables $\mathcal{E}$ and whose state is defined by the values of $\mathcal{V}$. Note that we explicitly assume that the basis process model is a sto-F SPT DPO DPP MU model, i.e., flow paths are stochastically determined by task outcomes; task timing is not affected by task outcomes; task outcomes and outcome probabilities are affected by task outcomes; and utility values are not affected by time spent in a particular state, but are affected by state transitions.

Because task outcomes are state-dependent, we will extend the TI task functions used in the previous section to include the model state s as explained below:

- ED(T, s) calls state-dependent Algorithm 16 to calculate the EUC of a general graph beginning in state $T(s)$.
- EDH(T, s) calls state-dependent Algorithm 17 to calculate the EUC of an HDP graph beginning in state $T(s)$.
- EDT(T, s) calls state-dependent Algorithm 18 to calculate the EUC of a non-HDP graph beginning in state $T(s)$.
- $\delta(T, i, s)$ returns the utility change due to outcome i of task T in state s.
- NextTask(T, i, s) returns the next task in the PF following outcome i of task T in state s.
- NextState(T, i, s) returns the next state in the SSF following outcome i of task T in state s.

Next, we shall extend the TI task field notation to account for state using the notation below:

- $T.p.s$: willingness probability (WP) for an HDP task in state $T(s)$.
- $T.i.s$: outcome i of state $T(s)$.
- $T.i.p.s$: probability of outcome i in state $T(s)$.
- $T.v.s$: value of SSF flow graph starting with state $T(s)$.
- $T.s$: represents state $T(s)$.

Given those functions and field notations, the state-dependent solution algorithms are defined below:

Algorithm 16 Find ED(X,s), the Expected Utility Change of a General SD Task Graph

```
TYPE ← T.type
if TYPE == HDP then
    return EDH(T, s) {Return EUC of HDP graph}
else if TYPE ≠ HDP OR TYPE ≠ End then
    return EDT(T, s) {Return EUC of non-HDP graph}
else
    return 0 {if End task, return zero }
end if
```

Algorithm 17 Find EDH(X,s), the Expected Utility Change of an HDP SD Task Graph

```
T.p.s ← 1 {Set HDP P_W to 1}
ETP1← EDT(T, s) {EUC of T with P_W = 1}
T.p ← 0 {Set HDP P_W to 0}
E_TP0 ← ET(T, s) {EUC of T with P_W = 0}
if E_TP1 ≤ E_TP0 then
    T.p.s ← 1 {Set P_W = 1 if highest EUC}
    restore HDP PF graph settings for P_W=1
    return E_TP1{Return E(T, s) with P_W = 1 if highest EUC}
else
    return E_TP0{Else return E(T, s) with P_W = 0}
end if
```

The state-dependent algorithms can require significantly greater computational effort than the SI algorithms, because for the SD task basis model, each HDP must be solved for each state it can be in. We will discuss LMLU performance in greater detail in the next section.

Algorithm 18 Find EDT(T,s), the Expected Utility Change of a Non-HDP SD Task Graph

1: **if** $T.s$ on Solved_Graph_List **then**
2: $E_T \leftarrow T.v.s$ {If graph already solved, assign stored value}
3: **else**
4: $n \leftarrow T.n$ {Number of outcomes}
5: $E_T \leftarrow 0$ {Initialize EUC to zero}
6: **for** $i = 1$ to n **do**
7: $E_T \leftarrow E_T + T.i.p.s \times(\Delta(T,i,s) + \text{E}(\text{NextTask}(T, i, s), \text{NextState}(T, i, s))$ {Sum the EUCs for each outcome}
8: **end for**
9: Solved_Graph_List $\leftarrow T.s$ {Enter $T(s)$ in stored solution list}
10: $T.v.s \leftarrow E_T$ {Store the solution value}
11: **end if**
12: **return** E_X

6.4.5 Performance Analysis of LMLU Solution Method

The performance of the LMLU algorithm depends on both the type of basis process model and the depth of the associated PF graph to be analyzed.

In general, the LMLU algorithm's performance is linear with the size of the SSF graph. The size of the SSF graph is dependent upon the type of process model on which it is based. For example, a state-independent (SI) basis process model results in a much smaller state space than a state-dependent (SD) basis process model.

First we will consider the performance of the most general case, the SD LMLU algorithms which are applied to SD basis process models. In that case, tasks within the basis process model are dependent on model state. That is, both the outcomes and the probability of each outcome are dependent on model state, and task outcomes can change model state. Thus, each task in the PF graph increases the number of potential model states by increasing the number of task outcomes in the PF graph overall. This means that the state space of a SD LMLU solution is exponential in depth and outcome.

For example, if state $T(s)$ has n outcomes, the set of potential new states generated by performance of task T is $S' = \{T_i(s_i), i = 1 \text{ to } n\}$. Each task is a step in a PF graph; therefore, the possible state space of an SSF graph grows at a rate that is on the order of $\mathcal{O}(n^d)$, where n is the number of outcomes for a task, and d is the depth of the PF graph. That implies that even a small 20-step PF graph with 4 outcomes for each task can potentially have a

relatively large state space of $4^{20} = 2^{40} \approx 10^{12}$ states.

Since the SD LMLU method solves for each distinct HDP/state combination, the number of HDP graph solutions can become very large for an HDP deep within a PF graph. From our previous example, an HDP 10 steps into the PF graph could potentially have 4^{10} states and thus require that $4^{10} \approx 10^6$ HDP graph solutions be calculated to solve that HDP.

Several strategies may be used to improve the efficiency of the SD LMLU algorithm. One strategy is to trade off computation with memory usage by storing each unique SSF graph solution. That can be effective if the SSF graph is structured so that many paths lead to overlapping values of $T(s)$. The strategy has been used in the LMLU algorithms specified in Section 6.4.4.

Another strategy that can reduce computational effort involves realization that the value of a particular PF graph depends only on the change to utility that results from traversal of the graph. If the utility variable under which the PF graph is evaluated is a Markov utility variable (as we have assumed for our basis process model), then it may be possible to "lump" certain states with common outcomes together and perform a single evaluation that applies to all of the lumped states. An example of a case in which lumping may be performed is an SPO model whose outcomes are not dependent on model state. Thus, only the state-dependent probabilities for each outcome need be calculated for each unique state. Similarly, an SPP model requires the calculation only of outcomes and not outcome probabilities.

If the tasks in a basis process model are all state-independent, the lumping-of-state strategy is the most efficient. Essentially, the SSF graph collapses into a structure identical to the PF graph, and only unique PF graphs need to be evaluated. Figure 6.6 provides an example of such a PF/SSF graph, in which the resulting states from task outcomes can be lumped into a composite state for the purposes of evaluating the SSF graph. In the figure, each hashed oval represents such a lumped state. The labels indicate how the states are lumped together, e.g., lumped state 3−6 represents states 3, 4, 5, and 6. In the case of lumped states for SI task graphs, the calculation effort is reduced to the order of $\mathcal{O}(\mid T \rightarrow T_i \mid)$, where $\mid T \rightarrow T_i \mid$ is the number of unique task-to-task transitions in the PF graph. The SI LMLU algorithms can be used to solve models that have SI tasks.

As an example of an SI LMLU solution, consider the basis process model in Figure 6.3.

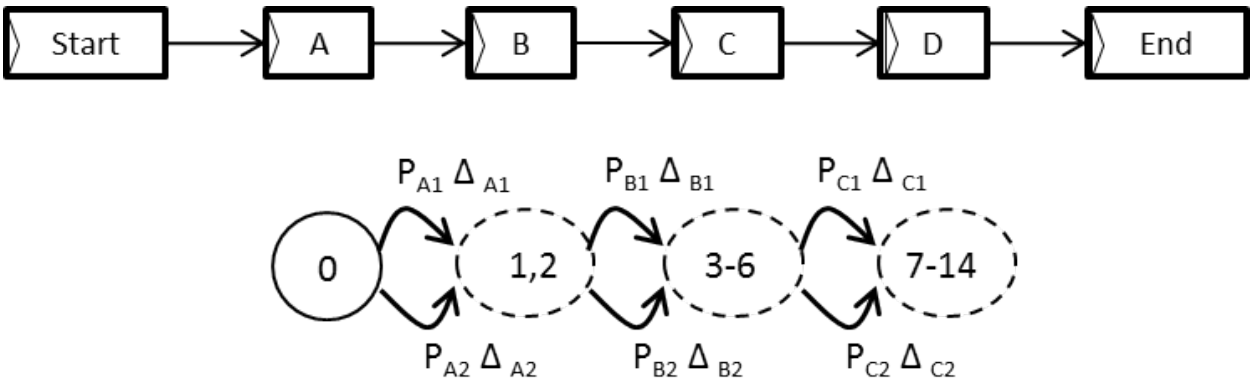

Figure 6.6: SI Task PF/SSF Graph with Lumped States

Suppose that all the tasks in that model are state-independent and that tasks J and F are HDPs to be solved. Also suppose that start and end tasks have no effect on the utility function. The problem of solving the associated PF graph for this model comes down to evaluating the value of each unique transition from one task to another. For example, solving for HDP F involves solving for the EUC of the graph $F \rightarrow G \rightarrow End$, i.e., $E(F)$, when P_W for F is 0 and when P_W for F is 1. Because G is an SI task, the EUC of transitioning from G to End need be calculated only once for the P_W comparison. Thus, the total number of EUC evaluations needed to solve the HDPs for the associated PF graph is $n = 2 \times$(number of unique HDP transitions) + (number of unique non-HDP transitions), which is on the order of $\mathcal{O}(\mid T \rightarrow T_i \mid)$.

6.5 A Modified Optimization Approach

In the previous section, we discussed the LMLU solution approach. It can be applied only to a certain restricted class of HITOP basis process models. However, it is possible to apply insights gained from the LMLU solution model to the more traditional optimization methods discussed in Section 6.3. Recall that the search space for an optimization method is defined by the feasible set $\mathbb{A}$. In general, $\mathbb{A} = [0, 1]$ for each HDP to be solved; however, given the conditions of the LMLU solution, the feasible set may be reduced to $\mathbb{A} = \{0, 1\}$ for each HDP. Recall that that was proven using the assumption that human decisions are based upon the maximization of a utility function. Thus, the search space even for traditional

optimization methods may be greatly reduced under the assumed LMLU conditions. For an SI process model, that means that it is necessary to evaluate only $2^{|HDP|}$ possible solutions, where $|HDP|$ is the number of HDPs in the model. Additionally, the LMLU restriction on models that use only single-token task outputs can be removed when we use an optimization solution method. An optimization technique enables the solution of a more general class of models, such as those using general AND and OR joins and splits.

For an SD process model, $|HDP|$ is instead the set of unique HDP/state combinations. $|HDP|$ (as previously discussed) can be considerably larger for an SD process model, depending upon the structure and depth of the associated SSF graph. Additionally, the set of HDP/state pairs to be evaluated must first be calculated, which may make such an approach less efficient than a traditional optimization approach for SD task models.

For both SI and SD process models, a discrete feasible set removes many of the problems involved with stopping criteria and the comparison of solutions required by direct search methods, and may, depending upon the model, lead to a tractable HDP solution method.

6.6 Conclusion

In this chapter, we have formally stated the MAUS HDP solution problem, provided some general background on optimization methods, described in detail the LMLU HDP solution method (including applicable model classes, algorithms, and theorems), and discussed the general efficiency of the LMLU algorithm and how insights gained from the LMLU approach may be applied to make general optimization approaches more efficient for certain classes of process models.

CHAPTER 7

CASE STUDIES

7.1 Introduction

In Chapter 2, we defined the conceptual model for studying a cyber-human system (CHS). In Chapter 3, we specified the HITOP mathematical formalism that can be used to build the CHS conceptual model. In Chapter 4, we developed the execution algorithms that can be used to implement HITOP. In Chapter 5, we defined MAUS, the solution framework in which an executable HITOP model can be used to characterize the human decision points (HDPs) within a CHS. In Chapter 6, we explored several solution methods that can be used within MAUS. In this chapter, we will apply a solution method to several case studies to demonstrate the usefulness of the HITOP model and MAUS.

We begin by describing the tool used to perform those case studies, the HITOP atomic formalism.

7.2 The HITOP Atomic Formalism

In order to perform case studies, we first had to construct an executable version of HITOP. Thus, we developed and implemented the executable HITOP atomic formalism as an atomic model formalism within the Möbius modeling framework. The formalism allowed us to construct executable HITOP models, and then use the extensive set of analytical and simulation tools available in Möbius to perform experiments using those models. We implemented the associated MAUS for each HITOP model using Möbius reward, study, and solution modules [60].

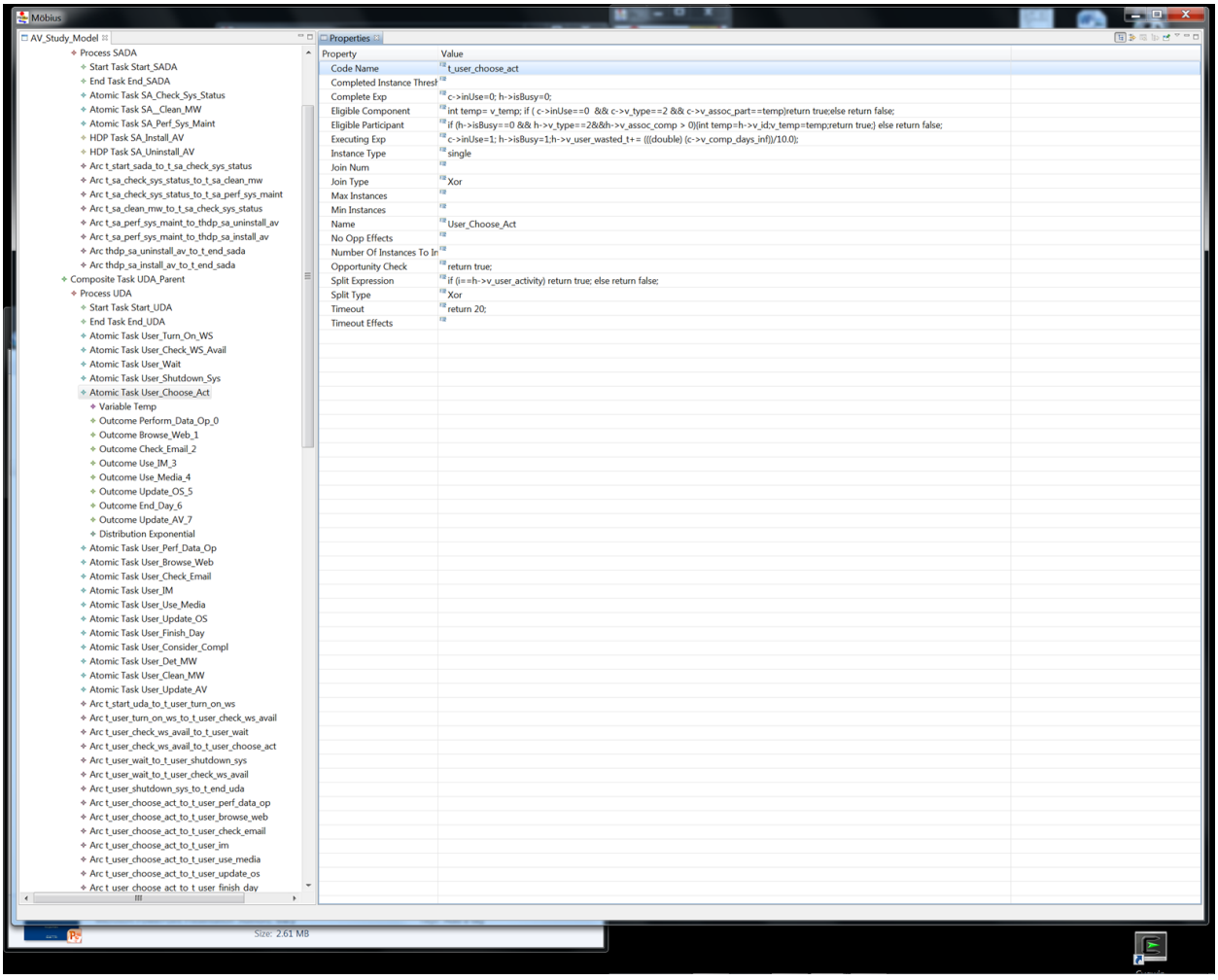

Figure 7.1: HITOP Atomic Model Editor

7.2.1 Implementation Details

The HITOP atomic formalism was defined using a graphics-based, XML-like format and an associated editor (see Figure 7.1). To create the model definition files, we used that format to define processes, participants, components, and tasks, along with their associated properties, variables, and functions (as defined in Chapter 3). We then used JAVA code and the model definition files to build C++ code, and we compiled and linked the C++ code to form an executable model which interfaced with the Möbius abstract functional interface (AFI).

7.2.2 Additions to Möbius

In order to implement a HITOP atomic formalism, we had to add several previously unavailable features to the set of Möbius capabilities. The first was the ability to handle *colored tokens*. HITOP is designed to simulate multiple process instance tokens moving through a process model simultaneously. Thus, each process instance token must have a color to distinguish it from other process instance tokens. In order to implement that feature in the HITOP atomic formalism, we created a colored token modeling structure.

The second capability added to Möbius was the *bag* structure. The bag structure is used to hold a multiset of objects. Because HITOP uses colored tokens, a bag was needed in order to hold multisets of colored tokens within the model. Thus, we created the bag modeling structure, and used it to implement the internal structure of tasks.

The third capability added to Möbius was the *parallel action* (PA). Typically, actions within the Möbius AFI are enabled and fired sequentially. However, in a HITOP model, many colored tokens share the same model structure, and thus several tokens can enable the same action in the model structure in parallel, i.e., at the same instant of time. Thus, to implement the HITOP atomic formalism, we added to Möbius the capability to enable multiple actions simultaneously from a single action structure. That capability may prove to be useful in future formalisms as well, as it allows a single model structure to be used at run-time by many different tokens. Before the PA was incorporated, a modeler needed to know the maximum number of processes to be simulated before run-time, and a corresponding

model structure needed to be built for each one.

With the PA, a single model structure can be used independently and simultaneously by multiple processes, the number of which can vary while the model is running. We call that capability *structural re-use* because every process may independently re-use a PA, even when it is currently in use by, i.e., has been activated by, another process. In addition to its current application in HITOP, the ability to simulate ad hoc parallel processes may prove useful in many other areas. For example, the PA structure can be used to model ad hoc process systems in which multiple virtual machines can be created or removed during system operation.

7.3 AV Case Study

In this section, we will describe a case study of a university's anti-virus (AV) installation policy and practice. A cyber-human system (CHS) model of the system was built using the HITOP formalism and analyzed using MAUS. The model was implemented using the HITOP atomic formalism and Möbius framework. To conduct this case study, we defined the CHS to be studied, collected data about the system, built a representative HITOP model, characterized the model for its HDP values, and then gathered experimental data for several configurations.

7.3.1 Case Study Scope

The CHS studied for this case study was a large university network, its system administrators, and its users. Specifically, we wanted to study how the decisions to install and uninstall anti-virus software would affect the security of the network, specifically malware (MW) infection rates.

It was university policy that anti-virus software should be installed on every university system; however, it was known that SAs sometimes either chose not to install anti-virus software, or chose to uninstall anti-virus software when it was installed. The purpose of this case study was to model the CHS relevant to those decisions in order to better understand

1) why system administrators might choose to install or uninstall anti-virus software, 2) how important this decision might be to overall system security metrics, and 3) what factors might be changed to influence greater compliance with the university anti-virus installation policy.

7.3.2 Data Collection

To gather data for the model, we conducted interviews with university network security staff and system administrators. Additionally, we conducted a university-wide survey of system administrators, and reviewed cyber-security-relevant university documents, such as the university's information security policy. Results of the data collection were used to determine system definition and configuration data, as well as to construct the utility functions for system administrators and the university cyber-security organization.

Using the collected data and our CHS case-study scope, we constructed a HITOP model called the *Anti-Virus Study Model*, described next.

7.3.3 Model Construction

A HITOP model called the *Anti-Virus Study Model* was constructed using the HITOP atomic formalism. In the model, we defined sets of participants, components, processes, and tasks, as described next.

Participants

Two types of participants were defined for the model: users and system administrators. A user represented a generic university staff, student, or faculty workstation (WS) user. Variables associated with the user characterized things like user training level and the amount of time wasted when using a WS for user daily activities. The user variables were assigned values as follows:

- training $\in [0, 1]$ represented the amount of security training the user had received, where 1.0 is the highest possible training level and 0.0 is no training; and

- wasted time $\in \mathbb{N}$ represented the amount of time wasted by the user during the daily use of an associated WS.

A system administrator represented the basic group-level system administrator whose job was to maintain the network and individual WSs for all the group's users. The system administrator could make decisions about whether or not to install or uninstall anti-virus software on the entire network. Variables associated with the system administrator characterized things like experience level, amount of extra work required, number of complaints from users, and positive and negative experiences with anti-virus software. The user variables were assigned values as follows:

- experience level $\in [0, 1]$ represented the experience level of the system administrator, where 1.0 is the highest level of experience and 0.0 is the lowest level of experience;
- extra work $\in \mathbb{N}$ represented the amount of extra work, above and beyond the system administrator's standard job duties, that was required because of malware infections and anti-virus software usage;
- complaints $\in \mathbb{N}$ represented the number of complaints received from system users because of wasted user time or system performance;
- positive AV experiences $\in \mathbb{N}$ represented the number of positive experiences related to anti-virus software use; and
- negative AV experiences $\in \mathbb{N}$ represented the number of negative experiences related to anti-virus software use.

Note that negative experiences with anti-virus software could occur because of events such as false positives or decreased system performance because of anti-virus installation. Positive anti-virus experiences could occur due to events such as detection of malware by anti-virus software, i.e., a true positive, or removal of malware with anti-virus software by users acting on their own.

The initial Anti-Virus Study model specified a single type of user and a single type of system administrator. Note that the Anti-Virus Study model could be expanded to characterize different user and system administrator types with different behaviors and utility functions.

Components

Three types of components were defined for the model: workstations, servers, and the system. A *workstation (WS)* component represented the computer used by the user to perform daily activities. Variables associated with a WS characterized the amounts of various data types, performance level, infection status, operating system (OS) status, anti-virus status, availability status, and malware alert status. A set of WS variables were assigned values as follows:

- $P \in \mathbb{N}$ represented the amount of public class data stored on the WS;
- $C \in \mathbb{N}$ represented the amount of confidential class data stored on the WS;
- $HR \in \mathbb{N}$ represented the amount of highly restricted class data stored on the WS;
- performance level $\in [0, 1]$ represented a general metric of how "fast" a WS was perceived to be by the user;
- infected $\in \{0, 1\}$ was the Boolean variable that was set to 1 if the WS was infected with a virus;
- OS type $\in \{0, 1\}$ was the Boolean variable that was set to 1 if the WS had a Windows OS and set to 0 else;
- OS auto $\in \{0, 1\}$ was the Boolean variable that was set to 1 if the OS was automatically updated, and set to 0 if the OS was manually updated by the user;
- AV installed $\in \{0, 1\}$ was the Boolean variable that was set to 1 if the anti-virus software was installed on the WS;

- AV auto $\in \{0, 1\}$ was the Boolean variable that was set to 1 if the anti-virus software was automatically updated on the WS, and set to 0 if the anti-virus software was manually updated by the user;
- available $\in \{0, 1\}$ was the Boolean variable that was set to 1 if the WS was available for use; and
- malware alert $\in \{0, 1\}$ was the Boolean variable that was set to 1 if malware had been detected on a WS.

Note that a malware alert would make a WS unavailable until the WS was "cleaned," i.e., the malware was removed by either the user or the system administrator.

A *server* component represented a system server that provided data to all users within a system. A server had the same variable set as a WS, but a different purpose and relationship to system users. For example, a server's operation affected all system users, whereas a single WS affected only a single user. Also, the type and amount of data stored on a server were different from those for a WS. For example, HR and C data were less likely to be stored on a server, and P data was more likely to be stored on a server and stored in greater amounts.

A *system* component represented the overall network administered by the system administrator. Variables associated with the system component characterized system configuration, number of WSs, number of servers, number of infected WSs and servers, system alerts, and anti-virus installation. A set of system variables were assigned values as follows:

- number of WS $\in \mathbb{N}$ represented the number of WSs in the system;
- number of servers $\in \mathbb{N}$ represented the number of servers in the system;
- WS infections $\in \mathbb{N}$ represented the number of WSs infected with malware in the system;
- SVR infections $\in \mathbb{N}$ represented the number of servers infected with malware in the system;
- WS alerts $\in \mathbb{N}$ represented the number of WSs with a malware alert in the system;

- SVR alerts $\in \mathbb{N}$ represented the number of servers with a malware alert in the system; and
- AV installed $\in [0, 1]$ was the Boolean variable that was set to 1 if anti-virus software was installed on all the WSs and servers in the system.

Those components were used by the participants defined in the previous section to perform the processes and tasks described next.

Processes and Tasks

Three basic processes were defined in the Anti-Virus Study model: the root, system administrator daily activities, and user daily activities.

The first process, known as the *root process*, begins the HITOP simulation and synchronizes the system administrator and user daily activities processes (see Figure 7.2). The root process starts with a start task and a single token (as indicated by a small circle with an embedded 1 to the left of the start task, as seen in Figure 7.2).

Note that the "Begin Daily Activities" task increments the simulation day counter variable *Days*. The simulation time is limited by the *Day Limit* variable to 91 days (about a standard three-month period) by the Repeat task, which directs process flow to the end task when *Days* >= *Day Limit*. The "Begin Daily Activities" task also splits the process into two parallel processes, represented by the SADA and UDA composite tasks.

The SADA task links process flow to the "System Administrator Daily Activities" child process, and the UDA task links process flow to the "User Daily Activities" child process. Note that the UDA task was normally set to be a multi-instance task, in which one instance is spawned for every active user in the system. For example, since the Anti-Virus Study model had three users, we simulated three instances of the "User Daily Activities" child process by setting the *instance number* expression in the UDA task to 3.

The Repeat task synchronizes the system administrator and user activities, and checks for the end of the simulation. The Repeat task will not become active until both the SADA task and the UDA task (and their associated child processes) have completed, i.e., until the

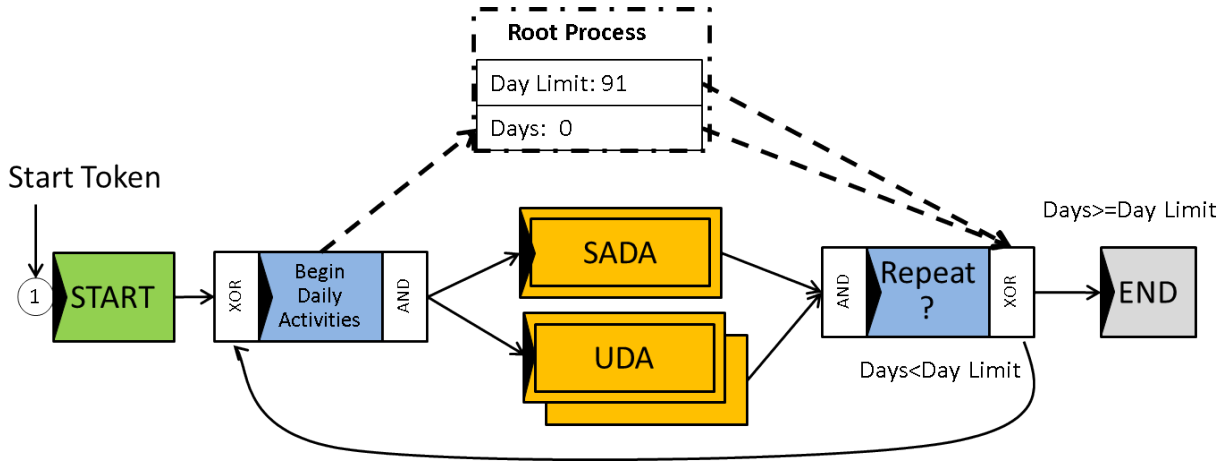

Figure 7.2: Root Process

representative PI tokens from each process arrive at the entering arcs for the Repeat task. If the day limit has not been reached, the Repeat task directs the process flow to the "Begin Daily Activities" task to start another daily simulation.

The end task is used to terminate the simulation.

Every HITOP model has a root process that starts model execution, spawns child processes, and ends model execution. Next, we will discuss the two child processes in the Anti-Virus Study model.

The "System Administrator Daily Activities" (SADA) process represents the system administrator activities, performed on a daily basis, that are related to anti-virus software and malware infections. Tasks such as checking system status, cleaning WSs infected with malware, and performing system maintenance are performed within the SADA process. Additionally, two HDPs were defined in the SADA process: *install AV* and *uninstall AV*. They represent the system administrator's decisions to install or uninstall anti-virus software on the system, respectively. Note that when anti-virus software is installed, it is installed on all computers within the system, and when anti-virus software is uninstalled, it is uninstalled on all computers within the system (see Figure 7.3).

Process flow paths to those HDPs are taken depending upon the current system state. That is, if anti-virus software is currently installed on the system, the process flow branches to the *uninstall AV* HDP. If anti-virus software is not installed, the process flow branches to the *install AV* HDP. While performing those HDPs, the system administrator can decide

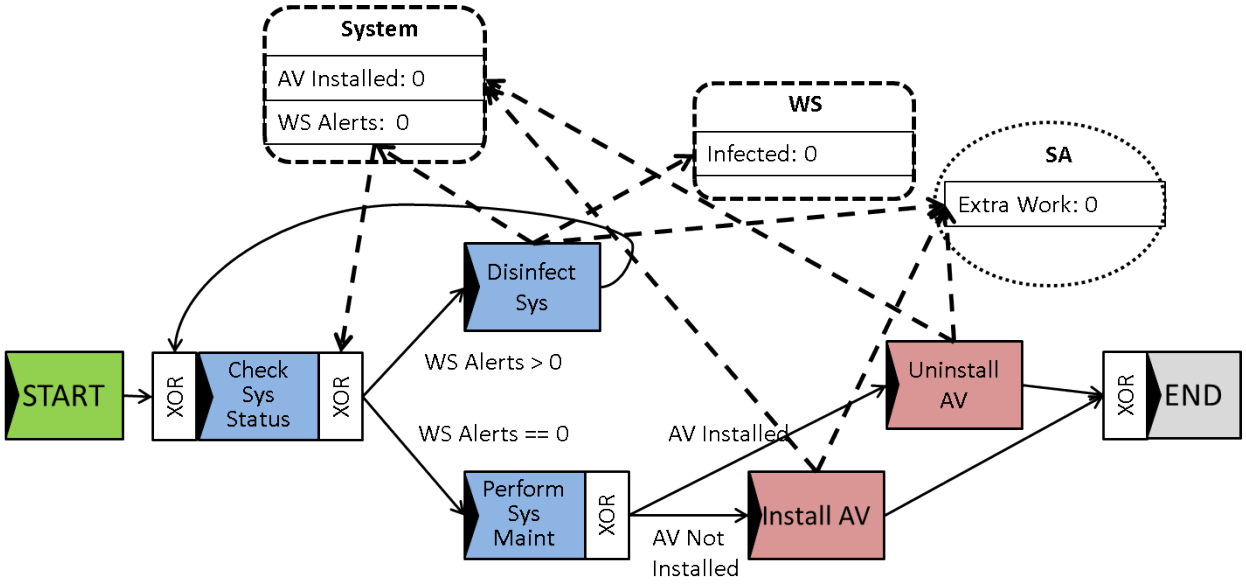

Figure 7.3: System Administrator Daily Activities Process

to install or uninstall anti-virus software on all the system workstations and servers. The system administrator is allowed to make that decision once a month, and an anti-virus install or uninstall requires a corresponding amount of extra work for the system administrator.

Recall that the probability of an HDP decision is the willingness probability P_W. Thus, the probability of the system administrator deciding to install or uninstall anti-virus software while performing the corresponding HDP is determined by the value of P_W for each HDP. When the Anti-Virus Study model is characterized within our solution framework, the effective willingness probability (EWP) for each HDP will be determined.

The "User Daily Activities" (UDA) process represents the activities performed by a user each day. That process includes such tasks as writing data to the workstation, checking email, browsing the Web, detecting malware, updating the WS software, and complaining. (See Figures 7.4 and 7.5.) The tasks in the UDA process are described next.

- Turn On WS pairs a user participant with a WS component and begins the user's workday.
- Check WS Available checks to see whether the selected WS is available for use, i.e., does not have a malware alert; if it is, the process flow is directed to task "Choose Act"; otherwise the process flow is directed to task "Wait."

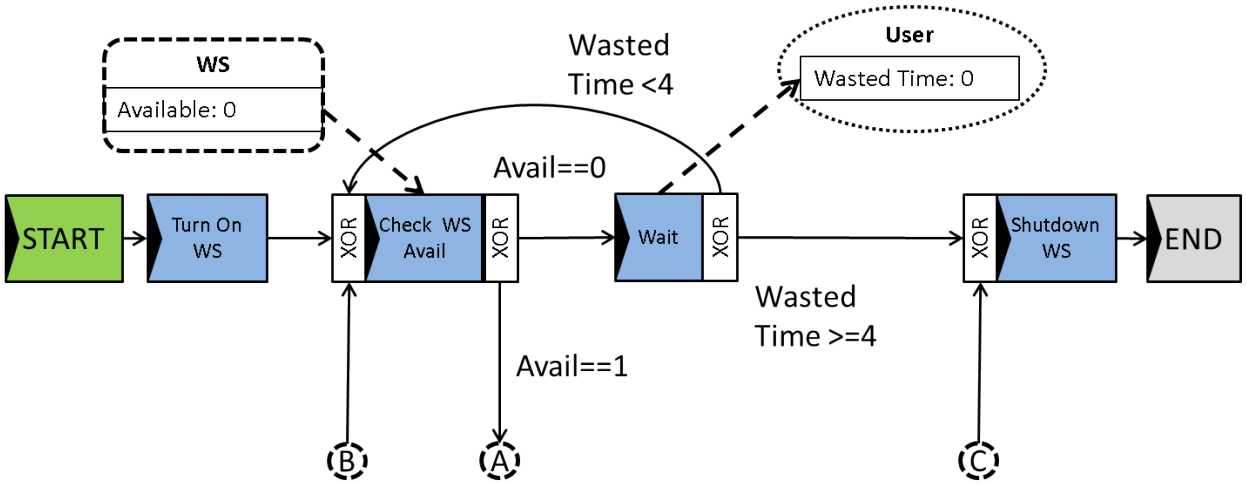

Figure 7.4: User Daily Activities Process, Part 1

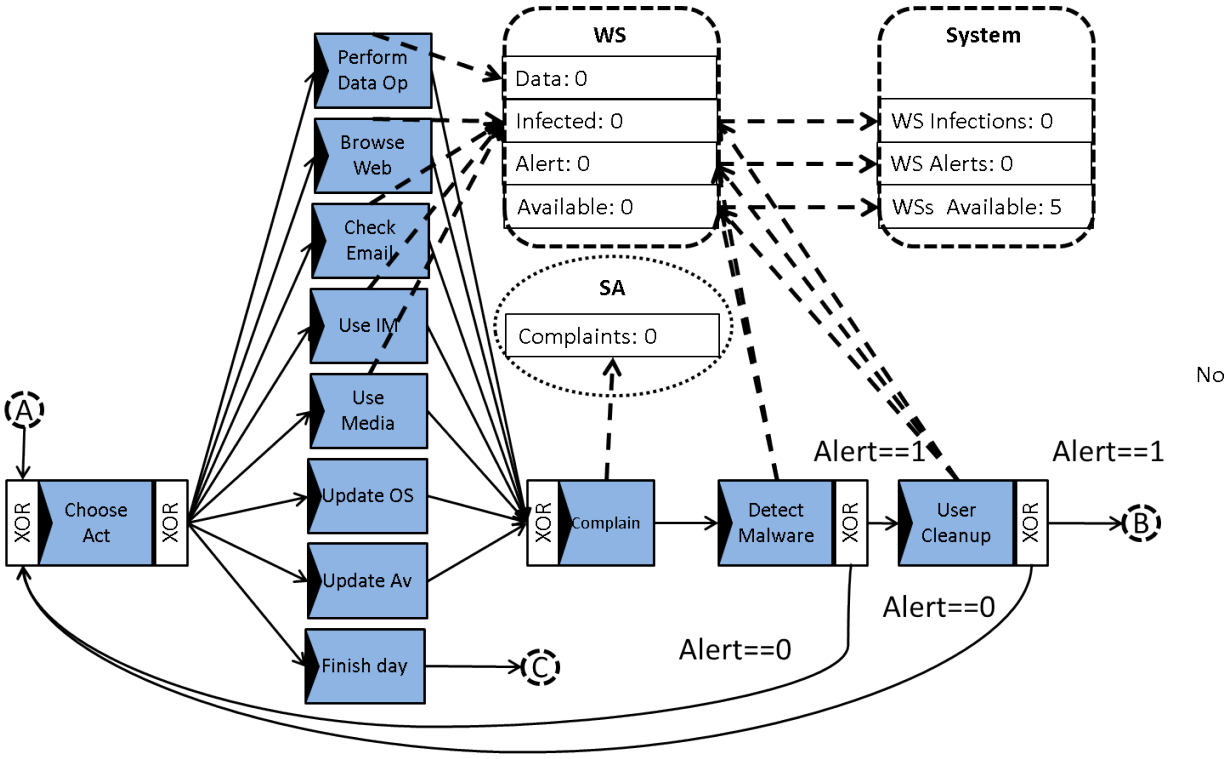

Figure 7.5: User Daily Activities Process, Part 2

- Wait waits for a period of time for the WS to become available. This task increases the amount of time the user considers "wasted" while waiting for the WS to become available. Once the wait period is over, if the user has not wasted too much time already (4 hours), the process flow is directed back to the "Check WS Avail" task; otherwise the process flow is directed to the "Shutdown WS" task.

- Shutdown WS shuts down the WS and ends user activities for the day.

- Choose Act chooses one of eight possible activities to perform, and process flow is directed to the chosen activity. Note that the choice of activity is determined stochastically via task outcomes, and the relative probability weights of the choices are set with a HITOP model parameter.

- Perform Data Op writes data to the WS. The type of data written, i.e., P, C, or HR, is determined stochastically via task outcomes. Process flow is directed from this task to the "Complain" task.

- Browse Web browses the Web. The WS can become infected by malware when this task is being performed, as determined by an outcome probability dependent on user training, anti-virus installation status, OS and anti-virus software update status, and the general malware threat environment. Process flow is directed from this task to the "Complain" task.

- Check Email performs user activities associated with email. The WS can become infected by malware when this task is being performed, as determined by an outcome probability dependent on user training, anti-virus installation status, OS and anti-virus software update status, and the general malware threat environment. Process flow is directed from this task to the "Complain" task.

- Use IM conducts an instant message chat, and the WS can become infected by malware when this task is being performed, as determined by an outcome probability dependent on user training, anti-virus installation status, OS and anti-virus software update

status, and the general malware threat environment. Process flow is directed to the "Complain" task.

- Use Media plugs media such as a cell phone or USB stick into the WS via a USB port. The WS can become infected by malware when this task is being performed, as determined by an outcome probability dependent on user training, anti-virus installation status, OS and anti-virus software update status, and the general malware threat environment. Process flow is directed from this task to the "Complain" task.

- Update OS updates the OS on the WS (if auto update is not on). This task sets the OS age variable for the WS to 0. Process flow is directed from this task to the "Complain" task.

- Update AV updates the anti-virus software on the WS (if anti-virus auto update is not on and anti-virus software is installed). This task sets the anti-virus age variable for the WS to 0. Process flow is directed from this task to the "Complain" task.

- Finish Day finishes activities for the day, and process flow is directed from this task to the "Shutdown WS" task.

- Complain files a complaint from the user with the system administrator with a probability that increases as user wasted time increases.

- Detect Malware detects malware with a probability that increases if anti-virus software is installed and increases in the user's training.

- User Cleanup cleans malware from a WS with a probability that increases with anti-virus installation and increases in the user's training.

Now that we have defined the participants, components, processes, and tasks in the Anti-Virus Study model, we are ready to define the measures of model value, i.e., the model utility functions.

Utility Functions

The Anti-Virus Study model has three basic utility functions: system administrator, security, and business.

The system administrator utility function measures the value of the system to the system administrator, and it was used to solve for the values of system administrator decisions, i.e., it was used to solve for the HDP P_W that maximizes system administrator utility. The system administrator utility function is defined by Equation 7.1:

$$U_{PART} = \beta_{PE}E + \beta_{PX}X + \beta_{PP}P + \beta_{PW}W, \tag{7.1}$$

where U_{PART} is a weighted, linear combination of embarrassment, E, defined in Equation 7.2; job experiences, X, defined in Equation 7.3; job performance, P, defined in Equation 7.4; and workload, W, defined in Equation 7.5. Each of the elements of U_{PART} is weighted by a β to ensure that $U_{PART} \in [-1, 1]$.

Embarrassment, E, indicates the level of embarrassment the system administrator feels because of malware infections of the system administrator's system or complaints from users. The embarrassment component of the system administrator utility function is defined by Equation 7.2:

$$E = \gamma_{EI}I + \gamma_{EC}C, \tag{7.2}$$

where I is the number of malware infections in the system, and C is the number of complaints received from users about the system. Both of those factors are summed over the evaluation period of three months. Each of the elements in E is weighted by a γ to ensure that $E \in [0, 1]$.

Job experiences, X, indicates the sum of positive and negative experiences, related to anti-virus software and malware infections, had by the system administrator over the evaluation period. The job experiences component of the system administrator utility function is defined by Equation 7.3:

$$X = (X_P - X_N)/(X_P + X_N), \tag{7.3}$$

where X_P is the number of positive experiences, and X_N is the number of negative experi-

ences. Because the difference between positive and negative experiences is divided by the sum of positive and negative experiences, it results in $X \in [-1, 1]$.

Job performance, P, indicates the system administrator's perception of his or her job performance. This perception is influenced by the number of system malware infections, the number of complaints, and the level of system availability maintained throughout the evaluation period. The job performance component of the system administrator utility function is defined by Equation 7.4:

$$P = \gamma_{PI} I + \gamma_{PC} C + \gamma_{PA} A, \tag{7.4}$$

where I is the number of malware infections in the system, C is the number of complaints received from users about the system, and A is the average system availability. Each of the elements in P is weighted by a γ to ensure that $P \in [0, 1]$.

Workload, W, indicates the system administrator's perception of his or her extra workload required by the number of system malware infections, anti-virus maintenance activities, and false alerts. The workload performance component of the system administrator utility function is defined by Equation 7.5:

$$W = \gamma_{WI} I + \gamma_{WM} M + \gamma_{WF} F, \tag{7.5}$$

where I is the number of malware infections in the system, M is the number of extra maintenance hours that the system administrator must perform related to the anti-virus software, and F is the number of false alerts, i.e., alerts for malware infection on a computer that is not infected. Each of the elements in W is weighted by a γ to ensure that $W \in [0, 1]$.

The values assigned to the utility function weights were determined by a survey of system administrators.

The security utility function captured the value of the model state per the security metrics of the university, and the function was used to evaluate the effective security utility for the characterized system model. The security utility function is defined by Equation 7.6:

$$U_{SEC} = \beta_{SIR} IR + \beta_{SA} A + \beta_{SD} D + \beta_{SR} R, \tag{7.6}$$

where U_{SEC} is a weighted, linear combination of the average infection rate per computer, IR, defined in Equation 7.7; the average computer availability, A, defined in Equation 7.8; the data unaffected probability, D, defined in Equation 7.9; and the percentage regulatory compliance rate, R, defined in Equation 7.10. Each of the elements of U_{SEC} is weighted by a β to ensure that $U_{SEC} \in [0, 1]$.

The average infection rate, IR, indicates the average number of infections per computer per evaluation period. The average infection rate component of the security utility function is defined by Equation 7.7:

$$IR = min\{I/n, 5\}, \tag{7.7}$$

where I is the number of malware infections in the system and n is the number of computers in the system. As is typical of utility functions that count events, R has a maximum threshold of 5, which means that in terms of security value, any average infection rate higher than 5 is as unacceptably bad as 5. R is defined such that $R \in [0, 5]$.

The average system availability, A, indicates the average percentage of time a computer is available for use when needed per evaluation period. The average system availability component of the security utility function is defined by Equation 7.8:

$$A = A_{total}/n, \tag{7.8}$$

where A_{total} is the sum of all the individual computer availabilities in the system and n is the number of computers in the system. A is defined such that $A \in [0, 1]$.

The average HR data unaffected probability, D, indicates the probability that a given block of HR data will remain unaffected by malware during the evaluation period. The average infection rate component of the security utility function is defined by Equation 7.9:

$$D = (1.0 - C_{HR}/D_{HR}), \tag{7.9}$$

where C_{HR} is the amount of HR data affected by malware and D_{HR} is the total amount of HR stored during the evaluation period. D is defined such that $D \in [0, 1]$.

The regulatory compliance rate R is the percentage of systems that have anti-virus installed per requirements. The average infection rate component of the security utility function is defined by Equation 7.9:

$$R = C_{AV}/n, \tag{7.10}$$

where C_{AV} is the average number of computers with anti-virus software installed and n is the total number of computers in the system. R is defined such that $R \in [0, 1]$.

The business utility captures the value of model events in terms of dollars, and it is used to evaluate the effective business utility for the characterized system model. The business utility function is defined by Equation 7.11:

$$U_{BUS} = C_{AV} + C_{MW} + C_{FINES} + C_{REP}, \tag{7.11}$$

where U_{BUS} is a linear combination of the anti-virus cost, C_{AV}, defined in Equation 7.12; the malware cost, C_{MW}, defined in Equation 7.13; the cost of regulatory and other punitive fines, C_{FINES}, defined in Equation 7.14; and the cost to the university's reputation, C_{REP}, defined in Equation 7.15. All the elements of U_{BUS} are measured in dollars.

Anti-virus cost, C_{AV}, measures the cost of installing and maintaining anti-virus software. The anti-virus cost component of the business utility function is defined by Equation 7.12:

$$C_{AV} = C_{install} n_{install} + C_{maint} H_{maint}, \tag{7.12}$$

where $C_{install}$ is the cost to install anti-virus software on one computer, including man-hours of work; $n_{install}$ is the number of computers on which anti-virus software was installed; C_{maint} is the cost per maintenance hour for anti-virus software; and H_{maint} is the total number of maintenance hours spent on anti-virus software.

Malware cost, C_{MW}, measures the cost of cleaning malware from an infected system and of restoring lost or corrupted data. The malware cost component of the business utility

function is defined by Equation 7.13:

$$C_{AV} = C_{clean} n_{clean} + C_{data} A_{data}, \tag{7.13}$$

where C_{clean} is the cost to clean a malware infection from one computer, including man-hours of work; n_{clean} is the number of computers that were cleaned of malware; C_{data} is the cost to restore data lost because of malware; and A_{data} is the total amount of data that is restored.

Regulatory cost, C_{REG}, measures the cost of fines for violating regulatory requirements. The regulatory cost component of the business utility function is defined by Equation 7.14:

$$C_{REG} = \sum_i E_i C_i, \tag{7.14}$$

where E_i is finable event i, and C_i is the cost of the fine for event i.

Reputation cost, C_{REP}, measures the cost to reputation for all reputation loss events. The reputation cost component of the business utility function is defined by Equation 7.15:

$$C_{REP} = \sum_i E_i C_i, \tag{7.15}$$

where E_i is a reputation loss event i, and C_i is the cost of reputation loss event i.

This concludes the Anti-Virus Study model description and definitions sections.

7.4 Experiments

In this section, we will describe how we used the Anti-Virus Study model (described in Section 7.3 and implemented using the HITOP atomic formalism described in Section 7.2) to perform several experiments to demonstrate the concepts discussed in this thesis.

First, we verified that the *Install AV Software* HDP is significant. Next, we characterized the Anti-Virus Study model for a selected configuration, i.e., solved for the optimum P_W values for the HDPs. Then, we evaluated a series of configurations using the characterized model.

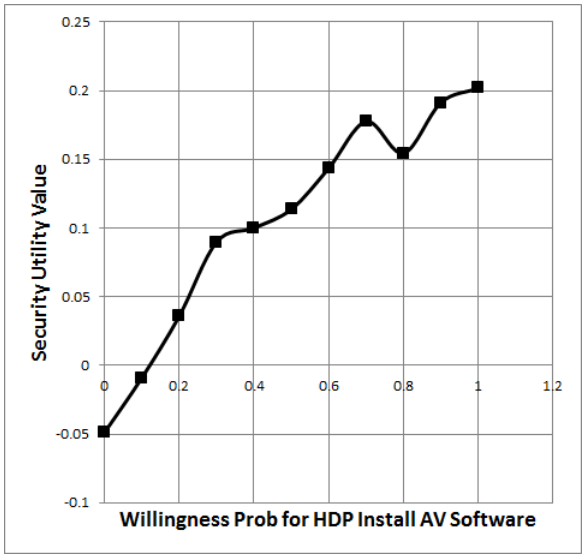

Figure 7.6: Significance of the *Install AV Software* HDP

7.4.1 Significance of *Install AV Software* HDP

First, we verified that the *Install AV Software* HDP has a significant effect on the model utility function. Recall that that is the second criterion for an HDP, i.e., the significance test. To verify the significance of this HDP, we ran a series of experiments that determined the security utility value for a range of willingness probability values for that HDP. The willingness probability for the *Install AV Software* HDP, P_{IN}, was varied from 0.0 to 1.0 in increments of 0.1. The results are shown in Figure 7.6. It is clear that P_{IN} has a significant effect on the security utility value. When P_{IN} is 0.0, the security utility is actually negative, and when P_{IN} is 1.0, the security utility is five times more positive. Thus, we confirmed that the *Install AV Software* HDP is *significant*. The *Uninstall AV Software* HDP can be similarly verified to be significant.

Once both HDPs have been verified to be significant, those HDPs may be characterized for each configuration, as demonstrated next.

7.4.2 Model Characterization

In this section, we will describe how we characterized the model. It involved selection of an optimum set of P_W values for the *Install AV Software* HDP and *Uninstall AV Software*

Table 7.1: Characterization Data

	$P_{IN} = 0.00$	$P_{IN} = 0.25$	$P_{IN} = 0.50$	$P_{IN} = 0.75$	$P_{IN} = 1.00$
$P_{UN} = 0.00$	-0.094	-0.094	-0.094	-0.094	-0.094
$P_{UN} = 0.25$	0.022	-0.010	-0.010	-0.038	-0.035
$P_{UN} = 0.50$	0.137	0.090	0.057	0.044	0.026
$P_{UN} = 0.75$	0.263	0.180	0.157	0.109	0.080
$P_{UN} = 1.00$	0.392	0.285	0.221	0.157	0.133

HDP. We will refer to the willingness probabilities for those HDPs, respectively, as P_{IN} and P_{UN}. Note that the HDP values can be set in different ways. For example, if the modeler has sufficient measured data, HDP values may be set based on empirical data gathered on the behavior of system participants. On the other hand, if the modeler has a validated model, HDP values may be set using solved values from an HDP solution method. In this section, we will utilize the exhaustive search method discussed in Section 6.3.1 and choose a solution granularity of 0.25 based on our estimation of a "good enough" solution, as discussed in Section 1.3.1.

The model we are characterizing is initialized to a state in which anti-virus software is installed. The probability of anti-virus software being installed at any point during the three-month simulation period is based on the solved values for P_{IN} and P_{UN}. Table 7.1 shows our experimental results. Figure 7.7 plots the values as a surface. Note that the highest participant utility, U_{PART}, is achieved at $\{P_{IN} = 0.0, P_{UN} = 1.0\}$. That means that the system administrator gets the most utility from the system by uninstalling the anti-virus software and not installing it again. Note also that the security utility value achieved with these values for P_{IN} and P_{UN} is lower than the *maximum* security utility value, which occurs when P_{IN} is 1.0 and P_{UN} is 0.0 (not shown in the figure). That is, the security utility is highest when the anti-virus is installed and has zero probability of being uninstalled.

It is with the settings of $\{P_{IN} = 0.0, P_{UN} = 1.0\}$ that we characterized the system for the configuration in which anti-virus software is initially installed. In the next section, we will conduct our configuration comparison assuming a consistent characterized model in which $\{P_{IN} = 0.0, P_{UN} = 1.0\}$.

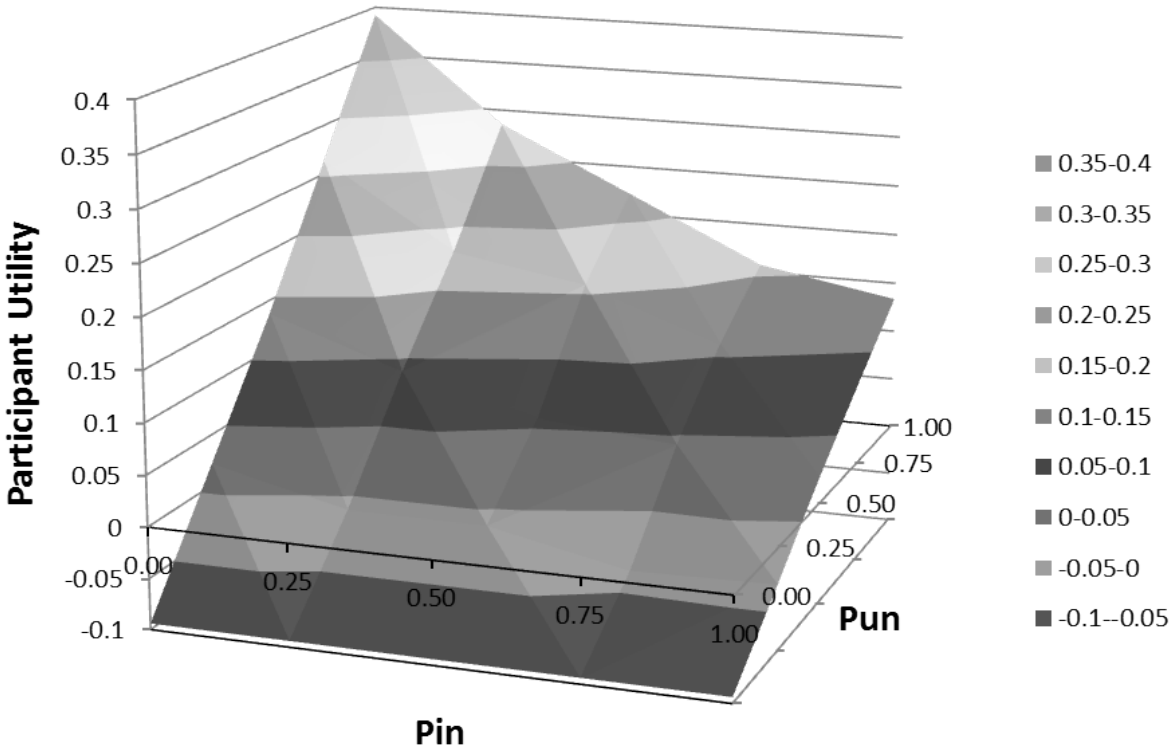

Figure 7.7: Characterization Data Viewed as a Surface

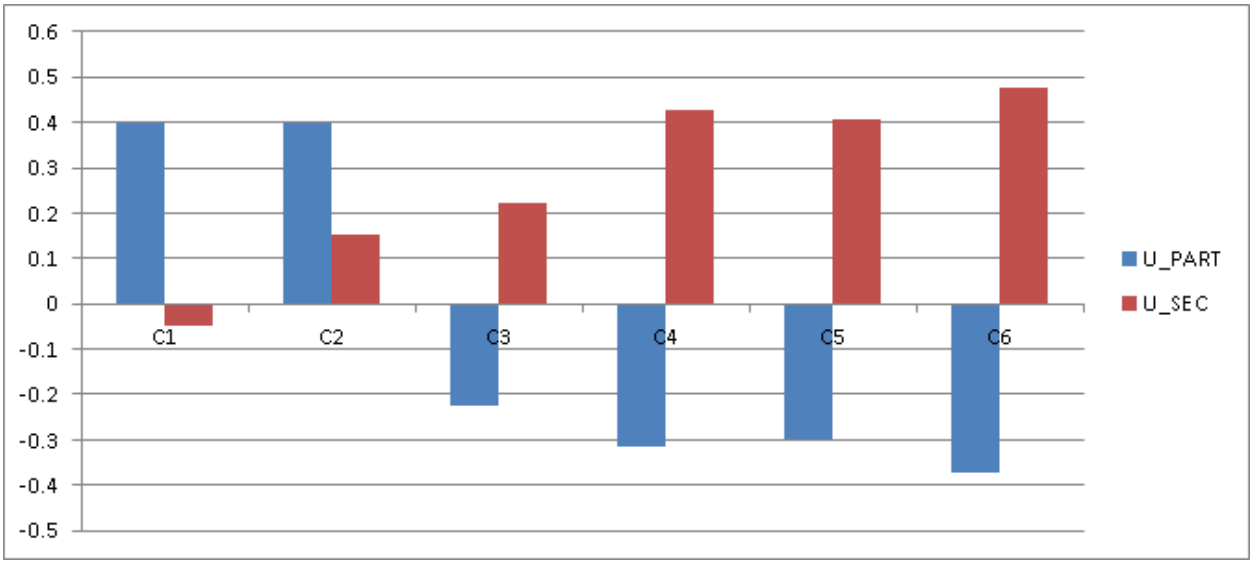

Figure 7.8: Configuration Comparison with Static Model

7.4.3 Configuration Comparisons

In this section, we will demonstrate several configuration comparisons that may be performed with a characterized model. We will compare the effect of initial anti-virus installation, automatic OS updates, and automatic anti-virus software updates on participant and security utility functions. In our experiments, we ran that comparison using both the characterized model and a baseline static model for comparison.

Configuration Comparison using Static Model

We performed an experimental comparison of system performance over a three-month period for six configurations. Factors that could be varied among configurations were whether anti-virus software was initially installed or not installed, anti-virus automatic update was turned on or off, and OS automatic update was turned on or off. Note that for this comparison, the system administrator was not allowed to install or remove anti-virus software, and thus this model provided a set of baseline configuration results for a system in which system administrators could not change the state of the system. Figure 7.8 shows the results.

Note that in the figure, the six experimental configurations are listed on the x-axis, and their corresponding utility values are listed on the y-axis. The participant utility function

Table 7.2: Configurations to be Analyzed

Configuration Number	1	2	3	4	5	6
AV Auto Setting	0	0	0	1	0	1
OS Auto Setting	0	1	0	0	1	1
AV Installed	0	0	1	1	1	1

U_{PART}, i.e., the system value to the system administrator, is indicated in blue, and the security utility value U_{SEC} is indicated in red.

In this series of experiments, six different configurations were examined. The experiments are summarized in Table 7.2. "AV auto setting" refers to the automatic anti-virus software update feature, and "OS auto setting" refers to the OS automatic update feature. If one of those settings is set to 1, it means that that feature is turned on for the simulation. Automatic updating lowers the probability of malware infection, but it does increase the amount of time the system is unavailable to the user, e.g., waiting for updates to download and install. That can result in more complaints to the system administrator.

The "AV installed" configuration variable controls whether anti-virus software is initially installed on the system, where a 1 means that anti-virus software is installed, and a 0 means that it is not installed. If anti-virus software is installed, it lowers the probability of malware infection and increases the probability of malware detection. However, it also introduces the possibility of false positives and increases the system administrator's workload and the number of complaints from users.

Note that configurations $3-6$ initially had anti-virus software installed, and configurations 1 and 2 did not. U_{PART} is positive when anti-virus software was not installed, and negative when anti-virus software was installed. That is consistent with our previous observations that the system administrator has a negative incentive to use anti-virus software. Also, note that U_{PART} becomes more negative when automatic OS and anti-virus software updates were turned on, mostly because of complaints from users about the reduced performance of their systems. U_{SEC}, on the other hand, becomes more positive when anti-virus software was installed and both OS and anti-virus updates were in automatic mode. Of course, that makes sense from a security-only standpoint, and it is with that sort of a system analysis,

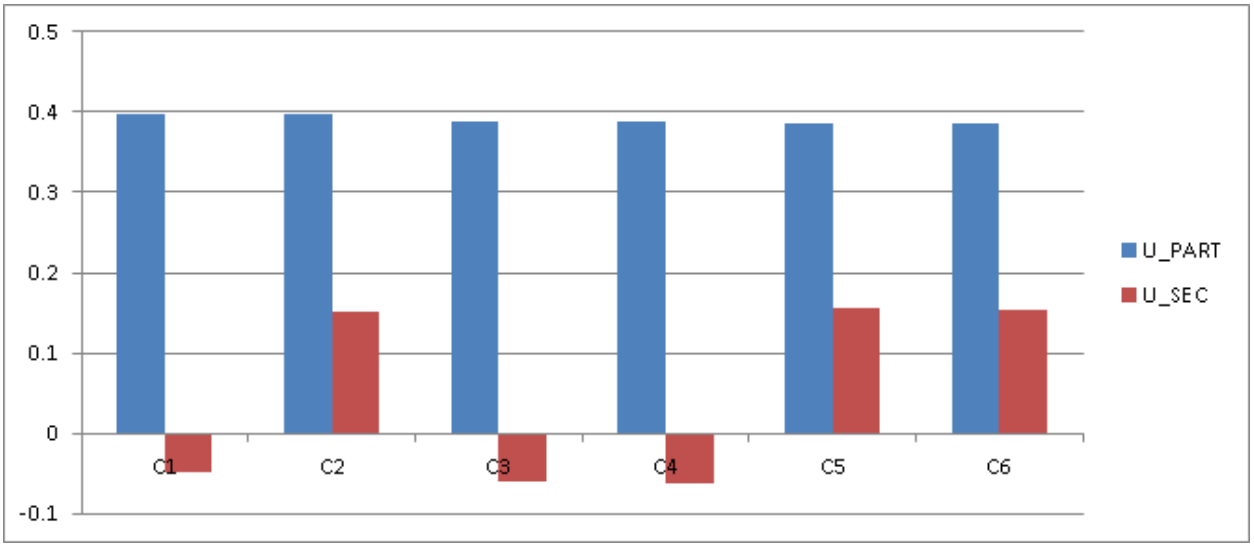

Figure 7.9: Configuration Comparison with Characterized Model

i.e., one in which a static system unaffected by system administrator actions is analyzed, that security configurations are typically selected. The security value achieved by configuration 6 in the static model is an example of an *ideal* security utility value.

The highest security utility values were achieved for the configurations in which the system administrator received the most negative utility values. Also, when the system administrator could not change the anti-virus software installation status of the system, the most secure configuration that also provided positive participant utility was configuration 2, in which anti-virus software was not installed, and the OS auto update was enabled. However, as we shall see in the next series of experiments, when the system administrator can decide to change the anti-virus configuration, the effective security utility value is very different.

Configuration Comparison using Characterized Model

In this section, we describe how we repeated the experiments of the previous section, but used the characterized model instead of the static model. That is, we used a model in which the HDP values were set to $\{P_{IN} = 0.0, P_{UN} = 1.0\}$ instead of a static HDP model. The intent was to assess what effect a characterized model would have on our overall configuration comparison results. The results are presented in Figure 7.9.

Note that U_{PART} values are uniformly positive and higher in value than their values in the

static model throughout the set of configurations. That makes sense, as the model has been characterized to optimize the participant's utility and not security utility. Also note that the highest U_{SEC} value is approximately one-third of its value in the static model. The new highest U_{SEC} value is an example of an *effective* security utility value, where the previous, higher value was an example of an *ideal* security utility value. Also note that configurations 3 and 4, while positive in the static model, are actually negative in the characterized model, because of system administrator actions. That is an example of counterintuitive system behavior (when viewed without knowledge of system administrator actions). It is also an example of how accounting for human decisions during system analysis can lead to better insights into system behaviors.

7.5 Model Validation

In this section, we will briefly discuss an approach for validating a HITOP model. Just as with traditional model validation, we validate a HITOP model by showing that the model as built provides useful information about the system modeled. The validation is achieved in two phases. Phase one demonstrates that the model produces results that are consistent with values from an existing data set. Phase two demonstrates that the model produces results that are useful for predicting values in a future or unknown data set. Next, we will discuss each of the phases.

7.5.1 Phase One: Validation with Current Data

Phase one involves validation of a HITOP model using an existing data set. Such a validation has two parts. Part one consists of validating the values of the model HDPs. Ideally, a comparison would be made between the HDP solution values and some measured data from the actual system. For example, in the case of the Anti-Virus Study model, if the expected willingness probability for the decision to install anti-virus software is solved to be 0.80, in order to validate the model, we would need real-world data on the actual likelihood that system administrators will make that choice. Part two of the phase one of validation would

involve comparison of other security metrics generated by the characterized model against their real-world counterparts. For example, there would be a comparison of model malware infection rates with actual system infection rate data.

The process of model experimental output generation and comparison with actual system data would likely be an iterative process, in which an insufficient match would require reentry into the model definition phase to adjust model structure and parameters until a sufficiently representative model could be constructed and validated.

7.5.2 Phase Two: Prediction of Future Data

Phase two of validation uses the phase one validated model to make predictions about future real-world system data. The process is as follows. Step one is to solve the model for a new system configuration that is to be analyzed. Step two is to change the configuration of the real-world system and gather performance data. Step three is to compare the simulation data with the real-world data. Step four is to re-enter the model definition phase as needed to revise the model until its predictions can be sufficiently validated. Step five is to repeat this process on multiple configurations as needed.

Note that for a model to be useful, a perfect match with real-world data is not required. In fact, because of the many uncertainties involved, a perfect match would be exceedingly difficult to obtain on a consistent basis. However, as one of the purposes of our formalism is to provide a decision-making tool, a model validated to the extent that predictions of general system trends are accurate would be a valuable decision-making tool. For example, if system managers were seeking to implement a new anti-virus software policy, it would be useful for them to know in advance whether a policy would improve security, and whether that change would be large or small. Also, if cost estimates were included via a business utility, a cost-benefit analysis could be conducted prior to the new policy's implementation. In that sense, HITOP provides a tool with which decision-makers from different domains can coordinate their actions via a common system "view."

Next, we will discuss how HITOP may be interfaced with other formalisms to produce a composed model.

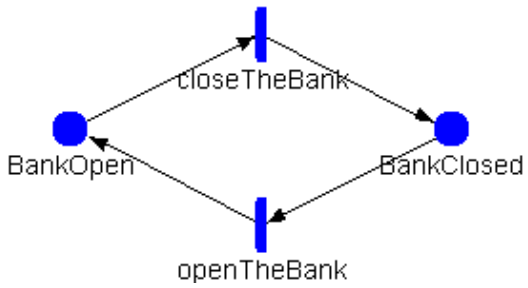

Figure 7.10: Bank SAN Model

7.6 The Bank Robbery Composed Model Case Study

In this section, we will briefly describe the technique for building a composed model using HITOP and other atomic modeling formalisms in the Möbius framework. We will use as an example a conceptual model of a bank, a bank robber, and two bank managers. The three different elements are most appropriately represented with three different atomic modeling formalisms. We will discuss each in turn and then describe how they may be linked into a single composed model.

7.6.1 The Bank

The bank model represents the physical structures and operations of a bank. In the example scenario, the only property of the bank that is relevant is whether it is open or closed. The bank can only be attacked when it is open (per our defined set of attacks).

Since the opening and closing of the bank can be modeled by a sequence of two events, bank opens and bank closes, a Stochastic Activity Network (SAN) model is an appropriate formalism in which to construct a model of the bank (see Figure 7.10).

7.6.2 The Bank Robber

The bank robber represents an attacker whose attack vectors require impersonation of a customer. The attacker has two attack goals: steal jewels from the vault, and steal money from the teller. Both goals require that the bank be open and that the attacker successfully

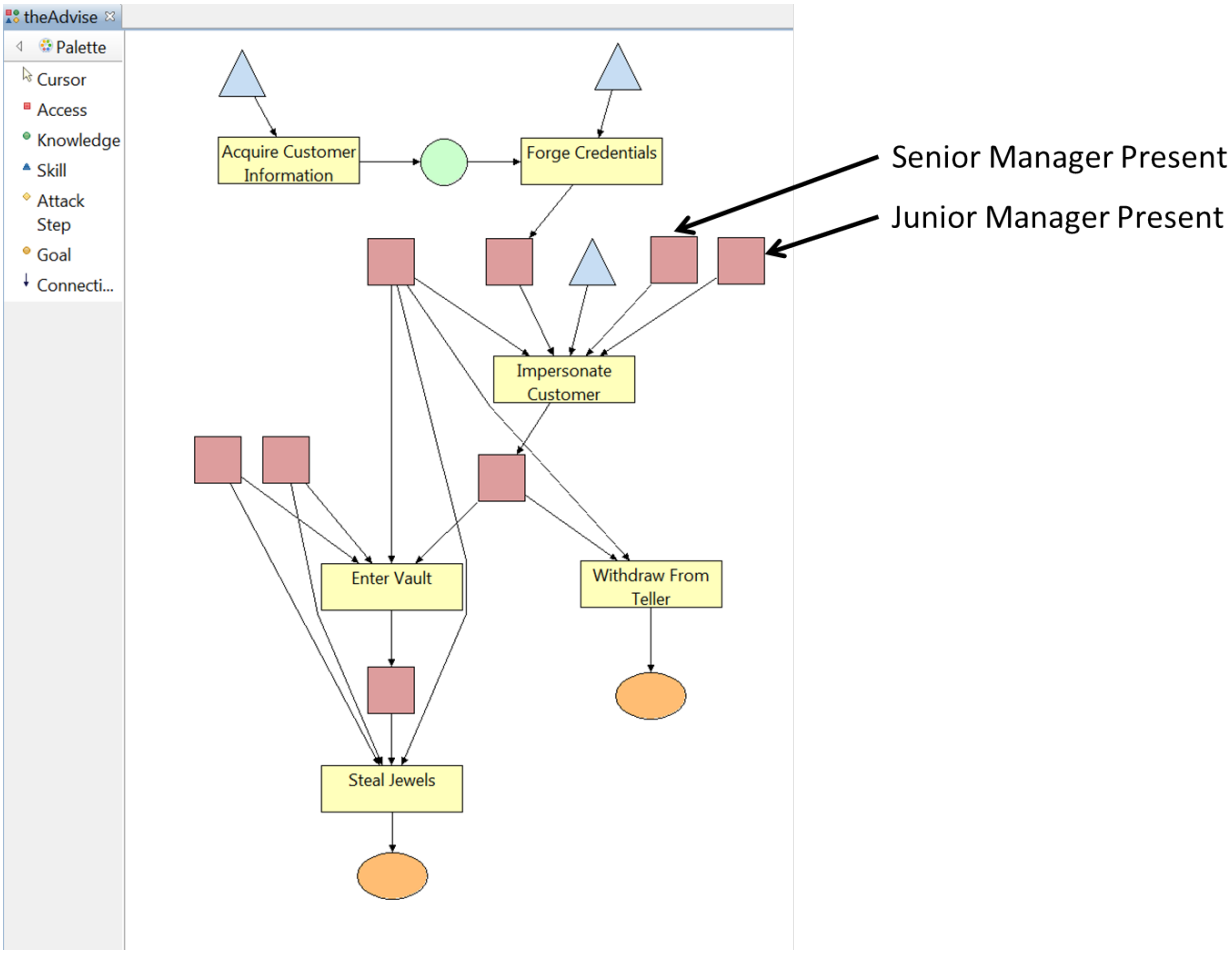

Figure 7.11: Attacker ADVISE Model

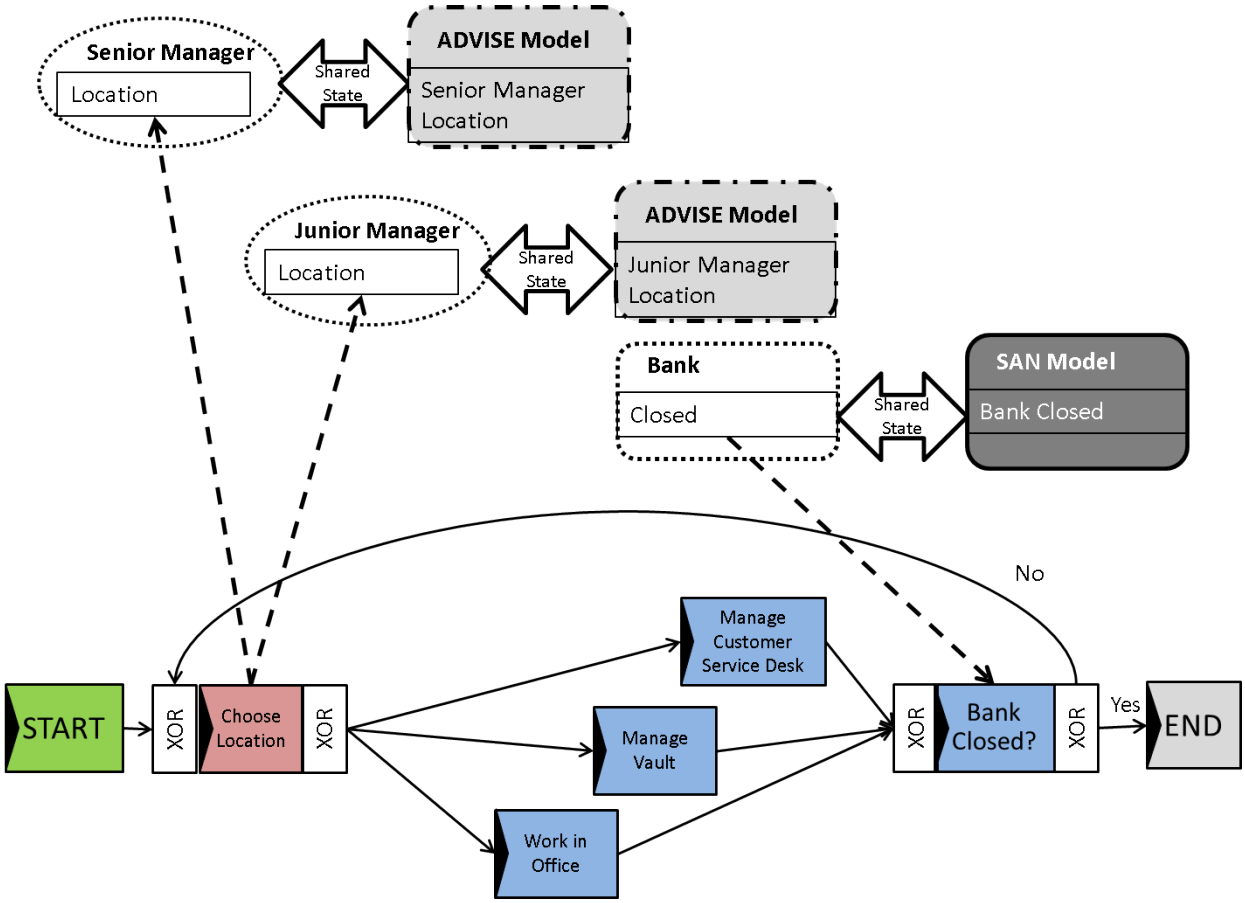

Figure 7.12: Managers HITOP Model

impersonate a customer. The probability of successfully of impersonating a customer is reduced by the presence of a senior or a junior manager. Thus, the probability of a manager being on duty at the customer service desk affects the overall probability of a successful attack.

Since the bank robber is supposed to be an attacker with various attack preferences, the robber is most appropriately modeled using the ADVISE atomic model formalism [42]. See Figure 7.11 for an example ADVISE model of the bank robber. The two left-most red boxes above the "Impersonate Customer" attack step represent the presence of the senior manager and the junior manager (discussed next).

7.6.3 The Bank Managers

The bank managers are a pair of managers who are on duty during bank hours. They are a senior manager and a junior manager. The purpose of a manager is to detect attempts by attackers to impersonate customers; thus, the presence of a manager reduces the overall probability of any attack vector involving impersonation of a customer. The senior manager is more experienced and has a higher probability of detecting fraud than the junior manager.

Bank policy states that a manager should always be somewhere on the bank floor during business hours. A manager can be either at the vault or at the customer service desk. Only one manager at a time will be at a given floor location. However, a manager can decide to violate bank policy and be in his or her office instead of on the bank floor. The manager derives some positive participant utility when in the office. The "choice of location" task is a HITOP HDP for each manager that represents that choice. Also, if there is a successful attack, the manager receives a large amount of negative utility. Thus, the bank manager's decision probability to follow bank policy and be on the floor, or not follow policy and be in the office, is dependent on the choice probability that provides the maximum utility for the manager. The problem is made more interesting because the behaviors of the senior and junior managers will affect one another. Also, the managers will both be attempting to maximize their own utility functions, leading to an interesting case in which multiple, perhaps conflicting utility optimization problems must be managed.

Since the activities of the two bank managers involve a process, the activities of the managers are most appropriately modeled using the HITOP atomic model formalism. See Figure 7.12 for an example HITOP model of the bank managers. Note that the participant state variables for the managers' locations are shared with the ADVISE model of the attacker, and the component state variables for the bank's closed status are shared with the SAN model of the bank.

7.6.4 The Composed Model

All of the atomic models described in the previous sections can be combined using a Möbius composed model. The models are linked through sharing of certain state variables. In

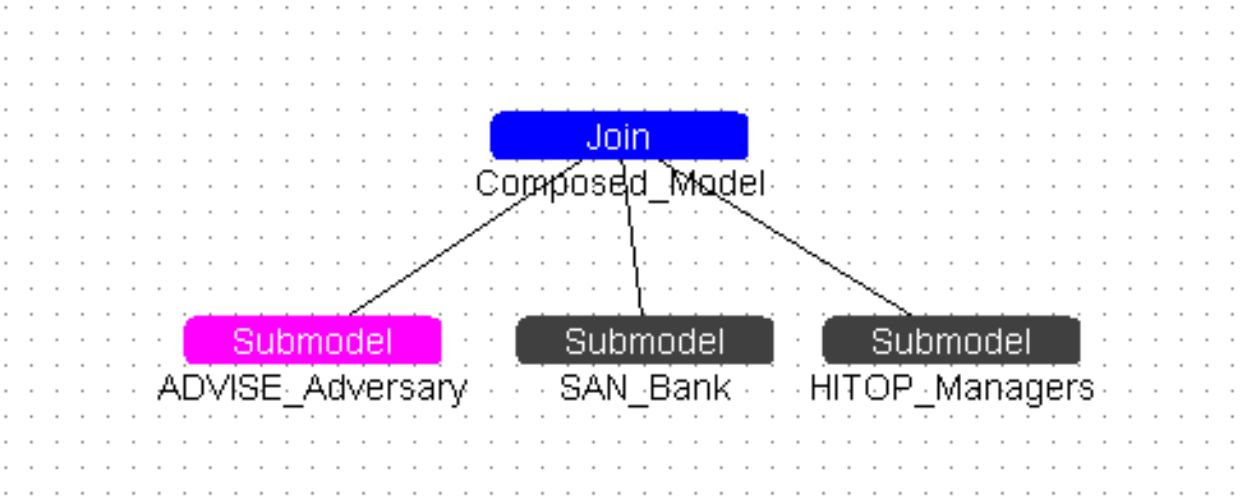

Figure 7.13: Bank Composed Model

our example scenario, the shared variables include the open or closed state of the bank, as determined by the SAN model; the achieved or not achieved attacker goal state from the ADVISE model; and the location state of each manager from the HITOP model. The composed model is shown in Figure 7.13.

To implement the composed model, for each configuration to be studied, we would characterize the composed model for each of the managers' "Choose Location" HDPs, and then solve the model for the measures of interest. For example, suppose that the HDP solution results in a senior manager's location choice vector of $[0.3, 0.5, 0.2]$ for [office, customer service desk, vault] and a junior manager's location choice vector of $[0.3, 0.2, 0.5]$. That information would be used to characterize the final composed model, which could then be solved for measures of interest, like the probability that the attacker can successfully steal jewels from the vault.

7.7 Conclusion

In this chapter, we discussed how HITOP and MAUS could be applied to a case study involving the decisions to install or uninstall anti-virus software on a large university network. We implemented the case study model using the HITOP atomic formalism and discussed

some sample results. We also discussed how a composed model might be constructed to illustrate the ability of HITOP to interface with and share state among multiple Möbius atomic formalisms.

CHAPTER 8

CONCLUSIONS AND FUTURE RESEARCH

8.1 Contributions

Humans and technology will continue to form systems that are vital to the welfare of society. As techno-human systems grow in complexity and importance, it becomes increasingly necessary to develop ways to measure and evaluate the effects of human beings on these systems. That is especially true for the numerous critical cyber-human systems (CHSs) in which we participate. The incidence and potential damage from cyber-crime, hacking, and malware grow each year, and in response, we must develop better measures and evaluations of CHS security to ensure that important CHSs stay safe and beneficial.

Of course, assessments of technology alone do not provide adequate measures. The human part of a CHS can be highly influential on security outcomes and must be taken into account when assessing CHS security. A method of evaluation for both humans and their associated cyber systems is needed. In this dissertation, we presented a novel approach for accounting for human actions, especially human decisions, in analysis of CHS security.

We presented an integrated modeling approach for identifying and quantifying how participants "within" a CHS can affect the system security state. We introduced the CHS conceptual model and defined the basic CHS element types. We introduced the OWC ontology and used it to relate system task performance to the defined CHS elements. We then introduced the concept of the Human Decision Point (HDP) and used it to identify the HDP as a special type of task within a CHS, i.e., a task whose outcomes are both influenced by human decisions and are important to measures of CHS cyber security. We introduced the Human-Influenced Task-Oriented Process (HITOP) modeling formalism as a means of specifying a mathematical CHS model given a conceptual CHS model and provided a for-

mal definition of its structure and state. We introduced the process-level and task-level viewpoints of HITOP and specified the execution algorithms for each. We introduced the Multiple-Asymmetric-Utility System (MAUS) modeling framework and described how it may be applied to generate quantitative measures of how human performance, especially human decisions, can affect CHS security. We described several simulation-based solution methods that could be used to solve the MAUS experimental framework, described several classes of solution models, and introduced the Linear Most Likely Utility (LMLU) analytical solution method, which can be used to solve certain classes of MAUS models. Finally, we applied our approach to a case study of a CHS involving system administrators and their use of anti-virus software in a large university network.

The overall goal of our work was to develop a formal method for modeling and analyzing human actions, especially human decisions, within complex systems. Our specific goal was to apply our developed method to the domain of cyber security. We achieved that goal by building HITOP, a modeling formalism that was capable of representing human participants within a CHS as distinct elements and providing quantitative measures of their effects on the system. HITOP is powerful because of its well-specified mathematical basis, yet provides a simple and intuitive interface to domain experts who are not modeling experts. MAUS provides an experimental framework that uses a HITOP model to analyze the importance of HDPs within the system and estimate the probability of their related decisions. Together, HITOP and MAUS provide a formal method to model and analyze human actions, especially decisions, within cyber-human and other complex techno-human systems.

8.2 Extensions and Applications

Although we achieved the general objectives of this thesis, there are many interesting potential extensions and application areas of this research. These areas can be grouped into theoretical work, modeling, and applications.

With respect to theoretical work, additional work could be done in extending the LMLU analytical solution method to more general classes of models, as well as identifying other classes of models for which the LMLU solution can be applied in an efficient way. Addi-

tionally, it was noted during our model simulations that the addition of process instance tokens resulted in longer simulation times. That was no doubt due to an effective simulation time that was a mathematical maximum of the set of individual token exponential execution times. A formal analysis of that effect, as well as a bounding analysis, would be an interesting theoretical addition to the execution specification of the HITOP model.

With respect to additional modeling work, the construction of a graphical interface front end for HITOP would greatly further the goal of developing a simple and user-friendly process modeling formalism. Additionally, full development of an executable composed HITOP-ADVISE-SAN model, as discussed in Chapter 7, would be a useful proof of concept. Also, the addition of an integrated and automatic LMLU solution method to the HITOP atomic formalism would be another step towards a useful domain-specific tool. Once a HITOP model and MAUS experimental framework have been defined, sufficient information is available to automatically characterize the HDPs in a system, thus saving the user a potentially laborious optimization problem.

As for applications, there is much interesting and useful work to be done in applying our formalism and solution method to real-world case studies. The AV Study model used only a subset of HITOP's capabilities, and application in the following areas is needed to fully explore all the possibilities of HITOP.

First, the ability of HITOP to handle multiple simultaneous participants of different types needs more extensive application and study. To see the benefit of reusing model structure (due to the parallel activity modelling construct), we must undertake a case study that examines the limits and benefits of using multiple tokens simultaneously.

Also, an application of HITOP to the study of real-world systems with already well-defined utility functions and data sets would be useful, especially for validation with "before and after" system empirical data.

Finally, application to a wider set of workflow patterns, such as complex OR and cancellation patterns, would truly demonstrate the general usefulness of HITOP with a variety of system types.

REFERENCES

[1] K. Vicente, *Cognitive work analysis: Toward safe, productive, and healthy computer-based work.* Lawrence Erlbaum, 1999.

[2] J. Mills, *Control: A history of behavioral psychology.* NYU Press, 2000.

[3] B. Skinner, *The behavior of organisms: An experimental analysis.* Appleton-Century, 1938.

[4] B. Skinner, *Science and human behavior.* Free Press, 1965.

[5] J. Anderson, *Cognitive psychology and its implications.* Worth Publishers, 2009.

[6] J. Hollands and C. Wickens, *Engineering psychology and human performance.* Prentice Hall, New Jersey, 1999.

[7] T. Sheridan and W. Ferrell, *Man-machine systems: Information, control, and decision models of human performance.* MIT press, 1974.

[8] J. Rasmussen, "Skills, rules, and knowledge: signals, signs, and symbols, and other distinctions in human performance models," *Systems, Man and Cybernetics, IEEE Transactions on*, no. 3, pp. 257–266, 1983.

[9] W. Rouse and N. Morris, "On looking into the black box: Prospects and limits in the search for mental models," *Psychological bulletin*, vol. 100, no. 3, p. 349, 1986.

[10] J. Hackman and G. Oldham, "Motivation through the design of work: Test of a theory," *Organizational behavior and human performance*, vol. 16, no. 2, pp. 250–279, 1976.

[11] J. Scott, "Rational choice theory," in *Understanding contemporary society: Theories of the present*, A. Halcli and G. Browning, Eds. Sage Publications London, 2000, pp. 126–138.

[12] H. Simon, "Rationality in psychology and economics," *Journal of Business*, pp. 209–224, 1986.

[13] R. Keeney and H. Raiffa, *Decisions with multiple objectives: preferences and value trade-offs.* Cambridge University Press, 1993.

[14] R. Keeney, "Multiplicative utility functions," *Operations Research*, vol. 22, no. 1, pp. 22–34, 1974.

[15] D. Messick and K. Sentis, "Estimating social and nonsocial utility functions from ordinal data," *European Journal of Social Psychology*, vol. 15, no. 4, pp. 389–399, 2006.

[16] M. Ali, "Probability and utility estimates for racetrack bettors," *The Journal of Political Economy*, pp. 803–815, 1977.

[17] C. Manski, "The structure of random utility models," *Theory and decision*, vol. 8, no. 3, pp. 229–254, 1977.

[18] M. Regenwetter, J. Dana, and C. Davis-Stober, "Transitivity of preferences," *Psychological Review*, vol. 118, no. 1, p. 42, 2011.

[19] M. Regenwetter, J. Dana, and C. Davis-Stober, "Testing transitivity of preferences on two-alternative forced choice data," *Frontiers in psychology*, vol. 1, 2010.

[20] R. Dukas, *Cognitive ecology: the evolutionary ecology of information processing and decision making*. University of Chicago Press, 1998.

[21] A. Rubinstein, "Economics and psychology? The case of hyperbolic discounting," *International Economic Review*, vol. 44, no. 4, pp. 1207–1216, 2003.

[22] H. Simon, "Theories of bounded rationality," *Decision and organization*, vol. 1, pp. 161–176, 1972.

[23] J. Broome, "Utility," *Economics and Philosophy*, vol. 7, no. 1, pp. 1–12, 1991.

[24] P. Fishburn and G. Kochenberger, "Two-piece von Neumann-Morgenstern utility functions," *Decision Sciences*, vol. 10, no. 4, pp. 503–518, 2007.

[25] J. Von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior (Commemorative Edition)*. Princeton University Press, 2007.

[26] S. Card, T. Moran, and A. Newell, *The psychology of human-computer interaction*. CRC, 1986.

[27] J. Carroll, "Human-computer interaction: psychology as a science of design," *Annual review of psychology*, vol. 48, no. 1, pp. 61–83, 1997.

[28] B. Kirwan, B. Kirwan, and L. Ainsworth, *A guide to task analysis: the task analysis working group*. CRC, 1992.

[29] K. Vicente and J. Rasmussen, "The ecology of human-machine systems ii: Mediating direct perception in complex work domains," *Ecological Psychology*, vol. 2, no. 3, pp. 207–249, 1990.

[30] H. Thimbleby and W. Thimbleby, *User interface design*. ACM Press, 1990.

[31] D. Cook, *Program evaluation and review technique: applications in education.* US Dept. of Health, Education, and Welfare, Office of Education, 1966, no. 17.

[32] W. Van der Aalst, A. ter Hofstede, and M. Weske, "Business process management: A survey," *Business Process Management*, pp. 1019–1019, 2003.

[33] S. White, "Process modeling notations and workflow patterns," *Workflow Handbook*, vol. 2004, pp. 265–294, 2004.

[34] E. Deborin, J. Basrai, T. Benedetti, R. Halchin, T. Mahfouz, N. Perera, B. Shamshabad, R. Spory, and R. Turakhia, *Continuous Business Process Management with HOLOSOFX BPM Suite and IBM MQSeries Workflow.* IBM Corp., 2002.

[35] S. White, "Introduction to BPMN," *IBM Cooperation*, pp. 2008–029, 2004.

[36] M. Dumas and A. ter Hofstede, "UML activity diagrams as a workflow specification language," *UML 2001The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pp. 76–90, 2001.

[37] W. Van Der Aalst and A. Ter Hofstede, "YAWL: yet another workflow language," *Information Systems*, vol. 30, no. 4, pp. 245–275, 2005.

[38] W. van Der Aalst, A. Ter Hofstede, B. Kiepuszewski, and A. Barros, "Workflow patterns," *Distributed and parallel databases*, vol. 14, no. 1, pp. 5–51, 2003.

[39] S. Jajodia, S. Noel, and B. OBerry, "Topological analysis of network attack vulnerability," *Managing Cyber Threats*, pp. 247–266, 2005.

[40] B. Schneier, "Attack trees," *Dr. Dobbs journal*, vol. 24, no. 12, pp. 21–29, 1999.

[41] S. Evans, D. Heinbuch, E. Kyle, J. Piorkowski, and J. Wallner, "Risk-based systems security engineering: Stopping attacks with intention," *Security & Privacy, IEEE*, vol. 2, no. 6, pp. 59–62, 2004.

[42] E. LeMay, W. Unkenholz, D. Parks, C. Muehrcke, K. Keefe, and W. Sanders, "Adversary-driven state-based system security evaluation," in *Proceedings of the 6th International Workshop on Security Measurements and Metrics.* ACM, 2010, p. 5.

[43] W. Van Der Aalst and K. Van Hee, *Workflow Management: Models, Methods, and Systems.* MIT Press, 2004.

[44] A. Beautement, R. Coles, J. Griffin, C. Ioannidis, B. Monahan, D. Pym, A. Sasse, and M. Wonham, "Modelling the human and technological costs and benefits of USB memory stick security," in *Managing Information Risk and the Economics of Security*, M. E. Johnson, Ed. Springer, 2009, pp. 141–163.

[45] A. Crystal and B. Ellington, "Task analysis and human-computer interaction: Approaches, techniques, and levels of analysis," in *Proceedings of the Tenth Americas Conference on Information Systems*, New York, New York, 2004.

[46] Q. Limbourg and J. Vanderdonckt, "Comparing task models for user interface design," in *The Handbook of Task Analysis for Human-Computer Interaction*, B. Webber, Ed. Lawrence Erlbaum Assoc., 2004, pp. 135–154.

[47] W. H. Sanders and J. F. Meyer, "A unified approach for specifying measures of performance, dependability, and performability," *Dependable Computing for Critical Applications*, vol. 4, pp. 215–237, 1991.

[48] S. Furnell, "Why users cannot use security," *Computers & Security*, vol. 24, no. 4, pp. 274–279, 2005.

[49] S. Furnell, P. Bryant, and A. Phippen, "Assessing the security perceptions of personal internet users," *Computers & Security*, vol. 26, no. 5, pp. 410–417, 2007.

[50] C. Herley, "So long, and no thanks for the externalities: the rational rejection of security advice by users," in *Proceedings of the 2009 workshop on new security paradigms workshop*. ACM, 2009, pp. 133–144.

[51] R. West, "The psychology of security," *Communications of the ACM*, vol. 51, no. 4, pp. 34–40, 2008.

[52] J. Rao, P. Kungas, and M. Matskin, "Logic-based web services composition: from service description to process model," in *Web Services, 2004. Proceedings. IEEE International Conference on*. IEEE, 2004, pp. 446–453.

[53] M. Juric, B. Mathew, and P. Sarang, *Business process execution language for web services*. Pakt, 2004.

[54] N. Russell, A. Ter Hofstede, and N. Mulyar, "Workflow controlflow patterns: A revised view," 2006.

[55] N. Russell, A. ter Hofstede, D. Edmond, and W. van der Aalst, "Workflow data patterns: Identification, representation and tool support," *Conceptual Modeling–ER 2005*, pp. 353–368, 2005.

[56] N. Russell, W. van der Aalst, A. ter Hofstede, and D. Edmond, "Workflow resource patterns: Identification, representation and tool support," in *Advanced Information Systems Engineering*. Springer, 2005, pp. 216–232.

[57] J. Mendling, H. Reijers, and W. van der Aalst, "Seven process modeling guidelines (7pmg)," *Information and Software Technology*, vol. 52, no. 2, pp. 127 – 136, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584909001268

[58] D. Eskins and W. Sanders, "The multiple-asymmetric-utility system model: A framework for modeling cyber-human systems," in *Quantitative Evaluation of Systems (QEST), 2011 Eighth International Conference on*, Sept. 2011, pp. 233 –242.

[59] W. Sanders and J. Meyer, "Stochastic activity networks: Formal definitions and concepts?" *Lectures on Formal Methods and PerformanceAnalysis*, pp. 315–343, 2001.

[60] D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster, "The Möbius framework and its implementation," *IEEE Trans. on Software Engineering*, vol. 28, no. 10, pp. 956–969, Oct. 2002.

[61] A. Adams and A. Blandford, "Bridging the gap between organizational and user perspectives of security in the clinical domain," *International Journal of Human-Computer Studies*, vol. 63, no. 1-2, pp. 175–202, 2005.

[62] G. Klein, *Sources of Power: How People Make Decisions*. MIT Press, 1999.

[63] J. Hackman, J. Pearce, and J. Wolfe, "Effects of changes in job characteristics on work attitudes and behaviors: A naturally occurring quasi-experiment," *Organizational Behavior and Human Performance*, vol. 21, no. 3, pp. 289–304, 1978.

[64] M. Evans, M. Kiggundu, and R. House, "A partial test and extension of the job characteristics model of motivation," *Organizational Behavior and Human Performance*, vol. 24, no. 3, pp. 354–381, 1979.

[65] C. Coello, "A comprehensive survey of evolutionary-based multiobjective optimization techniques," *Knowledge and Information systems*, vol. 1, no. 3, pp. 129–156, 1999.

[66] E. Jacquet-Lagreze and J. Siskos, "Assessing a set of additive utility functions for multicriteria decision-making, the UTA method," *European journal of operational research*, vol. 10, no. 2, pp. 151–164, 1982.

[67] J. Kleijnen, *Experimental design for sensitivity analysis, optimization, and validation of simulation models*. Wiley Online Library, 1997.

[68] J. Spall, *Introduction to stochastic search and optimization: estimation, simulation, and control*. John Wiley and Sons, 2003, vol. 64.

[69] J. Swisher, P. Hyden, S. Jacobson, and L. Schruben, "A survey of simulation optimization techniques and procedures," in *Simulation Conference Proceedings, 2000. Winter*, vol. 1. IEEE, 2000, pp. 119–128.

[70] R. Lewis, V. Torczon, and M. Trosset, "Direct search methods: then and now," *Journal of Computational and Applied Mathematics*, vol. 124, no. 1, pp. 191–207, 2000.

[71] T. Kolda, R. Lewis, and V. Torczon, "Optimization by direct search: New perspectives on some classical and modern methods," *SIAM review*, pp. 385–482, 2003.

[72] A. Khuri and S. Mukhopadhyay, "Response surface methodology," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 2, pp. 128–149, 2010.

[73] R. Hooke and T. Jeeves, "Direct search solution of numerical and statistical problems," *Journal of the ACM (JACM)*, vol. 8, no. 2, pp. 212–229, 1961.

[74] E. Anderson, M. Ferris, and A. G. S. of Management, "A direct search algorithm for optimization with noisy function evaluations," *SIAM journal on optimization*, vol. 11, no. 3, p. 837, 2001.