THE CYBER-PHYSICAL TOPOLOGY LANGUAGE: DEFINITION AND OPERATIONS

BY

CARMEN CHEH

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Professor William H. Sanders

ABSTRACT

As the number of security incidents and sophistication of those attacks increase, it is difficult to properly detect and diagnose malicious behavior. We conjecture that detection and diagnosis could be facilitated by an online world view that maintains information about the ability of a system to perform its intended function. We have thus developed the Cyber-Physical Topology Language (CPTL) to represent, exchange, and analyze information about a system in a dynamic fashion. In this thesis, we define a CPTL data model to represent cyber-physical assets within a system and the relations among them. We also define operations on CPTL that extract features of the system by generating a new CPTL data model that differs from existing CPTL data models in terms of topological, semantic and property changes. We then show how to integrate heterogenous data sources and detect intrusions by incorporating this model into a feedback loop. Finally, we show the applicability of our approach in an enterprise setting.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to thank my adviser, Prof. William H. Sanders, for his technical advice, encouragement, and support for my project. His guidance was an integral part of the success of this research. I would also like to thank Dr. Gabriel Weaver, who spent countless hours with me brainstorming, giving insights and advice, and pushing me to work even harder. I also thank Brett Feddersen, who helped me with implementing parts of my work and giving me advice about programming in general. Many thanks go to Jenny Applequist, who edited my horrible English in an insanely short amount of time. As a result, my English has greatly improved.

I thank the members of the PERFORM group, who have been a constant encouragement and source of joy. In particular, I thank Ahmed Fawaz, Uttam Thakore, Ken Keefe, Atul Bohara, Ron Wright, Dr. Robin Berthier, Varun Badrinath Krishna, David Huang, Mohammad Noureddine, Michael Rausch, and Benjamin Ujcich.

Last but not least, I thank my father and mother for their support and belief in me. In particular, I'm especially grateful to my mother, who flew to the U.S. to accompany me during the time of this effort, and had to endure the harsh, bitter winter of Illinois. I'm grateful to my father, who was always there to listen to my complaints.

This material is based in part upon work supported by the Army Research Office under Awards No. W911NF-09-1-0273 and W911NF-13-1-0086. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the Army Research Office. This material is also based upon work supported by the Maryland Procurement Office under Contract No. H98230-14-C-0141. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Army Re-

search Office or the Maryland Procurement Office. Finally, this material is in part based upon work supported by the Department of Energy under Award Number DE-OE0000097.[1]

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Over the past decade, the number of security incidents has been growing steadily. CERT reports a total of 60,463 security incidents in 2013 compared to 53,723 incidents in 2012 [1]. The volume of attacks that exploit a zero-day vulnerability increases by up to 5 orders of magnitude after disclosure of the vulnerability [2, 3]. Security devices are widely deployed in systems to prevent and detect those attacks. However, attackers are still able to gain access and compromise the confidentiality, integrity, and/or availability of systems. According to a 2012 report, even purpose-built intrusion detection systems detect just 5% of known intrusions into systems of larger organizations. In contrast, manual log reviews detect 8% of known intrusions [4].

Intrusion detection technologies are important and are often the basis for automated response systems. However, as typically employed, the effectiveness of IDSes is limited in two important ways.

First, IDSes usually provide a limited, siloed view of system behavior. The reason is that they typically collect data from a single architectural layer: *host*, *network*, or *application*. Host-based IDSes, such as OSSEC, collect data about events that occur on a device, such as rootkit detection. Network-based IDSes, such as Snort, collect data about events in a network in the form of packet captures. Finally, application-based IDSes collect specific data about events within a particular application. We provide an in-depth analysis of those data sources in Chapter 2.

Second, traditional IDSes output a large amount of alerts, of which many are false positives or false negatives. The numerous alerts can easily overwhelm a human administrator trying to look for evidence of intrusions. Based on our literature review, there appear to be

three main types of IDSes [5, 6, 7]; *signature-based*, *anomaly-based*, and *specification-based*. **Signature-based** IDSes detect intrusions by matching events in a system to predefined patterns of malicious behavior and have a high false negative rate. **Anomaly-based** IDSes characterize deviations from normal behavior as indications of an intrusion and typically have a high false positive rate. **Specification-based** IDSes classifies events as intrusions if the events do not match the formal specification of appropriate system behavior. If the system evolves faster than its specification, then specification-based IDSes would produce a high number of false alerts. Chapter 2 provides a more detailed analysis of the limitations of each type of IDSes.

Those limitations of traditional IDSes are the motivation behind the following three requirements for representing a target system to detect anomalous behavior and generate meaningful alerts that are comprehensible by humans, and machine-actionable to facilitate automated responses.

- The system model must be able to represent diverse types of data at different levels of abstraction in order to avoid data silos and a limited view of system behavior. Target systems such as enterprise systems and industrial control systems have various components with various levels of complexity. Events, both normal and malicious, occur at multiple architectural layers and impact system state. Therefore, the system model must integrate heterogenous data and account for multiple levels of abstraction.

- The system model must support operations that summarize the state of the target system in order to reduce the number of alerts that overburden end users. Therefore, we argue that operations that analyze alerts relative to different views of a target system can reduce alerts and provide higher-level, actionable information to network administrators and response systems.

- The system model must be updated in a timely manner. A system model must capture the current state of the target system. In order to realize different time requirements, views of the target system at different levels of detail may be constructed. In this manner, the operator or response system may adaptively tune the view of the target system in a manner that balances detail of the model with time requirements.

Our contribution. We address the above requirements and goals by developing the *Cyber-Physical Topology Language* (CPTL). We developed a preliminary version of CPTL which was expressed informally in [8], and provided an example of a CPTL-related service used for situational-aware computer networks in [9]. In this thesis, we give the formal definition of CPTL. Although CPTL encompasses a suite of operations to process and communicate information about systems of networked assets, in this paper we focus on the following contributions:

- a CPTL data model that constructs different worldviews of the target system,
- a set of operations on CPTL, specifically vertex contraction, that analyzes those worldviews, and
- a feedback loop to baseline system behavior using different worldviews of the target system provided.

We elaborate on each contribution and describe how it addresses our requirements.

CPTL data model. First, we describe the CPTL data model and its formal specification of the target system. The CPTL data model consists of three components: a *graph*, an *ontology*, and an *interpretation*. The *graph* describes the structure of interactions among entities in the target system. The *ontology* is based on Description Logic (DL) that provides a formal specification so that we can automatically deduce malicious behavior. Finally, the *interpretation* maps the ontology to the graph so that we can select specific data sources and relate them in a manner that explains user behavior.

The graph structure enables the CPTL data model to integrate event information across a variety of architectural layers of the target system. We want to not only baseline systems in terms of the behavior of *individual layers* but also in terms of the *interactions among those layers*. The working hypothesis here is that although an attacker may be able to compromise an individual layer's behavior, an IDS that characterizes normal behavior in terms of interactions among multiple layers will be much harder for an attacker to circumvent.

Ontologies and interpretation functions from Description Logics provide analysts and algorithms with machine-actionable semantics that are unambiguous. Those formalisms also

provide a way to reinterpret the same graph and attributes to construct a new worldview with new semantics.

The combination of graphs and ontologies allows us to incorporate additional data sources not traditionally found in IDS systems. User behavior may be integrated into the graphical model in addition to the traditional network and host-level information. Furthermore, interactions between devices and the physical environment may also be represented and reasoned about using our framework.

CPTL operations. Second, we introduce operations on CPTL to extract features from the target system. An operation on CPTL is characterized by three types of changes to the target system model: *topological*, *property*, and *semantic*. Topological changes affect the structure of the graph by modifying the entities in the target system and the relations between them. Property changes involve modifying the features of entities or features of relations. Finally, semantic changes affect the formal description of the graph by modifying the ontology and/or interpretation.

Now, we describe a set of operations on CPTL and their functionality. In particular, we introduce the operations **Join**, **Abstract**, and **Contract**.

Join: This operation operates on two CPTL models that represent two different worldviews of a target system. The result of the **Join** operation is a CPTL model that represents a combination of the worldviews of the target system by merging entities and relations in the views. **Join** is useful when we need to relate different information sources together or update the CPTL model with new events.

Abstract: This operation takes a single CPTL model and modifies it semantically by describing the graph using a higher-level formal description. We can use the **Abstract** operation to extract high-level descriptions of behavior patterns for analysis and specification.

Contract: This operation operates on a single CPTL model by taking a set of entities in the model and merging the set into a single general entity in the resultant CPTL model. That general entity represents the set of entities and inherits their relations as well as a

summary of their features. We can use **Contract** to generate summarized worldviews of the target system and extract higher-level features of the graph.

In this thesis, we will focus on the **Contract** operation, and will only briefly describe the other two operations (in Chapter 3). In general, these operations allow us to efficiently query the system state for underlying patterns and dynamically update the system state. We can also derive high-level features that are semantically related and intuitive for humans to reason about.

CPTL-aware feedback loop. We incorporate the CPTL data model within a feedback loop to baseline system behavior. The feedback loop consists of two stages: the *offline* analysis and the *online* checking. We describe the two components in further detail.

Offline: We maintain a CPTL model of the target system and perform CPTL operations to infer and corroborate state information across data sources. We also use the operations to obtain refined behavior profiles that are based on semantic features.

Online: We add events into our CPTL representation by asserting the event as a fact in the ontology. Then, we use existing reasoner tools to check that the ontology is satisfiable by the events, i.e., the events fulfill the axioms in the ontology. If an event causes the reasoner to raise an exception, we can either present the event along with the axioms it violated to a human administrator, or automatically block the event from occurring.

We use the concept of feedback loops to increase our level of understanding of the target system. The offline model informs the online model about appropriate behavior to enforce by deriving axioms that are asserted in the online model. The online model informs the offline model about changes in behavior and state. In addition, the online model provides the offline model with feedback about false positives so that the offline model can improve its specification of intrusions.

1.1 Use Case

In this thesis, we focus on malicious insiders as our threat model. Insider attacks pose a greater threat to systems than external attacks do, because of the intimate knowledge of the system and its defenses possessed by insiders [10]. Recent figures show that insider threats can cause tremendous damage; for example, two recent cases involved a financial loss of more than $1,000,000 in sales and the loss of more than $40,000,000 in documents [11]. Insider threats also result in greater financial damage than external threats do [12]. An insider also has legitimate physical and cyber access to the system, obviating the need to execute complicated attacks.

Malicious insiders can be categorized into two groups: *masqueraders* and *traitors*. Those terms have been adopted from [13]. A **masquerader** is an illegitimate user who steals a legitimate user's credentials and uses them to conduct malicious activity. On the other hand, a **traitor** is a legitimate user who violates system policy in order to compromise confidentiality, integrity, and/or availability of system assets.

In this thesis, we will look at the following insider attacks:

- Unauthorized exfiltration of data.
- Tampering with data.
- Misuse of system resources.

This thesis. In Chapter 2, we present related research efforts on IDSes and discuss their limitations. In Chapter 3, we define CPTL and its operations, and in Chapter 4, we describe its implementation. We apply CPTL and its operations to an enterprise use case in Chapter 5, and present an evaluation of our work in Chapter 6. Finally, we conclude and discuss future work in Chapter 7.

CHAPTER 2

RELATED WORK

In this chapter, we survey the current state of IDSes and their limitations. More specifically, Section 2.1 describes the sources of audit data that are used by IDSes, and discusses the need for a formal framework that integrates heterogenous data sources to gain a more comprehensive picture of intrusions occurring in a system. In Section 2.2, we compare the different types of IDSes and discuss their shortcomings. We also relate existing IDSes to our approach, and explain how our approach addresses their gaps.

2.1 Audit Sources

IDSes analyze data that have been collected in the system to find evidence of intrusions. The data are collected from different architectural levels of the system. Based on our literature review, there appear to be three main types of data sources [14]: *network*, *host*, and *application*.

Host level. Host-based IDSes collect data about events occurring in a computer system. Events include system calls, user commands, file accesses, system events, and running processes. This data source provides evidence of intrusions that occur within a computer system and is vulnerable to attacks that compromise the system and tamper with the collected data.

Network level. Network-based IDSes look at network traffic within a target system to detect intrusions. These IDSes are limited to attacks that propagate over a network, and are unable to detect physical attacks (for example, a virus located on a thumb drive) or social engineering attacks.

Application level. Application-based IDSes collect data about events occurring in a specific application. The data can be a combination of both host-level and network-level data sources. Since the collected data are specific to an application, the IDS is tailored to detect only intrusions that affect the application.

All three types of data sources provide important evidence of intrusions. Typically, systems employ multiple types of IDSes that monitor the system at the different levels. However, some attacks may manifest at all levels, but to an extent that is not detectable by individual IDSes. So we need a unified way to correlate data at different levels to be able to detect intrusions [15].

2.2 Intrusion Detection Systems

As mentioned in Chapter 1, there are three types of IDSes: signature-based, anomaly-based, and specification-based. Next, we will elaborate on these types of IDSes and describe their pros and cons.

2.2.1 Signature-based IDS

Signature-based IDSes rely on a database of signatures that describe attacks. The ability of an IDS to detect intrusions relies solely on that signature database. The database is updated from time to time with signatures of new attacks. The advantages of a signature-based IDS include its low false positive rate as well as its efficiency. However, signature-based IDSes are unable to detect unknown attacks or variants of an attack that do not follow the fixed signature in the database.

More et al. [16] attempts to improve on the current signature-based IDSes by abstracting the definition of signatures to a higher-level contextual description that is informed by vulnerability description feeds such as CVE, CVSS, and forums. More collects data streams from network monitors, host monitors, sensor data from other IDS modules, and security logs. Those data streams are asserted as facts in a knowledge base. Then, a reasoner is run

over the knowledge base to infer whether the collected data are indications of an attack. That approach is an improvement over existing signature-based IDSes because it can detect variants of attacks as long as the specification in the vulnerability description feeds covers the different variants. Furthermore, the approach combines heterogenous data sources to get a better picture of intrusions in the system. However, it still suffers from the weakness of signature-based IDSes, as it is unable to detect unknown attacks.

Just as in other approaches, we use ontologies to express domain knowledge and events. In contrast with other approaches, our model also expresses the relations among entities and is capable of describing not only signatures but also baselines and policies.

2.2.2 Anomaly-based IDS

Anomaly-based IDSes develop a baseline of normal behavior and detect intrusions by looking at how much events deviate from that baseline. Baselines are typically created using statistical distributions, machine-learning techniques, or data mining [14]. Baselining of user behavior is the main technique for detecting masquerader attacks [13] and is also recommended by CERT [17]. Since a masquerader is an outsider, he or she is probably unfamiliar with the legitimate user's behavior, and his or her actions would very likely deviate from the typical behavior of the legitimate user.

The ability of anomaly-based IDSes to detect attacks is dependent on the features and technique used to create and update the baseline. Anomaly-based IDSes are able to detect unknown attacks, but are known to have a higher false positive rate [7, 14, 18]. Creating an accurate baseline and updating it over time are hard tasks. Below, we describe an approach that attempts to provide a framework for creating baselines.

The Anomaly Detection at Multiple Scales (ADAMS) program [19] of the Defense Advanced Research Project Agency (DARPA) aims to detect and prevent insider threats through use of large-scale data. One of the products of that program is the Proactive Detection of Insider Threats with Graph Analysis And Learning (PRODIGAL) architecture [20].

The PRODIGAL architecture detects possible insiders by using models of user activity

at multiple levels of abstraction [21]. The contributions of the PRODIGAL architecture include:

- a specialized visual language called the *Anomaly Detection Language* (ADL) for expressing different combinations of algorithms, data, baselines, time periods, and clusters;
- novel algorithms that detect anomalies by modeling different aspects of normal behavior and comparing observables in a dataset to the normal behavior; and
- an application of the framework to an actual dataset consisting of monitored computer usage activity in a business organization.

The Anomaly Detection Language (ADL) consists of components that take in data, operate on the data, and output the result of the operation. Components can be chained together to form a pipeline of operations. Different components can perform different types of operations. The operations supported are statistical anomaly analysis, group detection that extracts communities of entities, filtering and partitioning of data into classes, aggregation of certain data values, and normalization by scaling of data [21]. The language is meant to be domain-independent.

In [20, 22], the PRODIGAL architecture is applied to a dataset reflecting computer usage activity in a real business organization. The authors looked at data from various audit sources, such as email, file accesses, login information, print jobs, visits to websites, and Instant Messaging (IM) activity. The organization consisted of approximately 5,500 people, and each user conducted approximately 1,000 actions per day, giving rise to 5.5 million records per day. Data were collected over a period of two months.

The authors defined six main threat scenarios, and for each scenario, constructed an anomaly detection algorithm that uses the ADL to combine indicators from the various audit sources. A red team was used to augment the data with instances of insider activity. The results obtained were promising with the highest area under the curve (AUC) being 0.979 [20].

We emphasize that our CPTL data model is complementary to PRODIGAL's ADL, because CPTL focuses on modeling an evolving target system whereas the ADL models fusion of data. Thus, we envision that CPTL could be used as a basis for representing ADL. Moreover, CPTL provides a formal definition of the domain of interest and allows for automatic inferencing of anomalies.

It is mentioned in [23] that the key to detecting attacks lies in the inexplicability of an event rather than its rarity. Explanations for events rely not only on the information surrounding the events but on external data sources. We can use CPTL to explain anomalies using additional data sources, such as the state and location of an asset or the presence of an external event, because these sources contribute to the state of the target system. We believe that CPTL and PRODIGAL can be combined to work in tandem, with PRODIGAL using CPTL operations to query the target system state.

2.2.3 Specification-based IDS

Specification-based IDSes formally define appropriate system behavior and detect events that deviate from the formal description. The ability to detect intrusions lies in the formal specification of the system. Thus, it is essential that the formal specification be accurate and up to date. The advantage of the specification-based IDS is that it detects any attack that does not adhere to the definition of appropriate behavior, and thus has a low false negative rate. This type of IDS is useful in environments that have more well-defined policies of acceptable behavior; for example, in cyber-physical systems [24]. However, it is hard to create and maintain an accurate specification. Next, we describe an approach that combines specification-based and anomaly-based detection to detect malicious insiders.

Maloof et al. [25] developed a system called *Exploit Latent Information to Counter Insider Threats* (ELICIT) that looks at information-use events. The events they looked at include searching, browsing, downloading, and printing of data. Based on those events, they developed detectors that use contextual information, such as location of users and machines, to profile users and the activity of groups of users during a specified time period. Those detectors were chosen based on analysis of events in the dataset, advice from domain experts,

and information from publicly known insider cases.

Some of the detectors included printing of documents to a distant printer (where distance was measured in terms of the number of floors from a user's office to the printer), and accessing of documents that were outside a user's social network (where the social network was determined based on email correspondence and project assignments). Those two detectors represented specifications of policies enforced by a system.

To evaluate the performance of ELICIT, the authors collected data from 284 days of network traffic and abstracted the data into 91 million information-use events performed by 3,900 users. Then, a red team developed scenarios based on publicly known insider cases and inserted events into the dataset. The results were promising; the area under the Receiver Operating Curve (ROC) curve was 0.92; detection rate was 0.84; and the average false-positive rate was 0.015.

Our work provides a formal framework that could model the data and detectors used in ELICIT. We could perform more complex analysis of the data and generate detectors that can be automatically updated.

CHAPTER 3

A CYBER-PHYSICAL TOPOLOGY LANGUAGE FOR SYSTEM MODELING

3.1 Motivation

In Chapter 1, we introduced the limitations of current IDSes and explained the need for a formal specification of a system model. The system model represents the state of the target system (e.g., an enterprise system, smart grid, or data center). The state of the target system encompasses (1) cyber-physical assets, (2) users, and (3) properties of and interactions among the assets and users. For example, in our use case in Chapter 5, we have an enterprise system that is the target system. The state of the enterprise system consists of (1) physical assets such as machines and printers, cyber assets such as files, and credentials; (2) users, such as employees; and (3) employees' accesses to files, possession of credentials by employees, and accesses to printers for printing of files.

Thus, we define the **Cyber-Physical Topology Language (CPTL)** as a mechanism to integrate and exchange diverse information at multiple levels of abstraction and perform operations on that information to obtain useful measures. The information is represented as a graph $G(V, E)$ that is described by an ontology $\mathcal{K}$ expressed in a description logic DL.

In the following sections, we will describe the definition of CPTL and its operations. We apply CPTL and its operations to two use cases in Chapter 5. In this chapter, we will use a small portion of the dataset in an enterprise setting to demonstrate the syntax and usage of CPTL and its operations.

3.2 Background

As mentioned previously, we have developed a representation of information in terms of a graph, ontology, and description logic, for which we will now provide some background information.

3.2.1 Graphical Data

With the advent of big data, many analytic techniques have been employed. Graph analytics is a popular and important technique for sifting through big data. Graphs are well-suited to modeling relationships among entities and thus have been used to model social networks like Facebook, biological networks, and communication and computer systems. For example, the Linked Data movement [26] is about connecting semantically related online entities that are modeled as a graph.

Graphical data are both visually intuitive and allows us to leverage theoretical tools such as graph theory for data analytics. We want to be able to process the information represented in a graph and present a higher-level understanding of the data. In addition, we want to be able to reason about the information at different levels of abstraction.

Graphical data are represented as a graph G that consists of vertices V and edges E. More formally, we define G as a *multi-directed graph*.

Definition 1 *A **multi-directed** graph G is an ordered triple*

$$G = (V, E, h) \tag{3.1}$$

where V is a set of vertices, E is a set of edges, and $h : E \to V \times V$ is a function mapping the edges E to an ordered tuple. We can thus refer to an individual edge as $e_i(v_1, v_2)$.

We represent a small portion of the dataset in an enterprise setting using a multi-directed

graph, as shown in Figure 3.1. The graph can be represented as $G = (V, E, h)$, where

$$V = \{v_1, v_2, \ldots, v_{12}\}$$
$$E = \{e_1, e_2, \ldots, e_{18}\}$$

$$\begin{aligned}
h(e_1) &= (v_1, v_4) & h(e_{10}) &= (v_5, v_7) \\
h(e_2) &= (v_2, v_3) & h(e_{11}) &= (v_5, v_7) \\
h(e_3) &= (v_2, v_5) & h(e_{12}) &= (v_9, v_6) \\
h(e_4) &= (v_3, v_6) & h(e_{13}) &= (v_9, v_7) \\
h(e_5) &= (v_3, v_7) & h(e_{14}) &= (v_{12}, v_9) \\
h(e_6) &= (v_7, v_8) & h(e_{15}) &= (v_5, v_8) \\
h(e_7) &= (v_4, v_6) & h(e_{16}) &= (v_{11}, v_8) \\
h(e_8) &= (v_7, v_{10}) & h(e_{17}) &= (v_{11}, v_{10}) \\
h(e_9) &= (v_5, v_7) & h(e_{18}) &= (v_5, v_{10})
\end{aligned}$$

The vertices V and edges E have a set of **vertex attributes** and **edge attributes**, respectively. The attributes are used as data to infer further semantic meaning about the graphical data as we will see in examples later in this section. The properties are represented as functions that map a vertex (or edge) to its respective attributes. We define the following two sets as well as the corresponding functions below.

S_V**:** This is a set of possible vertex attribute values.

S_E**:** This is a set of possible edge attribute values.

$A_V : V \to S_V$**:** This is a function mapping vertices in the graph to their respective vertex attributes.

$A_E : E \to S_E$**:** This is a function mapping edges in the graph to their respective edge attributes.

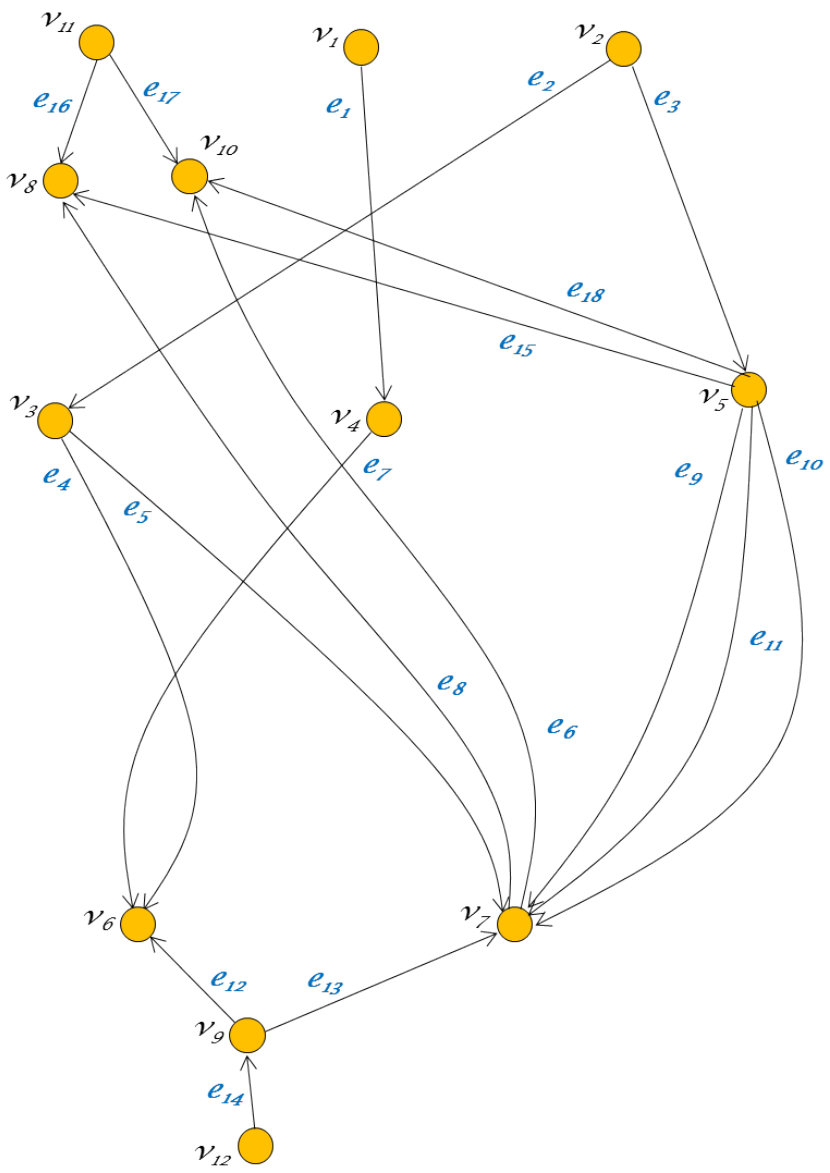


Figure 3.1: A multidirected graph $G(V, E, h)$. The vertices are labeled as v_i, and the edges are labeled as e_i.

In the following way, we can define the vertex and edge attributes for the graph in Figure 3.1 based on the dataset. The symbol "$\cdot$" denotes that there is no attribute value for that vertex or edge.

$$S_V = 2^{\text{String}}$$
$$S_E = 2^{(\text{Date,Integer,Integer,Integer})}$$

$$\begin{aligned}
A_V(v_1) &= \text{Alice} & A_V(v_7) &= \text{b.doc} \\
A_V(v_2) &= \text{Bob} & A_V(v_8) &= \text{Printer1} \\
A_V(v_3) &= \text{bob1} & A_V(v_9) &= \text{Documents} \\
A_V(v_4) &= \text{alice1} & A_V(v_{10}) &= \text{Printer2} \\
A_V(v_5) &= \text{bob2} & A_V(v_{11}) &= \text{Buildingl} \\
A_V(v_6) &= \text{a.doc} & A_V(v_{12}) &= \text{CompanyDoc}
\end{aligned}$$

$$\begin{aligned}
A_E(e_1) &= (\cdot, \cdot, \cdot, \cdot) & A_E(e_{10}) &= (\text{2014-05-28:1PM}, \cdot, \cdot, 5) \\
A_E(e_2) &= (\cdot, \cdot, \cdot, \cdot) & A_E(e_{11}) &= (\text{2014-05-28:2PM}, \cdot, \cdot, 1) \\
A_E(e_3) &= (\cdot, \cdot, \cdot, \cdot) & A_E(e_{12}) &= (\cdot, \cdot, \cdot, \cdot) \\
A_E(e_4) &= (\text{2014-02-10:10AM}, 50, 0, \cdot) & A_E(e_{13}) &= (\cdot, \cdot, \cdot, \cdot) \\
A_E(e_5) &= (\text{2014-06-02:11AM}, 10, 2, \cdot) & A_E(e_{14}) &= (\cdot, \cdot, \cdot, \cdot) \\
A_E(e_6) &= (\text{2014-05-28:1PM}, \cdot, \cdot, 5) & A_E(e_{15}) &= (\text{2014-05-28:1PM}, \cdot, \cdot, 5) \\
A_E(e_7) &= (\text{2014-02-15:4PM}, 2, 0, \cdot) & A_E(e_{16}) &= (\cdot, \cdot, \cdot, \cdot) \\
A_E(e_8) &= (\text{2014-05-28:2PM}, \cdot, \cdot, 1) & A_E(e_{17}) &= (\cdot, \cdot, \cdot, \cdot) \\
A_E(e_9) &= (\text{2014-10-20:3PM}, 8, 10, \cdot) & A_E(e_{18}) &= (\text{2014-05-28:2PM}, \cdot, \cdot, 1)
\end{aligned}$$

3.2.2 Ontologies

Description Logics (DL) is a family of languages that represent knowledge in a formal manner. DL models the relationships among entities in a domain of interest using *concepts, roles*, and *individual* names. The domain of interest is termed the *knowledge domain*. **Individual names** represent single entities in the domain, whereas **concept names** are sets of individuals, and **role names** represent binary relations between individuals. For example, an individual name might be *Alice* and a concept name might be **Person**, which represents the set of individuals that are instances of people; of which *Alice* is one. A role name might be **hasMachine**, representing the relationship between an instance of **Person** and an instance of **Machine**. So *Alice* could be related to *LAPTOP-1* via the role name **hasMachine**.

We can represent facts about a knowledge domain in terms of **axioms** that are logical statements expressing the relationships among concepts, roles, and individuals. The axioms must hold true in any instance of the knowledge domain. For instance, we could write an axiom that states that all instances of the **Person** concept are also instances of the **LivingOrganism**, concept whereas all instances of **Machine** concept are also instances of **Non-LivingOrganism** concept. That would be a subsumption axiom, which we will discuss in more detail later.

Description logics differs from traditional databases in several ways. First, description logics allow us to infer new knowledge from existing knowledge based on the defined axioms. Second, description logics can handle incomplete knowledge using the *Open World Assumption*. Under the **Open World Assumption** (OWA), a fact that is not explicitly stated as true can be either true or false. For example, we can write an axiom that defines a concept **MultiMachineUser** as the set of individuals who are instances of **Person** and are each related to more than two instances of **Machine** via the role name **hasMachine**. Then, under a closed world, *Alice* would not be an instance of **MultiMachineUser**. However, under the OWA, the fact that *Alice* does not have any other machines is not explicitly stated, and thus *Alice* may have other machines. So, *Alice* is an instance of **MultiMachineUser** unless it is asserted explicitly that she has only machine *LAPTOP-1*.

Now, we define an **ontology** as $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, which is expressed in a description logic

$DL(\mathbf{N_I}, \mathbf{N_C}, \mathbf{N_R}, \mathbf{N_F})$, which defines the knowledge domain of the graph G. An ontology consists of two types of axioms: *terminological* axioms $\mathcal{T}$ and *assertional* axioms $\mathcal{A}$. **Terminological** axioms express the "vocabulary of an application domain" using concepts and roles, whereas **assertional** axioms contain "assertions about named individuals in terms of this vocabulary" [27]. More formally, we define the symbols used in the definition of an ontology and description logic below.

$\mathbf{N_I}$: The individual names defined in the DL describe the set of possible identifiers of individuals.

$\mathbf{N_C}$: The concept names defined in the DL describe categories of individuals.

$\mathbf{N_R}$: The role names defined in the DL describes the set of possible relations between individuals.

$\mathbf{N_F}$: The feature names defined in the DL describes the set of possible values in a particular concrete domain $\mathcal{D} = (\Delta^{\mathcal{D}}, \mathcal{P}^{\mathcal{D}})$ where $\Delta^{\mathcal{D}}$ is a set and $\mathcal{P}^{\mathcal{D}}$ is a set of predicate names which refer to e.g., strings and integers.

***ABox* axioms:** Assertional (ABox) axioms describe the properties of individuals.

***TBox* axioms:** Terminological (TBox) axioms describe relationships between concepts and roles.

For example, we define a description logic $DL(\mathbf{N_I}, \mathbf{N_C}, \mathbf{N_R}, \mathbf{N_F})$ that represents the knowledge domain of the graphical data in Figure 3.1. This DL describes an enterprise setting in terms of cyber and physical assets. The people working in the enterprise system are given identities in the cyberworld consisting of their user names. An identity associated with a person is termed a **user account** or **user**, for short when the context is clear. The cyber assets in the system are a shared filesystem organized into directories, and data are stored in files within directories. We model user access in terms of three operations on files and directories: create, write or delete.

The components of the DL are formally given in Table 3.1, 3.2, and 3.3. The entries in each table are highlighted with differing shades of blue indicating the hierarchy of concept

and role names. A darker shade implies that the concept or role name is higher in the hierarchy, and the lighter-colored entries below it are members of the concept or role name.

$$\begin{aligned}\mathbf{N_I} = \{&\text{Person}_1, \text{Person}_2,\\ &\text{Identity}_1, \text{Identity}_2, \text{Identity}_3,\\ &\text{File}_1, \text{File}_2,\\ &\text{Directory}_1, \text{Directory}_2,\\ &\text{Building}_1, \text{Printer}_1, \text{Printer}_2, \}\end{aligned}$$

Concept Name	Description
PrinterGroup	An entity that represents the agglomeration of a printer, the files that were printed to it and the identities that used the printer.
PhysicalWorld	Entities that take a physical form
Person	A human being
Building	A physical building
Printer	A printer device
CyberWorld	Entities that do not have physical form but exist virtually
File	A resource that stores information
Directory	A collection of files or other directories
Identity	A virtual form of identification

Table 3.1: List of concept names in DL.

3.3 Cyber-Physical Topology Language

As mentioned in Section 3.1, we aim to formally define the *CPTL data model*. The CPTL data model informally consists of the following primitives:

Role Name	Description
contains	A **Directory** *contains* another **Directory** or **File**; i.e., a directory has another directory or file inside it
locatesPhysicalDevice	A **Building** *locatesPhysicalDevice* a **Printer**; i.e., the printer device is located within a building
hasIdentity	A **Person** *hasIdentity* an **Identity**; i.e., a person has an associated identity
prints	An **Identity** *prints* a **File** or an **Identity** *prints* to a **Printer** or a **File** is *prints* to a **Printer**; i.e., an identity prints information in a file to a printer
changePrinter	A **PrinterGroup** *changePrinter* to another **PrinterGroup**; i.e., a person in the first **PrinterGroup** printed the same file to the printer in the second **PrinterGroup**
shareFile	A **PrinterGroup** *shareFile* with another **PrinterGroup**; i.e., the same file is printed to the printers in both **PrinterGroup**s
shareIdentity	A **PrinterGroup** *shareIdentity* with another **PrinterGroup**; i.e., the same person printed to the printers in both **PrinterGroup**s
accesses	An **Identity** *accesses* a **Directory** or **File**; i.e., an identity interacts with a directory or file
write	An **Identity** *writes* a **File**; i.e., an identity writes information to a file
create	An **Identity** *create* a **Directory** or **File**; i.e., an identity makes a new directory or file
delete	An **Identity** *delete* a **Directory** or **File**; i.e., an identity removes a directory or file

Table 3.2: List of role names in DL.

Feature Name	Description
name	A *String* used for identification of an individual
numIdentity	An *Integer* that denotes the number of identities contained in an entity
numDelete	An *Integer* that denotes the number of deletions made by an individual to information
numInsert	An *Integer* that denotes the number of additions made by an individual to information
numFiles	An *Integer* that denotes the number of files contained in an entity

Table 3.3: List of feature names in DL.

- a *graph*, which is an ordered triple consisting of a set of vertices and edges, and a function mapping edges to endpoints. Vertices represent instances of objects in a system, whereas edges represent the actual relationship between the objects.

- an *ontology*, which represents the domain knowledge. The ontology contains *axioms* that assert general facts about the objects and relations among the objects in a target system.

- an *interpretation*, which maps the ontology to the graph. Then, we can reason about the graph using axioms defined in the ontology.

More formally, we define the CPTL data model as follows.

Definition 2 *A* ***CPTL data model*** *is*

$$(G, \mathcal{K})_{\mathcal{I}} \tag{3.2}$$

where G is a graph $G(V, E, h)$, $\mathcal{K}$ is an ontology expressed using description logic DL, and $\mathcal{I}$ is an interpretation of G that is a model of a subset of axioms $\mathbb{V}$ in $\mathcal{K}$.

In the next subsection, we will slowly introduce more detail about this formal definition of a CPTL data model.

First, we want to analyze the graphical data with respect to a particular knowledge domain. Thus, we represent the graphical data using axioms in the ontology.

We construct *ABox* axioms to assert facts about the vertices and edges in the graph G. In particular, we construct axioms to assert that each vertex is an instance of a concept whereas each edge is an instance of a role.

ABox axioms consist of concept assertions and role assertions. Concept assertions are of the form $C(a)$, which implies that the individual a is an instance of the concept C. These individuals correspond to vertices in the graph. Role assertions are of the form $R(a, b)$, which implies that the individual a is related to b by the role R. These roles correspond to edges in the graph.

Now, we can represent G in Figure 3.1 using the following *ABox* axioms.[1]

Person(Person_1)	**File**(File_2)
Person(Person_2)	**Directory**(Directory_1)
Identity(Identity_1)	**Directory**(Directory_2)
Identity(Identity_2)	**Building**(Building_1)
Identity(Identity_3)	**Printer**(Printer_1)
File(File_1)	**Printer**(Printer_2)

hasIdentity($\text{Person}_1, \text{Identity}_2$)	**prints**($\text{Identity}_3, \text{File}_2$)
hasIdentity($\text{Person}_2, \text{Identity}_1$)	**prints**($\text{Identity}_3, \text{File}_2$)
hasIdentity($\text{Person}_2, \text{Identity}_3$)	**contains**($\text{Directory}_1, \text{File}_1$)
create($\text{Identity}_1, \text{File}_1$)	**contains**($\text{Directory}_1, \text{File}_2$)
write($\text{Identity}_1, \text{File}_2$)	**contains**($\text{Directory}_2, \text{Directory}_1$)
prints($\text{File}_2, \text{Printer}_2$)	**prints**($\text{Identity}_3, \text{Printer}_1$)
write($\text{Identity}_2, \text{File}_1$)	**locatesPhysicalDevice**($\text{Building}_1, \text{Printer}_1$)
prints($\text{File}_2, \text{Printer}_1$)	**locatesPhysicalDevice**($\text{Building}_1, \text{Printer}_2$)
write($\text{Identity}_3, \text{File}_2$)	**prints**($\text{Identity}_3, \text{Printer}_2$)

The *ABox* axioms defined in the ontology deal with *explicit knowledge* about the individuals in the graph G. **Explicit knowledge** represents facts that have been explicitly asserted. In the following subsection, we will discuss how to represent *implicit knowledge* about the graph G. **Implicit knowledge** represents facts that are not explicitly asserted but rather have been inferred based on explicit knowledge and terminological *TBox* axioms.

[1]The role **prints** is originally a 4-ary relation. However, our definition of CPTL data model only covers binary relations. So, we model the role **prints** as three binary edges as shown in Figure 3.2.

3.3.1 Presenting Views of the Graphical Data

Now we are able to relate a graph to a knowledge domain. We also want to present different views of the graphical data for different applications. For example, users in different roles may use different views, or algorithms may be interested in different aspects of a target system. We formalize a view of a graph as an *interpretation*.

In traditional description logics, an **interpretation** $\mathcal{I}$ maps the ontology to the data in the domain of discourse, $\Delta^{\mathcal{I}}$, which is a set of individuals. That mapping function (or interpretation function) is represented by $\cdot^{\mathcal{I}}$. The interpretation function $\cdot^{\mathcal{I}}$ maps each concept name $A \in \mathbf{N_C}$ to a subset $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, each role name $r \in \mathbf{N_R}$ to a binary relation $r^{\mathcal{I}}$ on $\Delta^{\mathcal{I}}$, each individual name $a \in \mathbf{N_I}$ to an individual $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, and each feature name $f \in \mathbf{N_F}$ to a function $f^{\mathcal{I}}$ from $\Delta^{\mathcal{I}}$ to $\bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}$.

In contrast, our data do not consist of a set of individuals. Graphical data reflect existing relations between individuals. Those relations need to be preserved in the interpretation. So we want to be able to map the ontology to graphical data in the form $G = (V, E, h)$ mentioned in Section 3.2.1. Then, the vertices V correspond to $\Delta^{\mathcal{I}}$. The existence of edges E is modeled in the *ABox* axioms, and these axioms can be translated to the appropriate mapping in the interpretation function.

In addition, we want the interpretation $\mathcal{I}$ to be a *model* of a subset of axioms $\mathbb{V} \subseteq \mathcal{K}$ in the ontology $\mathcal{K}$. An axiom α holds in $\mathcal{I}$ (or $\mathcal{I}$ satisfies α) if the semantics of the axiom are satisfied by $\mathcal{I}$. Then, $\mathcal{I}$ is a **model** of a set of axioms if all the axioms are satisfied by $\mathcal{I}$. In other words, a model is an "abstraction of a state of the world that satisfies the set of axioms" [28]. In particular, $\mathbb{V}$ contains all the *ABox* axioms and a subset of *TBox* axioms $\mathbb{T}'$ in $\mathcal{K}$. We can express that more formally as $\mathcal{I} \models \mathbb{V}$, i.e., for every axiom $\alpha \in \mathbb{V}$, $\mathcal{I} \models \alpha$. This restriction on the interpretation is necessary to ensure that the graphical data are consistent with a subset of the ontology that represents the knowledge possessed by a user. Therefore, the interpretation satisfies *ABox* and a subset of *TBox* axioms.

Explicit knowledge. The semantics of the *ABox* axioms that describe the graph G is defined in the interpretation as follows. Concept assertions $C(a)$ translate to $a^{\mathcal{I}} \in C^{\mathcal{I}}$, whereas

role assertions translate to $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$ in the interpretation $\mathcal{I}$. Thus, the interpretation $\mathcal{I}$ covers the explicit knowledge expressed in *ABox* axioms.

Implicit knowledge. The interpretation is also able to represent implicit knowledge possessed by a user role. The implicit knowledge is expressed as a set of *TBox* axioms $\mathbb{T}' \subseteq \mathbb{V}$ that is a subset of the *TBox* axioms in $\mathcal{K}$, i.e., $\mathbb{T}' \subseteq \mathbb{A}$ where $\mathbb{A}$ represents the *TBox* axioms in $\mathcal{K}$.

TBox axioms consist of concept inclusion, concept equality, role inclusion, and role equality. The axioms are listed in Table 3.4 along with their syntax and semantics.

Axiom	Syntax	Semantics
Concept inclusion	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
Concept equality	$C \equiv D$	$C^{\mathcal{I}} = D^{\mathcal{I}}$
Role inclusion	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
Role equality	$R \equiv D$	$R^{\mathcal{I}} = S^{\mathcal{I}}$

Table 3.4: *TBox* axioms

Concept and role inclusion describe subsumption of concepts and roles, whereas concept and role equality describe equivalence of concepts and roles, respectively. In Table 3.4, concept C is subsumed by concept D which implies that an individual that is an instance of concept C is also an instance of concept D. Similarly, if two individuals are related by role R, they are also related by role S. Equivalence of concepts C and D implies that C is subsumed by D and D is subsumed by C. Similarly, role equivalence implies that R is subsumed by S and S is subsumed by R.

We describe the hierarchy of concepts and roles using concept and role inclusion, respectively in Table 3.1, 3.2.

$$\textbf{Person}, \textbf{Building}, \textbf{Printer} \sqsubseteq \textbf{PhysicalWorld}$$

$$\textbf{File}, \textbf{Directory}, \textbf{Identity} \sqsubseteq \textbf{CyberWorld}$$

$$\textbf{write}, \textbf{create}, \textbf{delete} \sqsubseteq \textbf{accesses}$$

The semantics of the *TBox* axioms reflect how the axioms are translated in the interpretation. Thus, *TBox* axioms express additional mappings of concept names to vertices and role names to edges.

Thus, the interpretation function $\cdot^{\mathcal{I}}$ maps concept names $A \in \mathbf{N_C}$ to a subset $A^{\mathcal{I}}$ of vertices V, role names $r \in \mathbf{N_R}$ to a subset $r^{\mathcal{I}}$ which is an edge in E if r appears in *ABox* axioms, individual names $a \in \mathbf{N_I}$ to a vertex $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, and feature names $f \in \mathcal{N}_{\mathcal{F}}$ to vertex attributes.[2]

The *ABox* axioms given in the previous section can then be translated into an interpretation function $\mathcal{I}$ as follows. If we added the axiom $\mathbf{Person} \sqsubseteq \mathbf{PhysicalWorld}$ to the set $\mathbb{V}$, then the interpretation would be modified to include $\cdot^{\mathcal{I}}(\mathbf{PhysicalWorld}) \rightarrow \{v_1, v_2\}$.

$$\Delta^{\mathcal{I}} = \{v_1, v_2, \ldots, v_{12}\} \tag{3.3}$$

$$\begin{aligned}
\cdot^{\mathcal{I}}(\mathbf{Person}) &\rightarrow \{v_1, v_2\} \\
\cdot^{\mathcal{I}}(\mathbf{Identity}) &\rightarrow \{v_3, v_4, v_5\} \\
\cdot^{\mathcal{I}}(\mathbf{File}) &\rightarrow \{v_6, v_7\} \\
\cdot^{\mathcal{I}}(\mathbf{Printer}) &\rightarrow \{v_8, v_{10}\} \\
\cdot^{\mathcal{I}}(\mathbf{Building}) &\rightarrow \{v_{11}\} \\
\cdot^{\mathcal{I}}(\mathbf{Directory}) &\rightarrow \{v_9, v_{12}\}
\end{aligned}$$

$$\begin{aligned}
\cdot^{\mathcal{I}}(\mathbf{hasIdentity}) &\rightarrow \{e_1, e_2, e_3\} \\
\cdot^{\mathcal{I}}(\mathbf{locatesPhysicalDevice}) &\rightarrow \{e_{16}, e_{17}\} \\
\cdot^{\mathcal{I}}(\mathbf{create}) &\rightarrow \{e_4\} \\
\cdot^{\mathcal{I}}(\mathbf{write}) &\rightarrow \{e_5, e_7, e_9\} \\
\cdot^{\mathcal{I}}(\mathbf{contains}) &\rightarrow \{e_{12}, e_{13}, e_{14}\} \\
\cdot^{\mathcal{I}}(\mathbf{prints}) &\rightarrow \{e_6, e_8, e_{10}, e_{11}, e_{15}, e_{18}\}
\end{aligned}$$

[2]We make the Unique Name Assumption (UNA) unlike traditional DL.

$$\cdot^{\mathcal{I}}(\text{Person}_1) \to \{v_1\} \qquad \cdot^{\mathcal{I}}(\text{File}_2) \to \{v_7\}$$
$$\cdot^{\mathcal{I}}(\text{Person}_2) \to \{v_2\} \qquad \cdot^{\mathcal{I}}(\text{Printer}_1) \to \{v_8\}$$
$$\cdot^{\mathcal{I}}(\text{Identity}_1) \to \{v_3\} \quad \cdot^{\mathcal{I}}(\text{Directory}_1) \to \{v_9\}$$
$$\cdot^{\mathcal{I}}(\text{Identity}_2) \to \{v_4\} \qquad \cdot^{\mathcal{I}}(\text{Printer}_2) \to \{v_{10}\}$$
$$\cdot^{\mathcal{I}}(\text{Identity}_3) \to \{v_5\} \quad \cdot^{\mathcal{I}}(\text{Building}_1) \to \{v_{11}\}$$
$$\cdot^{\mathcal{I}}(\text{File}_1) \to \{v_6\} \quad \cdot^{\mathcal{I}}(\text{Directory}_2) \to \{v_{12}\}$$

$$\cdot^{\mathcal{I}}(\text{name}) \to f_1$$
$$\cdot^{\mathcal{I}}(\text{timestamp}) \to f_2$$
$$\cdot^{\mathcal{I}}(\text{numInsert}) \to f_3$$
$$\cdot^{\mathcal{I}}(\text{numDelete}) \to f_4$$
$$\cdot^{\mathcal{I}}(\text{numPages}) \to f_5$$

$$f_1(v_1) = \text{Alice} \qquad f_1(v_7) = \text{b.doc}$$
$$f_1(v_2) = \text{Bob} \qquad f_1(v_8) = \text{Printer1}$$
$$f_1(v_3) = \text{bob1} \qquad f_1(v_9) = \text{Documents}$$
$$f_1(v_4) = \text{alice1} \quad f_1(v_{10}) = \text{Printer2}$$
$$f_1(v_5) = \text{bob2} \quad f_1(v_{11}) = \text{Building1}$$
$$f_1(v_6) = \text{a.doc} \quad f_1(v_{12}) = \text{CompanyDoc}$$

$$f_2(e_4) = \text{2014-02-10:10AM} \quad f_2(e_9) = \text{2014-10-20:3PM} \quad f_3(e_4) = 50 \quad f_4(e_4) = 0$$
$$f_2(e_5) = \text{2014-06-02:11AM} \quad f_2(e_{10}) = \text{2014-05-28:1PM} \quad f_3(e_5) = 10 \quad f_4(e_5) = 2$$
$$f_2(e_6) = \text{2014-05-28:1PM} \quad f_2(e_{11}) = \text{2014-05-28:2PM} \quad f_3(e_7) = 2 \quad f_4(e_7) = 0$$
$$f_2(e_7) = \text{2014-02-15:4PM} \quad f_2(e_{15}) = \text{2014-05-28:1PM} \quad f_3(e_9) = 8 \quad f_4(e_9) = 10$$
$$f_2(e_8) = \text{2014-05-28:2PM} \quad f_2(e_{18}) = \text{2014-05-28:2PM}$$

$$f_5(e_6) = 5 \quad f_5(e_{11}) = 1$$
$$f_5(e_8) = 1 \quad f_5(e_{15}) = 5$$
$$f_5(e_{10}) = 5 \quad f_5(e_{18}) = 1$$

The graphical data can now be analyzed by a user U with respect to a particular knowledge domain given by the ontology $\mathcal{K}$ expressed in DL.

Modeling edge attributes. Edge attributes and multi-edges are not modeled directly by the ontology. Instead, we model edge attributes by using *reified* relations and meta-roles to transform the description logic $DL(\mathbf{N_I}, \mathbf{N_C}, \mathbf{N_R}, \mathbf{N_F})$ and interpretation $\mathcal{I}(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ to the description logic $DL'(\mathbf{N'_I}, \mathbf{N'_C}, \mathbf{N'_R}, \mathbf{N'_F})$ and the interpretation $\mathcal{I}'(\Delta^{\mathcal{I}'}, \cdot^{\mathcal{I}'})$. That transformation can similarly model hyperedges as N-ary relations. **Reification** is the process of representing relations as a concept instead of a role [27].

We use the most straightforward strategy that is described as a design pattern in Ontology Design Patterns (ODP) catalog [29] as well as in World Wide Web Consortium (W3C) Working Group Note [30]. We first transform the description logic $DL(\mathbf{N_I}, \mathbf{N_C}, \mathbf{N_R}, \mathbf{N_F})$ to $DL'(\mathbf{N'_I}, \mathbf{N'_C}, \mathbf{N'_R}, \mathbf{N'_F})$ by adding concept names representing edges, role names representing the size of edges, and individual names representing the individual edges.

For an $r_i \in \mathbf{N_R}$, we construct the following additional concepts, roles, individual and feature names in DL'. For each role name, we add a concept name that represents the role. We add two role names that represent the outward bound edge and the inward bound edge. Finally, we add an individual name that represents the presence of an edge. This is described formally below.

$$\mathbf{N'_C} = \mathbf{N_C} \cup \mathbf{N_e} \text{ where } |\mathbf{N_e}| = |\mathbf{N_R}|$$
$$|\mathbf{N'_R}| = 2|\mathbf{N_R}|$$
$$\mathbf{N'_I} = \mathbf{N_I} \cup \mathbf{N_i} \text{ where } |\mathbf{N_i}| = |E|$$

$$c_i \in \mathbf{N_e}$$
$$r_i^1, r_i^2 \in \mathbf{N'_R}$$
$$e_i \in \mathbf{N_i}$$

To illustrate the process, we pick an edge e_5 that is an instance of the role **write** based on our interpretation function defined earlier. Then, we add the following concepts, roles, and individual names.

$$\mathbf{Write} \in \mathbf{N_e}$$
$$\mathbf{writeTo}, \mathbf{writeFrom} \in \mathbf{N'_R}$$
$$\text{Write}_1 \in \mathbf{N_i}$$

The additional elements in DL' are described as follows to model the existence of an edge. The role names are subsumed by the old role name in the following manner.

$$r_i^1, r_i^2 \sqsubseteq r_i \tag{3.4}$$

We add an assertion that the concept **Write** representing the role **write** must be linked to the relata of **write**; i.e., **Write** must have exactly one relation **writeTo** and exactly one relation **writeFrom**. The concept name is constructed as follows.

$$c_i \sqsubseteq (\geq 1 r_i^1.A \sqcap \leq 1 r_i^1.A) \sqcap (\geq 1 r_i^2.B \sqcap \leq 1 r_i^2.B)$$

Finally, we modify the *ABox* axioms representing the graphical data. For each edge e_i, we add the concept assertion $c_i(e_i)$. If the old *ABox* axiom contains the role assertion $r_i(a, b)$, we transform the axiom to $r_i^1(e_i, a)$ and $r_i^2(e_i, b)$.

Thus, for the individual edge given above, we modify the following axioms.

$$\textbf{Write}(\text{Write}_1)$$

$$\textbf{writeFrom}(\text{Write}_1, \text{Identity}_2)$$

$$\textbf{writeTo}(\text{Write}_1, \text{File}_1)$$

In other words, an edge can be associated with a new concept in DL'. For binary relations, the new concept has exactly two relations; each relation is between the individuals that were originally incident to the edge. For n-ary relations, the new concept has exactly n relations; each relation is between the individuals that were in the ordered set related to the edge.

Thus, we recapitulate our definition of a **CPTL data model** $(G, \mathcal{K})_{\mathcal{I}}$ as an interpretation $\mathcal{I}$ of a graph G that is a model of a subset of axioms $\mathbb{V}$ in the ontology $\mathcal{K}$.

The CPTL data model of Figure 3.1 is illustrated below in Figure 3.2. We can see that edges with edge attributes are modeled as additional vertices connected to the incident vertices of the edge. For example, edge e_7 is modeled as v_{14} with edges $e_{7.1}$ and $e_{7.2}$.

Inverse mapping of ontology to data. Now that we have defined a mapping from the ontology $\mathcal{K}$ to the data G using the interpretation $\mathcal{I}$, we can specify the inverse mapping from the data to the ontology. We use the inverse mapping when we perform operations on the CPTL data model.

We define five relations as follows.

$T_V : V \rightarrow \mathbf{N_C}$**:** This is a relation mapping vertices in the graph to concept names, i.e., $T_V(v) = \{\mathbf{c} | v \in \mathbf{c}^{\mathcal{I}}, \mathbf{c} \in \mathbf{N_C}\}$.

$T_E : E \rightarrow \mathbf{N_R}$**:** This is a relation mapping edges in the graph to role names, i.e., $T_E(e) = \{\mathbf{r} | e \in \mathbf{r}^{\mathcal{I}}, \mathbf{r} \in \mathbf{N_R}\}$.

$T_L : V \rightarrow \mathbf{N_I}$**:** This is a function mapping vertices in the graph to individual names, i.e., $T_L(v) = \mathbf{a}$ where $v = \mathbf{a}^{\mathcal{I}}, a \in \mathbf{N_I}$.

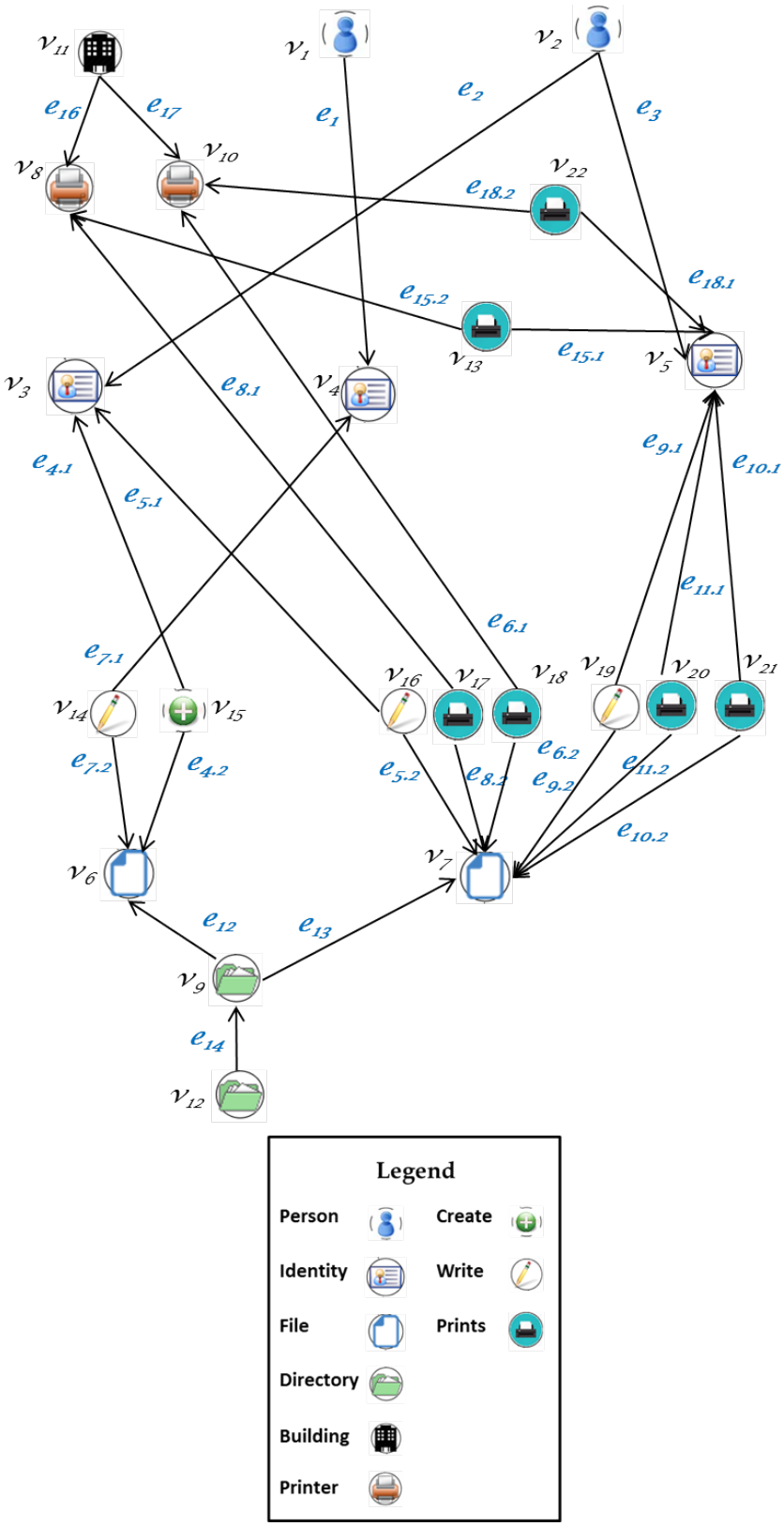


Figure 3.2: The CPTL data model $(G, \mathcal{K})_{\mathcal{I}}$. The vertices are labeled as v_i, and the edges are labeled as e_i. The concept name of each individual vertex is indicated by the icon and a legend is provided.

$F_V : V \rightarrow \mathbf{N_F}$**:** This is a relation mapping vertices in the graph to feature names, i.e., $F_V(v) = \{\mathbf{f} | v \in \mathit{Domain}(\mathbf{f}^{\mathcal{I}}), \mathbf{f} \in \mathbf{N_F}\}$.

$F_E : E \rightarrow \mathbf{N_F}$**:** This is a relation mapping edges in the graph to feature names, i.e., $F_E(e) = \{\mathbf{f} | e \in \mathit{Domain}(\mathbf{f}^{\mathcal{I}'}), \mathbf{f} \in \mathbf{N_F}\}$, and $\mathcal{I}'$ is the transformation of interpretation $\mathcal{I}$ to model edge attributes.

3.4 Operations on CPTL

An operation in CPTL creates a new data model from old data models, i.e., $f : X \rightarrow (G, \mathcal{K}_{n+1})_{\mathcal{I}_{n+1}}$ where $X \subset (G, \mathcal{K}_1)_{\mathcal{I}_1} \times (G, \mathcal{K}_2)_{\mathcal{I}_2} \times \cdots (G, \mathcal{K}_n)_{\mathcal{I}_n}$. The arity of the operation is determined by n, where $n = 1$ denotes a unary operation, $n = 2$ denotes a binary operation, and so on.

An operator is characterized by the (1) topological, (2) property, and (3) semantic changes applied to the data models. Topological changes involve modifying the structure of the graphs G_i, whereas property changes involve modifying the attributes of the vertices and edges. Semantic changes involve modifying the ontology K_i and/or the $\mathcal{I}_i$ in the data models.

Current operation set. We define a set of operations on CPTL: **Abstract**, **Join**, and **Contract**.

Abstract is a *unary* operator that takes a single CPTL model as input. We will formally define unary operators in Section 3.4.1. The operator **Abstract** involves only semantic changes; i.e., no changes are made to the graph structure. Instead, the interpretation is modified such that the current concept, role, and feature names used to describe the graph are replaced with concept, role, and feature names that subsume the current ones. The resultant CPTL model represents a higher-level description of the original CPTL model. The **Abstract** operation can be used to extract high-level descriptions of a graph.

Join is a *binary* operator that takes two CPTL models as input. The operator **Join** involves topological, semantic, and property changes. Vertices and edges from both CPTL models are merged together by a graph union operation in the resultant CPTL model. The

interpretation is modified to describe the combination of the two input CPTL models. We can use the **Join** operation to relate different views of a target system or add new individuals and relations to the model.

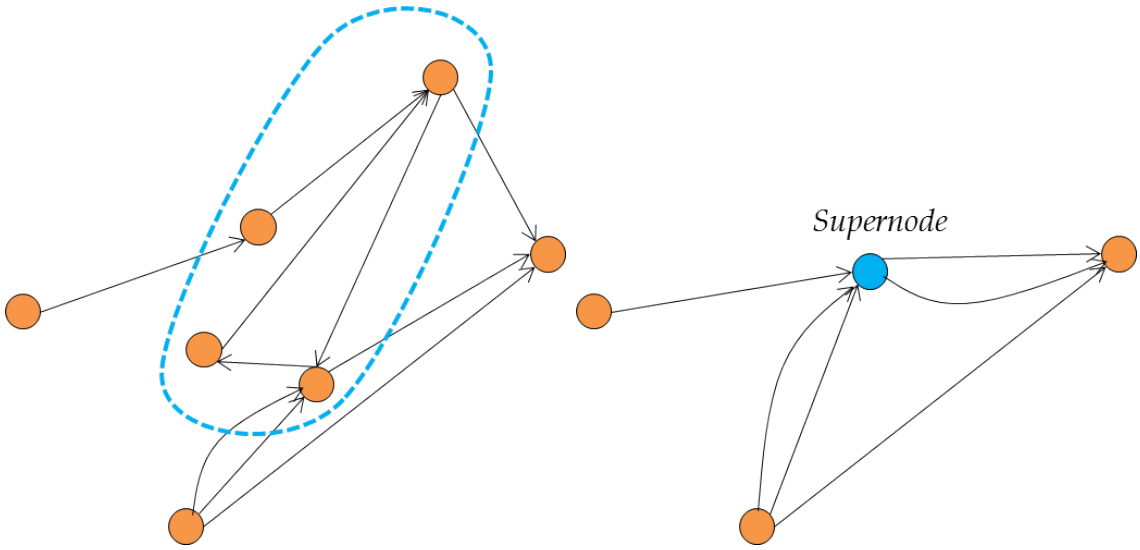
Finally, **Contract** is a *unary* operator that takes a single CPTL model as input. The operator involves topological, semantic, and property changes. It takes a set of vertices in the model and merges the set into a single vertex in the resultant CPTL model. That resulting vertex represents the set of vertices and inherits their neighborhood as well as a summary of their features. We can use **Contract** to generate different views of the target system and extract higher-level features of the graph.

We introduce the **Contract** operator as vertex contraction in the following subsections.

3.4.1 Unary Operations

Unary operations take in a single CPTL data model $(G, \mathcal{K})_{\mathcal{I}}$ to produce a new data model $(G', \mathcal{K}')_{\mathcal{I}'}$. We define a certain class of unary operations that are useful in the applications that we introduced earlier on. A unary operation is called **semantic-preserving** if the interpretation of the new graph G' is a model of the ontology that described the old graph G, i.e., $\mathcal{K} = \mathcal{K}'$. Two ontologies $\mathcal{K}$ and $\mathcal{K}'$ are equal if the following equations hold true.

$$
\begin{aligned}
\mathcal{K} &=< \mathcal{T}, \mathcal{A} > \text{ expressed using } DL(\mathbf{N_I}, \mathbf{N_C}, \mathbf{N_R}, \mathbf{N_F}) \\
\mathcal{K}' &=< \mathcal{T}', \mathcal{A}' > \text{ expressed using } DL'(\mathbf{N'_I}, \mathbf{N'_C}, \mathbf{N'_R}, \mathbf{N'_F}) \text{ and} \\
\mathbf{N_C} &= \mathbf{N'_C} \\
\mathbf{N_R} &= \mathbf{N'_R} \\
\mathbf{N_F} &= \mathbf{N'_F} \\
\mathcal{T} &= \mathcal{T}' \\
\mathcal{A} &= \mathcal{A}'
\end{aligned}
$$



(a) Original graph. (b) Resultant graph.

Figure 3.3: The original graph is given in (a). (b) shows the graph that results from the first contraction of the vertices within the blue region.

In other words, the interpretations of G' and G are models of $\mathcal{K}$. That implies that G' follows the same restrictions and constraints as G. A semantic-preserving operation is useful because it ensures that the resultant graph belongs to the same knowledge domain as the original graph.

In the remainder of this section, we describe how vertex contraction is a semantic-preserving operation with a useful application.

3.4.2 Vertex Contraction

Contraction operations produce a graph that is smaller than the initial graph by replacing a subset of vertices with a single vertex while preserving incident edges. The resultant graph differs topologically and in terms of the vertex and edge attributes. Figure 3.3 shows an example of topological contraction.

We introduce three different types of vertex contraction: *basic vertex contraction*, *extended vertex contraction*, and *extended vertex contraction with edge attributes*. Basic vertex contraction focuses on defining the supernode and its attributes, whereas extended vertex contraction additionally defines the edges incident to the supernode. Finally, extended ver-

tex contraction with edge attributes expands on extended vertex contraction by additionally defining the edge attributes. Basic vertex contraction is useful for summarization of a group of individuals; for example, for summarizing the level of security of a subnet using levels of patching on each machine in that subnet. We first define the basic vertex contraction operation, which is a lightweight operator with less expressivity but lower complexity. Then, we move on to define extended vertex contraction operation, which has higher expressivity that comes at the expense of higher complexity. Extended vertex contraction is useful for summarization of relations among a group of individuals; for example, for summarizing the number of modifications made to a group of files.

Basic vertex contraction. Informally, a vertex contraction of a set of vertices V_C in G produces a graph G' whose vertices and edges are identical to those of G with the exception of vertices in V_C. The set of vertices is replaced with a single vertex v' called the supernode.

We describe the **supernode** in terms of its (1) topology, (2) semantic meaning, and (3) properties. The supernode (1) is a vertex that is adjacent to the union of the neighborhood of vertices in V_C as illustrated in Figure 3.3, (2) is an instance of a concept C that is consistent with respect to the axiom $\mathbb{T}$, and (3) has attribute values that are defined as an aggregate representation of the selected vertices' attribute values given by the set of functions $\mathbf{F}$.

The basic vertex contraction involves defining the supernode and its attributes as a function of the selected vertices for contraction. Since the vertices are intentionally selected for contraction, the concept names of the vertices are known to be semantically related. In addition, the supernode is assigned a single concept name that represents the group of vertices selected. Thus, the concept name of the supernode is derived by substituting the concept names of the selected vertices into a defined function. Similarly, the feature names of the vertex attributes are known, because the vertices are intentionally selected for contraction. In addition, the supernode's attributes represent the aggregate of the attributes of the selected vertices. Thus, the feature values of the supernode are derived by substituting the feature values of the selected vertices into defined functions. The supernode is assigned feature names that represent the aggregate feature values.

We add a validity check to ensure that both (1) the supernode and edges incident to the

supernode and (2) the supernode's feature names and values are also consistent with respect to the axioms in the ontology $\mathcal{K}$. Therefore, the resulting interpretation is still a model of $\mathcal{K}$. So the semantic meaning of the new graph G' is still preserved.

The formal definition of basic vertex contraction is given below.

Definition 3 Basic vertex contraction *is a function* $\textbf{Contract}(V_C, (G, \mathcal{K})_{\mathcal{I}}, \mathbb{T}, \mathbf{F}) = (G', \mathcal{K})_{\mathcal{I}'}$ *where* $V_C \subseteq V(G)$.

Notation:

V_C**:** *The subset of vertices in the graph selected for contraction.*

$\mathbb{T}$**:** *The TBox axiom describing the concept name of the supernode.*

F: *The function describing the feature names of the supernode and their associated values, i.e.,* $\mathbf{F}$ *maps* $\mathbf{N_F}$ *to* $g : (\{(\mathbf{N_C}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i})\}) \rightarrow \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}$.

$N^+(W)$**:** *The neighborhood of vertices in* W *that have an outgoing edge from some vertex in* W*, i.e.,* $N^+(W) = \{v | e_i(v_1, v) \in E(G)$ *and* $v_1 \in W \subseteq V(G), v \notin W\}$.

$N^-(W)$**:** *The neighborhood of vertices in* W *that have an incoming edge from some vertex in* W*, i.e.,* $N^-(W) = \{v | e_i(v, v_1) \in E(G)$ *and* $v_1 \in W \subseteq V(G), v \notin W\}$.

Topological changes: *The resulting graph* G' *is described as follows:*

$$V(G') = V(G) - V_C + \{v'\} \tag{3.5}$$

$$E(G') = E(G) - \{e_i(v_1, v_2) | e_i \in E(G), v_1 \in V_C \text{ or } v_2 \in V_C\} + \tag{3.6}$$

$$\{e_i(v', v_1) | e_i \in E(G), v_1 \in N^+(V_C)\} + \{e_i(v_2, v') | e_i \in E(G), v_2 \in N^-(V_C)\} \tag{3.7}$$

The new vertex in G'*,* v'*, is called a* **supernode**.

Semantic changes: *The interpretation* $\mathcal{I}' = (\Delta^{\mathcal{I}'}, \cdot^{\mathcal{I}'})$ *is constructed as follows.*[3]

[3]We use the inverse mappings from Section 3.3.1 here.

$$\Delta^{\mathcal{I}'} = V(G') \tag{3.8}$$

$$\cdot^{\mathcal{I}'}(c) = \begin{cases} \cdot^{\mathcal{I}}(c) - \{v | c \in T_V(v), v \in V_C\} & \textit{where } c \in \{c_i | c_i \in T_V(v), v \in V_C\} \subseteq \mathbf{N_C} \\ \cdot^{\mathcal{I}}(c) & \textit{otherwise} \end{cases} \tag{3.9}$$

$$\cdot^{\mathcal{I}'}(r) = \begin{cases} \cdot^{\mathcal{I}}(r) - \{e_p(v_i, v_j) | r \in T_E(e_p(v_i, v_j)), v_i \in V_C \textit{ or } v_j \in V_C\} + \\ \quad \{e_q(v', v_j), e_r(v_i, v') | r \in T_E(e_q(v_k, v_j)) \cap T_E(e_r(v_i, v_m)), \\ \quad\quad v_k, v_m \in V_C \textit{ and } v_j, v_i \notin V_C\} \\ \quad\quad\quad \textit{where } r \in \{r_i | r_i \in T_E(e_p(v_i, v_j)), v_i \in V_C \textit{ or } v_j \in V_C\} \subseteq \mathbf{N_R} \\ \cdot^{\mathcal{I}}(r) \quad\quad \textit{otherwise} \end{cases} \tag{3.10}$$

$$\cdot^{\mathcal{I}'}(a) = \begin{cases} \phi & \textit{where } a \in \{a_i | T_L(v) = a_i, v \in V_C\} \subseteq \mathbf{N_I} \\ \cdot^{\mathcal{I}}(a) & \textit{otherwise} \end{cases} \tag{3.11}$$

$$\cdot^{\mathcal{I}'}(f) = \begin{cases} f^{\mathcal{I}'} : (G^{\mathcal{I}} = F^{\mathcal{I}} - (F^{\mathcal{I}} \cap V_C)) \rightarrow \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}, \\ \quad f^{\mathcal{I}'}(g) = f^{\mathcal{I}}(g) \\ \quad\quad \textit{where } f \in \{f_i | f_i \in A_V(v), v \in V_C\} \subseteq \mathbf{N_F} \\ \cdot^{\mathcal{I}}(f) \quad\quad \textit{otherwise} \end{cases} \tag{3.12}$$

We assign a concept name to the supernode v', i.e., v' is an instance of some concept C, such that C is consistent w.r.t. a given TBox axiom $\mathbb{T}$. The axiom $\mathbb{T}$ is a function that describes the relation between the concept name of the supernode and the concept names of the vertices in V_C. Then, we modify the interpretation in the following manner.

$$\cdot^{\mathcal{I}'}(C) = \cdot^{\mathcal{I}'}(C) \cup \{v'\} \tag{3.13}$$

Property changes: *For each feature name $f \in Domain(\mathbf{F})$, we calculate the attribute value of the supernode v' using the function $g = \mathbf{F}(f)$. The function g takes in as arguments the concept names of the selected vertices and their attribute values corresponding*

to a particular feature name $h \in \mathbf{N_F}$ that is semantically related to f. The output of g is $d \in \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}$. Then, we augment the interpretation in the following manner.

$$[\cdot^{\mathcal{I}'}(f)](v') = d \tag{3.14}$$

Validity check: *After assigning a concept name to the supernode and role names to the edges incident to the supernode, we run a reasoner to verify that the resulting interpretation $\mathcal{I}'$ is still a model of the ontology $\mathcal{K}$.*

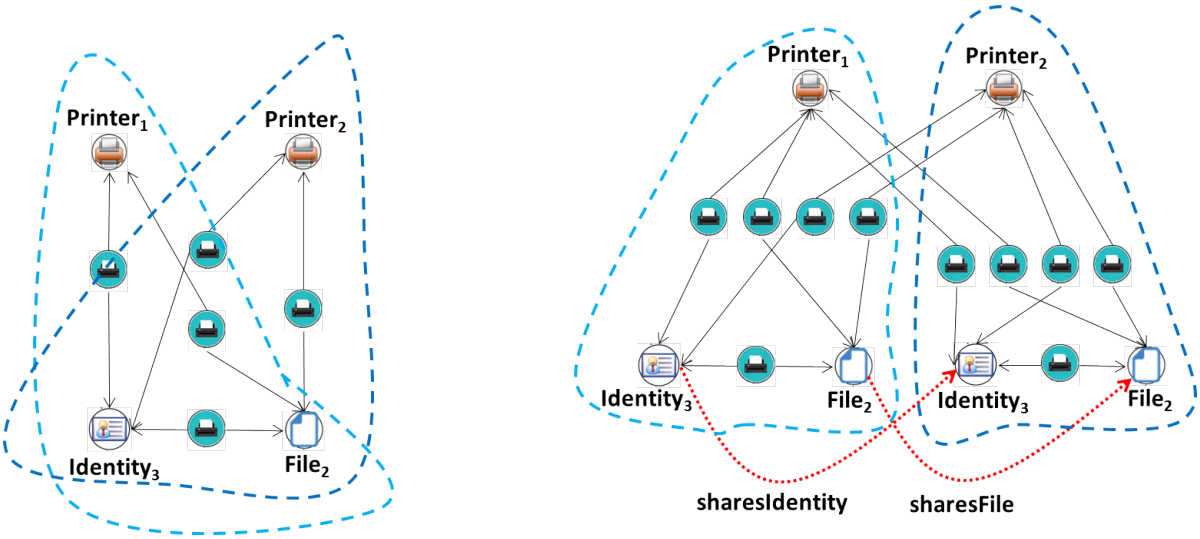
Example: In our enterprise system example in Chapter 5, employees print files to printers. When a person uses a printer other than his or her usual default printer choice, that may signify data exfiltration. However, that assumption could result in a false positive, perhaps if the unusual printer was used only because the default printer was not functioning. So we use contraction to infer the status of the printers.

Our hypothesis is that if a person prints a file F to printer P_1 and then subsequently prints F to another printer, P_2, it typically means that P_1 was not available. We will now look at a small snippet of the graph in Figure 3.2 and focus on the constituent print relations, along with the people and files associated with the printing.

In Figure 3.4b, two groups of vertices are selected for contraction. We give a detailed explanation of how multiple sets of vertices are contracted in Definition 6. Since the vertices selected from the two groups for contraction overlap, we transform the graph as shown in Figure 3.4b (see Definition 6 for more detail) so that the overlapping vertices (and its incident edges) are duplicated. Each group now contains distinct vertices. To model the fact that the duplicated vertices are the same entity, we add an edge of role name **sharesC** between them where **C** is a filler for the concept name of the original vertex. Then, we can contract the vertices as shown in Figure 3.4a and express the $\mathbb{T}$ axiom as

$$X \sqsubseteq (=1)\mathbf{numFiles}.=_{\text{count}\{F_1,\ldots,F_m\}} \sqcap (=1)\mathbf{numIdentity}.=_{\text{count}\{I_1,\ldots,I_k\}} \tag{3.15}$$

where $F_1, \ldots, F_m \in V_C$ are instances of **File** and $I_1, \ldots, I_k \in V_C$ are instances of **Identity**.



(a) Original graph. (b) Resultant graph after transformation.

Figure 3.4: The original graph is given in (a). (b) shows the transformed graph. The labels below the vertices represent their individual name whereas the labels below the edges represent their role name.

This axiom asserts that the supernode is related to exactly one **numFiles** feature with a value of the count of the number of files, and exactly one **numIdentity** feature with a value of the count of the number of credentials. The concept **PrinterGroup** is defined as $\mathbf{PrinterGroup} \equiv (= 1)\mathbf{numFiles}.\ \geq_0 \sqcap (= 1)\mathbf{numIdentity}.\ \geq_0$. So the concept of the resulting two supernodes is inferred to be **PrinterGroup**.

We also define the attributes of the supernode as the number of files and number of identities that print to the specified printer. So the function **F** is defined as follows.

$$\mathbf{F}(\mathbf{numFiles}) = g_1$$

$$g_1(\{(\mathbf{N_C^j}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_C\}) = \mathbf{count}(v_j), v_j \in \mathbf{File}^{\mathcal{I}}$$

$$\mathbf{F}(\mathbf{numIdentity}) = g_2$$

$$g_2(\{(\mathbf{N_C^j}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_C\}) = \mathbf{count}(v_j), v_j \in \mathbf{Identity}^{\mathcal{I}}$$

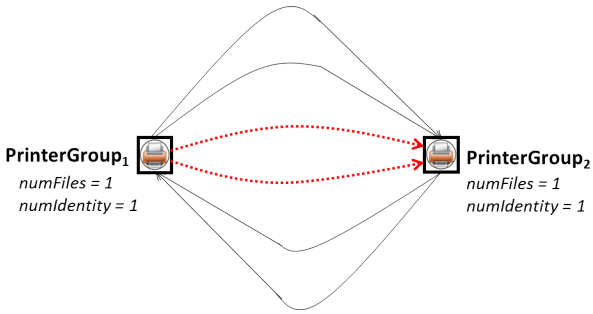


Figure 3.5: The resultant graph after application of basic vertex contraction.

The resulting graph is given in Figure 3.5. Using basic vertex contraction, we can profile the workload of printers by looking at the **PrinterGroup**'s vertex attributes **numFiles** and **numIdentity**. However, we can obtain more information using extended vertex contraction and infer the state of the printer, as we will show later.

Extended vertex contraction. In addition to defining the supernode, the extended vertex contraction also defines the edges incident to the supernode. The role names of the edges, unlike those in the basic vertex contraction, are not known to be semantically related. Instead, we can define functions that derive role names that represent a certain group of semantically related role names in the ontology $\mathcal{K}$. Then, we can derive the set of role names of the edges incident to the supernode by substituting the role names of the edges into the appropriate set of defined functions.

We describe the extended vertex contraction in an informal manner. Just as in basic vertex contraction, the supernode is an instance of a concept C that is consistent with respect to the axiom $\mathbb{T}$. In addition, the edges incident to the supernode are an instance of a set of roles that is consistent with respect to a subset of *TBox* axioms in $\mathbb{S}$. We add a validity check to ensure that (1) the supernode and edges incident to the supernode and (2) the supernode's feature names and values are also consistent with respect to the axioms in the ontology $\mathcal{K}$. Therefore, the resulting interpretation is still a model of $\mathcal{K}$. So the semantic meaning of the new graph G' is still preserved.

Definition 4 Extended vertex contraction *is a function*
Contract$(V_C, (G, \mathcal{K})_{\mathcal{I}}, \mathbb{T}, \mathbf{F}, \mathbb{S}) = (G', \mathcal{K})_{\mathcal{I}'}$ *where* $V_C \subseteq V(G)$.

Notation: *Refer to Definition 3 for more details.*

$\mathbb{S}$**:** *The set of TBox axioms describing the role names of the edges incident to the supernode.*

Topological changes: *Same as Definition 3.*

Semantic changes: *Refer to Definition 3.*

We first describe the procedure for each vertex in the neighborhood that has an outgoing edge from V_C; the same procedure applies to any vertex that has an incoming edge from V_C. For each vertex $w \in N^+(V_C)$, we assign role names to the edges that are incident to v' and w. $\mathbb{S}$ represents a set of TBox axioms, $\{R_1, \cdots, R_k\}$, that describes the relation between the role name of the edge incident to the supernode and a subset of the role names of the outgoing edges from V_C. The axioms $R_1 \cdots R_k$ are mutually disjoint; i.e., they involve disjoint sets of roles.

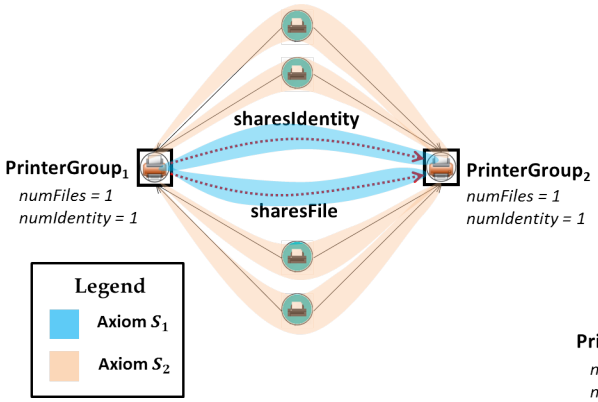
We look at a subset of axioms in $\mathbb{S}$ to choose the role names to assign to the new edges. The subset of axioms is determined by the role names assigned to the edges incident to w from vertices in V_C. For each axiom in that subset, say R_i, we assign a role name Y that is consistent w.r.t. R_i to the edge $e_q(v', w)$. Then, we augment the interpretation $\cdot^{\mathcal{I}'}$ in the following manner.

$$\cdot^{\mathcal{I}'}(Y) = \cdot^{\mathcal{I}'}(Y) \cup e_q(v', w) \tag{3.16}$$

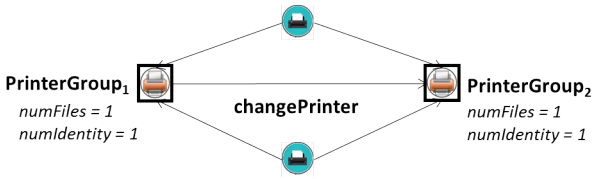
Property changes: *Same as Definition 3.*

Validity check: *Same as Definition 3.*

Example: We demonstrate the usage of extended vertex contraction using the previous example of attempting to infer the state of printers. Continuing to look at the graph in Figure 3.5, we can infer that a person has printed a file to both $\mathbf{Printer}_1$ and $\mathbf{Printer}_2$.The evidence for this inference is that two printer groups have been used by the same user to print the same file. This is represented in the graph by the occurrence of both a **sharesFile** and **sharesIdentity** edge between the two printer groups. In order to perform inference



(a) Grouping of edges based on the $\mathbb{S}$ axioms.



(b) Resultant graph.

Figure 3.6: The edges are grouped according to the $\mathbb{S}$ axioms, and (b) shows the resultant graph. The non-bold labels represent the feature values of the **PrinterGroup** concept.

upon this evidence, we construct $\mathbb{S}$, as the following two axioms.

$$X \equiv \textbf{sharesFile} \sqcap \textbf{sharesIdentity} \tag{3.17}$$

$$X \equiv \textbf{prints} \tag{3.18}$$

We define **changePrinter** as **sharesIdentity**$\sqcap$**sharesFile**, which represents the occurrence of both edges. The axiom $\mathbb{S}_1$ in Equation (3.17) means that the role name of the resulting edge is **changePrinter** when both **sharesFile** and **sharesIdentity** edges exist and if only one of either **sharesFile** and **sharesIdentity** exist, then the role name of the resulting edge takes on the existing edge's role name. Since both edges exist, the role name of the resulting edge is **changePrinter**.

The axiom $\mathbb{S}_2$ in Equation (3.18) means that we group edges that are instances of **prints** and assign the resulting edge a role name of **prints**. The set of edges selected by the two axioms is shown in Figure 3.6a.

The resulting graph is given in Figure 3.6b. We can now infer automatically that a person printed a file to the two printers. However, we cannot infer from this graph which printer was unavailable, because the information at hand only tells us that one of the two printers was unavailable. We can use contraction with edge attributes (which we will show later) to inform us about the order in which the file was printed to the printers.

Vertex contraction with edge attributes. In the previous paragraphs, we defined the basic and extended versions of vertex contractions. Both definitions covered the vertex attributes of the supernode. However, if the data model $(G, \mathcal{K})_{\mathcal{I}}$ models edge attributes as well, then we need to specify how the edge attributes differ in the contracted data model $(G', \mathcal{K})_{\mathcal{I}'}$.

The basic vertex contraction focuses on presenting a higher abstracted view of the group of selected vertices. However, it does not require maintenance of the attribute values of the edges incident to the group of vertices. Thus, we consider edge attributes only in the context of extended vertex contraction.

Much as in extended vertex contraction, we do not know in advance the role names and, by extension, the feature names of the edges. Thus, we adopt an approach similar to that of extended vertex contraction by associating the functions for calculating the aggregate attributes with an axiom α in $\mathbb{S}$. Thus, if an axiom α is used to deduce a role name for the edge, the feature values of the role name are derived by substituting the feature values of the selected edges into defined functions. The role name is assigned feature names that represent the aggregate feature values.

We modify Definition 4 in the following manner to create the following definition.[4]

Definition 5 Extended vertex contraction with edge attributes *is a function* $\mathbf{Contract}(V_C, (G, \mathcal{K})_{\mathcal{I}}, \mathbb{T}, \mathbf{F}, \mathbb{S}, \mathbf{H}) = (G', \mathcal{K})_{\mathcal{I}'}$ *where* $V_C \subseteq V(G)$.

Notation: *Refer to Definition 4 for more details.*

V_C**:** *The subset of vertices in the graph* W *selected for contraction.*

V_R**:** *The set of vertices representing edges incident to the selected vertices in* W.

H: *A set of tuples* $(\alpha, \mathbf{J})$ *where* $\alpha \in \mathbb{S}$ *and* $\mathbf{J}$ *is a function describing the feature names of the edge and their associated values. That is,* $\mathbf{J}$ *maps* $\mathbf{N_F}$ *to* $g : \{(\mathbf{N_C}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i})\} \rightarrow \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}$.

[4]In this definition of extended vertex contraction with edge attributes, the description logic DL and interpretation $\mathcal{I}$ have already been transformed to model edge attributes in the manner described in Section 3.3.1.

$N^+(W)$**:** *The neighborhood of vertices in* W *that have an outgoing edge from some vertex in* W*, i.e.,* $N^+(W) = \{v | e_p(v_1, d), e_q(d, v) \in E(G)$ *and* $v_1 \in W \subseteq V(G), v \notin W\}$.

$N^-(W)$**:** *The neighborhood of vertices in* W *that have an incoming edge from some vertex in* W*, i.e.,* $N^-(W) = \{v | e_p(v, d), e_q(d, v_1) \in E(G)$ *and* $v_1 \in W \subseteq V(G), v \notin W\}$.

Topological changes: *Same as Definition 4.*

Semantic changes: *Refer to Definition 4 for more details.*

We first describe the procedure for each vertex in the neighborhood that has an outgoing edge from V_C*; the same procedure applies to any vertex that has an incoming edge from* V_C*. For each vertex* $w \in N^+(V_C)$*, we use* $\mathbb{S}$ *to assign concept names (representing roles) to the new individual vertices representing* $e_i(v', w)$*.*

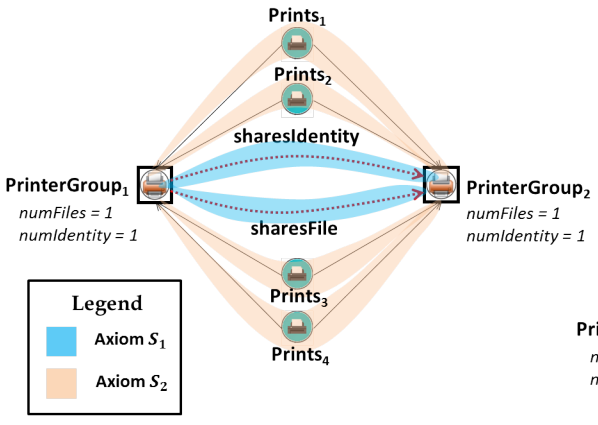
Property changes: *Refer to Definition 4 for more details.*

Then, we determine the edge attributes to be assigned to a_i*. The axiom* R_i *is mapped to the corresponding tuple in* $\mathbf{H}$ *to obtain the corresponding function* $\mathbf{J}$*. For each element* $f \in$ *Domain*$(\mathbf{J})$*, we calculate the attribute value of* a_i *using the function* $g = \mathbf{J}(f)$*. The function* g *takes in as arguments the role names of the selected edges and their attribute values corresponding to a particular feature name* $h \in \mathbf{N_F}$ *that is semantically related to* f*. The output of* g *is* $d \in \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}$*. Then, we augment the interpretation* $\cdot^{\mathcal{I}'}$ *in the following manner.*

$$[\cdot^{\mathcal{I}'}(f)](a_i) = d \tag{3.19}$$

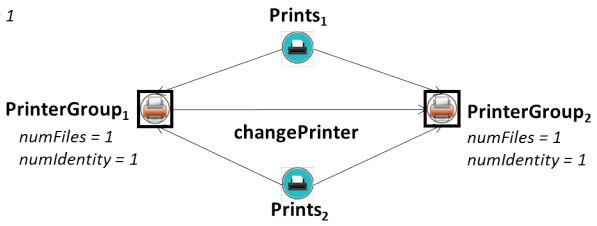
Validity check: *Same as Definition 4.*

Example: We continue with the previous example of inferring the state of printers and demonstrate the usage of vertex contraction with edge attributes. We can determine the order of the printing events by looking at the **timestamp** of the **prints** edges between the two **PrinterGroup** vertices. We want to determine the times of the first print jobs that are sent to the two printer groups. Then, we can determine the printer that is unavailable



	Prints$_1$	Prints$_2$	Prints$_3$	Prints$_4$
Timestamp	1PM	1PM	2PM	2PM

(a) Graph after contraction without changes to edges.



	Prints$_1$	Prints$_2$
Timestamp	1PM	2PM

(b) Resultant graph.

Figure 3.7: Partial attribute values for each **Prints** edge are given in the table at the bottom of (a). (We only show the hour of the timestamp in the table.) (b) shows the resultant graph after the $\mathbb{S}$ axioms and **H** mappings have been applied to the sets of edges. The table shows partial attribute values of the final edge (hours only).

as the head vertex of the edge with the lowest time. So we define **H** as a set of two tuples $(\mathbb{S}_1, \mathbf{J}_1)$ and $(\mathbb{S}_2, \mathbf{J}_2)$, where $\mathbf{J}_1$ and $\mathbf{J}_2$ are defined as follows.

$$\mathbf{J}_1 = \phi$$

$$\mathbf{J}_2(\mathbf{timestamp}) \rightarrow g_1$$

$$g_1(\{(\mathbf{N_C^j}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_R\}) = \min_{\mathbf{timestamp}}\{\mathbf{timestamp}(v_1), \ldots, \mathbf{timestamp}(v_k)\}$$

where $|V_R| = k$.

The mapping $\mathbf{J}_1$ defines the edge attributes for axiom $\mathbb{S}_1$, which is the **changePrinter** edge. $\mathbf{J}_1$ is a null mapping, which means that there are no feature names associated with the edge. The mapping $\mathbf{J}_2$ defines the edge attributes for axiom $\mathbb{S}_2$, which is the **prints** edge. $\mathbf{J}_2$ calculates the feature value for **timestamp** as the minimum **timestamp** value in the group of edges.

The resulting graph is shown in Figure 3.7b. We can see that the **Prints**$_1$ edge from **PrinterGroup**$_1$ to **PrinterGroup**$_2$ has a lower timestamp than the **Prints**$_3$ edge from

$\mathbf{PrinterGroup}_2$ to $\mathbf{PrinterGroup}_1$. So we can deduce that a file was printed to $\mathbf{PrinterGroup}_2$ and subsequently to $\mathbf{PrinterGroup}_1$. Thus, we can infer that $\mathbf{Printer}_2$ is likely not working.

We can extend the vertex contraction operation to handle multiple sets of vertices, $\{V_C^1, \ldots, V_C^m\}$, which results in the creation of m supernodes. The definition of vertex contraction easily extends to multiple supernodes. Semantic and property changes are applied to each set of vertices and the corresponding supernode. Topological changes differ in the following manner.

Definition 6 ***Multiple vertex contraction*** *is a function* $\boldsymbol{Contract}(\{V_C^1, \ldots, V_C^m\}, (G, \mathcal{K})_{\mathcal{I}}, \mathbb{T}, \mathbf{F}, \mathbb{S}, \mathbf{H}) = (G', \mathcal{K})_{\mathcal{I}'}$ *where* $V_C^1, \ldots, V_C^m \subseteq V(G)$.

Topological changes: *The resulting graph* G' *is a smaller graph where*

$$
\begin{aligned}
V(G') &= V(G) - \{V_C^1, \ldots, V_C^m\} + \{v'_1, \ldots, v'_m\} \\
E(G') &= E(G) - \{e_i(v_1, v_2) | e_i \in E(G), v_1 \in V_C^k \text{ or } v_2 \in V_C^k \text{ for } k \in [1, m]\} + \\
&\quad \{e_i(v'_j, v'_k) | e_i \in E(G), \exists v_t \in V_C^k \text{ s.t. } v_t \in N^+(V_C^j) \text{ for } j, k \in [1, m], j \neq k\} + \\
&\quad \{e_i(v'_k, v_1) | e_i \in E(G), v_1 \in N^+(V_C^k), v_1 \notin V_C^j, \forall j \in [1, m]\} + \\
&\quad \{e_i(v_2, v'_k) | e_i \in E(G).v_2 \in N^+(V_C^k), v_2 \notin V_C^j, \forall j \in [1, m]\}
\end{aligned}
$$

However, if there is a vertex that is contained by two sets, i.e., $v \in V_C^j, V_C^k, j \neq k$, then we model the sets V_C^j and V_C^k as containing two distinct vertices and connect the two vertices with an edge that has a role name of `shareC`, where `C` is a filler for the concept name of the vertex. More formally, $v_1 \in V_C^j$, $v_2 \in V_C^k$ and $e_{\text{sharesC}}(v_1, v_2) \in E(G)$ where $v \in C^{\mathcal{I}}$.

3.5 CPTL-Aware Feedback Loop

The goal of CPTL is to provide a formal specification of a target system. However, the state of the target system will evolve over time because of external events. So, as mentioned in Chapter 1, we need to update a CPTL model representation of the target system. In this

section, we present a framework for using CPTL and its operations in a dynamic manner to maintain a formal specification of a target system.

The general life cycle consists of two components: the *offline* analysis and the *online* checking.

Offline analysis. We maintain a CPTL model that represents the current state of the target system. We use operations to query the CPTL model for appropriate data so that we can perform analysis about the state of the system. In the context of intrusion detection, the analysis in which we are interested concerns obtaining of behavior profiles, inference of state information, and specification of an organization's policies. Throughout this chapter, we have presented small examples of the usage of vertex contraction. In each example, vertex contraction queries a relevant portion of the CPTL model and derives a model that represents a summary of certain features of the target system.

Online checking. We obtain system events and add them into our CPTL model by asserting them as facts into the ontology using $ABox$ axioms. Then, we use an ontological reasoner to infer the satisfiability of the ontology by the events. In the context of intrusion detection, the ontological reasoner checks that the events satisfy the axioms that represent appropriate behavior. If the ontology is not satisfied, then an event is indicative of malicious behavior. The events and axioms that constitute violations can be presented to a human administrator for further analysis or automatically blocked by the target system.

General overview. Now, we describe the information flow between the offline analysis and online checking components. The offline analysis passes axioms that are results of data analyses to the online checking component. The axioms enforce certain rules in the target system that must be satisfied by new events that occur in the system. For example, the axioms refer to appropriate behavior or signatures of attacks in the context of intrusion detection. In Chapter 5, we will introduce various axioms that attempt to model system policies or normal user behavior.

The online checking component updates the CPTL model with events and satisfiability

test results. The offline analysis will use the satisfiability test results to enhance the quality of the data analyses. For example, in the context of intrusion detection, the online checking will update the CPTL model with the events and axioms that were violated. The offline analysis incorporates that feedback about false positives and tunes the detectors to classify the events correctly. That concept of feedback loops is similar to Experience-based Access Management (EBAM) [31], which uses feedback from audit logs to drive the enforced control (EC) implemented in the operational access control system closer to the ideal model (IM) that represents the true permissions for the enterprise. Therefore, our approach to intrusion detection follows that life cycle so as to drive our detectors closer to the target function that discriminates between appropriate and inappropriate behavior.

CHAPTER 4

IMPLEMENTATION OF OUR APPROACH

In Chapter 3, we introduced the Cyber-Physical Topology Language (CPTL) data model and operations upon the data model. In this chapter, we will describe an implementation of the CPTL data model and vertex contraction.

4.1 Data Model

Recall from Chapter 3 that a CPTL data model $(G, \mathcal{K})_{\mathcal{I}}$ is an interpretation $\mathcal{I}$ of a graph G that is a model of a subset of axioms $\mathbb{V}$ in the ontology $\mathcal{K}$. We define a Java class `CPTLModel` that represents the CPTL data model and contains data fields representing the graph, ontology and interpretation. We describe the implementation of each component of the data model, that is, the graph, ontology, and interpretation, in the subsequent sections.

4.1.1 Graph

We implement the multi-directed graph G as an adjacency list `TreeMap<Vertex,TreeSet<Edge>> G`. The adjacency list maps each vertex $v \in V(G)$ to a list of edges $e_i \in E(G)$.

A vertex is implemented as a class `Vertex`. We implement the inverse mapping of the ontology to data described in Section 3.3.1 by adding data fields to the class `Vertex`. In particular, the relation mapping the vertex to its concept name, T_V is represented as the data field `type`, the relation mapping the vertex to its individual name, T_L is represented as the data field `INDname`, and the relation mapping the vertex to its feature names, F_V is

represented as the keys in the data field `attributes` that map to their corresponding values in A_V.

An edge is implemented as a class `Edge`. The function h is represented as a data field `endpoints` that is a list of size two. The inverse mapping of the ontology to data is implemented in much as in the `Vertex` class.

4.1.2 Ontology

The ontology $\mathcal{K}$ is expressed in OWL 2 Web Ontology Language (OWL 2) and serialized to an RDF/XML-formatted file that we name `base`. OWL 2 is a knowledge representation language that is endorsed by the W3C [32, 28].

The RDF/XML file `base` contains a description of the concept names $\mathbf{N_C}$, role names $\mathbf{N_R}$, individual names $\mathbf{N_I}$, feature names $\mathbf{N_F}$, and a list of *TBox* axioms.

We define a Java class `OntologyInterface` which represents the ontology $\mathcal{K}$. The class handles the creation, modification, and querying of the RDF/XML file using the OWL 2 API. A `CPTLModel` contains an instance of the class `OntologyInterface`.

4.1.3 Interpretation

In this section, we describe the process of parsing graphical data into a CPTL data model. Graphical data are stored in an XML file that is decorated with tags that correspond to concept names, role names, and feature names, along with their values. The node structure of the XML file describes the edges in the graph.

For each graphical data source, e.g., file accesses or print jobs, we build a parser that takes in an XML file and translates it into the graph $G(V, E, h)$. Based on the graph G, we generate *ABox* axioms.

We represent the subset of axioms $\mathbb{V} \subseteq \mathcal{K}$ as a separate RDF/XML file `smallbase` that contains a subset of the file `base`. The *ABox* axioms are added to `smallbase`. Then, we generate the interpretation $\mathcal{I}$ by running a HermiT reasoner [33] on the axioms contained in `smallbase`. The reasoner classifies the individuals into classes. For each class, we obtain the

list of individuals that are instances of the concept and add it to the interpretation function $\cdot^{\mathcal{I}}$.

The interpretation function $\cdot^{\mathcal{I}}$ is implemented as four `TreeMap`s that represent the mappings of concept names, role names, individual names, and feature names. In particular, $\mathtt{I_C}$ represents the mapping of concept names, $\mathtt{I_R}$ represents the mapping of role names, $\mathtt{I_N}$ represents the mapping of individual names, and $\mathtt{I_F}$ represents the mapping of feature names.

Edge attributes

For each role name $r_i \in \mathbf{N_R}$, we add a concept name c_i and two role names r_i^1, r_i^2 to the file `base`. Then, we generate $ABox$ axioms for each edge $e_i(a, b) \in E(G)$. The $ABox$ axioms are listed below.

$$c_i(e_i)$$
$$r_i^1(e_i, a)$$
$$r_i^2(e_i, b)$$

The generation of the $ABox$ axioms is accomplished by the function `generateABoxAxiom`, shown in Algorithm 1.

4.1.4 Updating the model

We can update a CPTL data model in four ways: (1) adding a vertex, (2) adding an edge, (3) removing a vertex, and (4) removing an edge. Those update operations are implemented as four separate functions, `addVertex`, `addEdge`, `removeVertex`, and `removeEdge`, respectively. The update operations affect the graph G and the interpretation functions $\cdot^{\mathcal{I}}$. We describe the details of each operation below.

Algorithm 1 The function `generateABoxAxiom`.

```
function generateABoxAxiom
    for all vertices v do
        **Add concept assertions about vertex.**
        indV ← getNamedIndividual(v.INDname)
        vType ← getOWLClassAssertionAxiom(v.type,indV)
        addAxiom (vType)

        **Add feature assertions about vertex.**
        for all (f, val) in v.attributes do
            vAttribute ← getOWLDataPropertyAssertionAxiom(f,indV,val)
            addAxiom (vAttribute)
        end for

        for all edges e incident to v do
            **Add concept assertion c_i(e)**
            indE ← getNamedIndividual(e.INDname)
            eType ← getOWLClassAssertionAxiom(e.type,indE)
            addAxiom (eType)

            **Add role assertion r_i^1 and r_i^2**
            startV ← e.endpoints(tail)
            endV ← e.endpoints(head)
            tailE ← getOWLObjectPropertyAssertionAxiom(e.type+"From",indE,startV)
            headE ← getOWLObjectPropertyAssertionAxiom(e.type+"To",indE,endV)
            addAxiom (tailE)
            addAxiom (headE)

            **Add feature assertion about edge e**
            for all (f, val) in e.attributes do
                eAttribute ← getOWLDataPropertyAssertionAxiom(f,indE,val)
                addAxiom (eAttribute)
            end for
        end for
    end for
end function
```

`addVertex` The vertex to be added, v, is first added to the graph G with an empty adjacency list. Then, we use the inverse mappings in v to update the respective interpretation functions by adding v to the appropriate list. In particular, $T_V(v)$, which is the vertex's data field `type`, is used as the key to reference the map $\mathtt{I_C}$. The keys in `attributes` are used to reference the map $\mathtt{I_F}$. Finally, $T_L(v)$, which is the vertex's data field `INDname`, is added to the map $\mathtt{I_N}$.

`addEdge` The edge to be added, $e(v_1, v_2)$, is first added to the graph G by addition of e to v_1 and v_2's adjacency list. Then, much as for `addVertex`, we use the inverse mappings in e to update the respective interpretation functions by adding e to the appropriate list. In particular, $T_E(e)$, which is the edge's data field `type`, is used as the key to reference the map $\mathtt{I_R}$. The keys in `attributes` are used to reference the map $\mathtt{I_F}$.

`removeVertex` The vertex to be removed, v, is first deleted from the graph G along with its adjacency list. We do not remove the edges incident to the vertex, because the edges are still used after deletion of the vertex to calculate edge attributes and role names. These edges will be removed by the function `removeEdge` later. We use the inverse mappings in v to remove v from the appropriate lists in the interpretation functions, using the same mechanism used for `addVertex`.

`removeEdge` The edge to be removed, $e(v_1, v_2)$ is first deleted from the graph G through removal of e from v_1 and v_2's adjacency list. Then, just as for `removeVertex`, we use the inverse mappings in e to remove e from the appropriate lists in the interpretation functions, using the same mechanism used for `addEdge`.

4.1.5 Generating Views of the Model

When we conduct analyses on the CPTL data model, we may only be interested in some relations or individuals. Therefore, we need to generate views of the CPTL data model by extracting sub-graphs. We implement this in the functions `RetainEdge` and `FilterEdge`.

`RetainEdge` This function takes in a set of edges, E, which represents the relations of interest. A clone of the current CPTL data model is generated. Then, we iterate through all the edges and remove those that are not contained in E. The cloned CPTL data model is then returned as the view.

`FilterEdge` This function takes in a set of edges, E, which represents the relations that are not of interest. Just as in `RetainEdge`, we generate a clone of the current CPTL data model and iterate through all the edges. We remove edges that are contained in E and return the cloned CPTL data model as the view.

4.2 Inferencing

We implement `inferVertexType` which infers the concept name of a vertex, and `inferEdgeType` which infers the role name of an edge. Both functions take in an axiom and use a HermiT reasoner to perform inferencing.

inferVertexType If the axiom is a subsumption axiom, then we call the `getSuperClasses` function of the reasoner, which returns the direct superconcept of the RHS expression of the axiom. If the axiom is an equivalence axiom, then we call the `getEquivalentClasses` function of the reasoner, which returns the equivalent concept names of the RHS expression of the axiom.

inferEdgeType This function mirrors `inferVertexType`, but, because of the separation of *TBox* axioms concerning roles and concepts in OWL 2, we add an extra step by mirroring the *TBox* axioms of roles to *TBox* axioms of the concepts that represent the roles. Then, we can use the same functions `getSuperClasses` and `getEquivalentClasses` to obtain the role name of the edges.[1]

[1]Note that this method does not deal with complex role inclusion axioms. However, we are currently investigating the use of the `PropertyChainAxiom`, which may help in our implementation.

4.3 Vertex Contraction

In this section, we describe our implementation of the vertex contraction operation.

We implement a class `CPTLOperations` that contains an object reference to an instance of `CPTLModel`, `oldModel`. All operations defined in the `CPTLOperations` are performed on `oldModel`. The result of the operations will be a new instance of `CPTLModel` that we name `newModel`.

4.3.1 Basic vertex contraction

We define a function `BasicContract` that takes a list of sets of vertices, `LinkedList<TreeSet<Vertex>>`; the number of sets of vertices to be contracted, m; a functor $\mathbb{T}$; and a functor $\mathbf{F}$. The algorithm for the function is given in Algorithm 2.

In brief, the work flow of `BasicContract` is as follows. First, we create a new instance of `CPTLModel`, called `newModel`, that copies the existing instance `oldModel`. Then, we define the supernodes and their attributes. Finally, we add the edges incident to the supernodes.

In more detail, we iterate through the sets $V_C^1, \ldots, V_C^m$. For each set of vertices, V_C^k, we create an empty instance of the `Vertex` class, v'_k, that represents the supernode.

We extract the concept names of the vertices in the set and add them to a list `vType`. The list `vType` is passed as a parameter to the functor $\mathbb{T}$. The functor $\mathbb{T}$ returns an axiom that describes the concept name of the supernode. The axiom is passed to the function `inferVertexType`, which uses a HermiT reasoner to infer the concept names that fulfill the axiom. One of the concept names is returned by `inferVertexType` and is added to the relation T_V through updating of the `type` data field.

The set of vertices V_C^k is passed to the functor $\mathbf{F}$. The functor $\mathbf{F}$ derives a set of feature names and corresponding values from the vertex attributes of vertices in V_C^k. The result of the functor $\mathbf{F}$ is assigned to the data field `attributes`.

Finally, `addVertex` is used to add the supernode vertex v'_k to the new data model, `newModel`, and `removeVertex` is used to remove the vertices in V_C^k from `newModel`.

Next, we iterate through all pairs of $\{V_C^j, V_C^k\}$ and $\{V_C^j, v_i\}$, where v_i represents vertices

Algorithm 2 The function `BasicContract`.

function BASICCONTRACT($\{V_C^1, \ldots, V_C^m\}$,$\mathbb{T}$,$\mathbf{F}$,verify)
newModel ← *clone(oldModel)*

Defines the concept name and vertex attributes of the supernodes
for all groups of vertices V_C^k **do**
vType ← *extract types of all vertices in* V_C^k
TAxiom ← $\mathbb{T}$*(vType)*
supernodeType ← *inferVertexType(TAxiom)*

vAttributes ← $\mathbf{F}$(*vertices in* V_C^k)

removeVertex(V_C^k)
supernode ← *new Vertex(supernodeType,vAttributes)*
addVertex*(supernode)*
end for

Defines the edges incident to supernodes
for all sets of vertices (V_C^k, W) **do**
edgeFrom ← *edges from* V_C^k *to the vertex group or singleton represented by W*
edgeTo ← *edges from the vertex group or singleton represented by W to* V_C^k

for all edges in edgeFrom **do**
eType ← *generalRelation*
eAttr ← *empty set*
newEdge ← *new Edge(eType,eAttr,(supernode,W))*
addEdge*(newEdge)*
end for
for all edges in edgeTo **do**
eType ← *generalRelation*
eAttr ← *empty set*
newEdge ← *new Edge(eType,eAttr,(W,supernode))*
addEdge*(newEdge)*
end for

removeEdge*(edgeFrom,edgeTo)*
end for

Verify validity of contraction operation
run HermiT reasoner over newModel
if model is satisfied **then** *newModel is valid*
else*Throw CPTLModelInvalidException*
end if
end function

that are not being contracted. For each pair, we find the edges between them, and if the pair of sets of vertices share some vertices, the `shareC` edge (as mentioned previously) is added to the list. Then, for each edge $e_i(v_1, v_2)$ where $v_1 \in V_C^j, v_2 \in V_C^k$, we create a new edge $e'(v'_j, v'_k)$ and associate the top-level role name that subsumes all other role names with the new edge e'. Similarly, for each edge $e_i(v_1, v_2)$ where $v_1 \in V_C^j, v_2 \notin V_C^1, \ldots, V_C^m$, we create a new edge $e'(v'_j, v_2)$, and for $e_i(v_2, v_1)$, we create a new edge $e'(v_2, v'_j)$. `addEdge` is used to add the new edge to the new data model, `newModel`. `removeEdge` is used to remove the old edges from `newModel`.

Finally, we verify the validity of the CPTL model by writing the new data structure `newModel` to an RDF/XML file and running the HermiT reasoner on the file. If the HermiT reasoner detects an inconsistency in the ontology, the contraction operation is invalid, and a `CPTLModelInvalidException` is thrown.

Complexity analysis. We define n to be the total number of vertices over all the m groups of vertices selected for contraction. We define p to be the total number of edges that are incident to at least one of the n vertices.

First, we analyze the computational complexity of the first *for* loop that creates supernodes. The functions $\mathbb{T}$,**F** require a single iteration over the vertices in V_C^k. The functions **removeVertex** and **addVertex** run in constant time. The function *inferVertexType* takes an amount of time that is at most exponential in the size of the axioms in the ontology [34]. So the total computational complexity for the first *for* loop is $O(n + m(2^A))$, where A is the size of the *TBox* axioms in the ontology. Typically, the size of the *TBox* axioms can be considered a constant, since it represents the domain of knowledge.

Next, we analyze the computational complexity of the second *for* loop, which adds the edges between the new supernodes and their neighbors. The total number of distinct vertices in the `newModel` will be $|V|-n+m$, where $|V|$ is the total number of vertices in the `oldModel`. Then, each edge incident to one of the n selected vertices is visited $|V| - n + m$ times, and added to the lists *edgeFrom* and *edgeTo* in $O(\log\, p)$ time. The functions **removeEdge** and **addEdge** run in constant time. So the total computational complexity for the second *for* loop is $O(p\log\, p(|V| - n + m))$.

Therefore, the total computational complexity of `BasicContract` is $O(n + m(2^A) + p\log\, p(|V| - n + m))$, which we can upper-bound by $O(|E||V|\log\, |E|)$ where $|E|$ is the total number of edges in the `oldModel`.

4.3.2 Extended vertex contraction with edge attributes

We define a function `ExtendedContract` that takes a list of sets of vertices, `LinkedList<TreeSet<Vertex>>`; the number of sets of vertices to be contracted, m; a functor $\mathbb{T}$; a functor $\mathbf{F}$; a functor $\mathbb{S}$; and a functor $\mathbf{H}$. The algorithm for this function is given in Algorithm 3.

The work flow of `ExtendedContract` is similar to that of `BasicContract` with the exception of the definition of the edges incident to the supernodes, which we describe next.

Just as for `BasicContract`, we iterate through all pairs of $\{V_C^j, V_C^k\}$ and $\{V_C^j, v_i\}$, where v_i represents vertices that are not being contracted. For each pair, we determine the edges between them and divide the edges into two categories based on the direction of the edge. For each category, we pass the edges to the functor $\mathbb{S}$. The functor $\mathbb{S}$ constructs the relevant *TBox* axioms based on the role names of the edges and returns a mapping of *TBox* axioms to edges that correspond to the axiom. For each *TBox* axiom, T, returned by $\mathbb{S}$, we create a new edge e_T. We pass the axiom to the function `inferEdgeType` which uses the HermiT reasoner to infer the role name of the resulting edge e_T. The role name returned by `inferEdgeType` is added to the relation T_E through updating of the `type` data field. We also pass the edges E_T corresponding to the *TBox* axiom T to the functor $\mathbf{H}$. The functor $\mathbf{H}$ derives a set of feature names and corresponding values from the vertex attributes of edges in E_T. The result of the functor $\mathbf{H}$ is assigned to the data field `attributes`. Then, `addEdge` is used to add the edge e_T to the new data model. Finally, `removeEdge` is used to remove the old edges from `newModel`.

Finally, the new CPTL model is verified through the same steps as for `BasicContract`.

Complexity analysis. The computational complexity of the first *for* loop is the same as that of `BasicContract`.

Algorithm 3 The function `Contract`.

function CONTRACT($\{V_C^1, \ldots, V_C^m\}$,$\mathbb{T}$,$\mathbf{F}$,$\mathbb{S}$,$\mathbf{H}$, verify)
newModel ← *clone(oldModel)*
Defines the concept name and vertex attributes of the supernodes
for all groups of vertices V_C^k **do**
vType ← *extract types of all vertices in* V_C^k
TAxiom ← $\mathbb{T}$*(vType)*
supernodeType ← *inferVertexType(TAxiom)*
vAttributes ← $\mathbf{F}$(*vertices in* V_C^k)
removeVertex(V_C^k)
supernode ← *new Vertex(supernodeType,vAttributes)*
addVertex*(supernode)*
end for

Defines the role names and edge attributes of edges incident to supernodes
for all sets of vertices (V_C^k, W) **do**
edgeFrom ← *edges from* V_C^k *to the vertex group or singleton represented by W*
edgeTo ← *edges from the vertex group or singleton represented by W to* V_C^k

eType ← *extract types of all edges in edgeFrom*
SAxiomSet ← $\mathbb{S}$*(eType)*
for all edges E that fulfill *SAxiomSet(i)* **do**
newEdgeType ← *inferEdgeType(E)*
eAttr ← $\mathbf{H}$(E)
newEdge ← *new Edge(eType,eAttributes,(supernode,W))*
addEdge*(newEdge)*
end for

eType ← *extract types of all edges in edgeTo*
SAxiomSet ← $\mathbb{S}$*(eType)*
for all edges E that fulfill *SAxiomSet(i)* **do**
newEdgeType ← *inferEdgeType(E)*
eAttr ← $\mathbf{H}$(E)
newEdge ← *new Edge(eType,eAttributes,(supernode,W))*
addEdge*(newEdge)*
end for

removeEdge*(edgeFrom,edgeTo)*
end for
Verify validity of contraction operation
run HermiT reasoner over newModel
if model is satisfied **then** *newModel is valid*
else*Throw CPTLModelInvalidException*
end if
end function

Next, we analyze the computational complexity of the second *for* loop which adds the edges between the new supernodes and their neighbors. Then, each edge incident to one of the n selected vertices is visited $|V| - n + m$ times, and added to the lists *edgeFrom* and *edgeTo* in $O(\log\, p)$ time. The functions **removeEdge** and **addEdge** run in constant time. The function *inferEdgeType* takes an amount of time that is at most exponential in the size of the axioms in the ontology. The maximum number of times that *inferEdgeType* is called in a single iteration of the loop can be upper-bounded by the number of role names in the ontology $|N_R|$. So the total computational complexity for the second *for* loop is $O(p\log\, p(|V| - n + m) + m|N_R|(|V| - n + m)(2^A))$.

Therefore, the total computational complexity of `Contract` is $O(n + m(2^A) + p\log\, p(|V| - n + m) + m|N_R|(|V| - n + m)(2^A))$, which we can upper-bound by $O(|E||V|\log\, |E| + |V|^2)$.

CHAPTER 5

APPLICATION OF OUR APPROACH

In this chapter, we apply the theory developed in Chapter 3 to an enterprise system. Our aim is to detect masqueraders and traitors in the system. In Chapter 1, we explained our choice of malicious insiders as our threat model. Now, we present our approach to detecting malicious insiders in the context of an enterprise system.

Recall from Chapter 1 that a masquerader is an outsider person who uses a legitimate user's credential (e.g., a user account) to perform malicious behavior. As mentioned in Chapter 2, a masquerader's actions is likely to deviate from the legitimate user's behavior. Thus, we can detect such deviations by baselining legitimate users' behavior in terms of activity type, location, amount of activity and timing. In particular, we develop baselines of each employee's writes to files and printer preferences. For example, an anomalous amount of writes to files signals possible tampering with data.

Traitors are much harder to detect because they are inherently legitimate users and can slowly trick the baselining into accepting potentially malicious behavior. We attempt to detect traitors by checking for misuse of system resources and data exfiltration. In particular, we use printing history to determine whether a person is printing large files or stealthily printing files over a long period of time using company printers. Studies have shown that a good number of insiders exfiltrate data physically through stealing or printing and it has been suggested that organizations monitor the printing behavior of employees [11, 12]. So we develop baselines of employee's printer preferences and detect unauthorized exfiltration of data when an employee prints to a distant printer.

In the subsequent sections, we provide a general overview of the application of CPTL, describe our dataset, and show how CPTL can be used for baselining and specification of

an organization's policies in the context of the dataset.

5.1 General Overview of Workflow

Before we go into the details of the use case, we describe the general workflow of representing a target system using CPTL and conducting analysis using the CPTL-aware feedback loop. The general workflow is given in Figure 5.1.

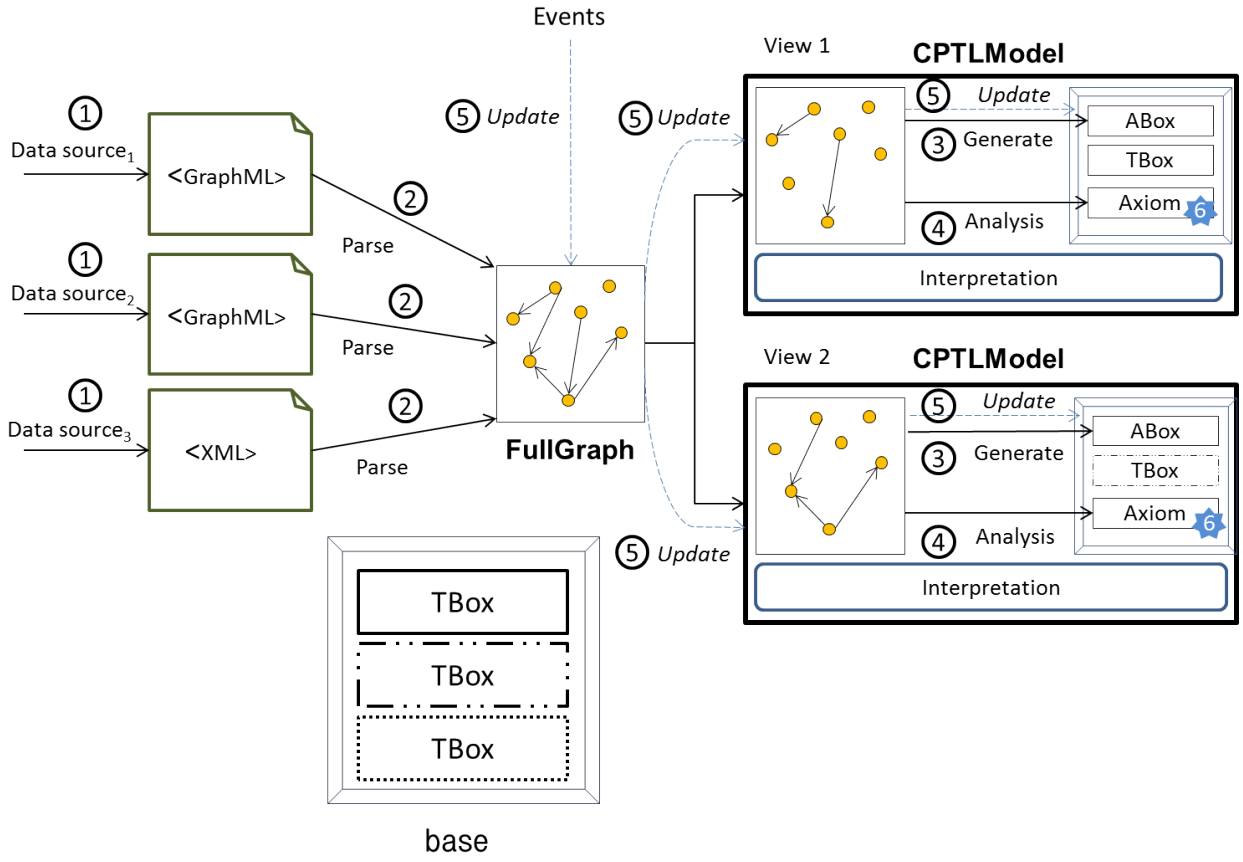


Figure 5.1: The general workflow of applying CPTL to a target system. The bold arrows represent the offline data processing and analysis, whereas the dotted arrows represent dynamic data flows and processing. The star-shaped icon labeled "6" represents a reasoner that is run over the ontology.

First, we obtain raw data from heterogenous sources of information such as logs, IDS alerts, and network packets. In step 1 of Figure 5.1, each set of data is parsed into either an intermediate GraphML or XML format that represents the graph G in the CPTL data

model. In step 2, we read the GraphML or XML file and populate the graph G in a CPTL data model with the information.

The full CPTL data model contains all the data and their relations. However, we want to conduct analyses that rely on some subset of the data. Therefore, we generate views of the full CPTL data model by extracting sub-graphs that contain relations and individuals of interest using the appropriate functions `FilterEdge` or `RetainEdge` that were described in Section 4.1.5.

The subset of *TBox* axioms that are necessary for analysis are listed in a separate RDF/XML file `smallbase`, represented in the figure by a beveled box. Then, the *ABox* axioms for the sub-graph are generated using the function `generateABoxAxiom` (illustrated in Algorithm 1) and added to `smallbase` in step 3.

Finally, we conduct offline analysis on the data in the view. The offline analysis results in the generation of axioms that are added to `smallbase` in step 4. When events occur in the system, they are added to the full CPTL data model and the corresponding views as shown in step 5. In step 6, a HermiT reasoner (represented by the star-shaped icon), is run over the views to check for violations; that, in turn, updates the offline analysis.

5.2 Data Sources

Following our workflow, in step 1, we obtain data sources about our target system which is an enterprise system. Our dataset consists of two components: commits to a *git* repository, and print jobs of a group of employees in a research group over a period of one year. We now describe the two components in detail.

***Git* repository.** The *git* repository is a shared research resource that contains papers, reports, source code, Web documents, spreadsheets, and presentations. Directories are generally organized according to projects. Several directories may belong to the same project. The *git* repository is accessed by 5 employees, and each employee can have multiple credentials.

We extract data from the *git* logs using the commands `git log --name-status` and

`git log ---stat`. The information provided by the commands is organized according to commits made by an associated credential. The commit information consists of the commit number, commit message, list of files modified, number of insertions and deletions made to the file in terms of lines, and timestamp of the commit.

We implemented a program *GitLogToXml* that parsed the information from the logs and constructed a graph using the *Boost Graph Library* (BGL) [35]. The *Boost* graph was then written into a GraphML file that represents the graph G in the CPTL data model. We implemented a function called `readGraphML` that takes in a GraphML file and populates a `CPTLModel` instance.

Print jobs. Employees print documents to printers that reside in buildings. In every building, there may be more than one printer on each floor. Employees must authenticate to a printer using their credentials before their print jobs are allowed to execute. An employee may submit his or her print jobs from more than one machine.

We collected the print job data of 6 employees using a Web interface that pulls data from the print events associated with credentials. In the process of collecting data, we realized that some employees who used Macintosh machines did not have their print jobs logged by the database, because their login IDs on the machine did not correspond to their credentials. Since we collected data only from the Web interface, we were unable to develop baselines for that group of employees. That loophole in the logging system illustrates the importance of collecting and relating data from heterogenous sources together. If we had access to the print logs of the individual printers, we could have identified the loophole earlier and corroborated information about printer state in our later examples.

The Web interface provides a list of print jobs for each employee. The print job information consists of the credentials of the employee, name of the file, number of pages printed, the machine used by the employee, and printer location.

We used JTidy [36] to convert the HTML file of the print jobs to an XML file. Then, we implemented a function `readPrintJob` that parses the information in the XML file, and populates a `CPTLModel` instance. The machine data does not provide enough information about the machine's location. Because of the low information value, we do not use those

data.

5.3 Ontology

In this section, we give a brief overview of the ontology (represented by the beveled box in Figure 5.1) that we developed to represent the domain knowledge of our target system. We describe the concept names, role names, feature names, and individual names used in the ontology, and the design decisions that we made.

Concept names. The main concept hierarchy that describes the domain knowledge of the target system is shown in Figure 5.2. An explanation of the terms in Figure 5.2 is given in Table 5.1a. A few other concepts that will be used in subsequent examples are explained in Table 5.1b.

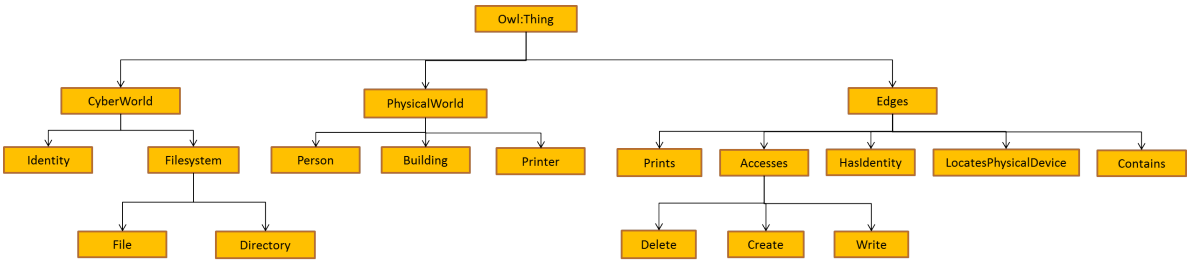


Figure 5.2: The concept hierarchy used in the ontology `base`. An arrow from concept A to concept B indicates that $B \sqsubseteq A$.

When we model a graph that has multi-edges or edges with attributes, we create a top-level concept name `Edges` that represent the role names. We add concept names that represent role names and assert that they are subsumed by `Edges`. We also add *TBox* axioms for the concepts that mirror the *TBox* axioms for the roles as explained in Section 4.2. In addition, we assert that each concept name, say `E`, has exactly one relation of type `eFrom` and one relation of type `eTo`, as described in Section 4.1.3.

Based on our understanding of the knowledge domain, we assert that all those classes are disjoint with each other, so that any individual can be a member of only one class.

Concept Name	Description
PhysicalWorld	Entities that have a physical form
Person	A human being
Building	A physical building
Printer	A printer device
CyberWorld	Entities that do not have a physical form but exist virtually
Identity	A virtual form of identification
Filesystem	A resource that manages how data are stored and accessed
File	A resource that stores information
Directory	A collection of files or other directories

(a) Description of concept names in the ontology.

Concept Name	Description
PrinterGroup	An entity that represents the agglomeration of a printer, the files that were printed to it, and the identities that used the printer.
Edges	Represents all the role names
ShareFile	See corresponding entry in Table 5.2b
SharePrinter	See corresponding entry in Table 5.2b
ChangePrinter	See corresponding entry in Table 5.2b

(b) Description of additional concept names in the ontology.

Table 5.1: The different shades of blue indicate levels in the concept hierarchy; the darker the shade, the higher a concept name is in the hierarchy. The lighter entries below a concept name are members of that concept name.

Role names. The main role hierarchy is shown in Figure 5.3 which illustrates the relations between entities in the target system. An explanation of the terms is given in Table 5.2a. A few other roles that will be used in subsequent examples are explained in Table 5.2b.

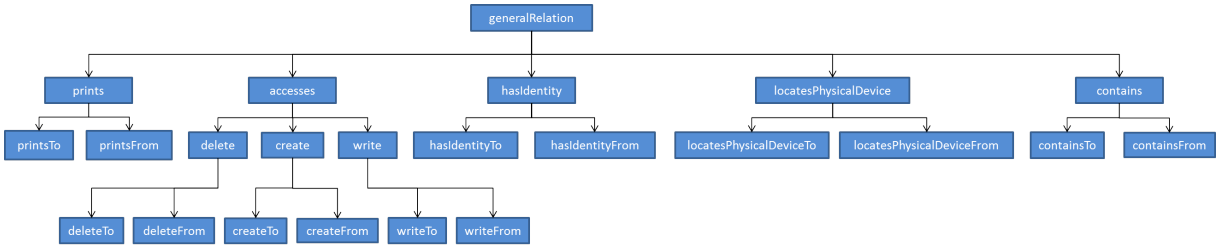


Figure 5.3: The role hierarchy used in the ontology `base`. An arrow from role A to role B indicates that $B \sqsubseteq A$.

A print event is a four-tuple consisting of the credential, file, machine, and printer. However, a CPTL data model only covers binary edges, whereas a print event is a hyperedge of size four. We intend to extend our definition of the CPTL data model to model hyperedges in Chapter 7 when we discuss future work. For now, we instead model the print event as three binary edges. Each edge is an instance of the `prints` role name. For a print event $(\texttt{Identity}_i, \texttt{File}_i, \texttt{Machine}_i, \texttt{Printer}_i)$ we create three edges: $\texttt{prints}(\texttt{Identity}_i, \texttt{File}_i)$, $\texttt{prints}(\texttt{Identity}_i, \texttt{Printer}_i)$, and $\texttt{prints}(\texttt{File}_i, \texttt{Printer}_i)$.

Feature names. Finally, we describe the feature names in the ontology. An explanation of the terms is given in Table 5.3a. A few other features that will be used in subsequent examples are explained in Table 5.3b.

We divide the timestamp of events into separate components: *year*, *month*, *day*, *hour*, *minute*, and *second*. That division allows us to conduct analysis at different time granularities.

Individual names. We assert that all individual names in the ontology are distinct to avoid problems with Open World Reasoning. In Open World Reasoning, an individual that is not asserted as distinct can be inferred by the *TBox* axioms to be the same individual as another individual. However, two individuals would never be treated as one. So, we "close" the world by constructing an axiom that asserts that all individuals are distinct.

Role Name	Description
contains	A **Directory** *contains* another **Directory** or **File**; i.e., a directory has another directory or file inside it
locatesPhysicalDevice	A **Building** *locatesPhysicalDevice* a **Printer**; i.e., the printer device is located within a building
hasIdentity	A **Person** *hasIdentity* an **Identity**; i.e., a person has an associated identity
prints	An **Identity** *prints* a **File** or an **Identity** *prints* to a **Printer** or a **File** is *prints* to a **Printer**; i.e., an identity prints information in a file to a printer
accesses	An **Identity** *accesses* a **Directory** or **File**; i.e., an identity interacts with a directory or file
write	An **Identity** *write* a **File**; i.e., an identity writes information to a file
create	An **Identity** *create* a **Directory** or **File**; i.e., an identity makes a new directory or file
delete	An **Identity** *delete* a **Directory** or **File**; i.e., an identity removes a directory or file

(a) Description of role names in the ontology.

Role Name	Description
changePrinter	A **PrinterGroup** *changePrinter* to another **PrinterGroup**; i.e., a person printed a file to the printer in the first **PrinterGroup** and subsequently to the printer in the second **PrinterGroup**
shareFile	A **PrinterGroup** *shareFile* with another **PrinterGroup**; i.e., the same file is printed to the printers in both **PrinterGroup**s
shareIdentity	A **PrinterGroup** *shareIdentity* with another **PrinterGroup**; i.e., the same person printed to the printers in both **PrinterGroup**s

(b) Description of additional role names in the ontology.

Table 5.2: The different shades of blue indicate levels in the role hierarchy; the darker the shade, the higher a role name is in the hierarchy. The lighter entries below a role name are members of that role name.

Feature Name	Description
name	A *String* used for identification of an individual
filetype	A *String* that denotes the extension of a file
year	An *Integer* that denotes the year that an event happened
month	An *Integer* that denotes the month that an event happened
day	An *Integer* that denotes the day that an event happened
hour	An *Integer* that denotes the hour that an event happened
minute	An *Integer* that denotes the minute that an event happened
second	An *Integer* that denotes the second that an event happened
floor	An *Integer* that denotes the floor where a printer is located
location	A *String* that denotes the printer's location on the floor
printerState	An *Integer* taking either 0 or 1 where 0 denotes not working and 1 indicates that the printer is in working condition
numDelete	An *Integer* that denotes the number of deletions made by an individual to information
numInsert	An *Integer* that denotes the number of additions made by an individual to information
numPages	An *Integer* that denotes the number of pages of a file printed

(a) Description of feature names in the ontology.

Feature Name	Description
numPrints	An *Integer* that denotes the number of times a person prints to a specific printer
numIdentity	An *Integer* that denotes the number of identities contained in an entity
numFiles	An *Integer* that denotes the number of files contained in an entity

(b) Description of additional feature names in the ontology.

Table 5.3: The feature names that are used in the ontology.

5.4 Profiling User's Behaviors

In this section, we introduce two detectors that baseline user's behaviors. To detect tampering with data, we implement a detector that profiles a person's writes to a file in Section 5.4.1. To detect data exfiltration, we propose profiling a person's printing behavior. So we implement a detector that profiles a person's printing behavior in Section 5.4.2.

5.4.1 Writes to files

We want a way to profile a person's writes to a file. Our aim is to detect an anomalous amount of modifications made by a person to a file as it could indicate tampering with data. The procedure we developed is as follows.

Initialization phase. First, we develop an initial baseline of each person's writes to every file in the file system for a month. We keep a `CPTLModel` data structure called `WriteHistory` that is a view of the full CPTL data model which contains writes to files. The `WriteHistory` is initialized with the first month's writes. We want to express a baseline that limits a person's writes to within three standard deviations of the mean number of modifications made to a file. Thus, we perform two contractions: one to obtain the average number of modifications and one to obtain the standard deviation.

We want to collapse the distinction between people and their credentials. So, we contract each `Person` (represented as P) and their associated credentials (represented as I) into a single supernode. So, the set of selected vertices is defined as $V_C = \{(P_k, I_1, \ldots, I_s) | P_k \in \texttt{Person}^{\mathcal{I}}, \exists e_r(P_k, I_q) \in E(G), e_r \in \texttt{hasIdentity}^{\mathcal{I}} \text{ for } q \in [1, s]\}$. We specify the concept name of the supernode using the *TBox* axiom $\mathbb{T}$ that is defined as $X \equiv T_V(P_k)$. We define the attributes of the supernode using the function $\mathbf{F}$ that is defined as follows.

$$\mathbf{F}(\texttt{name}) = g \tag{5.1}$$

$$g(\{(\mathbf{N}^{\mathbf{j}}_{\mathbf{C}}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_C \}) = \texttt{name}(v_j), v_j \in \texttt{Person}^{\mathcal{I}} \tag{5.2}$$

We specify the role names of the edges incident to the supernode using the *TBox* axiom $\mathbb{S}$.

$\mathbb{S}$ is defined as the set of a single axiom $X \equiv \texttt{write}$ that describes the grouping of edges that are instances of the `write` role name. Finally, we define the edge attributes for the first contraction as the average number of modifications, using the function $\mathbf{H}_1$ that is defined as a set of one tuple $(\mathbb{S}_1, \mathbf{J}_1)$, where $\mathbf{J}_1$ is defined as follows.

$$\mathbf{J}_1(\texttt{numInsert}) \rightarrow g_1$$
$$g_1(\{(\mathbf{N}^{\mathbf{j}}_{\mathbf{C}}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_R\ \}) = \text{average}\{\texttt{numInsert}(v_1), \ldots, \texttt{numInsert}(v_k)\}$$
$$\mathbf{J}_1(\texttt{numDelete}) \rightarrow g_2$$
$$g_1(\{(\mathbf{N}^{\mathbf{j}}_{\mathbf{C}}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_R\ \}) = \text{average}\{\texttt{numDelete}(v_1), \ldots, \texttt{numDelete}(v_k)\}$$

The edge attributes of the `write` edges are the average number of insertions and deletions made by a person P_k to a file F_i. The result of the first contraction is a CPTL data model called `avgModel`.

We define $\mathbf{H}_2$ for the second contraction to obtain the standard deviation as a set of one tuple $(\mathbb{S}_1, \mathbf{J}_2)$ where $\mathbf{J}_2$ is defined as follows.

$$\mathbf{J}_1(\texttt{numInsert}) \rightarrow g_1$$
$$g_1(\{(\mathbf{N}^{\mathbf{j}}_{\mathbf{C}}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_R\ \}) = \text{stddev}\{\texttt{numInsert}(v_1), \ldots, \texttt{numInsert}(v_k)\}$$
$$\mathbf{J}_1(\texttt{numDelete}) \rightarrow g_2$$
$$g_1(\{(\mathbf{N}^{\mathbf{j}}_{\mathbf{C}}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_R\ \}) = \text{stddev}\{\texttt{numDelete}(v_1), \ldots, \texttt{numDelete}(v_k)\}$$

The edge attributes of the `write` edges for that contraction are the standard deviation of the number of insertions and deletions made by a person P_k to a file F_i. The result of the second contraction is a CPTL data model called `stddevModel`.

We can create a variety of rules using this information. In this example, we will assert that P_k can make between $i_{\max} = \texttt{numInsert}_{\text{avgModel}} + 3 \times \texttt{numInsert}_{\text{stddevModel}}$ and $i_{\min} = \texttt{numInsert}_{\text{avgModel}} - 3 \times \texttt{numInsert}_{\text{stddevModel}}$ insertions and $d_{\max} = \texttt{numDelete}_{\text{avgModel}} + 3 \times \texttt{numDelete}_{\text{stddevModel}}$ and $d_{\min} = \texttt{numDelete}_{\text{avgModel}} - 3 \times \texttt{numDelete}_{\text{stddevModel}}$ deletions. In

Chapter 6, we will experiment with a few different rules to determine which one is the best at detecting violations.

So, we create a class $\texttt{P}_\texttt{k}\texttt{WriteF}_\texttt{i}$ that describes the baseline of modifications that person P_k makes to file F_i. The class is described by the following two axioms.

$$\texttt{P}_\texttt{k}\texttt{WriteF}_\texttt{i} \equiv \texttt{P}_\texttt{k}\texttt{WriteF}_\texttt{i} \sqcap (= 1)\texttt{numInsert}.(\geq_{i_{\min}}, \leq_{i_{\max}}) \sqcap (= 1)\texttt{numDelete}.(\geq_{d_{\min}}, \leq_{d_{\max}}) \tag{5.3}$$

$$\texttt{P}_\texttt{k}\texttt{WriteF}_\texttt{i} \equiv \texttt{Write} \sqcap (= 1)\texttt{writeTo}. =_{F_i} \sqcap (= 1)\texttt{writeFrom}. =_{P_k} \tag{5.4}$$

Axiom(5.4) defines the class as containing the writes made from P_k to F_i. More formally, the class contains individuals that are instances of the concept `Write` (which represents edges of type `write`) which has exactly one relation `writeTo` with a range of only F_i and exactly one relation `writeFrom` with a range of only P_k. Axiom(5.3) asserts that individuals of this class must have a number of modifications within the specified range.

Online checking. For each subsequent week, we add the axioms $\texttt{P}_\texttt{k}\texttt{WriteF}_\texttt{i}$ to the ontology. When events representing writes to files occur in the system, those events are asserted as facts in the ontology using *ABox* axioms. We run a HermiT reasoner for each event, and if the ontology is satisfiable, then the write event is permitted. The write event is added to `WriteHistory`. If the ontology is not satisfiable, the write event is shown to a human administrator who judges whether or not the write event is indeed malicious behavior. We simulate that judgment using a random number generator that outputs a real number between 0 and 1. The probability of the write events being malicious is assigned to be 0.2. If the write event is non-malicious, it is permitted and added to `WriteHistory`. If the write event is deemed malicious, it is denied.

Offline analysis. Now, the updated `WriteHistory` contains writes that have been permitted within the last week. We run the same contraction operations on `WriteHistory` and update the class axioms representing the baseline profiles.

5.4.2 Printing files

We want a way to profile a person's preference for printers. Our aim is to detect data exfiltration by noticing print events that involve use of a printer other than the user's default printer choice, when the unusual use cannot be explained by printer state. The procedure we developed is as follows.

Initialization phase. First, we develop an initial baseline of each person's choice of printer. We follow the same initialization process in Section 5.4.1 with the exception that we look at a different role name. We group each person's credentials together and perform contraction on the set of groups of people. So $V_C = \{(P_k, I_1, \ldots, I_s) | P_k \in \texttt{Person}^{\mathcal{I}}, \exists e_r(P_k, I_q) \in E(G), e_r \in \texttt{hasIdentity}^{\mathcal{I}} \text{ for } q \in [1, s]\}$, and $\mathbb{T}$ is specified as $X \equiv T_V(P_k)$. The function $\mathbf{F}$ is defined in Equations 5.1 and 5.2.

We group edges that are instances of role name `prints`; i.e., $\mathbb{S}$ is the set of a single axiom $X \equiv \texttt{prints}$. Finally, we specify the edge attributes of the `prints` role name as follows. The edge attribute is the number of `prints` edges. We define $\mathbf{H}$ as a set of a single tuple $(\mathbb{S}_1, \mathbf{J})$, where $\mathbf{J}$ is defined as follows.

$$\mathbf{J}(\texttt{numPrints}) \rightarrow g$$

$$g(\{(\mathbf{N}^{\mathbf{j}}_{\mathbf{C}}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_R \}) = \text{count}\{v_1, \ldots, v_k\}$$

where $|V_R| = k$.

Then, we can deduce each person's default choice of printer by comparing the frequencies of the person's print jobs sent to various printers.

Finally, we create a class $\texttt{S}_\texttt{k}\texttt{Print}$ that describes the printers that person S_k is likely to use for printing. The class is described by the following two axioms.

$$\texttt{S}_\texttt{k}\texttt{Print} \equiv \texttt{Prints} \sqcap \exists \texttt{printTo.Printer} \sqcap \exists \texttt{printFrom.Person} \sqcap (= 1)\texttt{printFrom}. =_{S_k}$$

$$\texttt{S}_\texttt{k}\texttt{Print} \equiv \texttt{S}_\texttt{k}\texttt{Print} \sqcap ((= 1)\texttt{printTo}. =_{P_l})$$

where P_1 is S_k's default printer.

We also initialize a `CPTLModel` instance `PrintJobDay` that is a view of the full CPTL data model that contains print jobs that occurred in a single day. The `PrintJobDay` will be updated with print jobs as they occur in the system, and once the day is over, the `PrintJobDay` will be re-initialized without any print jobs. The state of the printer, i.e., `printerState`, is set to "working" when the day begins.

Online checking. As print jobs arrive in the system, we assert them as facts in the ontology using *ABox* axioms. We then run a reasoner to detect inconsistencies within the ontology, and if the ontology is satisfiable, the print job is permitted. The print job is added to `PrintJobDay`. If the ontology is not satisfiable, the print job can be automatically blocked.

Offline analysis. Now, the updated `PrintJobDay` is used to infer the status of the printers and update the $\mathtt{S_kPrint}$ axioms.

First, we contract each printer (represented by P) along with the people who printed to it (represented by S) and files that were printed to it (represented by F). So, $V_C = \{(P_i, S_1, \ldots, S_k, F_1, \ldots, F_m) | P_i \in \mathtt{Printer}^{\mathcal{I}}, \exists e_r(S_j, P_i) \in E(G), e_r \in \mathtt{prints}^{\mathcal{I}}$ for $j \in [1, k]$, $\exists e_r(F_j, P_i) \in E(G), e_r \in \mathtt{prints}^{\mathcal{I}}$ for $j \in [1, k]\}$.

The `smallBase` ontology contains a specification of the concept name `PrinterGroup`. The concept `PrinterGroup` is defined by the following axiom
$\mathtt{PrinterGroup} \equiv (= 1)\mathtt{numFiles}. \geq_0 \sqcap (= 1)\mathtt{numPerson}. \geq_0$. We specify $\mathbb{T}$ as $X \sqsubseteq (= 1)\mathtt{numFiles}. =_{\text{count}\{F_1,\ldots,F_m\}} \sqcap$
$(= 1)\mathtt{numPerson}. =_{\text{count}\{S_1,\ldots,S_k\}}$, which would result in the reasoner's inferring that the supernode is of type `PrinterGroup`. The function $\mathbf{F}$ is defined as follows.

$$\mathbf{F}(\mathtt{name}) = g$$

$$g(\{(\mathbf{N^j_C}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_C\ \}) = \mathtt{name}(v_j), v_j \in \mathtt{Printer}^{\mathcal{I}}$$

Our hypothesis is that if a person prints a file F_1 to printer P_1 and then subsequently prints F_1 to another printer, P_2, it means that P_1 was not available. This is based on the assumption

that any person, malicious or not, would rarely print the same file to two different printers unless one of the printers was unavailable.

We detect the occurrence of the event that a person prints a file to different printers when two contracted supernodes share a `Person` vertex and a `File` vertex. We define $\mathbb{S}$ as the following set of axioms.

$$X \equiv \texttt{shareFile} \sqcap \texttt{shareIdentity} \tag{5.5}$$

$$X \equiv \texttt{prints} \tag{5.6}$$

The axiom $\mathbb{S}_1$ in (5.5) means that the role name of the resulting edge is `changePrinter` when both `shareFile` and `shareIdentity` edges exist, and that if only one of either `shareFile` or `shareIdentity` exists, then the role name of the resulting edge takes on the existing edge's role name. The axiom $\mathbb{S}_2$ in (5.6) describes the grouping of edges that are instances of the `prints` role name.

However, the indication of a `changePrinter` edge only informs us that one of the printers incident to the edge is unavailable; it does not indicate which one. So, we need edge attributes of the `prints` edge to inform us about the time of the print job. Therefore, we define $\mathbf{H}$ as

a set of two tuples $(\mathbb{S}_1, \mathbf{J}_1)$ and $(\mathbb{S}_2, \mathbf{J}_2)$, where $\mathbf{J}_1$ and $\mathbf{J}_2$ are defined as follows.

$$\mathbf{J}_1 = \phi$$

$$\mathbf{J}_2(\texttt{hour}) \rightarrow g_1$$

$$g_1(\{(\mathbf{N}^{\mathbf{j}}_{\mathbf{C}}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_R\ \}) = \min_{\texttt{hour}}\{(\texttt{hour}(v_1), \texttt{min}(v_1), \texttt{second}(v_1)), \ldots, \texttt{hour}(v_k), \texttt{min}(v_k), \texttt{second}(v_k))\}$$

$$\mathbf{J}_2(\texttt{minute}) \rightarrow g_1$$

$$g_1(\{(\mathbf{N}^{\mathbf{j}}_{\mathbf{C}}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_R\ \}) = \min_{\texttt{minute}}\{(\texttt{hour}(v_1), \texttt{min}(v_1), \texttt{second}(v_1)), \ldots, \texttt{hour}(v_k), \texttt{min}(v_k), \texttt{second}(v_k))\}$$

$$\mathbf{J}_2(\texttt{second}) \rightarrow g_1$$

$$g_1(\{(\mathbf{N}^{\mathbf{j}}_{\mathbf{C}}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_R\ \}) = \min_{\texttt{second}}\{(\texttt{hour}(v_1), \texttt{min}(v_1), \texttt{second}(v_1)), \ldots, \texttt{hour}(v_k), \texttt{min}(v_k), \texttt{second}(v_k))\}$$

where $|V_R| = k$.

The edge attributes of the `prints` edges are the times of the first print jobs that were sent to P_1 and P_2. With that information, we can identify the printer that is down as the head vertex of the edge with the lowest time.

Now, we can update the $\texttt{S}_\texttt{k}\texttt{Print}$ axioms with the list of printers that person S_k is allowed to use. For each person, if his or her default printer is now unavailable, we list printers that are currently available and *closest in distance* to the default printer's location. In this example, we define *distance* in terms of the number of floors and building location. In particular, printers that are located on the same floor are the closest, followed by printers on different floors of the same building; printers in different buildings are the farthest. We can exchange that definition for a more complex version by modifying the search criteria for printers.

Then, we update the axioms by modifying them as follows.

$$\mathtt{S_k Print} \equiv \mathtt{S_k Print} \sqcap ((= 1)\mathtt{printTo}.=_{P_1} \sqcup \cdots \sqcup = 1\mathtt{printTo}.=_{P_k}) \tag{5.7}$$

where $P_1, \ldots, P_k$ are the printers that are available and closest to the default printer.

However, there may be false positives when a user is printing a confidential document at a secure office printer rather than a common shared computer. We can use additional data sources about the type of printer and document to inform our decision-making process. On top of that, we can incorporate state information from the printers themselves to support or refute our current working hypothesis. Finally, to enhance our policy, we would also want data about the location of the machine that was used to send the print job.

5.5 Misuse of System Resources

5.5.1 Printing of copyrighted material

We want to detect printing of copyrighted material (specifically textbooks), that has been obtained through illegal means. Textbooks represent a good example because they have large numbers of pages. If the whole content of the textbook was printed at once, it would take a long time to print, thus causing congestion of the printer queue. This congestion could cause other people to look into the issue and detect the misuse of system resources. So people could evade detection by printing books chapter by chapter over a period of time. We use contraction to contract all files with the same name together and discover the total number of pages printed.

Of course, that may result in false positives, e.g., if a professor prints out many copies of an exam or a student prints out chapters of a thesis. Such occurrences can be explained by other data sources; for example, the exam date or the thesis deadline (and the knowledge that a student is working towards a thesis) can be correlated with the printing time. Data from diverse sources can be fused to gain a more comprehensive picture.

Initialization phase. First, we initialize a `CPTLModel` data structure called `PrintHistory` which is a view of the full CPTL data model that contains the print events of files. The `PrintHistory` will initially contain only the vertices that are instances of the `Person` or `Identity` concept name. We group each person's credentials together and perform basic contraction on the set of groups of people, much like the contraction operation used in profiling writes to a file (see Section 5.4.1). Therefore,

$$V_C = \{(P_k, I_1, \ldots, I_s) | P_k \in \texttt{Person}^{\mathcal{I}}, \exists e_r (P_k, I_q) \in E(G), e_r \in \texttt{hasIdentity}^{\mathcal{I}} \text{ for } q \in [1, s]\}$$

$$\mathbb{T} \rightarrow X \equiv T_V(P_k) \tag{5.8}$$

$$\mathbf{F}(\texttt{name}) = g \tag{5.9}$$

$$g(\{(\mathbf{N}^{\mathbf{j}}_{\mathbf{C}}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_C\ \}) = \texttt{name}(v_j), v_j \in \texttt{Person}^{\mathcal{I}} \tag{5.10}$$

Online checking. For each print job, we add the event to `PrintHistory`.

Offline analysis. Now, we want to enforce the policy that a person can only print less than a threshold number of pages of a particular file. We represent the threshold as `THRESHOLD_PAGES`.

Then, we contract each person individually (represented by P). Then, $V_C = \{P_k | P_k \in \texttt{Person}\}$, and $\mathbb{T}$ and $\mathbf{F}$ are defined using Equation 5.8, 5.9, 5.10.

We define $\mathbb{S}$ as the set of a single axiom $X \equiv \texttt{prints}$ that describes the grouping of edges that are instances of the `prints` role name. Finally, we define $\mathbf{H}$ as a set of one tuple $(\mathbb{S}_1, \mathbf{H}_1)$, where $\mathbf{J}_1$ is defined as follows.

$$\mathbf{J}_1(\texttt{numPages}) \rightarrow g_1$$

$$g_1(\{(\mathbf{N}^{\mathbf{j}}_{\mathbf{C}}, \bigcup_{1 \leq i \leq n} \Delta^{\mathcal{D}_i}) | v_j \in V_R\ \}) = \text{sum}\{\texttt{numPages}(v_1), \ldots, \texttt{numPages}(v_k)\}$$

where $|V_R| = k$. The edge attribute of the `prints` edges is the number of pages printed by a person of a specific file. If a `prints` edge has a feature value for `numPages` greater than the

THRESHOLD_PAGES, then the human administrator is presented with the information about the person, the file, and the number of pages of the file that was printed.

If the human administrator decides that the person violated the organization's policy, then the user can be denied print access, blocked from printing the same file again, or have a reduced threshold for the number of pages that may be printed.

CHAPTER 6

EVALUATION

In this chapter, we evaluate the performance of a CPTL data model, operations on that model, and the application of the CPTL-aware feedback loop in the context of intrusion detection within a small enterprise system example.

6.1 Implementation Performance

Chapter 4 described the implementation of the CPTL data structure and vertex contraction operations in Java. Now, we discuss experiments that we performed to determine the time and space complexity of using CPTL in the context of a small enterprise setting that was described in Chapter 5. All experiments were conducted on a Windows 7 Home Premium machine with 2.7 GHz CPU core and 4 GB of RAM.

6.1.1 CPTL Model

Space complexity. After parsing the data from the *git* logs and the print jobs, we obtained a graph of 2,684 vertices and 9,048 edges. The `CPTLModel` instance occupied 5.74 MB in memory. We wrote the *ABox* axioms representing the graph to an ontology RDF/XML-formatted file and measured the file size to be 9.81 MB. The number of RDF triples representing the graph was 94,668.

Time complexity. It took on average 9.538 seconds to load the entire graph of 2,684 vertices and 9,048 edges. We generated different views of the graph for each of our intrusion detectors (listed in Chapter 5). All the views retain the vertices of the original graph but only a subset of the edges. `PrintView` is used for generating axioms that detect suspicious

print jobs by looking at the printer location, as explained in Section 5.4.2. That view consists of all `prints` and `hasIdentity` edges. `PrintFileView` is used for generating axioms that detect printing of a large number of pages, as explained in Section 5.5.1. That view consists of `prints` edges between vertices which are instances of `Identity` and `File`, and `hasIdentity` edges. Finally, the `WriteFileView` is used for generating axioms that detect a suspicious amount of writes to files as explained in Section 5.4.1. That view consists of `write` edges and `hasIdentity` edges.

Table 6.1 shows the number of edges in each view, the time complexity of generating the view, and the space complexity of each view. We deduce that the space complexity is linear in the number of edges, whereas the time complexity of generating the views is relatively constant at ~ 0.3 seconds.

Table 6.1: Statistics for generating views of a CPTL data model.

	`PrintView`	`PrintFileView`	`WriteFileView`
Number of edges	3,619	1,217	317
Memory size (MB)	3.33	2.13	1.7
Average time to generate (s)	0.365	0.332	0.321

6.1.2 Vertex Contraction

In our use case examples, the selected number of vertices for the vertex contraction operation ranged between 8 and 35. Thus, the time complexity of the vertex contraction operation was more dependent on the number of edges, which ranged between 0 and 960.

We evaluate the time complexity of the vertex contraction operation with respect to the number of edges in the CPTL data model. Figure 6.1 shows the time taken for the vertex contraction operation, and Figure 6.2 shows the time taken for the verification step. As mentioned in Section 4.3.2, the time complexity of the contraction operation is on the order of $p\log\, p$ with respect to the number of edges, p. We plot the trend-line of $p\log\, p$ in Figure 6.1 and we can observe that the time complexity does follow the trend-line. The time complexity of verification is on the order of a second-order polynomial, and closely follows the trend-line as well.

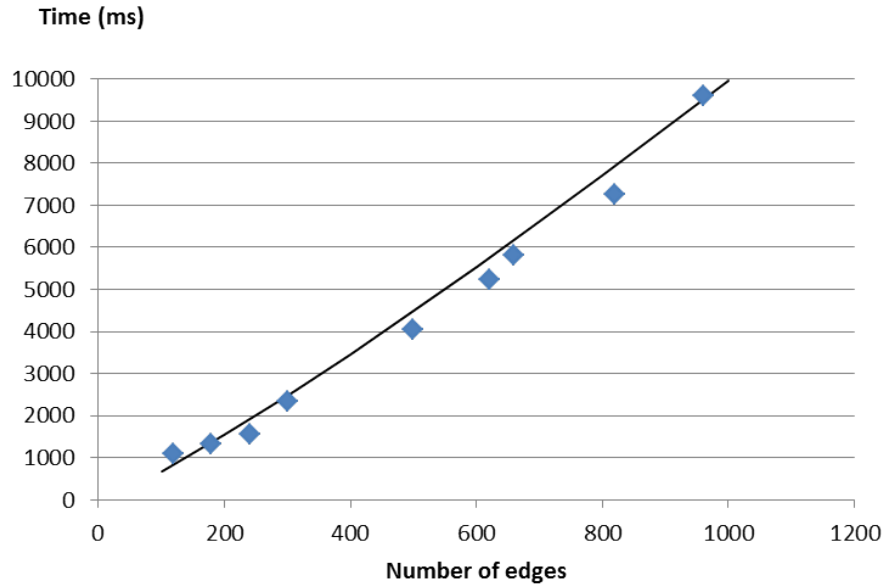


Figure 6.1: Plot of time taken for vertex contraction operation vs. number of edges incident to selected vertices for contraction.

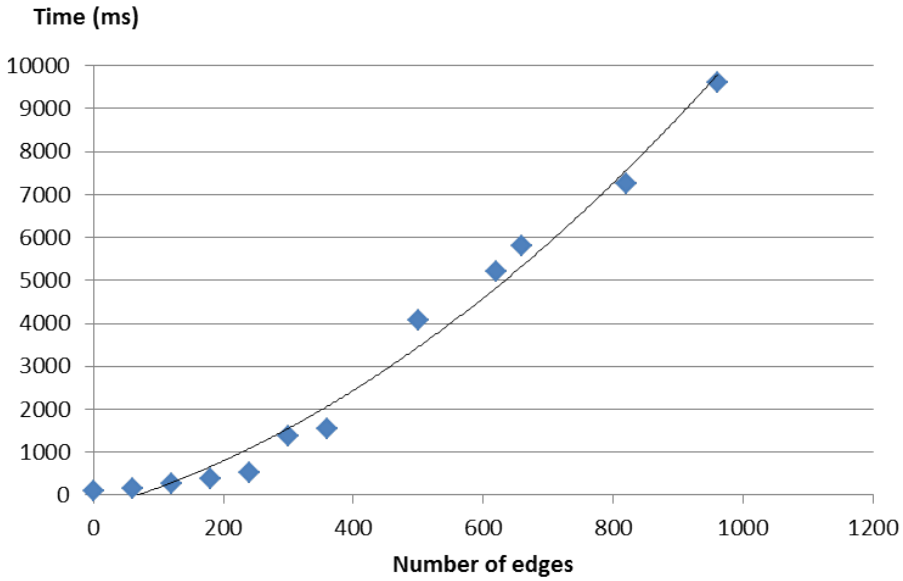


Figure 6.2: Plot of time taken for verifying validity of CPTL data model vs. number of edges in the CPTL data model.

We also evaluate the time complexity of concept and role inferencing. Since the number of axioms in the ontology is constant for all the vertex contraction operations, the time complexity is dependent on the axiom used for concept and role inferencing. Axioms that involved only one primitive concept or role name, like $X \equiv C, C \in \mathbf{N_C}$ or $X \equiv R, R \in \mathbf{N_R}$, took an average of 0.02 milliseconds. Axioms that involved additional operators, like $X \equiv C_1 \sqcap C_2, C_1, C_2 \in \mathbf{N_C}$ or $X \equiv R_1 \sqcap R_2, R_1, R_2 \in \mathbf{N_R}$, took an average of 3.68 milliseconds for concept names and 0.2 milliseconds for role names. Those results show that inferencing is fast for the axioms that we have defined.

Finally, we evaluated the time and space complexity of CPTL and the vertex contraction operation; it was satisfactory. We also measured the false positive and false negative rates of our detectors. The false negative rate was as low as 0.1, although the highest false negative rate was 0.45. The false positive rate was lower than 0.3 and can be reduced by the integration of other data sources that help to explain the occurrence of events. We will provide more details on this in the remainder of our chapter.

6.2 Use Case Performance

In this section, we evaluate the performance of our CPTL-aware feedback loop in the context of the detectors that we implemented in Chapter 5. For each detector, we specify the number of axioms generated and the time complexity of generating axioms, updating the CPTL model, and detecting violations.

6.2.1 Profiling User Writes

In Section 5.4.1, we described a detector that aims to flag anomalous writes to files on a per-person basis. Our results are given below.

Number of axioms. A total of 159 axioms were generated.

Offline analysis. We quantify the time taken for the offline analysis component in the CPTL-aware feedback loop. That component involves updating the CPTL data model and generating axioms for detection. We looked at two different rates of updating the offline analysis component: after a span of one week, and after a write was made. Figure 6.3 shows the time complexity of generating the axioms for baselining people's writes to files after a period of one week, and Figure 6.4 shows the time complexity for the individual write update rate. The average time complexity is 1.8 seconds for the weekly update rate and 1.3 seconds for the individual write update rate. We can see that the time varies from less than a second to a maximum of 6 seconds for the weekly update rate. The time complexity is acceptable since we are conducting the generation at the end of every week. The time complexity for the individual write update rate is below 3 seconds, which is also acceptable, since typically there are write commits only every few minutes.

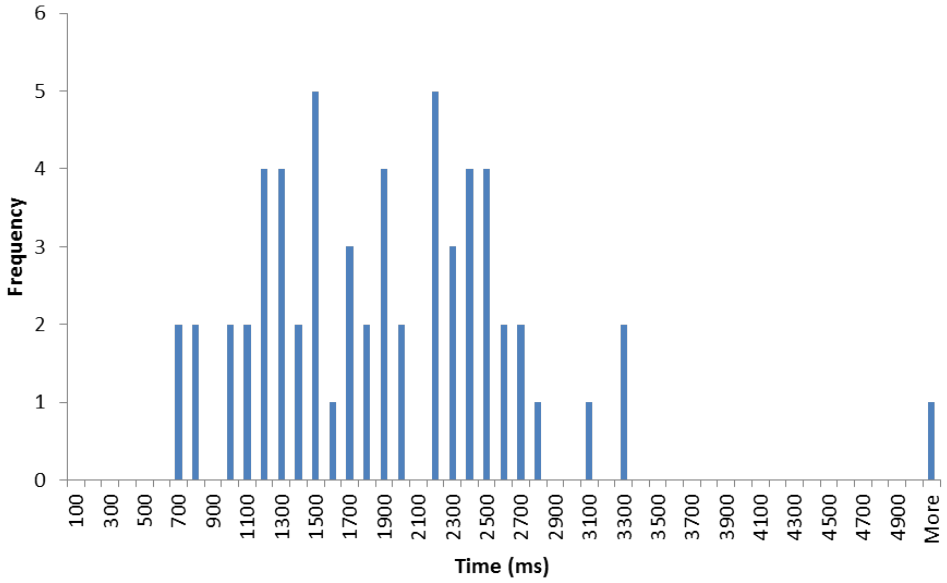


Figure 6.3: Histogram of time complexity of generating axioms that baseline writes to files with a weekly update rate.

Online checking. We quantify the time taken for the online checking component in the CPTL life cycle. That component involves running of a HermiT reasoner to check for consistency and updating of the CPTL data model. Figure 6.5 shows the time complexity

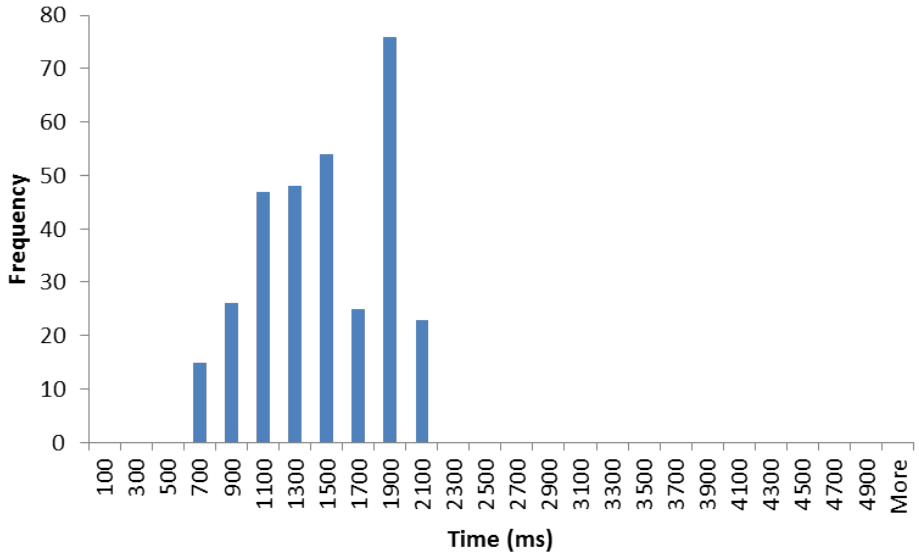


Figure 6.4: Histogram of time complexity of generating axioms that baseline writes to files with an individual write update rate.

of checking the consistency of the axioms. The average time complexity is 65 milliseconds. We can see that the time varies between 10 milliseconds and 750 milliseconds

The time complexity for updating the CPTL data model is consistently below 200 milliseconds. Thus, we can conclude that the most time-consuming aspect of the online checking phase is the consistency check.

Detection rate. In our dataset, we assume that all writes are non-malicious. So we need to generate synthetic test data that simulate malicious actions. We describe the procedure of generating synthetic test data as follows. First, we randomly choose a user-file pair and inject 10 malicious writes over a period of 10 weeks. Five user-file pairs are chosen, giving rise to 50 malicious writes in total.

We limit the files to *.tex* and *.pdf* files because our repository consists mainly of these two file types. For each selected user and file (U_1-F_1) pair, we randomly choose another file, say F_2, that has the same extension as the selected file F_1. Then, we randomly select a write made to F_2 and note down the number of modifications made to F_2. That number of modifications is used as the malicious write for the U_1-F_1 pair.

We also used the number of standard deviations to modify the baselines to be either more

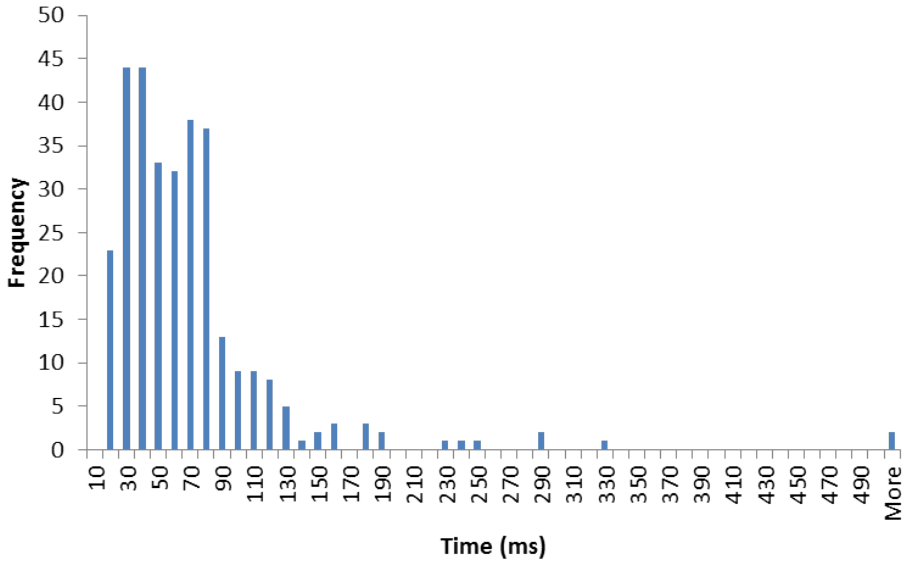


Figure 6.5: Histogram for time complexity of detecting anomalous writes to file.

specific or more general. The higher the number of standard deviations, the larger the range of possible modifications, and thus the baseline is more general.

We ran the procedure five times and obtained the average false positive and false negative rates. Figure 6.6 shows the false negative rate, and Figure 6.7 shows the false positive rate.

We can deduce that as the baseline is made more specific, the false negative rate decreases, and therefore more malicious writes are caught. The *.tex* files' malicious modifications are caught more often than the *.pdf* modifications because of the *.pdf* files' higher variance in modifications. In general, the weekly update rate outperforms the individual write update rate, which indicates that the modifications are relatively steady over a week but vary greatly from write to write.

On the other hand, as the baseline is made more specific, the false positive rate generally increases, with the exception of *.pdf* files with the individual write update rate. That trend is typical of IDSes that use baselining to detect malicious activity. However, the false positive rates across all configurations are within 0.05 of each other, which implies that we should focus on the false negative rate when choosing the specificity of the baseline. We can see that the weekly update rate greatly outperforms the individual write update rate in terms of false positives. Therefore, the best baseline would be a weekly update rate with a standard

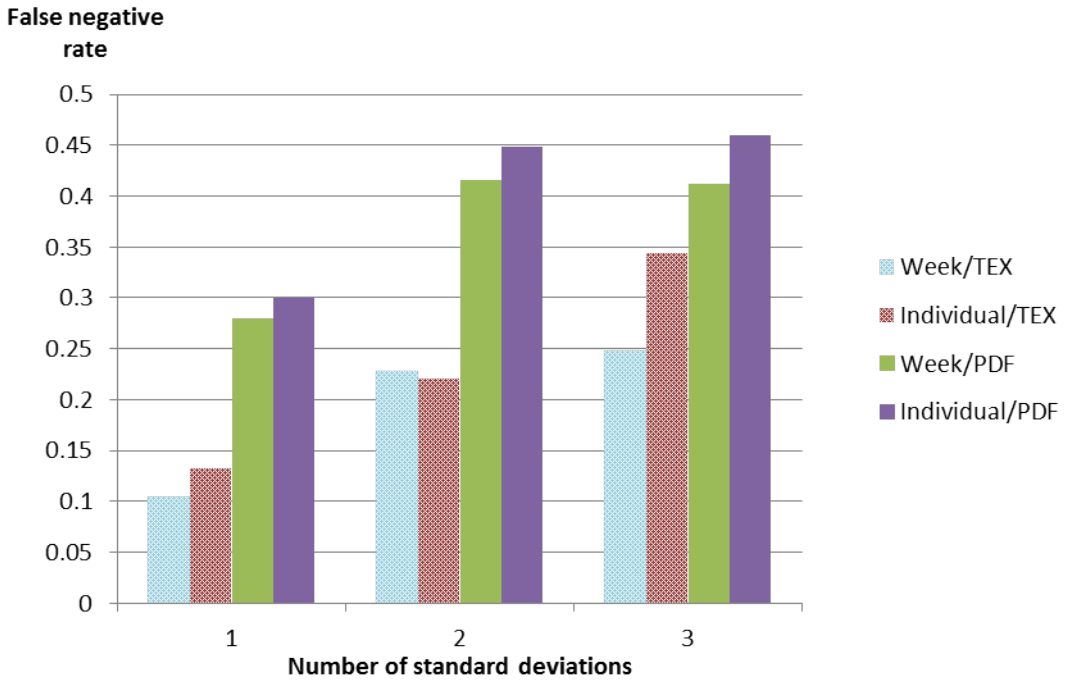


Figure 6.6: False negative rate of detecting anomalous writes.

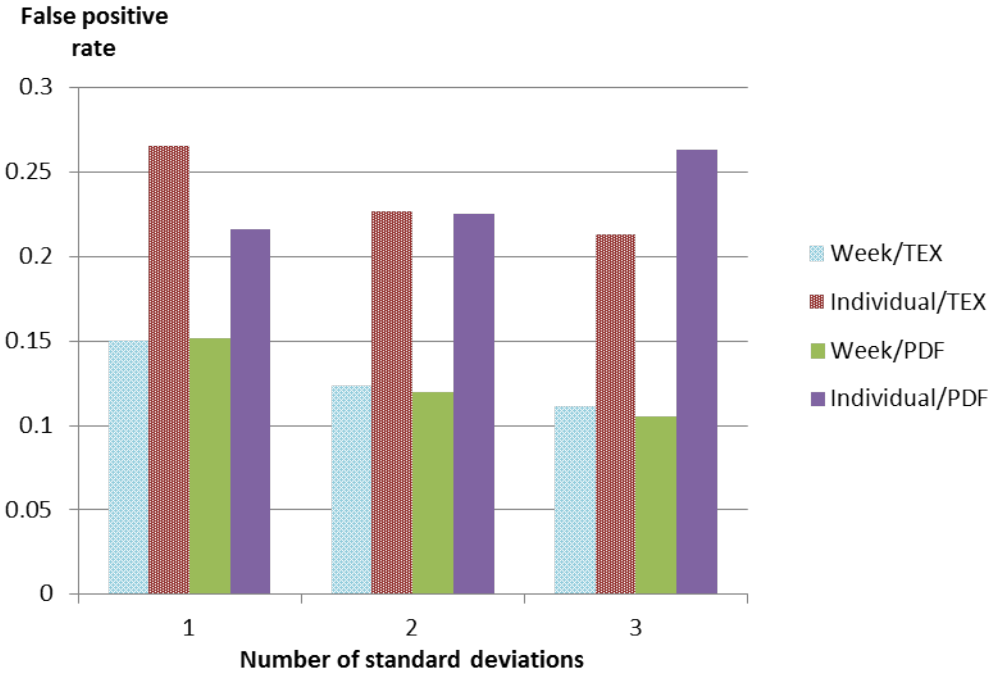


Figure 6.7: False positive rate of detecting anomalous writes.

deviation of 1.

6.2.2 Profiling User Printing

In Section 5.4.2, we described a detector that aims to detect data exfiltration based on people's default printer preferences, printer status, and printer location. Our results are given below.

Number of axioms. A total of 6 axioms were generated on a daily basis.

Offline analysis. We quantify the time taken for the offline analysis component in the CPTL lifecycle. Figure 6.8 shows the time complexity of generating axioms that specify printing behavior. The average time complexity is 296 milliseconds.

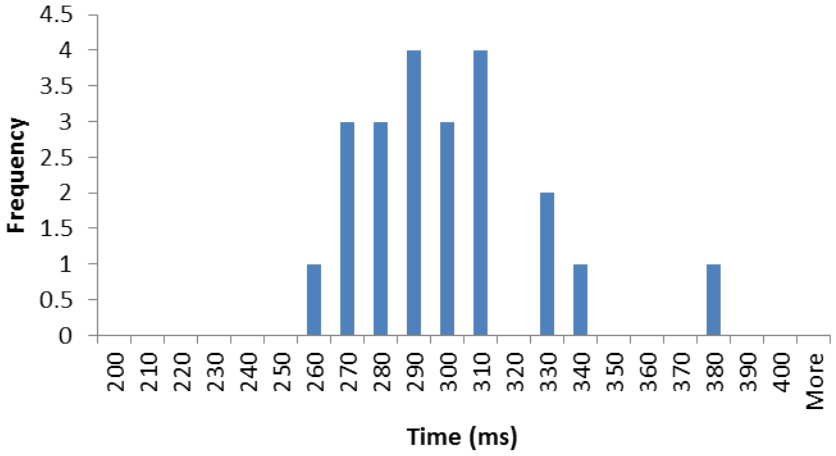


Figure 6.8: Histogram of time complexity of generating axioms that specify normal printing behavior.

Online analysis. We quantify the time taken for the online checking component of the CPTL lifecycle. Figure 6.9 shows the time complexity of checking the consistency of the axioms. The average time complexity is 89 milliseconds. The time varies between 60 milliseconds and 150 milliseconds, which is faster than the performance described in Section 6.2.1 because fewer axioms need to be checked for consistency.

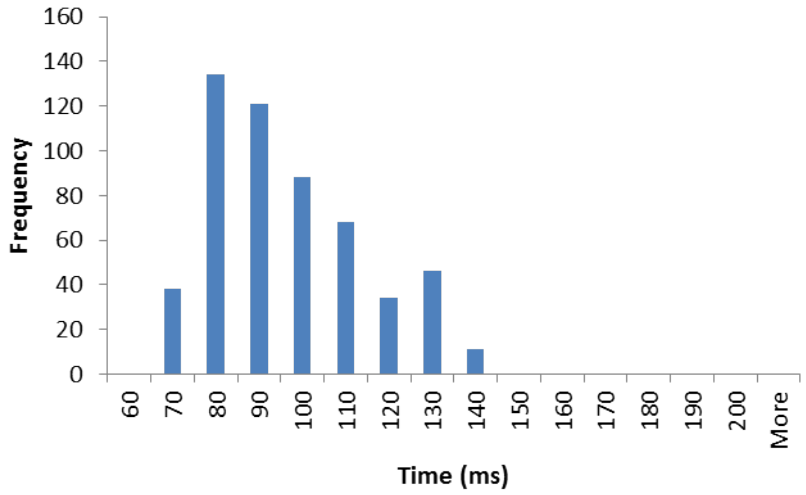


Figure 6.9: Histogram for time complexity of detecting illegal print jobs.

The time complexity for updating the CPTL data model is consistently below 60 milliseconds. Much as described in Section 6.2.1, the most time-consuming aspect of the online phase is the consistency check.

Detection rate. Unlike the baseline detector discussed in Section 6.2.1, the detection for printing is based on a policy specification that flags print jobs as suspicious if the person printed to a printer other than the default printer. So any print jobs that do not fulfill the specification are caught by the axiom. Thus, we only discuss false positives in this section.

There were 20 violations from a total of 3,594 print jobs. Out of the 20 violations, 14 were print jobs that were sent to a printer in a different building from the default printer, from a machine that was located close to the printer used. That evidence indicates that we could reduce the number of false positives if we had more information. The other six of the violations were print jobs that were sent to a neighboring printer close to the default printer even though the default printer was inferred to be working. However, we only infer the printer state based on our hypothesis of human behavior. In fact, someone may print a file to two different printers for an appropriate reason. For example, a secretary could want to save time by printing multiple copies of a report to two different printers. In the example of the six violations, the employee happened to find that the default printer was not working

and subsequently printed to the neighboring printer. Thus, there was no occurrence of a file's being printed to both printers. We can enhance this hypothesis with printer logs that indicate whether a printer is working.

6.2.3 Misuse of System Resources

In Section 5.5.1, we describe a detector that aims to detect abuse of system resources by keeping track of the number of pages of a file being printed. Our results are given below.

Offline analysis. We quantify the time taken for the offline analysis component in the CPTL lifecycle. Figure 6.10 shows the time complexity of updating the CPTL data model with print jobs. The time taken is exponentially related to the number of edges in the current CPTL data model. We can reduce the time complexity by discarding older data or ignoring print jobs when the file being printed is deemed to be of low value to the organization.

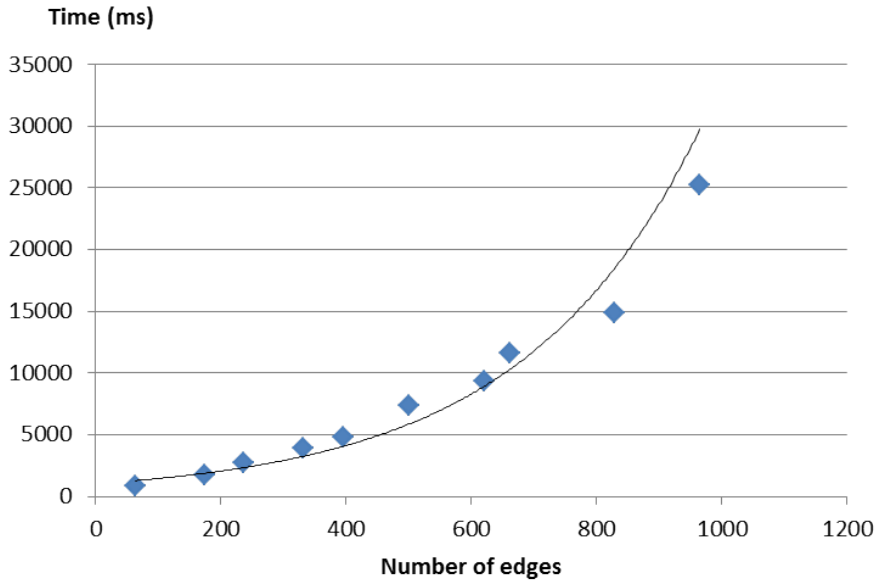


Figure 6.10: Plot of time taken for updating CPTL data model vs. number of edges in CPTL data model.

Online analysis. We quantify the time taken for the online checking component in the CPTL lifecycle. The time complexities of checking for violations and updating the model

are both consistently less than 1 millisecond, because this detector does not require that a reasoner be run over the data to check for consistency.

Detection rate. Just as described in Section 6.2.2, we only discuss false positives that do not fulfill the specification.

There were 4 violations from a total of 3,594 print jobs. Two of the violations were print jobs that involved multiple copies of the file for distribution purposes. To reduce false positives, we can use machine learning to detect such files. For example, a simple classifier could look for words such as "Distribution" or "ProblemSet". One of the violations was an accidental print job that was eventually canceled by the user. The last violation occurred because certain documents that were newly downloaded and subsequently printed were being assigned a default name. Thus, an accumulation of the print jobs of such documents culminated in a violation. We can use information such as the cancellation of a print job or knowledge about document naming to reduce false positives.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

In this thesis, we discussed the limitations of IDSes and explained the need for a language that describes an online "world view" that maintains information about the state of a system. We then presented a definition of the Cyber-Physical Topology Language and its operations that is grounded in graph theory and Description Logics. CPTL provides a formal specification of a target system and it can be queried by operations to obtain information about the target system. In particular, we introduced a vertex contraction operation that presents a summarized view of a target system. We proposed a CPTL-aware feedback loop that describes an approach to combine the more intensive analysis that uses operations, and the less intensive processing of events that occur in the target system.

To illustrate the practicality of our theoretical framework, we provided an implementation of CPTL and vertex contraction, and applied it to an enterprise setting to detect suspicious user behavior. We developed three detectors that used vertex contraction to extract features of the target system. Then, we simulated the arrival of events in the target system, the updating of the CPTL data model and the detectors, and the checking of the events for violations. The results showed that our approach is efficient and has a low false negative and false positive rate.

We conclude that our Cyber-Physical Topology Language and operations are a promising theoretical framework that allows for the maintenance of an "online world view" that can be used in the context of intrusion detection.

7.2 Future Work

We will now discuss the avenues for future work in terms of the theoretical framework, implementation, application, and evaluation of CPTL and its operations.

First, our current definition of CPTL does not model hyperedges. We worked around this caveat by modeling the `prints` hyperedge as three separate binary edges. We intend to explore different ways of extending the CPTL definition to model such occurrences of non-binary relations.

Second, we mentioned in Chapter 3 that we do not handle complex $\mathbb{S}$ axioms that involve role inclusions. However, those complex axioms are also useful. For example, if we had a CPTL data model that described the architecture of monitor placement in a data center environment, then we could perform contraction on a single physical machine to view the monitors residing on it. If a monitor on the machine was also monitoring another physical machine, we would want to represent the relation between the supernode and the other physical machine as that of sharing a monitor. That relation can be represented in terms of role inclusion; i.e., $\mathbf{shareMonitor} \equiv \mathbf{hasMonitor}^{-1} \circ \mathbf{hasMonitor}$.

However, no inference problems have been defined for role names with that amount of complexity. We could resort to brute-force methods. For example, we could manually search through all *TBox* axioms for role definitions that satisfy the given *TBox* axiom. Brute-force method, however, is a last resort. We will investigate other approaches to solving that problem.

We also intend to develop the other operations mentioned in Section 3.4, i.e., **Join** and **Abstract**. Eventually, we aim to develop a suite of operations that can be combined together to obtain interesting and useful features of the target system.

Third, our current implementation of the CPTL data model is a preliminary version that is meant to test the feasibility and practicality of our theoretical framework. We intend to optimize our implementation in terms of space complexity and time complexity.

Fourth, our dataset involves a small set of people and two data sources: *git* logs and print jobs. We plan to obtain a larger sample of people and augment our CPTL data model with other sources of information, such as printer logs and machine location. We also intend to

look into other methods of evaluating our detectors, such as simulation by red teams.

Finally, we intend to apply our CPTL life cycle to other target systems, such as the power grid and data centers. In general, we can also apply our CPTL operations to other areas, such as privacy. For example, in the context of exchange of information across organizations, we can use vertex contraction to present a higher-level architectural view of a target system without revealing specific details of the machines.

REFERENCES

[1] United States Computer Emergency Readiness Team, "US-CERT year in review," U.S. Department of Homeland Security, Year in Review Report, 2012.

[2] Symantec Corporation, "Internet security threat report," 2013 Trends, April 2014.

[3] L. Bilge and T. Dumitras, "Before we knew it: An empirical study of zero-day attacks in the real world," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382284 pp. 833–844.

[4] J. Flynn, "Intrusion Along the Kill Chain," in *Proceedings of BlackHat USA*. BlackHat, August 2012.

[5] N. Stakhanova, S. Basu, and J. Wong, "On the symbiosis of specification-based and anomaly-based detection," *Computers & Security*, vol. 29, no. 2, pp. 253 – 268, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404809000893

[6] S. Axelsson, "Intrusion detection systems: A survey and taxonomy," Chalmers University of Technology, Tech. Rep., 2000.

[7] H.-J. Liao, R. C.-H. Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16–24, 2013.

[8] G. A. Weaver, C. Cheh, E. J. Rogers, W. H. Sanders, and D. Gammel, "Toward a cyber-physical topology language: Applications to nerc cip audit," in *Proceedings of the First ACM Workshop on Smart Energy Grid Security*, ser. SEGS '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2516930.2516934 pp. 93–104.

[9] E. Rogers, W. Rogers, and G. A. Weaver, "Badger: The Networked Security State Estimation Toolkit," in *Proceedings of BlackHat USA*. BlackHat, August 2014.

[10] E. Schultz, "A framework for understanding and predicting insider attacks," *Computers and Security*, vol. 21, no. 6, pp. 526–531, Oct. 2002. [Online]. Available: http://dx.doi.org/10.1016/S0167-4048(02)01009-X

[11] M. L. Collins, D. Spooner, D. M. Cappelli, A. P. Moore, and R. F. Trzeciak, "Spotlight On: Insider Theft of Intellectual Property Inside the United States Government Involving Foreign Governments or Organizations," Software Engineering Institute, Technical Note, 2013.

[12] A. Cummings, T. Lewellen, D. McIntire, A. Moore, and R. Trzeciak, "Insider Threat Study: Illicit Cyber Activity Involving Fraud in the U.S. Financial Services Sector," Software Engineering Institute, Special Report, 2012.

[13] M. Salem, S. Hershkop, and S. J. Stolfo, "A survey of insider attack detection research," in *Insider Attack and Cyber Security: Beyond the Hacker*. Springer, 2008, pp. 69–90.

[14] A. Patel, M. Taghavi, K. Bakhtiyari, and J. C. Jnior, "An intrusion detection and prevention system in cloud computing: A systematic review," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 25 – 41, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S108480451200183X

[15] I. Corona, G. Giacinto, and F. Roli, "Adversarial attacks against intrusion detection systems: Taxonomy, solutions and open issues," *Information Sciences*, vol. 239, pp. 201–225, 2013.

[16] S. More, M. Matthews, A. Joshi, and T. Finin, "A knowledge-based approach to intrusion detection modeling," in *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on*. IEEE, 2012, pp. 75–81.

[17] National Cybersecurity and Communications Integration Center, "Combating the insider threat," U.S. Department of Homeland Security, Technical Publication, 2014.

[18] A. K. Jones and R. S. Sielken, "Computer system intrusion detection: A survey," Department of Computer Science, University of Virginia, Tech. Rep., 2000.

[19] Anomaly Detection at Multiple Scales (ADAMS) Broad Agency Announcement DARPA-BAA-11-04, Defense Advanced Research Projects Agency 2010, Arlington, VA.

[20] T. Senator, H. Goldberg, A. Memory, W. Young, B. Rees, R. Pierce, D. Huang, M. Reardon, D. Bader, E. Chow, I. Essa, J. Jones, V. Bettadapura, D. Chau, O. Green, O. Kaya, A. Zakrzewska, E. Briscoe, R. Mappus, R. McColl, L. Weiss, T. Dietterich, A. Fern, W.-K. Wong, S. Das, A. Emmott, J. Irvine, J.-Y. Lee, D. Koutra, C. Faloutsos, D. Corkill, L. Friedland, A. Gentzel, and D. Jensen, "Detecting insider threats in a real corporate database of computer usage activity," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2487575.2488213 pp. 1393–1401.

[21] A. Memory, H. Goldberg, and T. Senator, "Context-aware insider threat detection," in *Proceedings of the Workshop on Activity Context System Architectures*, 2013, pp. 44–47.

[22] W. Young, H. Goldberg, A. Memory, J. Sartain, and E. Ted, "Use of domain knowledge to detect insider threats in computer activities," in *Security and Privacy Workshops (SPW), IEEE*. IEEE, 2013, pp. 60–67.

[23] T. Senator, H. Goldberg, and A. Memory, "Distinguishing the unexplainable from the merely unusual: Adding explanations to outliers to discover and detect significant complex rare events," in *Proceedings of the ACM SIGKDD Workshop on Outlier Detection and Description*, ser. ODD '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2500853.2500861 pp. 40–45.

[24] R. Mitchell and I.-R. Chen, "A survey of intrusion detection techniques for cyber-physical systems," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 55:1–55:29, Mar. 2014. [Online]. Available: http://doi.acm.org/10.1145/2542049

[25] M. Maloof and G. Stephens, "elicit: A system for detecting insiders who violate need-to-know," in *Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, C. Kruegel, R. Lippmann, and A. Clark, Eds. Springer Berlin Heidelberg, 2007, vol. 4637, pp. 146–166.

[26] T. Berners-Lee, "Linked data," 2009. [Online]. Available: http://www.w3.org/DesignIssues/LinkedData.html

[27] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, Eds., *The Description Logic Handbook: Theory, Implementation, and Applications*. New York, NY, USA: Cambridge University Press, 2003.

[28] M. Krötzsch, F. Simančík, and I. Horrocks, "A description logic primer," *CoRR*, vol. abs/1201.4089, 2012. [Online]. Available: http://arxiv.org/abs/1201.4089

[29] M. Aranguren, E. Antezana, and R. Stevens, "N-ary relationship," *Ontology Design Patterns Public Catalog*, 2009. [Online]. Available: http://www.gong.manchester.ac.uk/odp/html/Nary_Relationship.html

[30] World Wide Web Consortium (W3C), "Defining n-ary relations on the semantic web," *W3C Working Group Note*, 2006. [Online]. Available: http://www.w3.org/TR/swbp-n-aryRelations/

[31] C. A. Gunter, D. Liebovitz, and B. Malin, "Experience-based access management: A life-cycle framework for identity and access management systems," *IEEE security & privacy*, vol. 9, no. 5, p. 48, 2011.

[32] World Wide Web Consortium (W3C), "OWL 2 web ontology language document overview," *W3C Recommendation*, 2012. [Online]. Available: http://www.w3.org/TR/owl2-overview/

[33] R. Shearer, B. Motik, and I. Horrocks, "Hermit: A highly-efficient owl reasoner." in *OWLED*, ser. CEUR Workshop Proceedings, C. Dolbear, A. Ruttenberg, and U. Sattler, Eds., vol. 432. CEUR-WS.org, 2008. [Online]. Available: http://dblp.uni-trier.de/db/conf/owled/owled2008.html

[34] World Wide Web Consortium (W3C), "OWL 2 web ontology language profiles (second edition)," *W3C Recommendation*, 2012. [Online]. Available: http://www.w3.org/TR/owl2-profiles/

[35] A. Lumsdaine, L.-Q. Lee, and J. G. Siek, *The Boost Graph Library: User Guide and Reference Manual*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[36] D. Raggett, "HTML Tidy library project," 2004. [Online]. Available: http://tidy.sourceforge.net