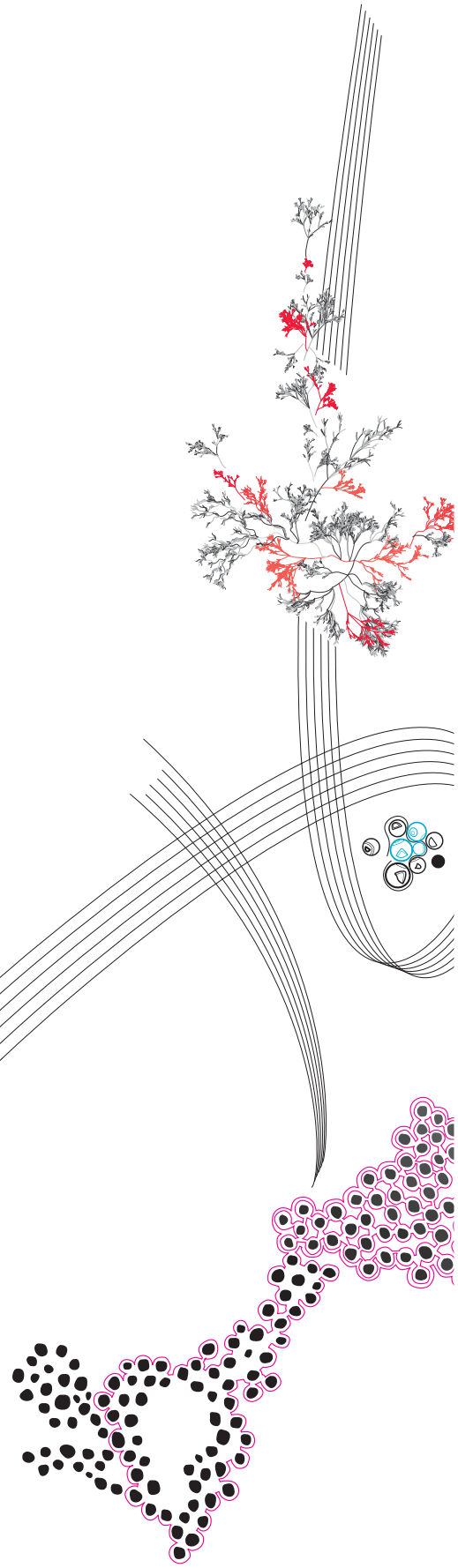


MSc Computer Science
Final Project

Model Checking DOGLog: Implementing Risk Assessment with Object-Oriented Disruption Graphs

Caz Saaltink

Supervisor: dr. Stefano Maria Nicoletti
Co-supervisor: dr.ing. Ernst Moritz Hahn
2nd supervisor: dr. Tom van Dijk
External supervisor: prof.dr. Giancarlo Guizzardi

June, 2025

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Abstract

Risk assessment in complex systems increasingly requires the consideration of both safety and security. Traditional methods often face challenges in unifying these aspects and explicitly modeling the objects at risk within a system. The WATCHDOG framework addresses these challenges by introducing object-oriented DisruptiOn Graphs (DOGs). DOGs combine fault trees, attack trees, and object graphs to provide a comprehensive modeling approach. To reason about these DOGs, WATCHDOG has DOGLog, a specialized three-layered logic. This thesis details the design and implementation of a model checker for DOGLog. The core contribution is the development and implementation of model checking algorithms for DOGLog, which utilize Binary Decision Diagrams (BDDs) and Multi-Terminal BDDs (MTBDDs) for efficient analysis. The developed algorithms support DOGLog's three layers: Boolean disruption propagation, probability calculations, and object-specific risk computations. The practical utility of DOGLog is demonstrated through a comprehensive case study of a cyber-physical pipeline system. This case study illustrates how ODF, the implemented tool, is used to analyze complex risk scenarios, identify optimal system configurations, and aid in the validation and refinement of risk models by uncovering subtle modeling issues. The thesis concludes by discussing current limitations and proposing future enhancements for both DOGLog and the broader WATCHDOG framework.

Keywords: fault trees, attack trees, binary decision diagrams, formal methods, risk assessment, model checking, object-oriented risk analysis

Contents

Chapter 1

Introduction

Fault trees are widely used in safety and reliability engineering to ensure that systems are designed to be robust against failures [18, 39]. They model how component failures propagate through a system to cause a system-level failure. This model enables engineers to identify critical components and design redundancies to prevent system failures. Attack trees are a similar model used in security engineering to model how an attacker can compromise a system [21, 46]. They assist security engineers in identifying possible attack vectors and designing appropriate defensive measures.

Research has shown that safety and security are closely related concerns that should often be considered together [12, 13, 19, 21, 50]. This recognition has led to the development of methods that combine fault trees and attack trees into unified models [19, 21, 25, 29]. Such integrated models enable engineers to analyze the interactions between system failures and attacks, leading to improvements in system robustness. However, safety and security requirements do not always align, i.e., sometimes improving a system's safety can come at the cost of its security, and vice versa. A notable example comes from a European luxury car manufacturer that implemented an automatic door unlocking mechanism for crash situations [47]. While this safety feature was intended to help in accidents, it created a security vulnerability where attackers could trigger the system by applying pressure to the car's roof, thereby gaining unauthorized access.

The WATCHDOG framework [31] is a novel approach that combines fault trees and attack trees with *object graphs* to model relationships between events/actions and *objects at risk*. By connecting both types of trees to objects, WATCHDOG allows us to analyze optimal properties of those objects to minimize the risk they are exposed to. Consider the example of the car crash door unlocking mechanism earlier. Figure 1.1 shows an *object-oriented DisruptiOn Graph* (DOG) modeling different events, actions, and objects in the system of a car. The DOG consists of three main components: an attack tree (left), a fault tree (right), and an object graph (bottom). The object graph shows relevant objects in the system and their relationships. An arrow between objects indicates that one object is a part of another object. For instance, the lock is part of the door, which in turn is part of the car. Events and actions can be connected to *conditions*, e.g., the event "Door locked" is connected to the condition "!Crash_Unlock". The condition consists of a negation operator "!" and the *object property* "Crash_Unlock", which represents the mechanism that unlocks the door when the car crashes. When a condition is connected to an event or action, it means that the event or action can only occur if the condition is satisfied. In this case, the door can only be locked if the crash unlocking mechanism is disabled. From a safety perspective, having the crash unlocking mechanism enabled is beneficial, as it prevents the "!Crash_Unlock" condition from being satisfied. However, from a security perspective, this

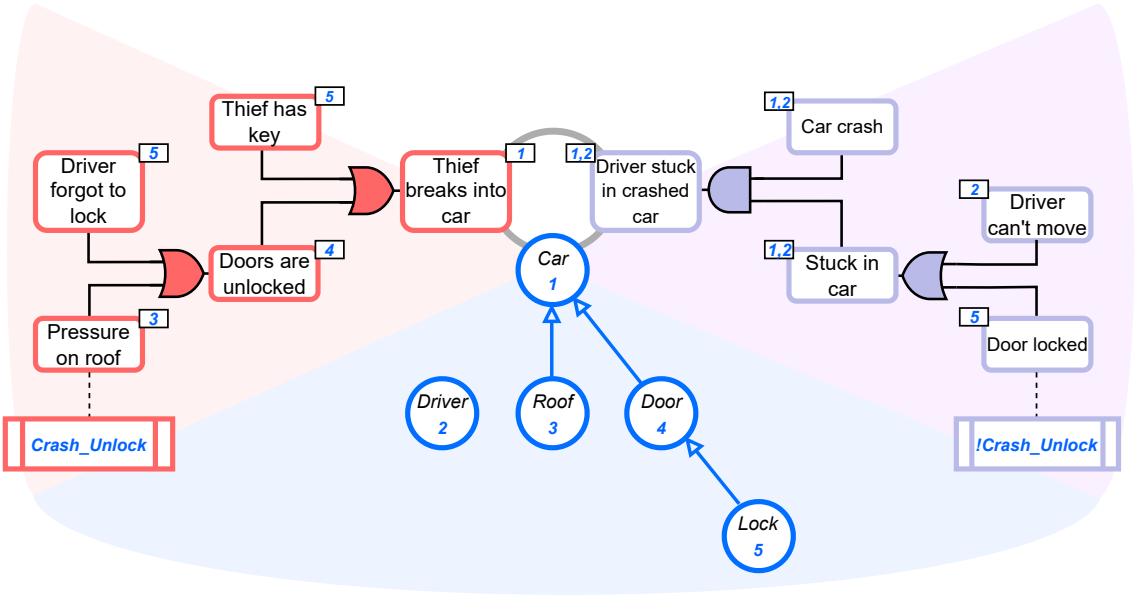


FIGURE 1.1: Example DOG for the car crash door unlocking mechanism

mechanism creates a vulnerability by allowing attackers to unlock the doors by applying pressure to the roof. Using the WATCHDOG framework, one can analyze the optimal configuration of object properties to minimize the risk exposure of the car.

Another important aspect of WATCHDOG is its ability to analyze risk exposure at the object level [31]. For example, with WATCHDOG, one can compute the minimum and maximum risk exposure of the car, or determine the most risky event or action the driver participates in. This is where the relation between objects and events/actions comes into play. These relationships are shown by the small blue numbers in the corners of the nodes in the attack and fault trees. For example, the action "Thief has key" only affects the lock, while the event "Car crash" affects both the driver and the car, which in turn affects the roof, door, and lock as they are part of the car (as indicated by the arrows).

To perform these risk calculations and determine optimal property configurations, Nicoletti et al. developed a custom logic called DOGLog [31]. In addition to the mentioned computations about objects, this logic also allows specification of DOG properties, such as "Is it possible for the driver to be stuck in the car given that the door is not locked?", or "What is the probability that the doors are not unlocked, but the thief has the key?" The thesis aims to develop algorithms to implement the DOGLog logic, and to create a tool for evaluating DOGLog properties on DOGs. This thesis is guided by the following research questions:

RQ1. How can model checking algorithms be designed and implemented for the three layers of DOGLog?

RQ2. How can DOGLog be used for performing risk analysis on a complex system, and what conclusions can be drawn from the application of DOGLog to such a system?

RQ3. What are the current limitations of the WATCHDOG framework and its associated logic DOGLog, and what are the most promising opportunities for future enhancements to improve its expressiveness and usability?

The report is structured as follows. First, Chapter 2 presents the necessary background information and formal definitions of fault trees, attack trees, binary decision diagrams, and the WATCHDOG framework. Next, we review related work in Chapter 3. In Chapter 4,

we present our proposed methodology for implementing the DOGLog logic. We then demonstrate the practical application of our work in a case study in Chapter 5. Chapter 6 discusses the findings, limitations, and potential future work. Finally, we conclude the report in Chapter 7.

Chapter 2

Preliminaries

2.1 Fault Trees

A fault tree is a directed acyclic graph (DAG) that models how component failures propagate through a system to cause a system-level failure. The structure consists of events connected by logical gates, which determine how failures combine and propagate.

Events in a fault tree are categorized into two types: basic events (BEs) representing atomic component failures, and intermediate events (IEs) representing composite failures that result from combinations of other events. These combinations are determined by logical gates, namely AND gates requiring all inputs to fail, and OR gates requiring at least one input to fail. The system-level failure that fault tree users are interested in preventing is at the root of the tree, and is often called the top event or top-level event (TLE).

The propagation of failures through the tree is determined by the gate types. For an AND gate, the output event occurs only when all input events occur simultaneously. For an OR gate, the output event occurs when any input event occurs.

Formally, we define a fault tree as follows:

Definition 2.1. A fault tree is a 3-tuple $(\mathtt{E}, t, ch)$ with $\mathtt{E} = \mathtt{IE} \cup \mathtt{BE}$ and $\mathtt{IE} \cap \mathtt{BE} = \emptyset$ where:

- $\mathtt{E}$ is the set of events
- $\mathtt{IE}$ is the set of intermediate events
- $\mathtt{BE}$ is the set of basic events
- $t : \mathtt{IE} \mapsto \{\mathtt{AND}, \mathtt{OR}\}$ is a function mapping intermediate events to the type of their gate
- $ch : \mathtt{IE} \mapsto \mathcal{P}(\mathtt{E}) \setminus \emptyset$ is a function mapping intermediate events to their children

The semantics of a fault tree F describes whether an event e fails, given that a set of basic events S fail [39].

Definition 2.2. The semantics of a FT F is a function $\pi_F : \mathcal{P}(\mathtt{BE}) \times \mathtt{E} \mapsto \{0, 1\}$ where $\pi_F(S, e)$ indicates whether e fails given the set S of failed BEs. It is defined as follows:

$$\pi_F(S, e) = \begin{cases} e \in S & \text{if } e \in \mathtt{BE} \\ \bigwedge\limits_{x \in ch(e)} \pi_F(S, x) & \text{if } e \in \mathtt{IE} \text{ and } t(e) = \mathtt{AND} \\ \bigvee\limits_{x \in ch(e)} \pi_F(S, x) & \text{if } e \in \mathtt{IE} \text{ and } t(e) = \mathtt{OR} \end{cases}$$

A set of basic events that cause the top event to fail is called a cut set. If the set is also minimal, i.e., removing any basic event from the set means the top event no longer fails, it is called a minimal cut set (MCS). Minimal cut sets are of interest to fault tree analysts, as they provide insight into which components are critical to the system's reliability. The dual of a cut set is a path set, which is a set of basic events such that if none of the events fail, it is impossible for the top event to fail. Similarly, a minimal path set (MPS) is a path set that is minimal.

2.2 Attack Trees

Attack trees are very similar to fault trees, but instead of modeling system failures, they model how an attacker can compromise a system. The structure of an attack tree is similar to that of a fault tree, with events representing possible attacks and logical gates determining how attacks combine and propagate. Instead of events, attack tree nodes are called attack steps (ASs), leafs are called basic attack steps (BASs), and the remaining nodes are intermediate attack steps (IASs). The gates in an attack tree are the same as in a fault tree, with AND gates requiring all inputs to succeed and OR gates requiring at least one input to succeed.

Definition 2.3. An attack tree is a 3-tuple $(\texttt{AS}, t, ch)$ with $\texttt{AS} = \texttt{IAS} \cup \texttt{BAS}$ and $\texttt{IAS} \cap \texttt{BAS} = \emptyset$ where:

- $\texttt{AS}$ is the set of attack steps
- $\texttt{IAS}$ is the set of intermediate attack steps
- $\texttt{BAS}$ is the set of basic attack steps
- $t : \texttt{IAS} \mapsto \{\texttt{AND}, \texttt{OR}\}$ is a function mapping intermediate attack steps to the type of their gate
- $ch : \texttt{IAS} \mapsto \mathcal{P}(\texttt{AS}) \backslash \emptyset$ is a function mapping intermediate attack steps to their children

The semantics of an attack tree A describes whether an attack step a has succeeded, given that a set of basic attack steps S have succeeded.

Definition 2.4. The semantics of an AT A is a function $\pi_A : \mathcal{P}(\texttt{BAS}) \times \texttt{AS} \mapsto \{0, 1\}$ where $\pi_A(S, a)$ indicates whether a has succeeded given the set S of succeeded BASs. It is defined as follows:

$$\pi_A(S,a) = \begin{cases} a \in S & \text{if } a \in \texttt{BAS} \\ \bigwedge\limits_{x \in ch(a)} \pi_A(S,x) & \text{if } a \in \texttt{IAS} \text{ and } t(a) = \texttt{AND} \\ \bigvee\limits_{x \in ch(a)} \pi_A(S,x) & \text{if } a \in \texttt{IAS} \text{ and } t(a) = \texttt{OR} \end{cases}$$

Similar to MCSs for fault trees, a minimal attack scenario (MAS) is a minimal set of basic attack steps that, when succeeded, guarantees the success of system compromise.

2.3 Binary Decision Diagrams

A Binary Decision Diagram (BDD), introduced by Lee [23] and further developed by Akers [2] and Bryant [7], is a directed acyclic graph that represents a Boolean function.

The structure consists of decision nodes representing variables, terminal nodes representing Boolean values, and edges representing variable assignments.

Each non-terminal node in a BDD represents a Boolean variable and has exactly two outgoing edges: a high edge (typically denoted by a solid line) representing the case where the variable is true, and a low edge (typically denoted by a dashed line) representing the case where the variable is false. The terminal nodes are labeled with $\top$ or $\bot$, representing the Boolean values *true* and *false* respectively.

The evaluation of a Boolean function represented by a BDD follows a path from the root node to a terminal node, with the path determined by the assignment of values to variables. At each decision node, the evaluation proceeds along the high edge if the corresponding variable is true, and along the low edge if it is false.

Definition 2.5. A Binary Decision Diagram is a 4-tuple $\text{BDD} = (V, \mathit{Low}, \mathit{High}, \mathit{Lab})$ where:

- V is the set of nodes
- $\mathit{Low} : V \mapsto V$ maps nodes to their low child
- $\mathit{High} : V \mapsto V$ maps nodes to their high child
- $\mathit{Lab} : V \mapsto \mathit{Vars} \cup \{\top, \bot\}$ maps non-terminal nodes to the variable they represent and terminal nodes to their value

Additionally,

- $\mathit{Vars}(\text{BDD})$ is the set of variables in the BDD,
- every $v \in V$ has an index $\mathit{Idx}(v) \in \mathbb{N}$ that determines the variable ordering, such that for all non-terminal nodes v, if $u = \mathit{Low}(v)$ or $u = \mathit{High}(v)$ then $\mathit{Idx}(v) < \mathit{Idx}(u)$
- $v_r \in V$ is the root node, also denoted $\mathit{Root}(\text{BDD})$, and $\mathit{Idx}(v_r) = 0$,

A Reduced Ordered Binary Decision Diagram (ROBDD), created by Bryant, is a type of BDD that enforces additional constraints to ensure efficiency and canonicity [7]. These constraints are:

1. No two distinct nodes represent the same function (reduced)
2. Variables appear in a fixed order along all paths from root to terminal nodes (ordered)
3. No variable node has identical high and low children (no redundant variables)

The canonicity of ROBDDs provides two important properties:

1. Equivalence checking of Boolean functions reduces to checking isomorphism of their ROBDDs
2. Tautologies and contradictions have unique representations: a single $\top$ node and a single $\bot$ node, respectively

From now on, we will refer to Reduced Ordered Binary Decision Diagrams as BDDs, unless explicitly stated otherwise.

2.3.1 Complement Edges

An often-used optimization for the BDD data structure is the use of *complement edges* [2]. This technique reduces the size of the BDD by allowing the representation of a Boolean function and its negation with the same subgraph.

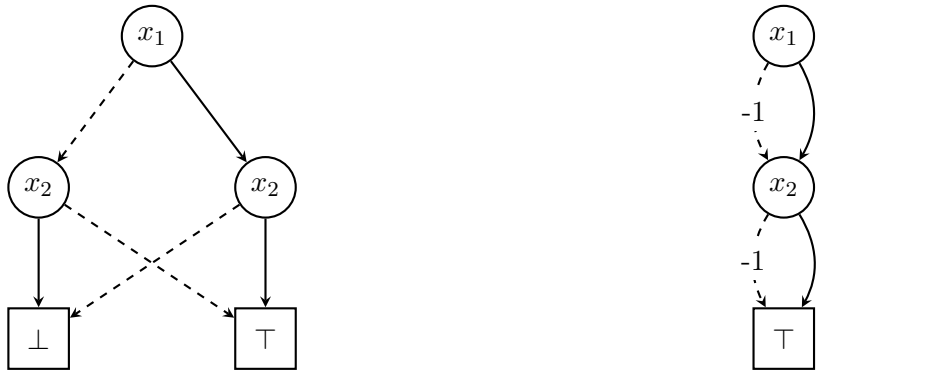
A complement edge indicates that the function represented by the node it points to should be negated. This is typically implemented by adding a "complement" attribute to each edge. When traversing a path from the root to a terminal node, the resulting value is inverted each time a complement edge is followed.

The use of complement edges has two main advantages. First, it can significantly reduce the number of nodes in the BDD. If a function and its negation are both needed within a larger BDD, only one of them needs to be explicitly represented. The other is represented by a complement edge pointing to the existing subgraph.

Second, it simplifies the negation operation. Negating a function represented by a BDD with complement edges can be done in constant time by simply complementing the edge pointing to the root of the BDD.

With complement edges, only a single terminal node, typically the $\top$ node, is required. The $\bot$ value is represented by a complement edge pointing to the $\top$ node. This further reduces the size of the BDD representation.

An example of this can be seen in Figure 2.1, which shows the BDD for the function $f(x_1, x_2) = x_1 \longleftrightarrow x_2$ with and without complement edges. In the standard BDD, two separate branches are required to represent the positive and negative cases. With complement edges, the BDD can reuse the same subgraph for both cases, with the low edge of x_1 complementing the result of the x_2 node. This reduces the number of nodes and simplifies the overall structure.



(A) Without complement edges.

(B) With complement edges.

FIGURE 2.1: Comparison of BDDs for $f(x_1, x_2) = x_1 \longleftrightarrow x_2$ with and without complement edges. The "-1" denotes a complement edge.

2.4 Multi-Terminal Binary Decision Diagrams

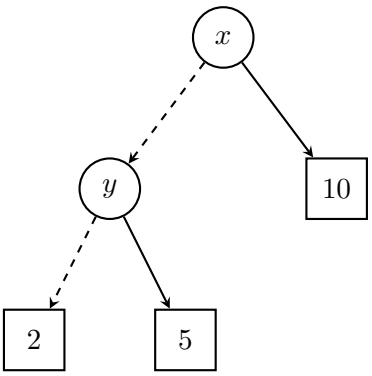
A Multi-Terminal Binary Decision Diagram (MTBDD) is a generalization of a BDD that allows for more than two terminal nodes [10, 14]. While a standard BDD represents a function mapping Boolean variables to a Boolean value ($\{\top, \bot\}$), an MTBDD can represent a function that maps Boolean variables to a value from an arbitrary set. The terminal nodes in an MTBDD are not restricted to Boolean values but can instead hold values from a specified domain, such as integers or real numbers.

In the context of this thesis, we are particularly interested in MTBDDs where the terminal nodes represent numeric values. This allows for the representation of functions of the form $f : \{\top, \bot\}^n \to \mathbb{R}$. Each path from the root to a terminal node corresponds to a specific assignment of the Boolean variables, and the value of the terminal node gives the

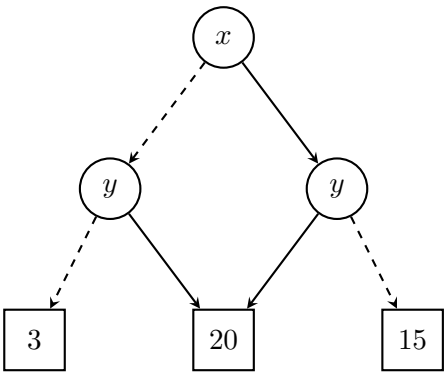
function's output for that assignment.

The key point of using numeric terminal values is the ability to perform arithmetic operations directly on the diagrams. For example, two MTBDDs can be combined using operations like addition, subtraction, or multiplication. These operations are applied element-wise on the functions they represent, which can be implemented efficiently using a recursive algorithm similar to the Apply algorithm used for BDDs [7]. This makes MTBDDs a powerful tool for quantitative analysis in, e.g., risk assessment, where numeric values associated with different system states need to be computed and combined.

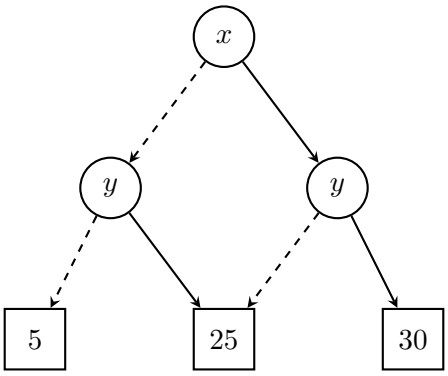
An example of MTBDD addition is shown in Figure 2.2. The figure shows two MTBDDs representing functions $f(x, y)$ and $g(x, y)$, and the resulting MTBDD for their sum $h(x, y) = f(x, y) + g(x, y)$. The terminal nodes of the resulting MTBDD are the sums of the corresponding terminal nodes of the input MTBDDs. For instance, for the variable assignment $(x = \top, y = \top)$, the path in the MTBDD for f leads to the terminal node with value 10, and the path in the MTBDD for g leads to the terminal node with value 20. The resulting path in the MTBDD for h leads to a terminal node with value $10 + 20 = 30$. Crucially, the assignments $(x = \top, y = \bot)$ and $(x = \bot, y = \top)$ both result in the value 25. This demonstrates the efficiency of MTBDDs, where both paths point to the same terminal node, avoiding redundancy and ensuring a compact representation.



(A) MTBDD for a function $f(x, y)$.



(B) MTBDD for a function $g(x, y)$.



(C) MTBDD for the resulting function $h(x, y) = f(x, y) + g(x, y)$.

Figure 2.2: An example of the addition of two MTBDDs.

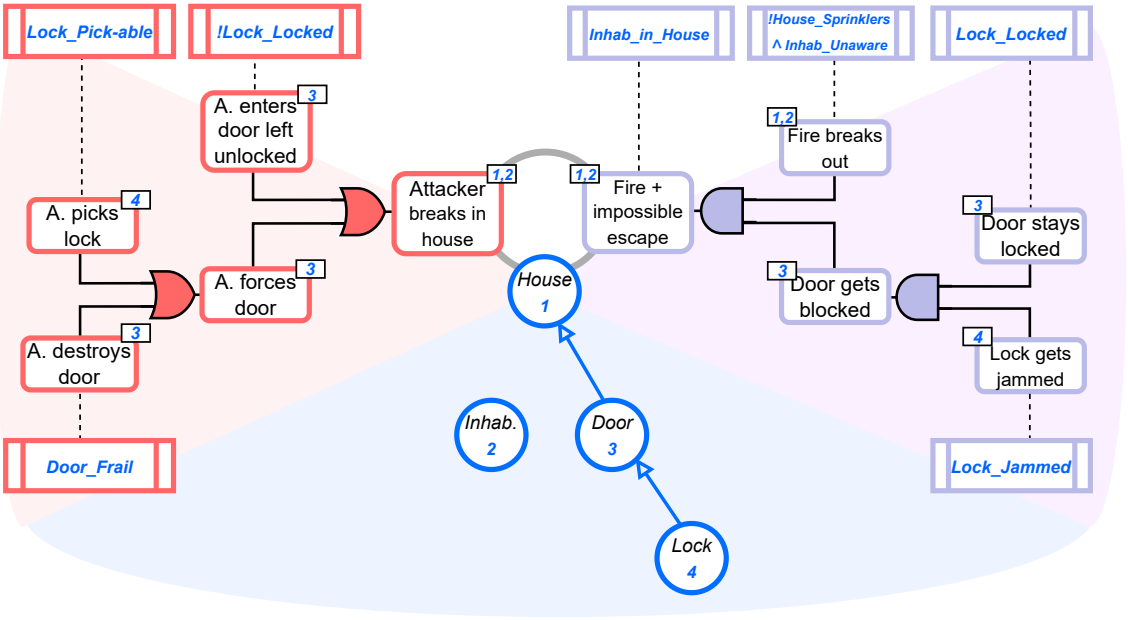


FIGURE 2.3: Example DOG, taken from an earlier version of [31], see [28].

2.5 WATCHDOG

WATCHDOG is a framework for ontology-aware risk assessment that combines the Common Ontology of Value and Risk (COVER) [45] with commonly used formal methods such as fault and attack trees to enable object-oriented risk assessment [31]. At its core, WATCHDOG introduces Object-oriented Disruption Graphs (DOGs)—a formalism that combines attack trees and fault trees with object graphs (OGs), to allow explicit modeling of Objects at Risk (OaRs) and their properties.

Figure 2.3 shows an example DOG modeling risks to a house and its inhabitants. The attack tree (red, left side) models how an attacker might break into the house by, e.g., destroying a frail door or entering through a door left unlocked. The fault tree (purple, right side) models how a fire could lead to inhabitants being trapped. The object graph (blue, bottom) makes explicit that the system involves an inhabitant and a house, and that the lock is part of the door, which in turn is part of the house. Object properties like "Lock_Locked" and "Door_Frail" can affect how attacks and failures propagate through the system. In this example, the lock being locked helps prevent the house from being broken into, while it can risk trapping inhabitants in case of a fire, which makes locking the lock a double-edged sword. This example demonstrates how DOGs combine attack/fault trees with explicit object modeling, and how object properties can affect risk propagation.

The key innovation of WATCHDOG is that it makes objects and their roles in the system under analysis explicit. For example, when analyzing risks related to a door being compromised, WATCHDOG allows explicitly modeling that both the door itself and its lock are objects at risk, with properties like "door is frail" or "lock is pickable" that affect how attacks and failures can propagate through the system.

To reason about DOGs, WATCHDOG provides a three-layered logic called DOGLog:

- Layer 1 allows reasoning about how disruptions propagate through the system, considering both the attack/fault tree structure and object properties
- Layer 2 enables probabilistic reasoning about the likelihood of disruptions occurring

- Layer 3 supports analyzing risk levels, allowing questions about which objects face the highest risks

2.5.1 Formal Definitions

In this section, we summarize the formal definitions of the key concepts in WATCHDOG [31]. We begin by giving the formal definition of Object-oriented Disruption Graphs, which consist of four main components:

Definition 2.6. An Object-oriented Disruption Graph (DOG) G is a tuple (A, F, O, B) where A is an attack tree, F is a fault tree, O is an object graph and B is a disruption knowledge base.

Both attack trees and fault trees in a DOG can be viewed as instances of a more general structure called a disruption tree:

Definition 2.7. A Disruption Tree (DT) T is a tuple (N, E, t) where:

- (N, E) is a rooted directed acyclic graph
- $t : N \mapsto \{\texttt{OR}, \texttt{AND}, \texttt{LEAF}\}$ is a function such that for $v \in N$, $t(v) = \texttt{LEAF}$ if and only if v is a leaf
- $ch : N \mapsto 2^N$ gives the set of children of a node
- R_T is the unique root of the tree T

A scenario vector $\vec{b} = (b_1, \ldots, b_n)$ represents the state of leaves in a disruption tree, where $b_i = 1$ if the i-th leaf is disrupted and $b_i = 0$ otherwise. The universe of scenarios $\mathcal{S}_T = 2^{\text{LEAF}_T}$ represents all possible combinations of leaf states in tree T, where each leaf can be either disrupted (1) or operational (0).

The behavior of a disruption tree is specified by its structure function f_T (see Definition 2.8), which, based on the currently disrupted leaves, indicates for each node whether it is disrupted [39].

In general, when a distinction between attack trees and fault trees is needed, we use subscripts $\__A$ and $\__F$, respectively. For example, $\mathcal{S}_A$ is the universe of scenarios for an attack tree, and $\vec{b}_F$ is a fault scenario. Additionally, N_F and N_A are the sets of nodes in fault and attack trees, respectively.

Definition 2.8. The structure function of a disruption tree T is a function $f_T : \mathbb{B}^n \times N \mapsto \mathbb{B}$ that indicates whether a node $v \in N$ is disrupted in a given scenario $\vec{b}$.

$$f_T(\vec{b}, v) = \begin{cases} b_i & \text{if } v = v_i \in \texttt{LEAF} \\ \bigvee_{v' \in ch(v)} f_T(\vec{b}, v') & \text{if } v \in \texttt{IE} \text{ and } t(v) = \texttt{OR} \\ \bigwedge_{v' \in ch(v)} f_T(\vec{b}, v') & \text{if } v \in \texttt{IE} \text{ and } t(v) = \texttt{AND} \end{cases}$$

WATCHDOG introduces Object Graphs (OGs) to explicitly specify objects that participate in safety- and security-related events/actions in the system, as well as to model relationships between objects.

Definition 2.9. An Object Graph (OG) O is a tuple (N_O, E_O, OP, cOP) where:

- N_O is the set of nodes representing Objects at Risk (OaRs)
- $E_O \subseteq N_O \times N_O$ is the set of directed edges representing the *parthood* relation between OaRs

- OP is the set of atomic propositions representing object properties of OaRs
- $cOP\colon N_O \mapsto 2^{OP}$ is a function that returns the set of properties of a node $v \in N_O$

Additionally, $ch\colon N_O \mapsto 2^{N_O}$ gives the set of *parts* of a node and O has a unique root, denoted R_O.

Similar to scenario vectors for disruption trees, a configuration vector $\vec{b_O} = (o_1, \ldots, o_m)$ represents the state of properties in an object graph:

Definition 2.10. A configuration $\vec{b_O}$ is a Boolean vector assigning values to properties $op \in OP$. $f_O\colon \mathbb{B}^n \times OP \mapsto \mathbb{B}$ is a valuation function where $f_O(\vec{b_O}, op) = 1$ if and only if property $op \in OP$ equals 1 given configuration $\vec{b_O}$. The set of all possible configurations is denoted by $\mathcal{C}$.

The final component of a DOG is its Disruption Knowledge Base (DKB), which establishes formal relationships between events/actions and the objects that participate in them.

Definition 2.11. A Disruption Knowledge Base (DKB) is a tuple (D, Im, Pa, Pr) where:

- $D = N_A \cup N_F \cup N_O$ is the entity domain, where N_A, N_F and N_O are pairwise disjoint
- $Im : N_A \cup N_F \mapsto \mathbb{R}_{\geq 0}$ assigns an impact factor to events/actions
- $Pa : N_A \cup N_F \mapsto 2^{N_O}$ returns the set of participating objects for an event/action
- $Pr : N_A \cup N_F \mapsto 2^{OP}$ returns the preconditions, where $Pr(v) = \bigcup_{o \in Pa(v)} cOP(o)$

For each event or action v, there is also an associated condition $Cond(v)$, which is a Boolean formula over the preconditions $Pr(v)$. The atomic propositions in $Cond(v)$ are $op \in Pr(v)$, and they can be connected with the usual logical operators $\wedge$ and $\neg$ (other operators can be derived from these).

Let $Form$ be the set of all Boolean formulae over preconditions, then we can evaluate conditions using a function f_{Cond}:

Definition 2.12. The condition evaluation function $f_{Cond}\colon \mathbb{B}^n \times Form \mapsto \mathbb{B}$ takes as input a configuration vector $\vec{b_O}$ and conditions $Cond(v)$. We have $f_{Cond}(\vec{b_O}, Cond(v)) = 1$ if and only if $\vec{b_O}$ satisfies $Cond(v)$.

To account for object properties in disruption propagation, WATCHDOG extends the basic structure function with conditions from the DKB. For an element v in N_A or N_F to be disrupted, two conditions must be met:

- The element must be disrupted according to the basic structure function
- The conditions $Cond(v)$ must evaluate to true

This is captured by the extended structure function f_T°:

Definition 2.13. The extended structure function of a disruption tree T is a function $f_T^\circ : \mathbb{B}^n \times \mathbb{B}^n \times N \mapsto \mathbb{B}$ that takes as input a scenario $\vec{b}$, a configuration $\vec{b_O}$, and a node $v \in N$. It is defined as:

$$f_T^\circ(\vec{b}, \vec{b_O}, v) = \begin{cases} b_i \wedge f_{Cond}(\vec{b_O}, Cond(v)) & \text{if } v = v_i \in \texttt{LEAF} \\ \bigvee_{v' \in ch(v)} f_T^\circ(\vec{b}, \vec{b_O}, v') \wedge f_{Cond}(\vec{b_O}, Cond(v)) & \text{if } v \in \texttt{IE} \text{ and } t(v) = \texttt{OR} \\ \bigwedge_{v' \in ch(v)} f_T^\circ(\vec{b}, \vec{b_O}, v') \wedge f_{Cond}(\vec{b_O}, Cond(v)) & \text{if } v \in \texttt{IE} \text{ and } t(v) = \texttt{AND} \end{cases}$$

2.5.2 DOGLog Syntax

DOGLog is structured in three syntactic layers, where each layer has its distinct purpose:

- Layer 1 (ϕ): Reasons about disruption propagation through the system
- Layer 2 (ψ): Reasons about disruption probabilities
- Layer 3 (ξ): Reasons about safety and security risk levels

The syntax is defined as follows:

$$\begin{aligned}
&\text{Layer 1: } \phi ::= a \mid \neg\phi \mid \phi \wedge \phi \mid \phi[a \mapsto \mathit{bool}] \mid \mathrm{MRS}(\phi) \\
&\text{Layer 2: } \psi ::= \mathrm{P}(\phi) \bowtie p \mid \neg\psi \mid \psi \wedge \psi \mid \psi[e \mapsto q] \\
&\text{Layer 3: } \xi ::= \mathsf{MostRisky}_{*}(o) \mid \underset{\max}{\mathsf{TotalRisk}}(o) \mid \underset{\min}{\mathsf{TotalRisk}}(o) \\
&\qquad\qquad \mid \mathsf{OptimalConf}(o) \mid \xi[\mathit{op} \mapsto \mathit{bool}]
\end{aligned}$$

In layer 1, atomic propositions a can represent any element in an AT or FT, or any object property, i.e., $a \in N_A \cup N_F \cup OP$. The syntax $[a \mapsto \mathit{bool}]$ represents setting evidence: assigning Boolean values to events, actions or object properties to construct what-if scenarios. With $\mathrm{MRS}(\phi)$ we identify minimal risk scenarios (MRSs): minimal assignments on leaves of the FT and AT, such that formula ϕ is satisfied.

Layer 2 formulae contain $\mathrm{P}(\phi)$ to denote the probability that ϕ holds, with $\bowtie \in \{<, \leq, =, \geq, >\}$ representing comparison operators for probability thresholds. Setting probabilistic evidence is done via $[e \mapsto q]$, where $e \in N_A \cup N_F$ is an event/action and $q \in [0, 1]$ is a probability value.

The main operators in layer 3 analyze risks for a specific object o. $\mathsf{MostRisky}_{*}(o)$ (with $* \in \{\boldsymbol{A}, \boldsymbol{F}\}$) identifies the most risky action ($\mathsf{MostRisky}_{\boldsymbol{A}}$) or event ($\mathsf{MostRisky}_{\boldsymbol{F}}$) for an object o. $\underset{\min}{\mathsf{TotalRisk}}(o)$ and $\underset{\max}{\mathsf{TotalRisk}}(o)$ compute the minimum and maximum total risk o is exposed to, by considering all possible configurations of object properties. $\mathsf{OptimalConf}(o)$ determines the configuration of object properties that minimizes the total risk for o. Finally, object properties $\mathit{op} \in OP$ can be assigned Boolean values via $[\mathit{op} \mapsto \mathit{bool}]$ to create configuration-specific scenarios.

Chapter 3

Related Work

3.1 Specification and Verification of Fault Tree and Attack Tree Properties

Researchers have developed formal methods support for specifying and verifying system properties of dynamic fault trees and attack trees, or combinations thereof. Methods exist for formally specifying properties using temporal logics, as well as techniques for verifying these properties through model checking [3, 36, 48].

The formal methods involved in property specification and verification for dynamic fault trees and attack trees typically involve two key aspects: (1) specification languages for expressing properties, and (2) analysis techniques for verifying these properties. For specification, temporal logics like Computation Tree Logic (CTL) can be used to express properties about the ordering and timing of events. For example, CTL can specify that a condition must hold for all paths and all states of a system model.

The analysis of fault trees and attack trees often involves transformation to formal models that enable automated verification [3, 36, 4]. Common target formalisms include timed automata, Petri nets, and Markov models.

These formal models can then be analyzed using model checking tools to verify properties expressed in temporal logic, such as:

- Safety properties: ensuring bad states are never reached (which is the same as stating that some good property is always satisfied)
- Liveness properties: ensuring desired states are eventually reached

However, none of these methods allow reasoning about the structure of the fault tree or attack tree itself. Instead, they rely on the fact that the tree can be transformed into some other formalism, for which specification and verification tools already exist.

3.2 Custom Logics for Fault Trees and Attack Trees

Recent work has introduced several logics for analyzing fault trees and attack trees. These logics provide formal languages for expressing and verifying properties about such trees, each focusing on different aspects.

3.2.1 Boolean Fault Tree Logic (BFL)

BFL [26] introduces a two-layered logic for formally specifying and verifying properties of fault trees.

The first layer provides Boolean operators and allows setting evidence and computing minimal cut/path sets. The second layer adds quantifiers and independence checking between formulae.

With BFL, one can:

- Investigate what-if scenarios by setting evidence
- Check independence between formulae
- Compute minimal cut and path sets
- Find upper/lower bounds for failing elements

The logic supports checking whether specific status vectors satisfy a property, computing all satisfying status vectors, and generating counterexamples when properties do not hold.

Two implementation approaches have been developed: (1) a BDD-based algorithm [26], which cleverly exploits the properties of BDDs to efficiently compute minimal cut/path sets and independence checks, and (2) a QSAT-based approach that translates BFL formulae to Quantified Boolean Formulae (QBF) for solving [43] using a QBF solver. These approaches complement each other, as different types of queries may be better suited for one implementation over the other [43]. At the time of writing, only the QSAT-based approach has been implemented, while the BDD-based approach is still under development.

3.2.2 Probabilistic Fault Tree Logic (PFL)

PFL [30] extends BFL to reason about probabilities in fault trees. In addition to Boolean reasoning, PFL enables:

- Checking probability bounds on events
- Setting probabilistic evidence
- Computing actual probability values

For example, with PFL one can check that "the probability of the top event occurring is less than 0.1" or compute "the probability that gate G1 fails given that gate G2 has failed". To achieve this, PFL combines layer 1 of BFL with two new layers:

- Layer 1: Boolean reasoning (inherited from BFL)
- Layer 2: Probability bounds and evidence, and independence checking
- Layer 3: Probability value computation and evidence

PFL uses BDDs to compute probabilities, in a way that seems similar to the algorithm proposed in [37]. An implementation of PFL does not yet exist.

3.2.3 Attack Tree Metrics Logic (ATM)

ATM [27] adapts the concepts from BFL and PFL to attack trees, allowing it to evaluate a variety of security metrics, including but not limited to attack cost, attack probability, and required attacker skill level. For example, one can use ATM to check that "the cost of a DDoS attack is less than 1000 given that a machine is compromised" or compute "the probability that an attacker can compromise a machine given that they have access to the network".

ATM consists of four layers:

- Layer 1: Boolean reasoning and minimal attacks (inherited from BFL)
- Layer 2: Metric bounds and setting attribution evidence (similar to layer 2 of PFL)

- Layer 3: Metric computation and setting attribution evidence (similar to layer 3 of PFL)
- Layer 4: Quantification over layer 1 and 2 formulae (similar to layer 2 of BFL)

ATM uses BDDs to compute security metrics. However, since an attacker makes conscious decisions about which actions to take, whereas faults are random events, the computation of probabilities in ATM uses a different approach than in PFL. The approach is based on the one proposed in [24]. No implementation of ATM exists at the time of writing.

3.2.4 Relation to WATCHDOG and DOGLog

The logics BFL, PFL and ATM serve as important precursors to DOGLog [31], each contributing valuable concepts:

- From BFL: the concept of layered logic specification and the ability to reason about Boolean properties, minimal cut/path sets, and setting evidence to investigate what-if scenarios
- From PFL: quantitative analysis capabilities, particularly for probability computation
- From ATM: the ability to compute security metrics

WATCHDOG extends beyond these logics by:

- Integrating both fault trees and attack trees in a unified framework
- Adding explicit modeling of objects and their properties
- Computing risk values for objects that participate in events and actions

This makes WATCHDOG more suitable for risk assessment where both safety and security aspects need to be considered. Further, WATCHDOG's object-oriented approach allows for reasoning about the impact of object properties on the propagation of risk in the system. This can be crucial, as noted in [19]: "when considered jointly, safety and security requirements or measures [can] lead to conflicting situations." For example, an exit door should be easily accessible in case of a fire (safety requirement), but should be secure against unauthorized access (security requirement) [13, 47, 12].

Note that while there is overlap in the capabilities of DOGLog, BFL, PFL, and ATM, none of these logics fully subsume another. Each logic introduces unique features or focuses on different aspects of risk and security analysis. As a result, no single logic contains all the expressiveness or analysis capabilities of another.

Chapter 4

Methodology

This chapter addresses the first research question: *How can model checking algorithms be designed and implemented for the three layers of DOGLog?* To answer this, the chapter details the design of the model checking algorithms for these layers and the subsequent implementation of these algorithms.

4.1 Model Checking

4.1.1 Layer 1 Formulae

Model checking formulae in layer 1 involves translating the formulae into BDDs. Layer 1 formulae reason about disruption propagation in the DOG using atomic propositions from the set $N_A \cup N_F \cup OP$. The translation from formulae to BDDs follows Algorithm 1, which takes a DOG G and a formula ϕ as input and produces a BDD B_G^ϕ representing the formula ϕ over G. The algorithm uses translation functions $\mathfrak{f}_A$ and $\mathfrak{f}_F$ to translate FT and AT nodes of the DOG into BDDs, while also taking their object properties into account.

Below, operations on BDDs are in bold (e.g., $\mathbf{\wedge}$ for conjunction, $\mathbf{\neg}$ for negation, and $\mathbf{\neg\exists}$ for negated existential quantification).

Definition 4.1. Let $B_a(x)$ be a BDD with a single node v with $Lab(v) = x$, $Low(v) = \bot$ and $High(v) = \top$.

Definition 4.2. The translation function of a disruption tree T is a function $\mathfrak{f}_T \colon N_T \to \texttt{BDD}$ that takes as input an element $v \in N_T$.[1]

$$\mathfrak{f}_T(v) = \begin{cases} B_a(v) \mathbf{\wedge} B_C(Cond(v)) & \text{if } v \in \texttt{LEAF} \\ \mathbf{\bigvee}_{v' \in ch(v)} \mathfrak{f}_T(v') \mathbf{\wedge} B_C(Cond(v')) & \text{if } v \in \texttt{IE} \text{ and } t(v) = \texttt{OR} \\ \mathbf{\bigwedge}_{v' \in ch(v)} \mathfrak{f}_T(v') \mathbf{\wedge} B_C(Cond(v')) & \text{if } v \in \texttt{IE} \text{ and } t(v) = \texttt{AND} \end{cases}$$

where $B_C(Cond(v))$ is the translation of the condition $Cond(v)$ into a BDD, as defined below.

Definition 4.3. The translation of a condition $Cond(v)$ into a BDD is given by the function $B_C \colon Form \to \texttt{BDD}$ that takes as input a Boolean formula over preconditions $Pr(v)$:

$$B_C(\varphi) = \begin{cases} B_a(op) & \text{if } \varphi = op \in OP \\ \mathbf{\neg}\, B_C(\varphi') & \text{if } \varphi = \neg\varphi' \\ B_C(\varphi') \mathbf{\wedge} B_C(\varphi'') & \text{if } \varphi = \varphi' \wedge \varphi'' \end{cases}$$

[1] This definition is adapted from an unpublished draft of [31].

We also allow priming variables: if we have $V = \{x_i\}_{i=1}^n$, then $V' = \{x_i'\}_{i=1}^n$. Then we let $B_G^\phi[V \curvearrowright V']$ denote the BDD obtained by replacing each variable x_i in B_G^ϕ with x_i'. Furthermore, we have $V' \subset V \equiv (\bigwedge_i x_i' \Rightarrow x_i) \wedge (\bigvee_i x_i' \neq x_i)$, meaning that the number of variables set to *true* in V' is strictly less than the number of variables set to *true* in V. We use RESTRICT as defined in [7].

Algorithm 1 Compute B_G^ϕ from DOG G and ϕ (adapted from an unpublished draft of [31])

Input: DOG G, formula ϕ
Output: BDD B_G^ϕ

function L1BDD(G, ϕ)
  if $\phi = a$ **then**
    if $a \in N_A$ **then return** $\mathfrak{f}_A(a)$
    else if $a \in N_F$ **then return** $\mathfrak{f}_F(a)$
    else if $a \in OP$ **then return** $B_a(a)$
  else if $\phi = \neg\phi'$ **then return** $\neg$(L1BDD(G, ϕ'))
  else if $\phi = \phi' \wedge \phi''$ **then return** L1BDD(G, ϕ')$\wedge$L1BDD(G, ϕ''))
  else if $\phi = \phi'[a_i \mapsto bool]$ **then return** RESTRICT(L1BDD(G, ϕ'), $x_i, bool \in \mathbb{B}$)
  else // ϕ = MRS(ϕ')
    $V \leftarrow Vars$(L1BDD(G, ϕ')) $\setminus$ OP // *minimality does not apply to OPs*
    return L1BDD(G, ϕ')$\wedge(\neg\exists V'.(V' \subset V) \wedge$L1BDD($G, \phi'$)$[V \curvearrowright V'])$

We define some predicates to distinguish between different types of nodes in a BDD that is output by Algorithm 1.

Definition 4.4. Let G be a DOG and $B = (V, Low, High, Lab)$ be a BDD. For a node $v \in V$, we define the following predicates:

$$\mathcal{T}_{\text{OP}}(G, v) \iff Lab(v) \in OP$$
$$\mathcal{T}_{\text{AT}}(G, v) \iff Lab(v) \in N_A$$
$$\mathcal{T}_{\text{FT}}(G, v) \iff Lab(v) \in N_F$$

where OP, N_A, and N_F are the sets of object properties, attack tree nodes, and fault tree nodes in G respectively.

The BDD B_G^ϕ, computed by Algorithm 1, represents all satisfying assignments for a given layer 1 formula ϕ. This BDD is then utilized to answer two types of queries:

1. **Check**: This query determines if a specific assignment of truth values to all variables in ϕ satisfies the formula. The variables include any object properties ($op \in OP$), attack tree nodes ($a \in N_A$), and fault tree nodes ($f \in N_F$) that are part of ϕ. The process involves:
   - Constructing B_G^ϕ using Algorithm 1.
   - Traversing B_G^ϕ from its root node according to the provided assignment. At each decision node, the path follows the high edge if the node's variable is true in the assignment, and the low edge if it is false.
   - If this traversal leads to the $\top$ terminal node, the assignment satisfies ϕ. If it leads to the $\bot$ terminal, it does not.
2. **Compute All Minimal Scenarios**: This query identifies all minimal scenarios to the non-object-property variables (i.e., variables from $N_A \cup N_F$) that satisfy a formula ϕ.

- If ϕ depends on object properties, their values must be specified as part of the query.
- The MRS(ϕ) operator, as detailed in Algorithm 1, is then used to only return the minimal risk scenarios.
- Each path from the root to the $\top$ terminal in this resulting BDD corresponds to a minimal risk scenario for ϕ.

It is important to note that for minimal risk scenarios, while no sub-scenario satisfies ϕ, adding attacks or faults (i.e., creating a super-scenario) does not guarantee that the new scenario will still satisfy ϕ. This is because ϕ can contain negations, leading to potentially non-monotonic (also called *incoherent*) behavior.

4.1.2 Layer 2 Formulae

In addition to the usual Boolean operators, layer 2 formulae allow checking probabilities, as well as setting probabilistic evidence on atomic propositions in $N_A \cup N_F$.

Semantics. The probability semantics of DOGLog involve calculating the probability of a given layer 1 formula ϕ under a specific configuration $\vec{b}_O$. This is done by considering all possible fault scenarios $\vec{b}_F \in \mathcal{S}_F$ in the fault tree and computing the probability of each scenario [31]. For each fault scenario, we then determine the maximal probability of successfully attacking the attack tree nodes in ϕ under the given configuration. Calculating the probability in this way reflects Assumption 4 in [31], which states that the attacker knows which elements have failed before choosing their actions.

$\alpha\colon \texttt{BAS} \cup \texttt{BE} \mapsto [0,1]$ is a function that assigns a probability to each basic node. The probability $\rho(\phi, \vec{b}_O)_{A,F}$ is calculated using the equation [31]:

$$\rho(\phi, \vec{b}_O)_{A,F} = \sum_{\vec{b}_F \in \mathcal{S}_F} \mathsf{Prob}(\vec{b}_F) \times \mathbb{P}_A(\mathsf{Set}(\phi, \vec{b}_F, \vec{b}_O))$$

Here, $\mathsf{Prob}(\vec{b}_F)$ represents the probability of a fault scenario $\vec{b}_F$, which is calculated with $\mathsf{Prob}(\vec{b}_F) = \prod_{i=1}^{k} b_i \times \alpha(v_i) + (1 - b_i) \times (1 - \alpha(v_i))$, and $\mathbb{P}_A(\mathsf{Set}(\phi, \vec{b}_F, \vec{b}_O))$ represents the maximal probability of successfully attacking the AT nodes in ϕ given the fault scenario and configuration.

The $\mathbb{P}_A(\phi)$ equation [31]:

$$\mathbb{P}_A(\phi) = \max_{\vec{b}_A \in [\![\phi]\!]_A} \prod_{v \in \vec{b}_A} \alpha(v)$$

is used to compute the probability of an attack succeeding by taking the maximum product of the probabilities $\alpha(v)$ of the activated attack steps $v \in \vec{b}_A$ in the minimal attack scenarios of ϕ (denoted $\vec{b}_A \in [\![\phi]\!]_A$).

$\mathsf{Set}(\phi, \vec{b}_F, \vec{b}_O)$, is used to adjust the formula ϕ based on the fault scenario $\vec{b}_F$ and configuration $\vec{b}_O$ [31]. It replaces the atomic propositions $a \in N_F \cup OP$ in ϕ with their corresponding truth values derived from the fault scenario and configuration, allowing the computation of the probability of the adjusted formula.

The algorithm. To compute the probabilities in practice, we use Algorithm 2, which is based on two algorithms: $\texttt{BDD}_{\texttt{DAG}}$ from [8, 24], and the algorithm in Fig. 7 of [37] (which we will refer to as $\texttt{p}_{\texttt{TLE}}$ from now on).

Algorithm 2 Compute probability $\rho(\phi, \vec{b_O})_{A,F}$ from DOG G, formula ϕ, and configuration $\vec{b_O}$

Input: DOG G, formula ϕ, configuration $\vec{b_O}$
Output: probability $\rho(\phi, \vec{b_O})_{A,F}$

function L2Prob($G, \phi, \vec{b_O}$)
  $B^{\phi}_{G} = (V, Low, High, Lab) \leftarrow$ L1BDD(G, ϕ)
    // With variable order: $\forall o \in OP, f \in N_F, a \in N_A.\ Idx(o) < Idx(f) < Idx(a)$
  $r \leftarrow v_r$ *// Start at root node*
  while $Lab(r) \in OP$ **do** *// Skip over OP nodes until reaching the first FT or AT node*
    if $f_O(\vec{b_O}, Lab(r)) = 1$ **then**
      $r \leftarrow High(r)$
    else
      $r \leftarrow Low(r)$
  return CalcNodeProb(G, B^{ϕ}_{G}, r)

function CalcNodeProb(G, B^{ϕ}_{G}, r)
  Let $V' \subseteq V$ be the set of nodes in B^{ϕ}_{G} that are a descendant of r
  $P \leftarrow V' \mapsto -$
  $P[\bot] \leftarrow 0$, $P[\top] \leftarrow 1$ *// Set terminal nodes*
  for all $v \in V'$ in bottom-up order **do**
    if $Lab(v) \in N_A$ **then**
      $p_{Low} \leftarrow P[Low(v)]$
      $p_{High} \leftarrow P[High(v)] \cdot \alpha(Lab(v))$
      $P[v] \leftarrow \max(p_{Low}, p_{High})$
    else if $Lab(v) \in N_F$ **then**
      $p_{Low} \leftarrow P[Low(v)] \cdot (1 - \alpha(Lab(v)))$
      $p_{High} \leftarrow P[High(v)] \cdot \alpha(Lab(v))$
      $P[v] \leftarrow p_{Low} + p_{High}$
  return $P[r]$

In [37], the $\texttt{p}_{\texttt{TLE}}$ algorithm computes the probability of the TLE of a fault tree, based on the fault tree's BDD representation. The algorithm is not restricted to the TLE, but can be used to compute the probability of any event in the fault tree.

In [8, 24], the $\texttt{BDD}_{\texttt{DAG}}$ algorithm enables the computation of multiple metrics on ATs, including the probability of an attack step succeeding. For the *unital semiring*, we take $D_* = ([0,1], \max, \cdot, 0, 1)$, which is also *absorbing* [24].

In DOGLog, we can use FTs and ATs together, so we need to somehow combine the two algorithms. To do this, we first create a single BDD of the formula using Algorithm 1. For the variable ordering of the BDD, we make sure that variables of OP precede variables of N_F, which in turn precede variables of N_A. First, the BDD is traversed until we passed all OP nodes, and we reach the first FT or AT node. We then calculate the probability $\rho(\phi, \vec{b}_O)_{A,F}$ using a bottom-up approach, where we first calculate the probabilities of the AT nodes using $\texttt{BDD}_{\texttt{DAG}}$, and then use $\texttt{p}_{\texttt{TLE}}$ to calculate the probabilities of the FT nodes on top of the AT probabilities.

While a formal correctness proof is beyond the scope of this practically oriented thesis, we present an informal argument for the correctness of our approach.

It is clear that $\texttt{BDD}_{\texttt{DAG}}$ correctly computes the probabilities of the AT nodes in our BDD. The purpose of Set is to adjust the formula ϕ based on the fault scenario and configuration, because the attacker knows which elements have failed before choosing their actions (recall Assumption 4). In the BDD, this is taken care of by the variable ordering [7], where the variables of the FT and object properties are ordered before the variables of the AT. This ordering ensures that when evaluating the BDD, all decisions about object properties and fault states are made before any decisions about attack actions. As a result, by the time we reach any attack node in the BDD, we have complete information about which components have failed and which object properties are true. This directly implements Assumption 4 by ensuring that attack probabilities are calculated with full knowledge of the system's fault state.

The $\texttt{p}_{\texttt{TLE}}$ algorithm normally computes the probability of the TLE of a fault tree, based on the fault tree's BDD representation. In this BDD, the computation ends at the terminal $\top$ and $\bot$ nodes, which represent probability 1 and 0, respectively. In our combined approach, the last FT node on a path in the BDD is parent to either a terminal node or an AT node (or both). If it is parent to an AT node, the probability of the AT node is already calculated by the $\texttt{BDD}_{\texttt{DAG}}$ algorithm, and we multiply with this probability instead of 0 or 1, which matches the desired semantics. If instead it is parent to a terminal node, we multiply with 0 or 1 like usual, which also matches the desired semantics.

4.1.2.1 Probability Calculation Example

Let us demonstrate the algorithm with a concrete example. We will analyze the DOG shown in Figure 4.1, which is based on the DOG in Figure 2.3, but with one modification: the *Lock_ Locked* object property is removed. To keep the notation concise, we use abbreviations for the DOG nodes as defined in Table 4.1. To make the example easier to follow, we will color nodes according to their type: AT nodes are **red**, FT nodes are **purple**, and object properties are **blue**. For this example, we will work with the formula $\varphi = \boldsymbol{FD} \wedge \boldsymbol{DGB} \vee \boldsymbol{EDLU} \wedge \boldsymbol{FBO}$.

In this example, we use probability values for the basic nodes as shown in Table 4.1. Moreover, we consider the following configuration: $\vec{b}_O = (\boldsymbol{LP} = 1, \boldsymbol{LJ} = 1, \boldsymbol{DF} = 1, \boldsymbol{HS} =$

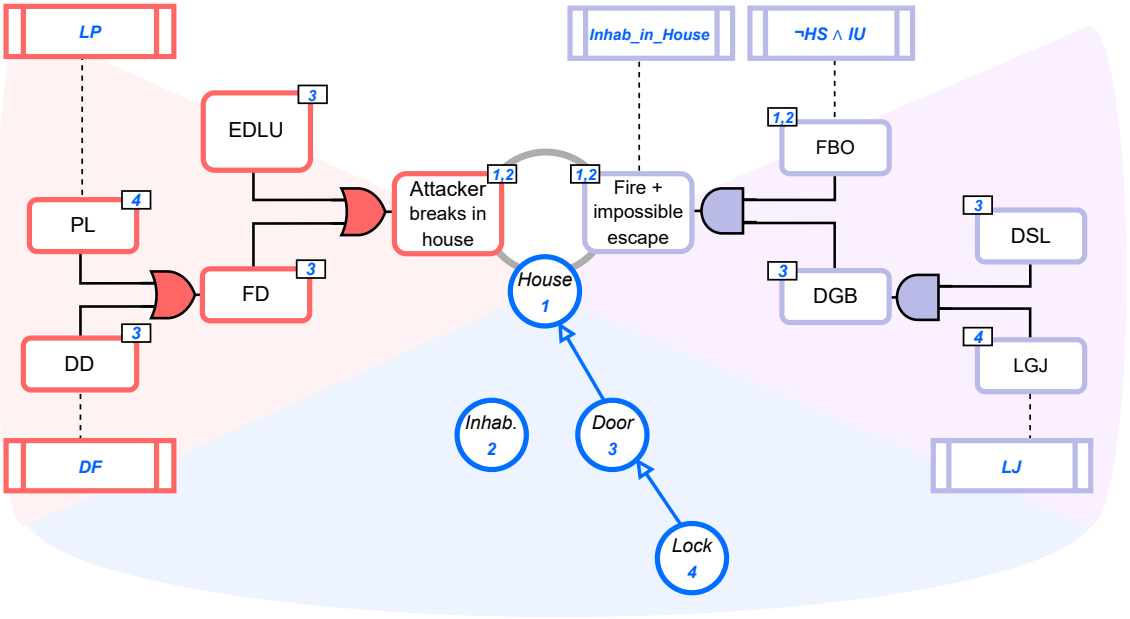


FIGURE 4.1: DOG for the example in Section 4.1.2.1, adapted from Figure 2.3, with abbreviated node names (see Table 4.1).

TABLE 4.1: Mapping between abbreviations in Figure 4.1 and full terms in Figure 2.3, as well as probabilities for basic nodes.

Abbr.	Full Term	$\alpha(v)$	Abbr.	Full Term	$\alpha(v)$
DD	A. destroys door	0.13	***HS***	*House_Sprinklers*	–
DF	*Door_Frail*	–	***IU***	*Inhab_Unaware*	–
DGB	Door gets blocked	–	***LGJ***	Lock gets jammed	0.70
DSL	Door stays locked	0.20	***LJ***	*Lock_Jammed*	–
EDLU	A. enters door left unlocked	0.17	***LP***	*Lock_Pick-able*	–
FBO	Fire breaks out	0.21	***PL***	A. picks lock	0.10
FD	A. forces door	–			

$0, \boldsymbol{IU} = 1$). We have

$$\begin{aligned} \mathfrak{f}_A(\boldsymbol{FD}) &= (\boldsymbol{PL} \wedge \boldsymbol{LP}) \vee (\boldsymbol{DD} \wedge \boldsymbol{DF}) \\ \mathfrak{f}_A(\boldsymbol{EDLU}) &= \boldsymbol{EDLU} \\ \mathfrak{f}_F(\boldsymbol{DGB}) &= (\boldsymbol{LGJ} \wedge \boldsymbol{LJ}) \wedge \boldsymbol{DSL} \\ \mathfrak{f}_F(\boldsymbol{FBO}) &= \boldsymbol{FBO} \wedge (\neg \boldsymbol{HS} \wedge \boldsymbol{IU}) \end{aligned}$$

The BDD. The resulting BDD for φ under the given configuration is shown in Figure 4.2, annotated with probability computations. The probability values are calculated differently for AT and FT nodes.

For AT nodes, the *Low* edge inherits the value of its target node unchanged, while the *High* edge computes the product of the target node's value and the current node's probability. The node's final value is the maximum of its *Low* and *High* edge values. This matches $\mathtt{BDD}_{\mathtt{DAG}}$. For FT nodes, we follow $\mathtt{p}_{\mathtt{TLE}}$. Both edges of a node v multiply their target values by different probabilities: the *Low* edge by $1 - \alpha(v)$ (probability of non-occurrence) and the *High* edge by $\alpha(v)$ (probability of occurrence). The node's value is the sum of both edge values.

Manual calculation. Table 4.2 shows the manual calculation of the probability according to the semantics. The first three columns represent all possible combinations of the fault tree variables, $\vec{b}_F \in \mathcal{S}_F$. For each combination, we calculate:

- $\mathsf{Prob}(\vec{b}_F)$: the probability of this fault scenario occurring
- $\mathbb{P}_A(\mathsf{Set}(\varphi, \vec{b}_F, \vec{b}_O))$[2]: the maximal probability of a successful attack under this fault scenario (and the given configuration)
- Their product: the contribution of this scenario to the total probability

The sum of all products (0.050078) gives us $\rho(\varphi, \vec{b}_O)_{A,F}$, which matches the result from our BDD-based computation.

Comparing the two methods. As we can see, the results from the BDD and the manual calculation are equal. Both methods perform an exhaustive analysis of all possible fault scenarios $\vec{b}_F \in \mathcal{S}_F$. While the table explicitly lists each scenario and its contribution, the BDD implicitly represents these scenarios through its paths from the root to terminal nodes.

A path in the BDD starts at the root node, and ends at one of the terminal nodes. Now, let us consider subpaths of these paths, such that they end at the first AT node they encounter (if any), and let us refer to this AT (or terminal) node as Δ.

Every such subpath corresponds to one or more rows in the table. If Δ is the terminal node $\top$ or $\bot$, the value of $\mathbb{P}_A$ in those rows should equal 1 or 0, respectively. If the subpath ends at an AT node, the value of $\mathbb{P}_A$ should be equal to the value of the node.[3] The value of Δ is calculated by $\mathtt{BDD}_{\mathtt{DAG}}$, and is exactly equal to $\mathbb{P}_A(\mathsf{Set}(\varphi, \vec{b}_F, \vec{b}_O))$, as discussed in the previous section.

All edges until Δ have a value that is determined by the source node v ($\alpha(v)$ for the *High* edge, $1 - \alpha(v)$ for the *Low* edge). Multiplying these values along the path gives us

[2] Abbreviated as $\mathbb{P}_A$ in Table 4.2.

[3] For example, the subpath $\boldsymbol{DSL} \dashrightarrow \boldsymbol{FBO} \longrightarrow \boldsymbol{EDLU}$ corresponds to the two rows where $\boldsymbol{DSL} = 0$ and $\boldsymbol{FBO} = 1$. We have $\Delta = \boldsymbol{EDLU}$, which has a probability value of 0.17 computed by $\mathtt{BDD}_{\mathtt{DAG}}$, and the $\mathbb{P}_A$ value for these rows is also 0.17.

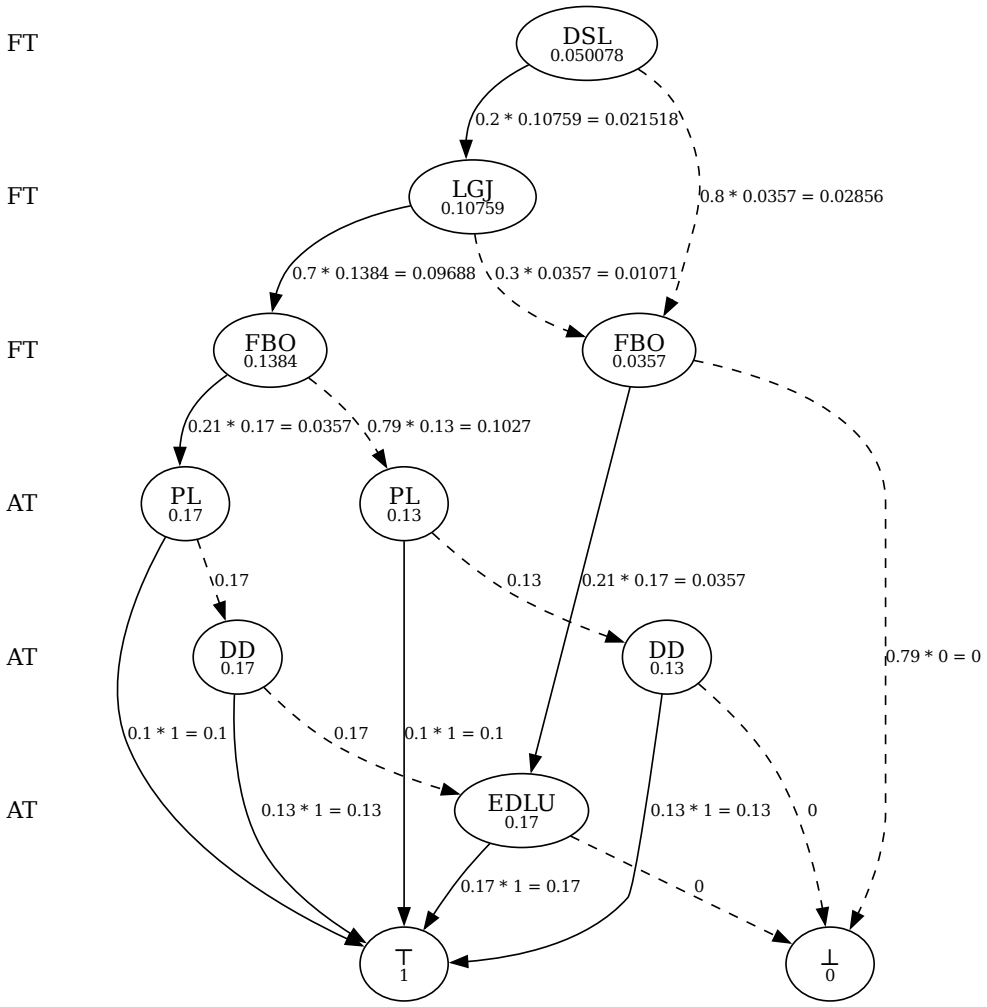


FIGURE 4.2: BDD for $\boldsymbol{FD} \wedge \boldsymbol{DGB} \vee \boldsymbol{EDLU} \wedge \boldsymbol{FBO}$, annotated with probability computation, with FT nodes before AT nodes. See Table 4.1 for abbreviations and used probabilities.

TABLE 4.2: Step-by-step calculation of $\rho(\varphi, \vec{b_O})_{A,F}$ by enumerating all fault scenarios and their corresponding attack probabilities.

DSL	***LGJ***	***FBO***	$\mathsf{Prob}(\vec{b_F})$	$\mathbb{P}_A$	$\mathsf{Prob}(b_F) \cdot \mathbb{P}_A$
0	0	0	0.18960	0	0
0	0	1	0.05040	0.17	0.008568
0	1	0	0.44240	0	0
0	1	1	0.11760	0.17	0.019992
1	0	0	0.04740	0	0
1	0	1	0.01260	0.17	0.002142
1	1	0	0.11060	0.13	0.014378
1	1	1	0.02940	0.17	0.004998
				$\rho(\phi, \vec{b_O})_{A,F} =$	0.050078

the probability of the fault scenario, which should be the same as the sum of the $\mathsf{Prob}(\vec{b_F})$ values for the corresponding rows in the table.[4] Ultimately, the value of the root node is the sum of the products of each subpath value and its corresponding Δ node value, which is the same as the sum of the products in the table.

Attacks first counterexample. To demonstrate the importance of the order of AT and FT nodes in the BDD, we present a counterexample. Figure 4.3 shows the BDD for the same formula, but with AT nodes before FT nodes. The resulting probability value is 0.0357, which is different from the correct value of 0.050078. This discrepancy arises because the BDD does not correctly represent the semantics of the formula.

Attack tree nodes have the unique property that they only influence the resulting probability value if the attack step is performed. If the attack step is not performed, the probability value remains unchanged. To maximize the resulting probability, only the minimum necessary attack steps should be performed.

In the BDD in Figure 4.3, you can already see this going wrong in the path ***PL*** $\longrightarrow$ ***EDLU*** $\longrightarrow$ .... Both ***PL*** and ***EDLU*** are performed, which—if the attacker knows which fault tree nodes have failed—is never necessary to satisfy the formula ***FD*** $\wedge$ ***DGB*** $\vee$ ***EDLU*** $\wedge$ ***FBO***: the attacker can always choose to perform either ***FD*** or ***EDLU*** to satisfy the formula.

4.1.3 Layer 3 Formulae

Layer 3 formulae analyze risks for specific objects. Each formula takes a single object as input and determines: the most risky event or action in which it participates, the total risk it is exposed to, or the optimal configuration of object properties to minimize the risk exposure. Setting evidence is also supported for layer 3 formulae, but only for object properties.

[4] Example: the subpath ***DSL*** $\dashrightarrow$ ***FBO*** $\longrightarrow$ ***EDLU*** has two edges. The probability of ***DSL*** $\dashrightarrow$ ***FBO*** is 0.8, and the probability of ***FBO*** $\longrightarrow$ ***EDLU*** is 0.21 (in the BDD this is the first number in the edge label). The product of these values is 0.168. The $\mathsf{Prob}(\vec{b_F})$ values in the table for the two rows corresponding to this path are 0.05040 and 0.11760, and their sum is 0.168.

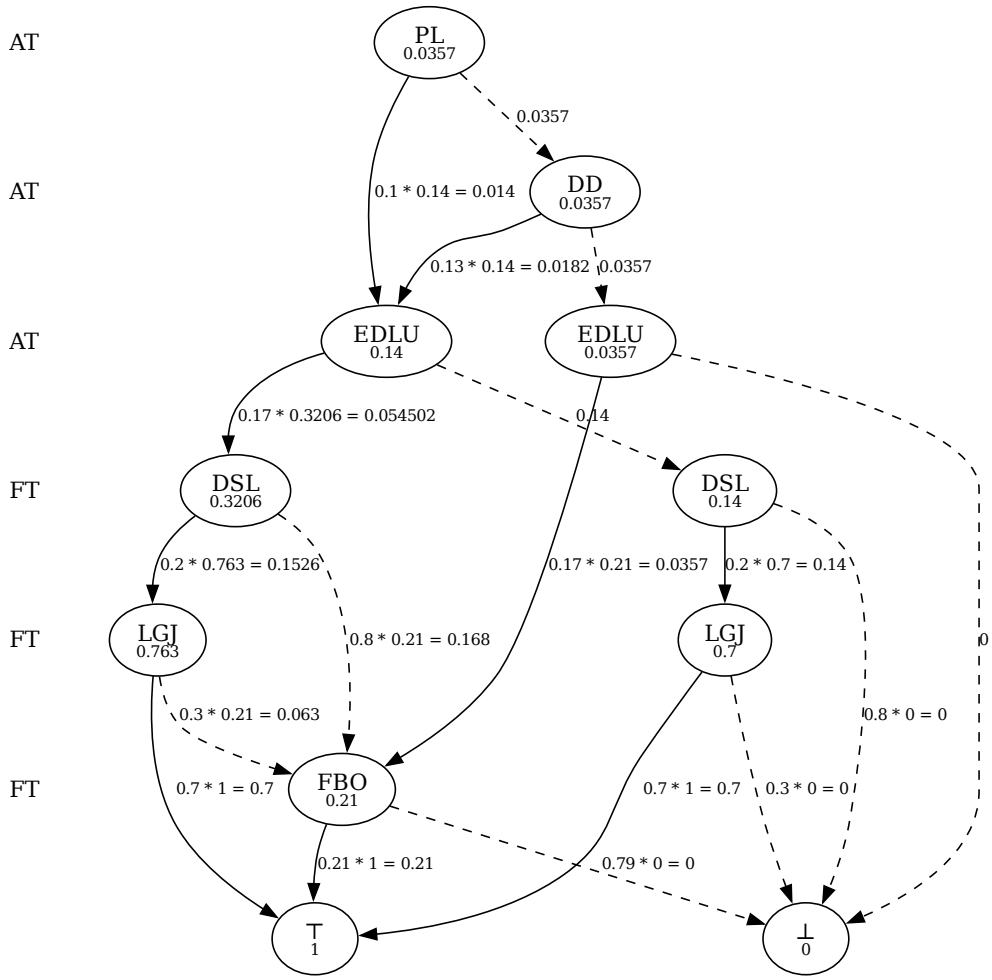


FIGURE 4.3: BDD for $\boldsymbol{FD} \wedge \boldsymbol{DGB} \vee \boldsymbol{EDLU} \wedge \boldsymbol{FBO}$, annotated with probability computation, with AT nodes before FT nodes. See Table 4.1 for abbreviations and used probabilities.

Layer 3 provides four functions:

- $\mathsf{MostRisky}_*$ (with $* \in \{\boldsymbol{A}, \boldsymbol{F}\}$): Returns the most risky event ($\mathsf{MostRisky}_{\boldsymbol{F}}$) or action ($\mathsf{MostRisky}_{\boldsymbol{A}}$) for an object
- $\underset{\text{min}}{\mathsf{TotalRisk}}$ and $\underset{\text{max}}{\mathsf{TotalRisk}}$: Return the minimum and maximum total risk an object is exposed to, based on the different possible configurations of object properties.
- $\mathsf{OptimalConf}$: Returns the configuration of object properties that minimizes the total risk to an object

When we set evidence in layer 3, we effectively constrain the set of possible configurations $\mathcal{C}$ to a subset of configurations that satisfy the evidence.

For the connection between objects and DT elements, the Pa function is used (defined in Definition 2.11). More specifically, [31] defines $(\!|o|\!)_*$ (with $* \in \{\boldsymbol{A}, \boldsymbol{F}\}$) as the set of all events ($(\!|o|\!)_{\boldsymbol{A}}$) or actions ($(\!|o|\!)_{\boldsymbol{F}}$) in which o participates, and for which a satisfying risk scenario and configuration exists:

$$(\!|o|\!)_* = \left\{ a \in N_* \mid o \in Pa(a) \wedge \exists \vec{b}_*, \vec{b}_O . f_*^\circ(\vec{b}_*, \vec{b}_O, a) = 1 \right\}$$

Furthermore, [31] defines the $\mathsf{objRiskVal}$ function as the total risk an object is exposed to given a configuration $\vec{b}_O$:

$$\mathsf{objRiskVal}(o, \vec{b}_O) = \sum_{a \in (\!|o|\!)_{\boldsymbol{A}} \cup (\!|o|\!)_{\boldsymbol{F}}} \rho(a, \vec{b}_O)_{A,F} \cdot Im(a)$$

Then, the functions $\mathsf{MostRisky}_*$, $\underset{\text{min}}{\mathsf{TotalRisk}}$, $\underset{\text{max}}{\mathsf{TotalRisk}}$, and $\mathsf{OptimalConf}$ are defined as follows:

$$\begin{aligned}
\mathsf{MostRisky}_*(o) &= \operatorname*{argmax}_{a \in (\!|o|\!)_*} \max_{\vec{b}_O \in \mathcal{C}} \left(\rho(a, \vec{b}_O)_{A,F} \cdot Im(a) \right) \\
\underset{\text{min}}{\mathsf{TotalRisk}}(o) &= \min_{\vec{b}_O \in \mathcal{C}} \mathsf{objRiskVal}(o, \vec{b}_O) \\
\underset{\text{max}}{\mathsf{TotalRisk}}(o) &= \max_{\vec{b}_O \in \mathcal{C}} \mathsf{objRiskVal}(o, \vec{b}_O) \\
\mathsf{OptimalConf}(o) &= \operatorname*{argmin}_{\vec{b}_O \in \mathcal{C}} \mathsf{objRiskVal}(o, \vec{b}_O)
\end{aligned}$$

4.1.3.1 Implementation.

The implementation of each layer 3 functions requires similar computational steps: each function iterates over all elements in which the object participates, and then it compares risk values across all possible configurations. The risk values are used differently depending on the specific function: $\mathsf{MostRisky}_*$ maximizes over individual elements, while $\underset{\text{min}}{\mathsf{TotalRisk}}$, $\underset{\text{max}}{\mathsf{TotalRisk}}$ and $\mathsf{OptimalConf}$ operate on the sum of risks across all elements, for each configuration.

Finding all elements in which the object participates, i.e., $\{a \in N_* \mid o \in Pa(a)\}$, is a trivial task. However, ensuring that a satisfying risk scenario and configuration exist for each element requires more thought.

Initially, verifying the existence of satisfying scenarios appears like a task perfectly suited for a SAT solver. Alternatively, this verification could be performed using BDDs by checking that the constructed BDD is not the terminal node $\bot$. However, SAT solvers are often more efficient than BDDs for this specific task. The key observation is that the

probability calculations require BDDs regardless of the verification method chosen. Since we can use the same BDD for both probability calculation and scenario verification, we avoid the need for additional computation.

In the BDDs, configuration choices are represented by outgoing edges of OP nodes. The system that results from these configuration choices is reflected in the non-OP nodes that are children of OP nodes. The BDD rooted at the direct child of the last OP node represents the system under a specific configuration, and this part of the BDD is used for probability calculations.

Definition 4.5. Given a DOG G and a BDD $B = (V, Low, High, Lab)$ for some formula, we define the set of configuration reflection nodes $\Gamma(G, B)$ as:

$$\Gamma(G, B) = \{v \in V \mid \exists u \in V.\ \mathcal{T}_{\mathrm{OP}}(G, u) \wedge v \in \{Low(u), High(u)\} \wedge \neg \mathcal{T}_{\mathrm{OP}}(G, v)\}$$

MostRisky. To find the most risky event or action for an object, we need to compare the risk values of all elements in which the object participates. For each element, we construct a BDD using Algorithm 1. The BDD is used to verify that a satisfying risk scenario and configuration exist, after which the same BDD is used for probability calculations.

For each element, all possible configurations of object properties must be considered. We calculate the probabilities of the nodes in the BDD that are direct children of OP nodes but are not OP nodes themselves; each of these nodes represents the system under a (set of) configuration(s). The highest probability value across all configurations, multiplied by the element's impact, equals the maximum risk value for that element.

Algorithm 3 Find most risky event/action for object o

Input: DOG G, object o, type $* \in \{\boldsymbol{A}, \boldsymbol{F}\}$
Output: element $a \in (\!|o|\!)_*$ with highest risk value, i.e., $\mathsf{MostRisky}_*(o)$

```
1:  function MostRisky(G, o, *)
2:      maxRisk ← −1, maxElement ← −
3:      for all a ∈ {a ∈ N_* | o ∈ Pa(a)} do
4:          BDDa = (V, Low, High, Lab) ← L1BDD(G, a)
5:          if BDDa ≠ ⊥ then    // Check that satisfying scenario and configuration exist
6:              V' ← Γ(G, BDDa)
7:              risk ← −1
8:              for all v ∈ V' do
9:                  p ← CalcNodeProb(v)    // Using function from Algorithm 2
10:                 risk ← max(risk, p · Im(a))
11:             if risk > maxRisk then
12:                 maxRisk ← risk
13:                 maxElement ← a
14:     return maxElement
```

TotalRisk. The $\mathsf{TotalRisk}_{\min}$ and $\mathsf{TotalRisk}_{\max}$ functions determine the minimum and maximum total risk an object is exposed to across all possible configurations of object properties. This calculation differs from that of $\mathsf{MostRisky}_*$. $\mathsf{MostRisky}_*$ compares individual elements to find the one with the highest potential risk. In contrast, $\mathsf{TotalRisk}_{\min}$ and $\mathsf{TotalRisk}_{\max}$ consider the cumulative risk from all relevant elements for each specific configuration. The function $\mathsf{objRiskVal}$, defined previously, calculates this cumulative risk for a given object o and configuration $\vec{b}_O$. It sums the risk contributions $(\rho(a, \vec{b}_O)_{A,F} \cdot Im(a))$ of all elements a in which o participates. A BDD is required for each such element a to compute $\rho(a, \vec{b}_O)_{A,F}$.

Because the number of configurations grows exponentially with the number of object properties, a direct approach of iterating through every possible configuration to calculate objRiskVal and then finding the minimum or maximum is computationally infeasible. A more efficient method using a symbolic representation of configurations is necessary.

We utilize Multi-Terminal Binary Decision Diagrams (MTBDDs) for this purpose. An MTBDD is constructed for each element a in which the object o participates. This MTBDD, denoted $MTBDD_a$, maps every possible configuration $\vec{b_O}$ to the risk contribution of element a under that configuration, i.e., $\rho(a, \vec{b_O})_{A,F} \cdot Im(a)$. Paths from the root to terminal nodes in $MTBDD_a$ represent specific configurations. The value of a terminal node is the risk contribution of a for the configuration(s) leading to it.

The individual MTBDDs, each representing the risk contribution of a single element a for object o, are combined to determine the total risk. This combination is achieved by summing these MTBDDs using the MTBDD addition operation. The sum is calculated as:

$$MT_{sum}(o) = \bigoplus_{a \in (\!|o|\!)_{\boldsymbol{A}} \cup (\!|o|\!)_{\boldsymbol{F}}} MTBDD_a$$

This resulting sum, $MT_{sum}(o)$, is itself an MTBDD. It symbolically represents the $\mathsf{objRiskVal}(o, \vec{b_O})$ function for an object o, where each path from the root to a terminal node corresponds to a specific configuration $\vec{b_O}$. The value of that terminal node is the total risk $\mathsf{objRiskVal}(o, \vec{b_O})$ for that specific configuration. The set of all terminal values in $MT_{sum}(o)$, denoted $Terms(MT_{sum}(o))$, therefore contains all possible values of $\mathsf{objRiskVal}(o, \vec{b_O})$. Consequently, $\underset{\text{min}}{\mathsf{TotalRisk}}(o)$ is found by taking the minimum value in $Terms(MT_{sum}(o))$. $\underset{\text{max}}{\mathsf{TotalRisk}}(o)$ is found by taking the maximum value in $Terms(MT_{sum}(o))$. This MTBDD-based method efficiently calculates total risk by avoiding the explicit enumeration of all configurations. The algorithm for computing the total risk for an object o is shown in Algorithm 4.

OptimalConf. From the semantics, we can see that OptimalConf is very similar to $\underset{\text{min}}{\mathsf{TotalRisk}}$. The only difference is the use of the argmin function instead of the min function. This means that we need to find the configuration that minimizes the total risk. We can use the same MTBDD from ConfigsToRiskMTBDD (see Algorithm 4) to find the configuration that minimizes the total risk. Instead of returning the minimum value of the terminals, we return the configurations that correspond to the paths leading to this minimum value. This process is detailed in Algorithm 5.

4.2 Implementation

The implementation of the DOGLog logic and the supporting framework is realized through a tool named Object-oriented Disruption Framework (ODF) (find the repository at [42]). ODF is designed to be used from the command line. A user interacts with the tool by first creating an `.odf` input file. This file defines the DOG model, including its constituent attack trees, fault trees, and object graph, as well as the DOGLog formulae to be evaluated. The user then invokes ODF via a command-line interface, providing the path to this `.odf` file. ODF processes the input, constructs the internal representation of the DOG, evaluates the specified DOGLog queries, and presents the results directly in the terminal. The overall workflow of the tool, from user input to result presentation, is summarized in Table 4.3.

Algorithm 4 Compute total risk for object o

Input: DOG G, object o, function $f \in \{\min, \max\}$
Output: minimal or maximal total risk value for object o

function TotalRisk(G, o, f)
  $MT_{sum} \leftarrow$ ConfigsToRiskMTBDD(G, o)
  return $f(Terms(MT_{sum}))$ *// Return the min or max of the terminals*

function ConfigsToRiskMTBDD(G, o)
  $MTs \leftarrow \emptyset$ *// Set of MTBDDs, one for each element*

  for all $a \in \{a \in N_A \cup N_F \mid o \in Pa(a)\}$ **do**
    $BDDa = (V, Low, High, Lab) \leftarrow$ L1BDD(G, a)
    if $BDDa \neq \bot$ **then**
      $V' \leftarrow \Gamma(G, BDDa)$

      $MTBDDa \leftarrow$ CreateMTBDD($BDDa, V', Im(a)$)
      $MTs \leftarrow MTs \cup \{MTBDDa\}$

  return SumMTBDDs(MTs)

function CreateMTBDD($BDDa, V', Im(a)$)
  $MTBDDa \leftarrow BDDa$
  for all $v \in V'$ **do**
    $p \leftarrow$ CalcNodeProb(v)
    Replace node v in $MTBDDa$ with terminal node with value $p \cdot Im(a)$
  return $MTBDDa$

function SumMTBDDs(MTs)
  $MT_{sum} \leftarrow 0$
  for all $m \in MTs$ **do**
    $MT_{sum} \leftarrow MT_{sum} + m$
  return MT_{sum}

Table 4.3: High-level workflow of the ODF tool.

Step	Description
1.	The user prepares an `.odf` input file.
2.	ODF parses the `.odf` file and constructs an internal DOG model.
3.	The tool evaluates the DOGLog queries defined in the input file, using the algorithms detailed in Section 4.1.
4.	Results of the evaluation are presented to the user via command-line output.

Algorithm 5 Find optimal configuration for object o

Input: DOG G, object o
Output: Set of configurations C that minimize total risk for o

```
function OptimalConf(G, o)
    MT_sum ← ConfigsToRiskMTBDD(G, o)
    C ← FindPathsToMinTerminal(MT_sum)
    return C

function FindPathsToMinTerminal(M)
    v_min ← ∞
    C ← ∅
    S ← Stack()    // Initialize an empty stack
    Push(S, (Root(M), ∅))    // Push root node and empty path

    while S is not empty do
        (n, P) ← Pop(S)    // n is node, P is current path (configuration)
        if n is a terminal node then
            if Lab(n) < v_min then
                v_min ← Lab(n)
                C ← {P}
            else if Lab(n) = v_min then
                C ← C ∪ {P}
        else
            x ← Lab(n)
            P_low ← P ∪ {(x, 0)}
            Push(S, (Low(n), P_low))
            P_high ← P ∪ {(x, 1)}
            Push(S, (High(n), P_high))
    return C
```

4.2.1 Technology

ODF is developed using Python version 3.13. Python was chosen for its ease of use and the author's familiarity with the language. Alternative languages such as Rust and C++ were considered. However, they were ultimately not chosen due to concerns that the time required for learning and mastering these languages would detract from other critical aspects of the thesis research.

ODF uses some key libraries to provide its functionality.

At the core of ODF's analytical capabilities are Binary Decision Diagrams (BDDs) and Multi-Terminal Binary Decision Diagrams (MTBDDs). These data structures are managed using a fork [41] of the `dd` library [49]. This fork fixes some bugs that were found during the development of ODF and adds some additional features to the `BDD` class. The `dd` library is a wrapper around `CUDD`, a C library for manipulating BDDs and MTBDDs [11]. `dd` was chosen for its support for both BDDs and MTBDDs, as well as its ergonomic Python API. Another potential option was `OxiDD` [17], but its Python bindings do not support MTBDDs at the time of writing.

The ODF tool processes input files written in a custom domain-specific language (DSL). These files, typically with an `.odf` extension, define the DOG (Attack Tree, Fault Tree, and Object Graph) and the DOGLog formulae to be evaluated. Parsing of this DSL is handled by the `Lark` library, a parsing toolkit for Python [22].

For the representation and manipulation of graph structures, such as the attack trees, fault trees, and object graphs, the `NetworkX` library is used [15]. The library provides a comprehensive set of tools for creating and analyzing complex graphs. Inside ODF, it is mainly used to check for connectivity, acyclicity, and to find all descendants of a node in the object graph.

ODF is used as a command-line tool, taking `.odf` files as input. The tool parses the input files, constructs the DOG, and then uses algorithms explained in Section 4.1 to check the DOGLog formulae. The results of the analysis are then presented to the user in structured and colored text output.

4.2.2 Numerical Precision

For some calculations, namely for layer 2 probabilities (Algorithm 2) and $\mathsf{MostRisky}_*$, ODF utilizes the Python `Fraction` class.[5] This class represents numbers with integer numerators and denominators. This representation allows for exact arithmetic, thereby avoiding floating-point rounding errors. Given that Python integers support arbitrary precision, limited only by available memory [9], these calculations are performed with exact precision.

The `CUDD` library, used for MTBDD operations, employs double-precision floating-point numbers for terminal node values. This affects MTBDD addition, which is used for $\mathsf{TotalRisk}_{\min}$, $\mathsf{TotalRisk}_{\max}$, and $\mathsf{OptimalConf}$ (see Algorithms 4 and 5). Therefore, these calculations may have floating-point rounding errors.

4.2.3 Optimal Configuration Representation

The $\mathsf{OptimalConf}$ function (Algorithm 5) finds object property configurations that minimize total risk. It does this by identifying paths to the terminal node with the minimum risk value in the sum MTBDD, $MT_{sum}(o)$.

[5] https://docs.python.org/3.13/library/fractions.html

It is well-known that different variable orderings can create (MT)BDDs of very different sizes [7]. The variable order in an MTBDD affects its structure, and therefore the paths that are found. ODF applies sifting to $MT_{sum}(o)$ before path extraction to get a more consistent representation. Sifting reorders MTBDD variables to try to minimize its size [38]. A smaller MTBDD can provide a clearer, more compact representation of optimal configurations.

For instance, consider three object properties OP_1, OP_2, OP_3. Suppose the optimal configurations (those yielding minimum risk R_{min}) require $OP_1 = true$ and $OP_3 = true$, while OP_2 can be *true* or *false* (a "don't care" condition). One variable order resulting from sifting (e.g., OP_1, OP_3, OP_2) might lead to a single path representing this optimal set: $\{OP_1 \mapsto true, OP_3 \mapsto true\}$. Here, OP_2 might not appear on the path if the MTBDD structure captures its irrelevance. Another sifting run might produce a different order (e.g., OP_1, OP_2, OP_3). This could result in two explicit paths being returned by Algorithm 5:

- $\{OP_1 \mapsto true, OP_2 \mapsto false, OP_3 \mapsto true\}$
- $\{OP_1 \mapsto true, OP_2 \mapsto true, OP_3 \mapsto true\}$

Both representations (a single path with an implicit don't care, versus two explicit paths) describe the same set of optimal conditions, with the first being more concise.

The sifting algorithm in `CUDD` (used by ODF) is not deterministic. Different runs might produce different variable orderings. This means the paths returned by Algorithm 5 for optimal configurations can vary between runs. However, these different sets of configurations are always logically equivalent. The minimal risk value itself also remains consistent. So, while the identified optimal configurations are valid, their specific path representation may change.

4.2.4 Testing Methodology

ODF is supported by an extensive automated test suite developed using the `pytest` framework. The testing strategy aims for comprehensive coverage of the tool's functionality using, among others, unit, integration, and error-handling tests. The test suite is structured to mirror the codebase.

Core logic and layers. The tests systematically cover the core components of the DOGLog implementation:

Parsing and model transformation. The parser's ability to correctly interpret DOGLog syntax and the subsequent transformation into internal models (Disruption Trees, Object Graphs, Configurations) are thoroughly validated. This includes testing various syntactic constructs and model validation rules.

Layer 1 (boolean logic and BDDs). Extensive tests verify the correct construction of Binary Decision Diagrams (BDDs) for a wide range of DOGLog formulae, from simple atomic propositions to complex nested expressions involving all Boolean operators. Satisfaction checking of these formulae against different configurations is also a key focus.

Layer 2 (probability calculations). The evaluation of probabilities for DOGLog formulae is tested, ensuring that calculations are correct under various configurations and when influenced by probabilistic evidence.

Layer 3 (risk aggregation). Tests cover risk aggregation functions (e.g., `TotalRisk`), identification of critical nodes (e.g., `MostRisky`), and the search for optimal configurations (`OptimalConf`). The construction of Multi-Terminal BDDs (MTBDDs) for risk calculations is also validated.

Utility functions. Core utility functions, such as Depth-First Search (DFS) algorithms used for BDD and MTBDD traversal and analysis, are independently tested.

Key features and operators. Specific DOGLog features and operators receive dedicated testing:

Evidence handling. We test the correct application and precedence of both Boolean (layer 1 and 3) and probabilistic (layer 2) evidence. This includes scenarios with single, multiple, nested, and conflicting evidence, as well as evidence interacting with node conditions and configurations.

MRS operator. The MRS operator, found in layer 1, is tested in various contexts: with simple and complex formulae, in conjunction with evidence, and in nested forms, ensuring its semantics are correctly implemented for identifying minimal risk scenarios.

BDD traversal. Algorithms involving BDD traversal are tested on various BDD structures, including single-node BDDs, BDDs with and without shared subtrees, and BDDs with complemented edges. We also take care to test the proper handling of complemented edges by including BDD structures where nodes can be reached via both complemented and non-complemented paths.

Test types and coverage. Different types of tests are used to cover different parts of the implementation:

Unit tests. Individual functions and classes, particularly those implementing core algorithms (e.g., BDD construction, probability computation), are tested in isolation.

Integration tests. The interaction between different components is verified. Layer-specific check functions serve as integration points for functionalities within that layer.

Scenario testing. The test suite covers a wide range of scenarios, including:

- Formulas representing tautologies and contradictions.
- Simple and deeply nested formula structures.
- Empty or trivial inputs for evidence and configurations.
- Probabilistic evidence with boundary values (0 and 1).
- Models with unusual structures (e.g., single-child intermediate nodes, nodes with no participating objects).
- Unsatisfiable nodes or conditions due to evidence.

Error handling. The implementation's robustness is tested by providing invalid inputs or creating inconsistent states. Tests verify that appropriate exceptions (e.g., `InvalidNodeEvidenceError`) are raised and that warnings are issued for non-critical issues (e.g., unused variables in configurations).

Chapter 5

Case Study

This chapter addresses the second research question: *How can DOGLog be used for performing risk analysis on a complex system, and what conclusions can be drawn from the application of DOGLog to such a system?* It presents a detailed case study to demonstrate the practical application of the DOGLog logic and the ODF tool.

5.1 Introduction

This chapter presents a case study to demonstrate the practical application of the WATCHDOG framework. We construct an object-oriented disruption graph (DOG) based on a Boolean logic Driven Markov Processes (BDMP) model from existing literature [20]. This process demonstrates the capability of translating models from other formalisms into a DOG. To fully showcase the features of WATCHDOG, we augment the translated DOG with additional object properties and safety-security interactions. Finally, we utilize DOGLog, our logic for reasoning over DOGs, to analyze the case study. This analysis highlights the power of DOGLog in assessing system risks and evaluating the impact of different configurations.

5.2 Pipeline System

The case study focuses on a hypothetical cyber-physical system designed for transporting a polluting substance, as described in [20]. The core of the system is a pipeline. This pipeline is equipped with pumps to propel the substance and valves to control its flow. Sensors are distributed along the pipeline. These sensors continuously measure pressure and flow within each section. The operation of each pump and valve is managed by a Remote Terminal Unit (RTU). Each RTU communicates with a central Control Center (CC). The RTUs have several key responsibilities. They (1) collect data from nearby pressure and flow sensors, (2) control the operational speed of pumps and the state (open/closed) of valves, (3) transmit data and alarm signals to the CC and receive instructions from it, (4) exchange pressure measurements and shutdown signals with adjacent RTUs.

A critical safety function of the RTUs is to ensure that the pipeline pressure does not surpass a predefined maximum, P_{max}, to prevent the pipeline from rupturing under excessive pressure. Each RTU also computes the pressure differential, $\Delta P = |P_n - P_{n-1}|$, between its own sensors and those of a neighboring RTU. If this ΔP exceeds a threshold ΔP_{max}, the RTU triggers an alarm to the CC. The CC then instructs all RTUs to halt pumps and close valves. Concurrently, the RTU issues a shutdown signal to its neighboring RTUs. The ΔP_{max} threshold is typically breached when a pipeline rupture occurs. This is

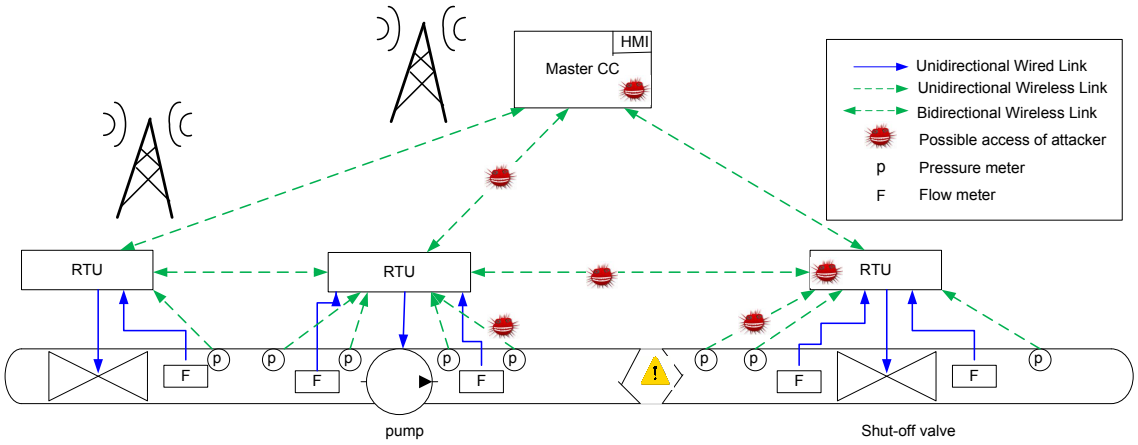


FIGURE 5.1: Schematic representation of the case study architecture (from [20]).

because the pressure before the break becomes significantly higher than the pressure after it, resulting in a large differential. A safety measure, termed the "Reflex Action", empowers each RTU to independently stop its controlled pump or close its valve upon reaching ΔP_{max} or receiving a shutdown order from another RTU. This action does not require prior instruction from the CC and serves as a high-priority redundant safety mechanism. The system architecture is depicted in Figure 5.1.

Kriaa et al. [20] make several assumptions about the physical setup and communication infrastructure. RTUs are assumed to be locally installed on pumps and valves, connected via wired links. Sensors, being more dispersed along the pipeline, utilize wireless links to communicate with RTUs. Given the extensive length of the pipeline (hundreds of kilometers) and the pipeline's distance from the CC (a hundred kilometers), communication is presumed to rely on a GSM network. The industrial protocols employed are Modbus/TCP for RTU-CC communication, Modbus/RTU for inter-RTU communication, and WirelessHART for sensor-RTU communication. These assumptions are used by safety and security experts to accurately estimate the parameters of events in their model [20].

5.2.1 BDMP Modeling

The risk analysis of the pipeline system is supported by a BDMP model, depicted in Figure 5.2 (from [20]). This model outlines various scenarios that can lead to the top event: pollution of the environment. We first introduce the BDMP concepts that are relevant to this case study. Then, we describe the BDMP model itself.

5.2.1.1 BDMP Concepts

Boolean logic Driven Markov Processes (BDMP) are graphical models for analyzing system reliability, first introduced in [5]. The formalism was later extended to incorporate security aspects, as detailed in [33, 34]. They extend traditional formalisms like fault trees and attack trees by incorporating state-dependencies and dynamic behaviors through Markov processes associated with events [35]. Similarly to fault and attack trees, BDMP typically feature a hierarchical structure, breaking down a main undesired event (top event) into more elementary events.[1]

[1]Though BDMP have one *main* top event, BDMP can have multiple top events [5]. The BDMP in Figure 5.2 has two top events: "attack occurrence" and "Pollution".

Common gate types define the logical relationships between events. The BDMP in Figure 5.2 utilizes:

- OR gates: The event occurs if at least one of the child events occurs.
- AND gates: The event occurs if all child events occur.
- Priority AND (PAND) gates: The event occurs if all child events occur in a specific, predefined order. In Figure 5.2, this is visually represented as an AND gate with a triangle symbol on top.

BDMP can contain different types of leaves. Figure 5.2 utilizes the following types:

- Failure in operation: Depicted by an icon of a broken component with a black exclamation mark (e.g., "pipe break accidentally"). These events occur during component operation, with an exponentially distributed time to failure.
- Failure on demand: Also depicted by an icon of a broken component with a red exclamation mark (e.g., "valves on demand failure to close"). These events occur with a specific probability when a component is called upon to function (e.g., 5e-5).
- Attacker Action: Represented by an icon of a crosshair (e.g., "access to CC"). These events have an associated exponentially distributed time to success (e.g., "10 days").
- Instantaneous Security Event (ISE): Shown as a crosshair icon, with "ISE" label in red (e.g., "high pumping pressure activation"). These occur with a specific probability when activated (e.g., 0.4).
- House Events represent system configurations that can be set to true or false prior to checking the model. In Figure 5.2, the node "No reflex action" represents such a condition.

BDMP also use triggers, shown as red dotted arrows in Figure 5.2, to model dynamic dependencies. Each leaf in a BDMP is associated with its own Markov process. This process can be in one of two modes: 'standby' or 'active' [5]. The behavior of a leaf depends on its current mode. For leaves associated with an exponentially distributed time to occurrence (like attacker actions or failures in operation), the process effectively 'ticks' or progresses towards occurrence only while in the 'active' state. For leaves associated with a probability of occurrence (like instantaneous security events or failures on demand), this probability is evaluated when the leaf's process transitions from the 'standby' state to the 'active' state. At the moment of this transition, the event occurs with its specified probability.

The mechanism that determines whether a leaf's process is in the 'standby' or 'active' state is governed by what Bouissou and Bon [5] term process selectors. A process selector, denoted X_i for an element i, is a Boolean function that dictates which of the two modes is currently selected for that element's Markov process. The value of X_i (1 for 'active' and 0 for 'standby') is determined by the state of the BDMP according to specific rules. If element i is a root of the tree structure, its process selector X_i is set to 1 (active). Otherwise, X_i is also 1, unless either:

1. The origin of a trigger pointing to element i has its structure function equal to 0 (meaning the triggering condition is not met).
2. Element i has at least one parent gate in the fault-tree, and all of these parent gates have their own process selectors equal to 0.

In essence, a leaf's process defaults to the 'active' mode unless it has a trigger condition which is not activated, or none of its direct parents are active. Triggers allow the state of

one part of the system to dynamically change the operational mode (and thus the behavior) of other components. This allows for the representation of complex sequential dependencies and dynamic system behaviors.

5.2.1.2 The BDMP Model

The BDMP in Figure 5.2 simultaneously represents three primary types of scenarios as considered by Kriaa et al. [20]: attack scenarios, accidental scenarios, and hybrid scenarios. Attack scenarios involve malicious actions. Accidental scenarios are based on component failures or other unintended events. Hybrid scenarios combine both malicious attacks and accidental failures, highlighting potential interactions between safety and security events.

Pollution occurs if the pipeline breaks and the protection system concurrently fails to react. The protection system encompasses the detection of a pipeline break by RTUs and the subsequent system shutdown, initiated either by the reflex action or by commands from the CC. Failure of the protection system can happen in two ways: it might be deactivated by an attacker prior to a break, or it could fail accidentally. This is represented by the OR gate labeled "possible scenarios" in the BDMP. Failure caused by an attacker is represented at the second layer of the BDMP using a Priority AND gate, labeled "attack protection syst then pipeline break". This gate signifies that the protection system must fail (its left input) *before* the pipeline breaks (its right input) for the combined event to occur.

The attack scenario, as modeled, assumes attacks follow a Poisson process with an estimated occurrence rate of once every five years. A typical attack involves deactivating the protection system and then causing a pipeline breach, for instance, through a water-hammer phenomenon. The water-hammer effect is achieved by suddenly closing a downstream valve while high-velocity, high-pressure flow is propagating, causing a shock. The preparation phase for an attacker includes gaining access to the SCADA system.[2] This could be through physical or remote control of the CC, physical access to an RTU, or network intrusion via communication links (RTU-CC or sensor-RTU). Subsequently, the attacker must understand the system's operation to effectively deactivate its protections. Once access and understanding are achieved, the attack steps to deactivate protections are considered quasi-instantaneous. To disable the RTUs' reflex action, an attacker might jam inter-RTU communication, preventing breach signals from reaching other RTUs. The model includes a configurable event, "No reflex action", which can be set to true or false to represent the presence or absence of this local safety measure in the system. After these preparations, the attacker can induce a water-hammer by creating high pumping pressure and abruptly closing a downstream valve, causing a pressure surge that can rupture the pipeline at its weakest point.

Accidental scenarios leading to pollution occur if the pipeline breaks accidentally and the protection system subsequently fails. Protection failure in this context can mean no instructions are given by the RTU, or there is an on-demand failure of equipment (valves and pumps) to respond correctly. The former case (no RTU instruction) can result from RTU failure or if the RTU does not react because it receives no CC instruction and its reflex action is not triggered. The BDMP details these accidental events as safety leaves.

Hybrid scenarios involve a combination of accidental and malicious events. For example, an attacker might remotely deactivate the protection system but then abandon the attack (e.g., failing to create a water-hammer). If the pipeline then breaks accidentally before the

[2]SCADA (Supervisory Control And Data Acquisition) systems are used for high-level supervision of industrial processes. In the context of this pipeline, the SCADA system includes the CC for overall monitoring and command, the RTUs which execute local real-time control logic, and the field devices (sensors, pumps, and valves) that the RTUs interface with and control.

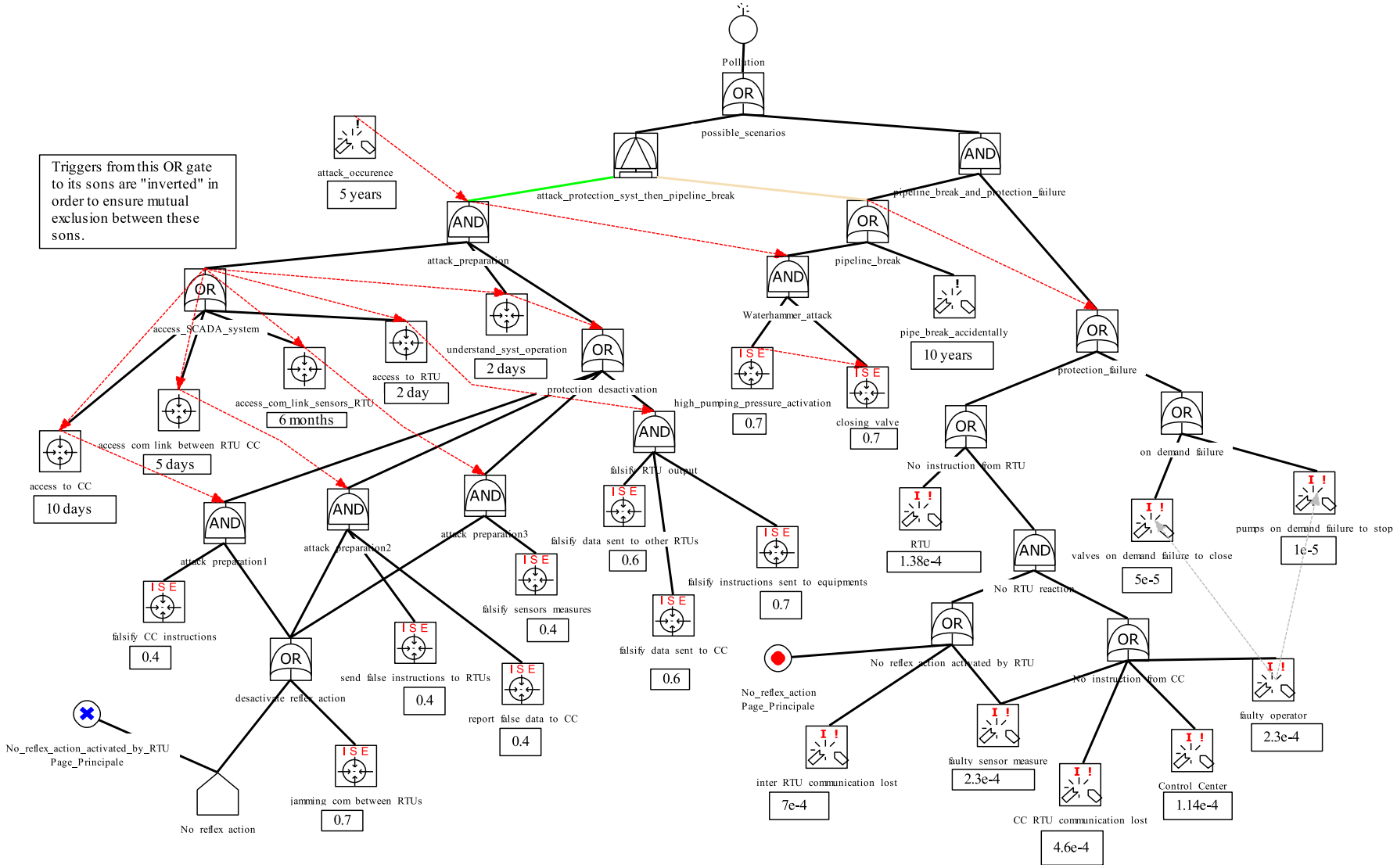


FIGURE 5.2: The source BDMP of the case study (from [20]).

deactivation is detected and remedied, pollution occurs. Kriaa et al. [20] note that this specific scenario has a very low probability, as it assumes the protection system deactivation remains undetected until an accidental break.

5.3 Creating the DOG

As previously mentioned, the BDMP has a hierarchical structure, similar to attack and fault trees. This similarity allows for a relatively straightforward translation of the BDMP structure into the attack and fault trees of the DOG. Although BDMP can theoretically combine attacks and faults within the same sub-trees, this was not a prominent feature in the case study's BDMP. The BDMP models the "pipeline break" node as an event that can be triggered either by an attack or accidentally. The hybrid scenario discussed in the source paper involves an attack that first disables the protection system, followed by an accidental pipeline break. The authors noted that this specific scenario has a very low probability. Consequently, for the translation into the DOG model, this particular situation is disregarded as it is not directly modelable with the DOG formalism's separation of attack and fault trees. Given its very low probability, this simplification is considered acceptable.

The construction of the fault tree begins by largely replicating the "protection failure" subtree from the BDMP. To model the "No reflex action" house event from the BDMP, an event named **No reflex action** is defined in the fault tree. This event is assigned a probability of 1 and is guarded by the negation of object property **Reflex_action_enabled**. This approach ensures that the model accurately reflects the safety measure's status: the **No reflex action** event is guaranteed to occur if the reflex action is disabled, and does not occur if it is enabled. The top-level event in the fault tree is termed **Accidental pollution**. The first child of this event is the **Protection failure** sub-tree. The other child is **Pipeline breaks accidentally**, corresponding to one of the "pipeline break" children in the BDMP. This structure accounts for the entirely accidental scenarios from the source BDMP.

For the attack tree, instead of the accidental pipeline break, the attack-induced pipeline break, i.e., the **Waterhammer attack** from the BDMP, is used. This **Waterhammer attack** becomes the first child of the top-level attack event, **Attacker causes pollution**. The other child of this top-level event requires careful construction. This subtree is based on the "attack preparation" subtree in the BDMP. In the BDMP, the "attack preparation" node has a child OR gate labeled "access SCADA system". This gate represents various methods to gain access to the SCADA system. Each child of this OR gate (representing an access method) is linked to a specific attack preparation step, which it activates using a trigger. Since attack trees do not feature triggers, these access nodes are instead modeled as children of their respective attack preparation nodes in the DOG's attack tree. The "attack preparation" node from the BDMP is renamed to **Attack protection system** in the attack tree. This **Attack protection system** node is an AND gate. Its children include **Understand system operation** and **Protection deactivation**, corresponding to the remaining children of the BDMP's "attack preparation" node. The **Protection deactivation** node retains the same children as in the BDMP, with each of these children having an additional child derived from the access nodes of the BDMP.

The object graph is then created based on the objects identified in the system. Figure 5.1 depicts the schematic representation of the case study architecture and helps identify these objects. This figure shows the *Control center*, *RTU*s, *Pumps*, *Valves*, and *Sensors*. Although the *Pipeline* itself is not explicitly labeled in the schematic, it is a central object in this case study.

The source paper also mentions the *SCADA system*, which is also included as an object. A *SCADA system* typically comprises a *Control center*, *RTU*s, and field-level equipment. In this case the field-level equipment consists of *Sensors*, *Pumps*, and *Valves*. A parent object called *Equipment* is introduced. This *Equipment* object is the parent of *Pumps*, *Valves*, and *Sensors*, and a child of the *SCADA system*. This structuring allows for analyzing risks to all equipment collectively, leveraging WATCHDOG's parthood relationship, in addition to analyzing risks for each equipment type individually.

Objects can have multiple parents. Since *Pumps*, *Valves*, and *Sensors* are also components of the *Pipeline*, parthood relationships are drawn accordingly.

Three types of links are also considered as objects: the link between an *RTU* and the *Control center*, links between *RTU*s, and links between *Sensors* and *RTU*s. These are modeled as distinct objects because they can possess different properties, such as being encrypted or wireless. While one could argue that these links are parts of the objects they connect (or vice versa), they are modeled as standalone objects. This decision avoids an automatic implication of risk transfer between connected components due to WATCHDOG's Assumption 3 (if an object participates in an event, its parts also participate). For instance, an incorrect sensor measurement should not automatically imply that the link between the sensor and the *RTU* is also at risk. If in a specific case such a dependency were desired, the model could be adapted, for example, by having the link also participate in the relevant event.

If the pipeline breaks and the protection system fails, significant environmental pollution occurs. Thus, the *Environment* is added as an object.

Lastly, an *Operator* object is included, as **Faulty operator** is considered a possible cause of the **No instruction from control center** event in the fault tree.

Next, the appropriate objects are connected to each element in the attack and fault trees. Probabilities are copied from the BDMP to the DOG. For elements in the BDMP that have a time to occurrence, corresponding probabilities are assigned in the DOG based on reasonable estimations.

These steps result in the initial DOG shown in Figure 5.3.

One of the unique features of WATCHDOG is its ability to analyze the impact of different configurations of object properties on the risk exposure of the system. Additionally, since these object properties connect the attack and fault trees of a DOG, they allow for modeling various safety-security interactions between measures. In the literature, safety-security interactions have been identified in four different categories [19, 35]:

- **Conditional dependency**: Fulfilling a safety requirement depends on a security requirement being fulfilled or vice-versa.
- **Mutual reinforcement**: Fulfilling a safety requirement reinforces the fulfillment of a security requirement, or a safety measure also improves security, or vice-versa.
- **Antagonism**: Fulfilling a safety requirement hinders the fulfillment of a security requirement, or a safety measure also decreases security, or vice-versa.
- **Independence**: No interaction between safety and security requirements or measures.

Mutual reinforcement and antagonism are also termed *synergy* and *conflict*, respectively. Using object properties, different security and safety measures can be modeled, which may be, for example, in synergy or conflict with each other. For conflicting properties, it is possible to analyze which configuration most effectively reduces risk across both the attack and fault domains. For measures in synergy, the optimal value is generally known, but

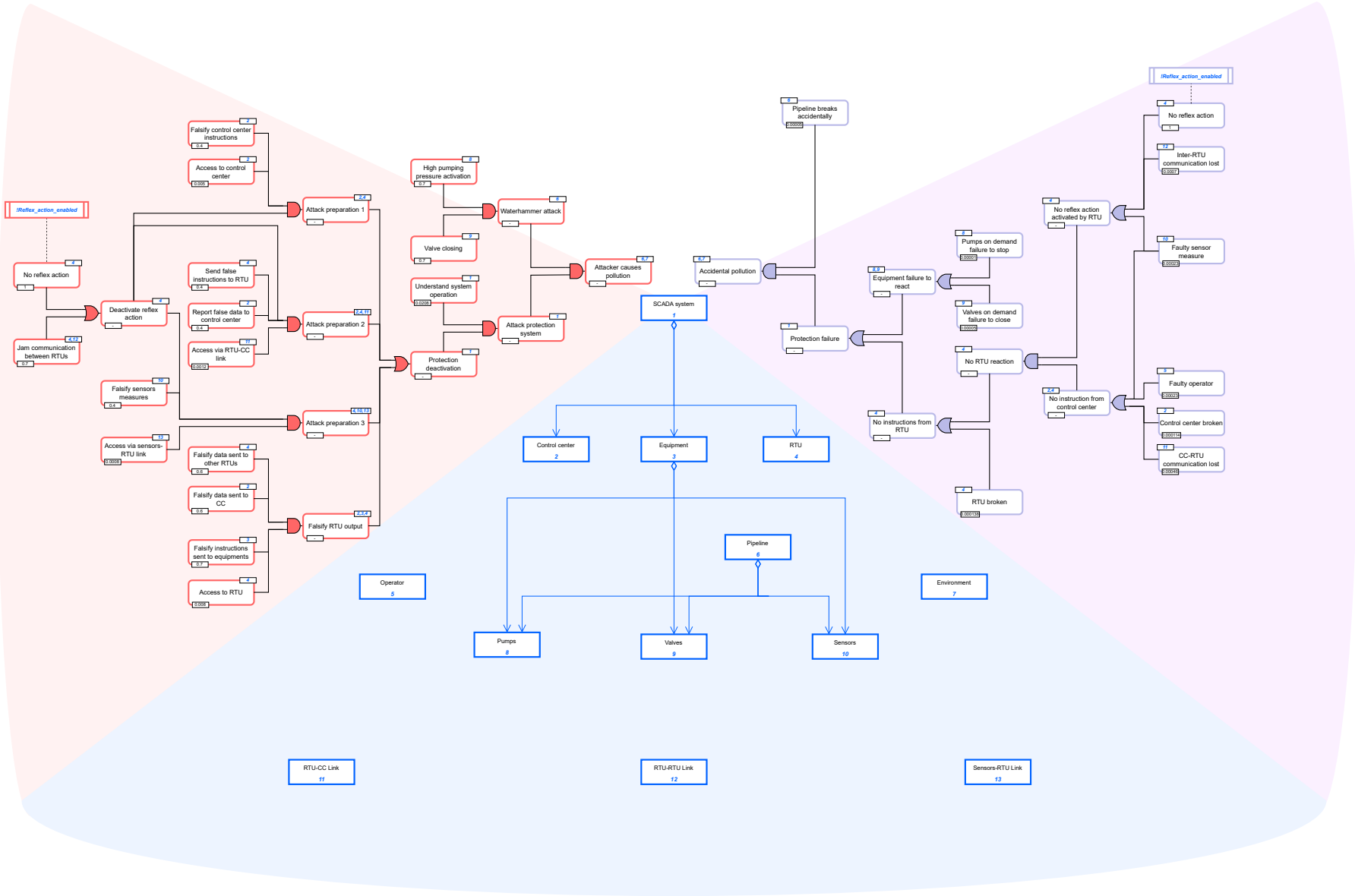


FIGURE 5.3: The initial DOG of the case study, derived from the BDMP before the added object properties.

their impact on risk reduction can be quantified. This information can support informed decisions regarding investment in potentially costly measures.

The initial model includes one object property, **Reflex_action_enabled**, indicating whether the reflex action mechanism is enabled or disabled. This property appears in both the attack and fault trees of the DOG, signifying it is not independent. Since it appears negated on both sides (i.e., enabling it reduces risk on both sides), it is synergetic. To make the case study more comprehensive, additional object properties are introduced, some of which are conflicting.

First, consider replacing the current wireless links between RTUs with wired links. Wired links would incur higher costs but could be justifiable if the risk reduction is significant. Making this link wired would prevent an attacker from jamming inter-RTU communication. Additionally, it would prevent the accidental loss of inter-RTU communication on the fault tree side. This object property is named **Wireless_RTU_RTU_link**. It is synergetic (where 'false' for **Wireless_RTU_RTU_link** implies wired and thus beneficial).

Next, consider encrypting the communication between the RTUs and the CC. Encryption would hinder an attacker from exploiting the RTU-CC link. However, RTUs often have limited computational resources, and encryption could introduce delays in processing instructions due to decryption requirements. This delay might be significant enough to introduce an additional cause for **No instruction from control center**, termed **Delayed instructions**. This scenario exemplifies a conflicting property: **RTU_CC_comm_encrypted**, which enhances security at the expense of safety.

As mentioned in Section 5.2, the system incorporates a *Reflex Action*. It is conceivable that a faulty sensor measurement could trigger this action accidentally (a false positive). A possible countermeasure is to allow overriding the reflex action from the control center. This would enable an operator to prevent an unnecessary shutdown. However, this override capability could also be exploited by an attacker with access to the control center or the RTU-CC link. The object property **Remote_CC_override_enabled** models this configuration choice. An attack **Override reflex action** is created as a sibling to **Deactivate reflex action**. Their common parent becomes **Deactivate or override reflex action**. **Attack preparation 1** (access to CC) and **Attack preparation 2** (access to RTU-CC link) are connected to this new parent node. **Attack preparation 3** (access to RTU) remains connected to **Deactivate reflex action**, as this attack path does not grant the ability to override from the CC. On the fault tree side, a parent node **Pipeline rupture** is added to **Pipeline breaks accidentally**. **Accidental waterhammer** is added as another child of **Pipeline rupture**. This **Accidental waterhammer** node is guarded by a negated **Remote_CC_override_enabled** property, as enabling the override can prevent an accidental waterhammer.

We can specify more detailed causes for **RTU broken**. This can result from hardware failure or a firmware bug. A firmware bug could be newly introduced by an update or an existing, undiscovered flaw. Since new software versions often contain bugs, the ability to roll back firmware to a previous version might be beneficial. The object property **Allow_firmware_rollback** is introduced for this. However, an attacker could trigger this rollback to exploit a known vulnerability in an older firmware version to gain **Access to RTU**. Presumably, finding and exploiting vulnerabilities in older firmware is easier than in the latest version, leading to possible security risks.

In the fault tree, the pipeline can break accidentally or due to an accidental waterhammer. Different materials could be chosen for pipeline construction. A stronger material, though more expensive, would be less prone to breaking accidentally. If a break did oc-

cur, it would presumably be smaller if a stronger material was used, reducing its impact. This is modeled by creating two children for **Pipeline breaks accidentally**: **Pipeline strong material break** and **Pipeline weak material break**. These children have different probabilities and impacts. **Pipeline strong material break** is conditioned on **Strong_material**, while **Pipeline weak material break** is conditioned on ¬**Strong_material**.

Similarly, the probability of **Faulty sensor measure** can be reduced by adding redundant sensors. If one sensor fails, a redundant sensor could still provide accurate measurements. This is modeled with two children events having different probabilities but the same impact, as the consequence of a faulty sensor measure remains the same. This object property is named **Redundant_sensors**.

Lastly, probabilities are assigned to the newly created basic events, and impact values are ascribed to all nodes. For elements that were further specifications of existing BDMP nodes, probability values similar to the original ones are used. For other new nodes, values are chosen that seem reasonable in the context of existing node values.

Assigning impact values required a different approach, as there was no direct comparison in the BDMP. The top-level events are assigned an impact of 100. Other nodes receive impacts based on their relative significance. Both top-level events (**Attacker causes pollution** and **Accidental pollution**) have the same impact, reflecting equally catastrophic outcomes.

Various DOGLog functions (e.g., $\mathsf{TotalRisk}_{\max}$, $\mathsf{TotalRisk}_{\min}$, $\mathsf{OptimalConf}$) use the sum of risks of all elements in which a particular object participates. Care must be taken when assigning impact values to nodes that merely represent multiple pathways to an event, to avoid double counting. For example, in the attack tree, there are multiple ways to achieve **Protection deactivation**, all leading to the same outcome. The chosen modeling approach assigns an impact to the parent **Protection deactivation** node, and not to its children representing the specific methods. Assigning impacts to the children as well would lead to double counting. Alternatively, assigning impacts only to the children and zero to the parent would cause the risk to grow disproportionately with each additional pathway. For instance, adding a fifth child to the **Protection deactivation** OR gate would increase its probability and also add the impact of the new child, distorting the overall risk assessment.

These modifications and additions result in the DOG depicted in Figure 5.4.

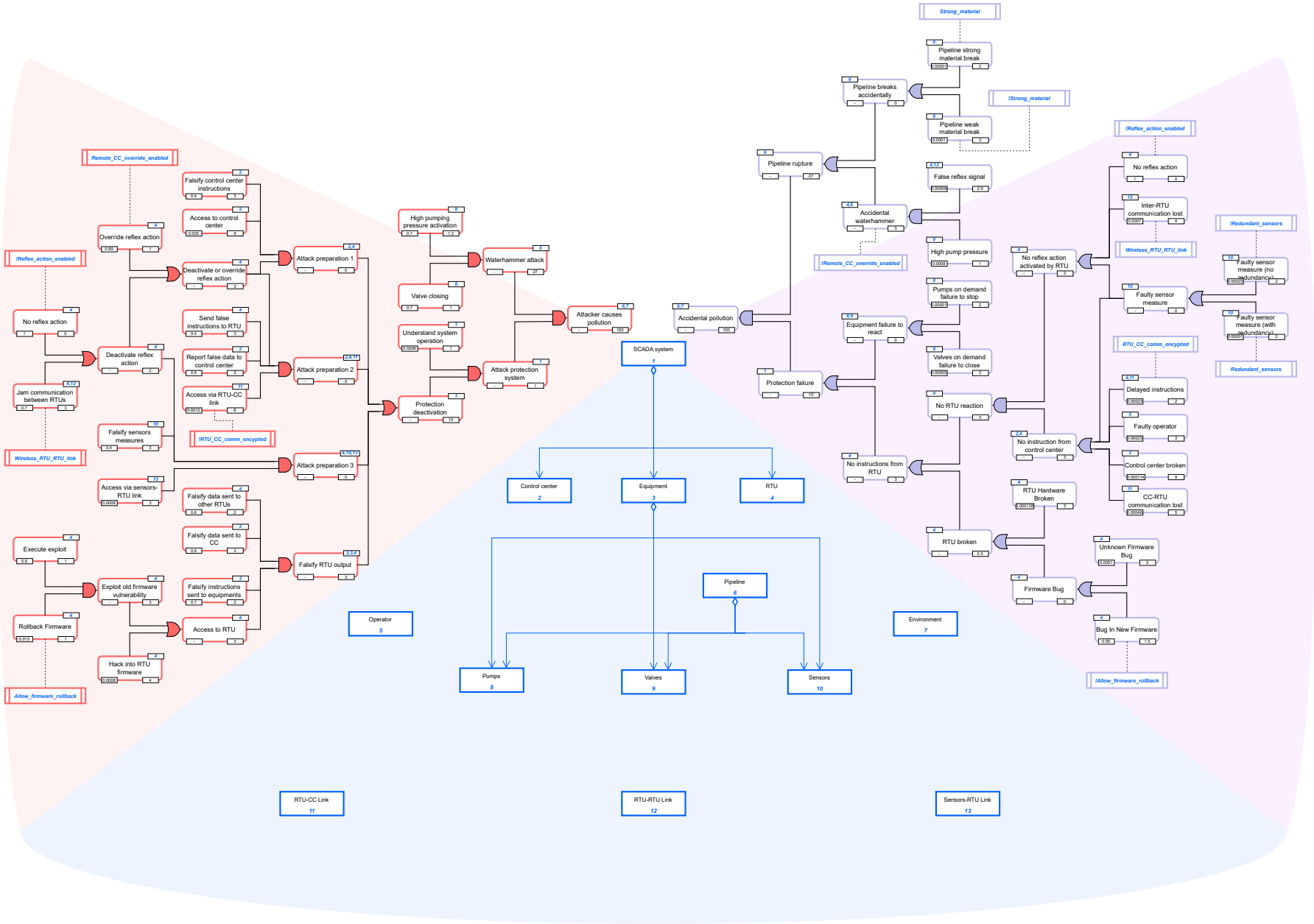


FIGURE 5.4: The refined DOG of the case study, incorporating additional object properties and impact values.

5.4 DOGLog Analysis

The DOG for this case study contains many objects. The *SCADA system* is set as the root object in the object graph because it has the most child objects. This does not necessarily mean it is the most important object in the system.

First, the risks associated with the *Pipeline* are investigated. The *Pipeline* is a crucial element of the system, as its failure could have catastrophic consequences. The maximum and minimum total risk for the *Pipeline* are calculated to see its risk range. This also shows to what extent different configurations can influence the amount of risk on the *Pipeline*.

```
--------------------------------------------------------------------------------
Processing Formula 1:
  MaxTotalRisk(Pipeline)
--------------------------------------------------------------------------------
  Maximum Total Risk: 13.236485099322643

--------------------------------------------------------------------------------
Processing Formula 2:
  MinTotalRisk(Pipeline)
--------------------------------------------------------------------------------
  Minimum Total Risk: 13.230548285373462
```

The analysis shows a maximum total risk of 13.2365 and a minimum total risk of 13.2305 for the *Pipeline*. The maximum risk is 0.04% larger than the minimum risk. This small difference means that the *Pipeline*'s risk is not very sensitive to the different configurations.

Next, another important element, the *Environment*, is investigated. In the DOG, this element is only attached to the most catastrophic events—the two top-level events. This set of events is a strict subset of the events attached to the *Pipeline*.

```
--------------------------------------------------------------------------------
Processing Formula 3:
  MaxTotalRisk(Environment)
--------------------------------------------------------------------------------
  Maximum Total Risk: 0.0034829395170428747

--------------------------------------------------------------------------------
Processing Formula 4:
  MinTotalRisk(Environment)
--------------------------------------------------------------------------------
  Minimum Total Risk: 0.0002561253929022441
```

The results for the *Environment* are quite different. The maximum total risk is 3.4829e-03, and the minimum total risk is 2.5613e-04. This means a suboptimal configuration can increase the risk to the *Environment* by a factor of 13.60. This is a large factor. However, in absolute terms, the difference between the maximum and minimum risk for the *Environment* is comparatively small. The set of events in which the *Environment* participates is a strict subset of the events in which the *Pipeline* participates. This implies that the absolute difference between the maximum and minimum total risk for the *Environment* must be less than or equal to that of the *Pipeline*.

It is unexpected that the *Environment* is so sensitive to different configurations, while the *Pipeline* is not. The cause of this difference should be found in one or more of the events where the *Pipeline* participates but the *Environment* does not. These include two subtrees: **Waterhammer attack** in the attack tree, and **Pipeline rupture** in the fault tree. The fault tree subtree, **Pipeline rupture**, should show some sensitivity to configurations. This is because it is influenced by two object properties: **Remote_CC_override_-**

enabled and **Strong_material**. The attack tree subtree, **Waterhammer attack**, on the other hand, is not influenced by any object properties. Therefore, it should be completely insensitive to different configurations.

An investigation is conducted to determine how much of the *Pipeline*'s risk can be attributed to the **Waterhammer attack** subtree. To do this, the probability of the **Waterhammer attack** node is calculated. This probability is then multiplied by its attached impact value of 27. Since this subtree is not influenced by any object properties, the empty configuration can be used for the probability calculation.

```
--------------------------------------------------------------------------------
Processing Formula 5:
  {}
  P(Waterhammer_attack) < 1
--------------------------------------------------------------------------------
    INFO: P(Waterhammer_attack) = 49/100 (~0.49)
  Result: True
```

The calculated probability is 0.49. Multiplying this by the impact value gives a risk of 13.23.

This risk represents a significant part of the *Pipeline*'s total risk. It is 99.9959% of the minimum total *Pipeline* risk and 99.9510% of the maximum total *Pipeline* risk. This shows that the large risk of the **Waterhammer attack** overshadows the risk of other events in which the *Pipeline* participates. This makes it seem as if the *Pipeline*'s risk exposure is not very sensitive to different configurations. In this specific example, these surprising results prompted a re-examination of the source BDMP. This re-examination revealed an oversight in our initial interpretation of the BDMP. The source BDMP correctly models that the protection system must be compromised before a **Waterhammer attack** can be successfully executed. This dependency is represented in the BDMP using a trigger pointing from "attack preparation" to "Waterhammer attack". Our initial DOG did not capture this. This oversight was not apparent until the DOGLog results were analyzed. DOGLog played a crucial role in uncovering this misinterpretation, which is a valuable insight.

Before continuing with the case study, we updated the DOG to accurately reflect this understanding of the BDMP. We expect this will lead to more realistic and interesting results. The correction involves making the existing **Attack protection system** node a child of the **Waterhammer attack** node. This change ensures that the DOG correctly models the prerequisite that the protection system must be compromised for the **Waterhammer attack** to occur. This aligns with the dependency modeled by a trigger in the source BDMP. With this change, the DOG is logically equivalent to a structure without the redundant direct link between **Attack protection system** and **Attacker causes pollution**. However, it was decided to retain **Attack protection system** as a direct child of **Attacker causes pollution** as well, to maintain a more familiar-looking tree structure. Figure 5.5 shows the DOG with this correction.

The risk of the *Pipeline* is calculated again, this time using the adapted DOG.

```
--------------------------------------------------------------------------------
Processing Formula 1:
  MaxTotalRisk(Pipeline)
--------------------------------------------------------------------------------
  Maximum Total Risk: 0.007421275290642874
```

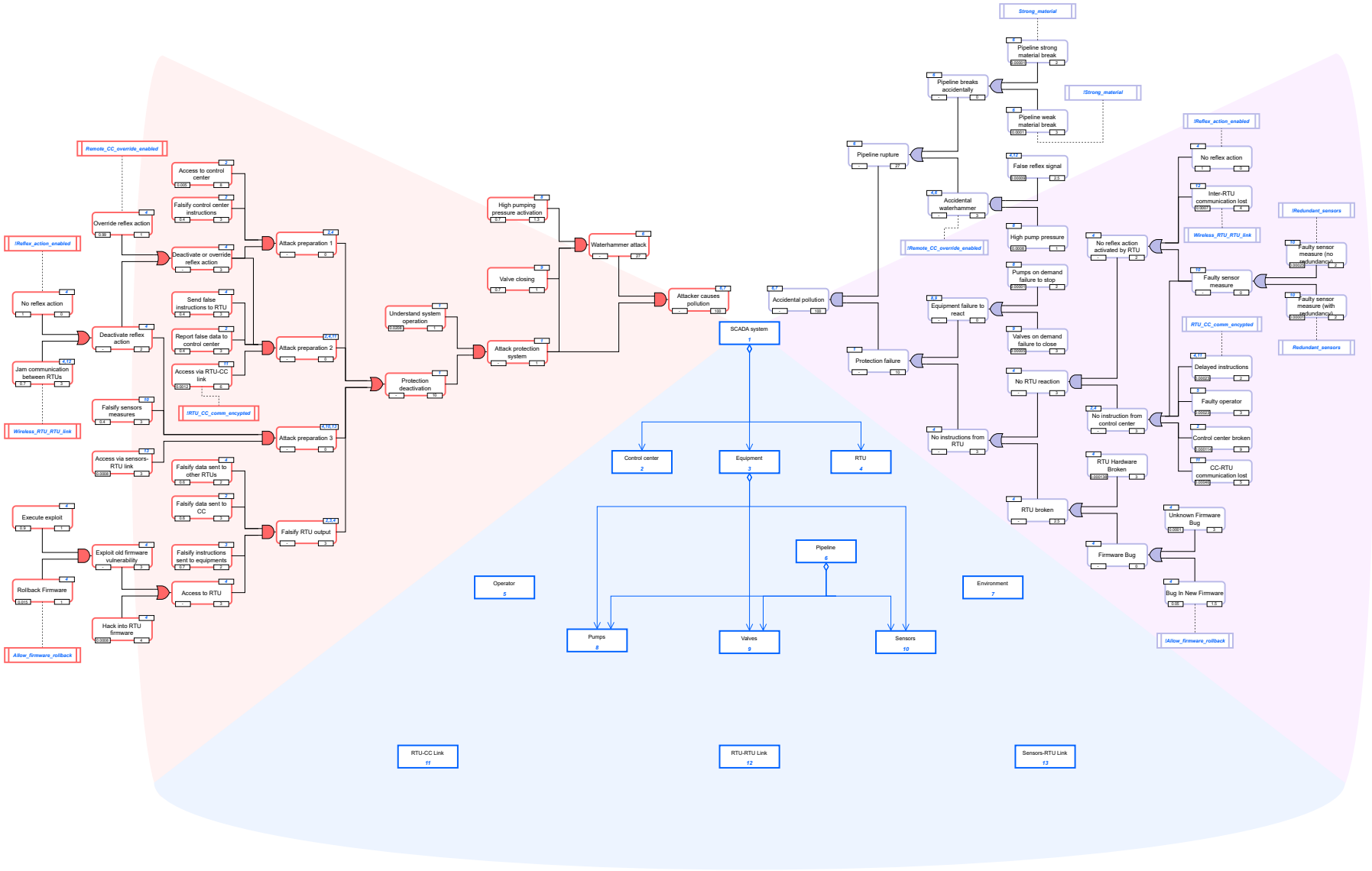


FIGURE 5.5: Corrected DOG where the **Attack protection system** node is a child of the **Waterhammer attack** node.

```
--------------------------------------------------------------------------------
Processing Formula 2:
  MinTotalRisk(Pipeline)
--------------------------------------------------------------------------------
  Minimum Total Risk: 0.0006037624678622442
```

The new maximum total risk is 7.4213e-03, and the new minimum total risk is 6.0376e-04. The maximum total risk is 1129.17% larger than the minimum total risk, a slightly smaller factor than we saw with the *Environment* before. This percentage seems more plausible than the previous 0.04%.

```
--------------------------------------------------------------------------------
Processing Formula 3:
  MaxTotalRisk(Environment)
--------------------------------------------------------------------------------
  Maximum Total Risk: 0.0034829395170428747

--------------------------------------------------------------------------------
Processing Formula 4:
  MinTotalRisk(Environment)
--------------------------------------------------------------------------------
  Minimum Total Risk: 0.0002561253929022441
```

The risk values for the *Environment* do not change. This might seem strange at first, since an extra condition was added to the **Waterhammer attack**. This addition should decrease its probability, thereby decreasing the probability of the top-level event **Attacker causes pollution**, and therefore the risk of the *Environment*. However, the added condition was already a condition for **Attacker causes pollution**, so its probability did in fact not change.

Let us now investigate how much of the *Pipeline*'s risk can be attributed to the **Waterhammer attack** subtree in the adapted DOG. To find this percentage, the previous method (calculating the probability using the empty configuration) cannot be used. This is because the **Waterhammer attack** is now influenced by object properties. Instead, the $\mathsf{TotalRisk}_{\max}$ and $\mathsf{TotalRisk}_{\min}$ functions are used to find the risk of the **Waterhammer attack**. However, these functions only take objects as parameters, so they cannot be used directly. A new pseudo-object can be created with an arbitrary, unique name. This pseudo-object is then attached only to the **Waterhammer attack** event. We call this pseudo-object *WAO* (Waterhammer Attack Object).

```
--------------------------------------------------------------------------------
Processing Formula 6:
  MaxTotalRisk(WAO)
--------------------------------------------------------------------------------
  Maximum Total Risk: 0.000936175968

--------------------------------------------------------------------------------
Processing Formula 7:
  MinTotalRisk(WAO)
--------------------------------------------------------------------------------
  Minimum Total Risk: 5.54770944e-05
```

In the maximum risk scenario, the **Waterhammer attack**'s risk is 12.61% of the *Pipeline*'s risk. In the minimum risk scenario, it is 9.19% of the *Pipeline*'s risk. These percentages seem more representative of the intended scenario than the previous 99.9510% and 99.9959% values.

Now, let us examine the fault tree. The fault tree has some unique properties. At two points in the tree, an object property influences the impact or probability of an event. This is not natively supported in DOGLog. However, it can be achieved by creating an intermediate node with an `OR` gate. This gate has two children with different impact or probability values. One child has a condition that is true if the object property is true, and the other has a condition that is true if the object property is false. For **Faulty sensor measure**, the **Redundant_sensors** object property is used to change the probability: if there are multiple redundant sensors, it reduces the probability of a **Faulty sensor measure**. Similarly, **Pipeline breaks accidentally** is influenced by the **Strong_material** object property. Using a stronger material reduces the probability of breakage. It also slightly reduces the impact, since the break would presumably be smaller.

All other object properties that appear in the fault tree also appear in the attack tree. Some of these properties are in *synergy* (**Reflex_action_enabled**, **Wireless_RTU_RTU_link**). Others are in *conflict* (**Remote_CC_override_enabled**, **RTU_CC_comm_encrypted**, **Allow_firmware_rollback**). The optimal configuration is examined to see what can be concluded about the conflicting properties. Optimal configurations are specific to individual objects. So, the optimal configurations for some of the most important objects are examined: *Environment*, *Pipeline*, *SCADA system*, and *RTU*.

```
--------------------------------------------------------------------------------
Processing Formula 8:
  OptimalConf(Environment)
--------------------------------------------------------------------------------
    INFO: There is one optimal configuration with a risk value of
        0.0002561253929022441
  Optimal Configurations:
    - {Reflex_action_enabled: True, Strong_material: True,
        Remote_CC_override_enabled: False, Redundant_sensors: True,
        Allow_firmware_rollback: False, Wireless_RTU_RTU_link: False}

--------------------------------------------------------------------------------
Processing Formula 9:
  OptimalConf(Pipeline)
--------------------------------------------------------------------------------
    INFO: There is one optimal configuration with a risk value of
        0.0006037624678622442
  Optimal Configurations:
    - {Reflex_action_enabled: True, Strong_material: True,
        Remote_CC_override_enabled: False, Redundant_sensors: True,
        Allow_firmware_rollback: False, Wireless_RTU_RTU_link: False}

--------------------------------------------------------------------------------
Processing Formula 10:
  OptimalConf(SCADA_system)
--------------------------------------------------------------------------------
    INFO: There is one optimal configuration with a risk value of
        0.057970446012327835
  Optimal Configurations:
    - {Redundant_sensors: True, Allow_firmware_rollback: True,
        Reflex_action_enabled: True, Wireless_RTU_RTU_link: False}
```

```
------------------------------------------------------------------------------
Processing Formula 11:
  OptimalConf(RTU)
------------------------------------------------------------------------------
    INFO: There is one optimal configuration with a risk value of
        3.4721460015144587
  Optimal Configurations:
    - {Reflex_action_enabled: True, Wireless_RTU_RTU_link: False,
        Remote_CC_override_enabled: False, RTU_CC_comm_encrypted: False,
        Redundant_sensors: True, Allow_firmware_rollback: True}
```

There is one conflicting property between the configurations of *Environment* with *Pipeline*, and *SCADA system* with *RTU*. The conflicting property is **Allow_firmware_rollback**. It is set to false for the *Environment* and *Pipeline*, and true for the *SCADA system* and *RTU*. Since the *Environment* and *Pipeline* might be considered more important to protect, one might be inclined to set this property to false. On the other hand, the combined risk of the *SCADA system* and *RTU* is higher than that of the *Environment* and *Pipeline*. This suggests that more attention should be paid to the *SCADA system* and *RTU*. To make a more informed decision, the influence of this property on the risk of these objects is investigated. The optimal configuration for these objects is calculated again, but this time evidence is set for the **Allow_firmware_rollback** property to the opposite value.

```
------------------------------------------------------------------------------
Processing Formula 12:
  (OptimalConf(Environment) [Allow_firmware_rollback: 1])
------------------------------------------------------------------------------
    INFO: There is one optimal configuration with a risk value of
        0.003467626368441233
  Optimal Configurations:
    - {Remote_CC_override_enabled: True, Redundant_sensors: True,
        Reflex_action_enabled: True, Wireless_RTU_RTU_link: False,
        Strong_material: True}

------------------------------------------------------------------------------
Processing Formula 13:
  (OptimalConf(Pipeline) [Allow_firmware_rollback: 1])
------------------------------------------------------------------------------
    INFO: There is one optimal configuration with a risk value of
        0.004693802336441233
  Optimal Configurations:
    - {Remote_CC_override_enabled: True, Redundant_sensors: True,
        Reflex_action_enabled: True, Wireless_RTU_RTU_link: False,
        Strong_material: True}

------------------------------------------------------------------------------
Processing Formula 14:
  (OptimalConf(SCADA_system) [Allow_firmware_rollback: 0])
------------------------------------------------------------------------------
    INFO: There is one optimal configuration with a risk value of
        0.5257458934717115
  Optimal Configurations:
    - {Redundant_sensors: True, Reflex_action_enabled: True,
        Wireless_RTU_RTU_link: False, Remote_CC_override_enabled: False}
```

```
--------------------------------------------------------------------------------
Processing Formula 15:
  (OptimalConf(RTU) [Allow_firmware_rollback: 0])
--------------------------------------------------------------------------------
    WARNING: Evidence {'Allow_firmware_rollback': False} made node '
        Exploit_old_firmware_vulnerability' unsatisfiable.
    WARNING: Evidence {'Allow_firmware_rollback': False} made node '
        Rollback_firmware' unsatisfiable.
    INFO: There is one optimal configuration with a risk value of
        4.186653303125821
  Optimal Configurations:
    - {Reflex_action_enabled: True, Wireless_RTU_RTU_link: False,
        Remote_CC_override_enabled: False, RTU_CC_comm_encrypted: False,
        Redundant_sensors: True}
```

These results show that the risk increases more for the *Environment* and *Pipeline* than for the *SCADA system* and *RTU*. The risks of the *Environment* and *Pipeline* increase by 1253.88% and 677.43%, while the risks of the *SCADA system* and *RTU* increase by 806.92% and 20.58%. These larger relative increases, along with the opinion that the *Environment* and *Pipeline* are more important to protect, lead to the conclusion that the **Allow_firmware_rollback** property should be set to false.

As mentioned before, three object properties are in conflict between the attack tree and fault tree. It seems that the previous optimal configuration results already decided which option to pick for these properties. However, directly comparing risk values between attack and fault trees can be difficult or misleading. This is because of their different natures: the attacker makes choices, while the fault tree involves purely random events. Therefore, it is a good idea to investigate the influence of these properties on the risk of both trees individually. This can be done by calculating the optimal configuration for the attack tree and fault tree separately. Again, pseudo-objects are used to achieve this.[3]

```
--------------------------------------------------------------------------------
Processing Formula 16:
  OptimalConf(ACPO)
--------------------------------------------------------------------------------
    INFO: There is one optimal configuration with a risk value of
        0.00020547072
  Optimal Configurations:
    - {Allow_firmware_rollback: False, Reflex_action_enabled: True,
        Remote_CC_override_enabled: False, Wireless_RTU_RTU_link: False}


--------------------------------------------------------------------------------
Processing Formula 17:
  OptimalConf(APO)
--------------------------------------------------------------------------------
    INFO: There is one optimal configuration with a risk value of
        3.0796844123278364e-07
  Optimal Configurations:
    - {Reflex_action_enabled: True, Strong_material: True,
        Remote_CC_override_enabled: True, Redundant_sensors: True,
        Allow_firmware_rollback: True, Wireless_RTU_RTU_link: False}
```

Out of the three conflicting properties, one property, **RTU_CC_comm_encrypted**, is found to be not conflicting in practice; it is not present in either of the optimal configurations! The reason for this is different for the two trees. Regarding the attack tree, the reason is similar to what was seen before: the attack it influences, **Attack preparation**

[3] We create *ACPO* and *APO* for the top-level events of the attack tree and fault tree, respectively.

2, is not the most optimal attack. For the fault tree, however, the reason is different, as fault trees are not based on choices but on random events. Instead, we need to look at the subtree rooted at **No RTU reaction** more closely to find out why this property is not present in the optimal configuration. **No RTU reaction** is an AND gate with two children. **No reflex action activated by RTU** has three children, two of which are disabled by the optimal configuration: **No reflex action** and **Inter-RTU communication lost**. Therefore, for **No RTU reaction** to be true, **Faulty sensor measure** must be true. Since **Faulty sensor measure** is also a child of **No instruction from control center**, this means that all other children of **No instruction from control center** are irrelevant outside of the **No instruction from control center** subtree. This explains why **RTU_CC_comm_encrypted** is not present in the optimal configuration for the fault tree, but is present in the optimal configuration for *RTU*, since *RTU* participates in both **No instruction from control center** and **Delayed instructions**. Consequently, following the optimal configuration of *RTU*, **RTU_CC_comm_encrypted** should definitely be set to false.

For each of the remaining properties, their influence on the risk of the attack tree and fault tree is investigated. This is done similarly to before, by setting evidence for the **Remote_CC_override_enabled** and **Allow_firmware_rollback** properties to the opposite value. First for **Remote_CC_override_enabled**:

```
--------------------------------------------------------------------------------
Processing Formula 18:
  (OptimalConf(ACPO) [Remote_CC_override_enabled: 1])
--------------------------------------------------------------------------------
    INFO: There is one optimal configuration with a risk value of
        0.002018016
  Optimal Configurations:
    - {Allow_firmware_rollback: False, Reflex_action_enabled: True}

--------------------------------------------------------------------------------
Processing Formula 19:
  (OptimalConf(APO) [Remote_CC_override_enabled: 0])
--------------------------------------------------------------------------------
    INFO: There is one optimal configuration with a risk value of
        3.101857918359319e-07
  Optimal Configurations:
    - {Redundant_sensors: True, Allow_firmware_rollback: True,
        Reflex_action_enabled: True, Wireless_RTU_RTU_link: False,
        Strong_material: True}
```

The risk of the attack tree increases by 882.14%, and the risk of the fault tree increases by 0.72%. Clearly, with an increase factor of almost 10, the attack tree is much more sensitive to this property than the fault tree, suggesting that we should follow the optimal configuration for the attack tree where this property is set to false. Fortunately, this also aligns with the optimal configurations for *Environment*, *Pipeline*, and *RTU*, which all have this property set to false.

Now, the same is done for **Allow_firmware_rollback**. Unlike **Remote_CC_override_enabled**, this property was also a conflicting property in the optimal configurations between *Environment* and *Pipeline*, and *SCADA system* and *RTU*:

```
--------------------------------------------------------------------------------
Processing Formula 20:
  (OptimalConf(ACPO) [Allow_firmware_rollback: 1])
--------------------------------------------------------------------------------
    INFO: There is one optimal configuration with a risk value of
        0.0034673184
  Optimal Configurations:
    - {}

--------------------------------------------------------------------------------
Processing Formula 21:
  (OptimalConf(APO) [Allow_firmware_rollback: 0])
--------------------------------------------------------------------------------
    INFO: There is one optimal configuration with a risk value of
        5.0292570019171146e-05
  Optimal Configurations:
    - {Remote_CC_override_enabled: True, Redundant_sensors: True,
        Reflex_action_enabled: True, Wireless_RTU_RTU_link: False,
        Strong_material: True}
```

The risk of the attack tree increases by 1588%. The risk of the fault tree increases by 16230%. This time, the fault tree is much more sensitive to this property than the attack tree, by more than a factor of 10. This would side with the optimal configurations for *SCADA system* and *RTU*, which both have this property set to true.

Ultimately, the results show that outcomes can vary drastically depending on which objects or which tree is being examined. This case study illustrates the utility of DOGLog in navigating such complexities. By facilitating the analysis of system risks considering both safety and security aspects, DOGLog helped identify non-obvious interactions. For example, the *Pipeline*'s initial risk insensitivity resulted from the dominant **Waterhammer attack**. Adjusting the model based on this insight yielded more plausible results. The investigation of object properties revealed synergies and conflicts. Clear optimal settings were found for non-conflicting properties. **Strong_material**, **Reflex_action_enabled**, and **Redundant_sensors** should be set to true. **Wireless_RTU_RTU_link** should be set to false. While appearing conflicting at first sight, **RTU_CC_comm_encrypted** was also found non-conflicting in practice and should be set to false. Determining settings for the conflicting properties, **Allow_firmware_rollback** and **Remote_CC_override_enabled**, yielded interesting results. For **Remote_CC_override_enabled**, we found that the attack tree is much more sensitive to this property than the fault tree, suggesting that we should follow the optimal configuration for the attack tree where this property is set to false. This also happened to align with the optimal configurations for objects *Environment*, *Pipeline*, and *RTU*. The analysis of **Allow_firmware_rollback** gave conflicting results. Analysis prioritizing key objects suggested false, while analysis comparing attack and fault tree sensitivity suggested true might be better. These conflicting findings highlight that a single optimal configuration may not exist. The best choice depends on the analysis perspective and specific risk priorities. WATCHDOG provides the framework to explore these trade-offs and support informed decision-making.

Chapter 6

Discussion and Future Work

This chapter addresses the third research question: *What are the current limitations of the WATCHDOG framework and its associated logic DOGLog, and what are the most promising opportunities for future enhancements to improve its expressiveness and usability?* It reflects on the findings of the case study, discusses the limitations of the current WATCHDOG framework and DOGLog, and proposes several directions for future work.

The case study highlighted the role of DOGLog in model validation and refinement. Analysis with DOGLog initially produced unexpected results concerning the risk sensitivity of the *Pipeline* object. Specifically, the risk associated with the *Pipeline* seemed surprisingly insensitive to different configurations. This finding prompted a re-examination of both the source BDMP model and our DOG translation, which led to the discovery of a misinterpretation of a dependency related to the **Waterhammer attack**. The original BDMP correctly modeled that the protection system must be compromised before a **Waterhammer attack** could be executed. This dependency was initially missed in our DOG model. This experience underscores an iterative modeling process: an initial model is constructed, analysis reveals surprising insights, which leads to model correction, followed by re-analysis. The use of DOGLog was instrumental in uncovering this subtle oversight. Such errors might be easily missed when using a more traditional approach, where the model is analyzed in its entirety without zooming in on smaller parts of the model. In this way, formal analysis with DOGLog can act as a valuable validation step, enhancing the accuracy and reliability of the risk model.

6.1 WATCHDOG Limitations and Possible Extensions

Object-oriented DisruptiOn Graphs (DOGs) are, in their current form, static models. This means they cannot represent the temporal aspects of system behavior. The case study highlighted that translating dynamic models, such as BDMP, to DOGs can result in a loss of temporal information. Over time, researchers have proposed various methods to extend traditional fault and attack trees to incorporate a sense of time [29, 39]. Furthermore, numerous formalisms beyond fault and attack trees are utilized for risk analysis. These include BDMP [5] and Probabilistic Event Graphs [6], among others, with a recent survey by Nicoletti et al. [29] compiling several such approaches. Given that DOGs represent a new formalism, their initial focus on static, more simple, modeling is understandable.

The WATCHDOG framework introduces novel contributions, particularly in its object-oriented approach to risk assessment, grounded in ontological principles from the COVER framework [45]. The authors of WATCHDOG state that DOGs are the first formalism to explicitly account for objects participating in events and to aggregate the risk of those

events for different objects [31]. Key distinguishing features of the WATCHDOG framework, enabled by DOGs and the associated DOGLog logic and DOGLang query language, include:

- The ability to model diverse objects that are at risk within the system.
- The ability to model how these objects participate in various events and attacks.
- The functionality to aggregate the risk posed by events and attacks specifically for different objects.
- The capability to model the *parthood* relation, where one object is a part of another.

WATCHDOG defines risk as the product of impact and probability. Impact values are assigned to all disruption tree nodes. Probabilities, on the other hand, are explicitly assigned only to basic nodes; for all other nodes, probabilities are calculated based on the probabilities of their child events and the logical gates connecting them. Essentially, WATCHDOG requires that events in which objects participate have probabilities and impacts. The concept of using probabilities and impacts for events is also present, or can be readily incorporated, in several other risk analysis formalisms. Examples include Failure-Attack-CounTermeasure FACT graphs [44] and Attack Tree Bow-ties [1, 32], as noted in the survey by Nicoletti et al. [29]. Consequently, the object-oriented modeling additions introduced by WATCHDOG could potentially be applied to these other formalisms, provided they meet the necessary foundational requirements for WATCHDOG.

More broadly, the object-oriented reasoning capabilities of WATCHDOG might be adaptable even to formalisms where not every event or attack has a direct probability value. For instance, BDMP utilize rates of occurrence for some events instead of direct probabilities. While the direct translation of "risk" in the WATCHDOG sense requires further investigation in such contexts, it could be fruitful to explore object-oriented queries in these formalisms. One might investigate questions such as "what is the maximal rate of occurrence of an event affecting object o, given certain object properties?" Similarly, one could explore "what is the optimal configuration of object properties to minimize the rate of occurrence of events impacting object o?" Such explorations could pave the way for integrating WATCHDOG's object-oriented paradigm with a wider array of existing risk analysis techniques, thereby enhancing their expressiveness and analytical depth. Future research could focus on developing formal semantics to bridge WATCHDOG with these diverse formalisms.

6.2 Potential Enhancements to DOGLog and WATCHDOG

This section discusses several potential enhancements to DOGLog and the WATCHDOG framework. These were identified during the course of this research and the case study analysis. These suggestions, offered as potential directions for future work, aim to improve the expressiveness, usability, and modeling capabilities of WATCHDOG and DOGLog.

6.2.1 Nodes as Parameters for Layer 3 Functions

During the case study, a limitation in the current DOGLog logic became apparent. Specifically, layer 3 functions, such as $\underset{\text{min}}{\mathsf{TotalRisk}}$, $\mathsf{MostRisky}_*$, and $\mathsf{OptimalConf}$, are defined to accept only objects as parameters. In our analysis, we encountered a scenario where we needed to determine the risk contribution of a specific attack tree node: **Waterhammer attack**. To work around this limitation, we introduced a pseudo-object. This pseudo-object, named *WAO* (Waterhammer Attack Object), exclusively participated in the **Wa-**

terhammer attack node. By analyzing this pseudo-object, we could indirectly assess the risk of the targeted attack node. This approach works because layer 3 functions operate on the set of disruption tree elements in which an object participates.

However, the necessity of creating pseudo-objects suggests a potential area for improvement in DOGLog. The restriction of layer 3 functions to object parameters appears somewhat arbitrary. Relaxing this limitation to allow any node from a disruption tree as a parameter to layer 3 functions would be beneficial. Such an extension would offer greater flexibility in risk analysis. It would also eliminate the need for ad-hoc solutions like pseudo-objects, leading to more direct and intuitive queries.

6.2.2 Direct Probability Calculation and Comparison

The current DOGLog layer 2 syntax focuses on comparing the probability of a layer 1 formula ϕ against a constant threshold. For example, one can query if $\mathrm{P}(\phi) < 0.5$. While this is useful, determining the exact probability value of ϕ is less direct. Theoretically, one could approximate the value by repeatedly querying against different thresholds. This could perhaps employ a binary search strategy. However, this method is cumbersome and inefficient for the user. Our current implementation mitigates this by logging the computed probability value to the terminal whenever a probability comparison is evaluated. This provides the user with the exact value. Nevertheless, the language itself still requires framing the query as a comparison, even if the comparison aspect is not the primary interest. A more straightforward approach would be to allow the term $\mathrm{P}(\phi)$ to stand alone in the grammar. When used this way, it would directly yield the probability value of ϕ, similar to how layer 3 functions return specific risk values. This would enhance the usability of layer 2 for direct probability queries. It is important to note that such a standalone $\mathrm{P}(\phi)$ expression would result in a numeric value, not a Boolean. Consequently, it could not be combined using logical operators (e.g., $\wedge, \vee, \neg$) in the same way that probability comparisons like $\mathrm{P}(\phi) < 0.5$ can.

Furthermore, the current syntax does not support direct comparison of probabilities of two different formulae. For instance, it is not possible to express a query like "is the probability of ϕ_1 greater than the probability of ϕ_2?". Extending layer 2 to allow expressions such as $\mathrm{P}(\phi_1) > \mathrm{P}(\phi_2)$ would enable more complex and nuanced probabilistic analyses. This would allow for direct ranking of event likelihoods within the logic itself.

6.2.3 Support for Variables and Arithmetic Operations

Building upon the idea of directly calculating probability values, DOGLog could be further enhanced by incorporating variables and arithmetic operations. Currently, values computed by DOGLog functions are only presented as final outputs. Examples include risk values from layer 3, or potential probability values from an extended layer 2. There is no mechanism within DOGLog to store these values or perform further calculations on them.

We propose extending DOGLog to support:

- Variable assignment: Allowing users to assign the result of a DOGLog function (e.g., a standalone $\mathrm{P}(\phi)$ or $\underset{\text{min}}{\mathsf{TotalRisk}}\ (\mathrm{o})$) to a variable.
- Arithmetic operations: Enabling standard arithmetic operations on these variables and numeric literals. These include addition, subtraction, multiplication, and division.

- Comparisons: Allowing comparisons between variables and numeric literals. These include (in)equality, less than (or equal to), greater than (or equal to).

Such features would significantly increase the expressive power of DOGLog. In the case study, for instance, we frequently performed calculations outside the DOGLog environment, e.g., comparing different risk values. We also calculated relative increases in risk under different configurations, e.g., the 1129.17% increase for the risk of *Pipeline* between the maximal and minimal risk scenarios. Integrating these capabilities into DOGLog itself would allow for more complex and self-contained analyses. This would involve introducing new statement types for variable assignment. It would also require expressions for arithmetic and comparison, akin to those found in general-purpose programming languages. Additionally, a dedicated 'print' statement would be valuable. This would allow users to output the values of variables or intermediate calculations directly from their DOGLog queries.

6.2.4 Modeling Conditional Probabilities and Impacts in DOGs

The case study highlighted a new modeling requirement. This is the need to adjust the probability or impact of a disruption tree node based on the state of object properties. For example, the probability of a **Faulty sensor measure** might decrease if **Redundant_sensors** are installed. Similarly, the impact of a **Pipeline rupture** might be lower if the pipeline is constructed using a **Strong_material**.

Our current approach to model such conditionality within the DOG formalism involves a workaround. We create two (or more) sibling nodes in the disruption tree. Each sibling represents a different probability or impact value. These sibling nodes are then guarded by mutually exclusive conditions based on object properties. For instance, one node **Pipeline strong material break** is active if **Strong_material** is true. Another node **Pipeline weak material break** is active if ¬**Strong_material** is true. While this workaround achieves the desired effect, it leads to a more verbose DOG structure.

A more elegant solution would be to extend the WATCHDOG framework to directly support conditional probability and impact values for nodes. This means the probability or impact of a node would not be a static value. Instead, it could be determined by a function that maps combinations of object properties to a specific value. This extension would require only slight changes to the semantics of WATCHDOG. Primarily, the probability attribution function for basic nodes, $\alpha\colon \texttt{BAS} \cup \texttt{BE} \mapsto [0,1]$, and the impact attribution function, $Im\colon N_A \cup N_F \mapsto \mathbb{R}_{\geq 0}$, would need to be redefined. Their new signatures would incorporate object configurations: $\alpha'\colon (\texttt{BAS} \cup \texttt{BE}) \times \mathcal{C} \mapsto [0,1]$ and $Im'\colon (N_A \cup N_F) \times \mathcal{C} \mapsto \mathbb{R}_{\geq 0}$. Here, $\mathcal{C}$ represents the set of all possible configurations of object properties. This change integrates well with the existing semantics. The functions α and Im are only used in contexts where a specific configuration $\vec{b_O}$ is already available, namely in the calculation of $\rho(\phi, \vec{b_O})_{A,F}$ for layer 2, and $\mathsf{objRiskVal}(o, \vec{b_O})$ and $\mathsf{MostRisky}_*(o)$ for layer 3. Thus, making these attributions dependent on $\vec{b_O}$ is a natural extension.

To incorporate this into the DOG definition language, several approaches can be considered. A straightforward method is to introduce support for a ternary operator. This operator would be used directly within the probability and impact attributes of nodes. The syntax for such a solution, using *val* for a numeric value expression and *cond* for a

condition expression over object properties, could be:

$$\begin{aligned} val &::= c \mid cond\,?\,val : val \\ cond &::= op \mid \neg cond \mid cond = cond \mid cond \neq cond \mid cond \wedge cond \mid cond \vee cond \\ &\quad \mid cond \implies cond \mid (cond) \end{aligned}$$

Here, c represents a constant numeric value (e.g., a probability or impact). *op* refers to an object property (atomic proposition). The *cond* expression reuses the familiar Boolean operators for combining object properties, as already used for node conditions in DOGs.

Another, more powerful, option could be to allow the definition of small functions. These functions would take object properties as input and return the corresponding probability or impact value. This approach, however, introduces complexities. These complexities relate to the types of statements allowed within such functions (e.g., if-statements, loops). This could potentially build upon the previously discussed extensions for variables and arithmetic in DOGLog. For the immediate use case of conditional probabilities and impacts as seen in the case study, the ternary operator approach is likely sufficient, and is simpler to implement. It would offer a concise way to express conditional values directly in the DOG.

6.3 Correctness and Verification of ODF

The correctness of ODF is critical for reliable risk assessment using DOGLog. The implementation is validated through an extensive automated test suite, as detailed in Section 4.2.4. For layer 2 probability calculations, our approach combines the proven $\texttt{BDD}_{\texttt{DAG}}$ and $\texttt{p}_{\texttt{TLE}}$ algorithms. The manual calculation example in Section 4.1.2.1 demonstrates exact agreement between the BDD-based computation and exhaustive enumeration, providing additional confidence in this component. Together with the test suite, this provides a degree of confidence in the tool's correctness. However, it is important to acknowledge the inherent limitations of this approach.

Testing can only verify behavior for the specific inputs and scenarios included in the test suite. While these tests can find implementation bugs, they cannot prove their absence. This limitation is particularly significant for risk assessment tools, where incorrect results could lead to poor safety and security decisions.

For applications in safety-critical domains, formal verification would provide stronger correctness guarantees. Such verification could involve:

1. **Formal specification.** Expressing DOGLog semantics in a theorem prover such as Coq, Isabelle/HOL, or Lean.
2. **Verified algorithms.** Developing the algorithms with machine-checked proofs of correctness with respect to the formal specification.
3. **Equivalence proofs.** Demonstrating that the implementation faithfully realizes the intended semantics through formal proofs.

However, such formal verification is a substantial research undertaking that extends well beyond the scope of this thesis. The development of verified implementations typically requires significant additional effort and specialized expertise in formal methods.

The current ODF implementation provides a solid foundation for DOGLog analysis. It has been validated through extensive testing and demonstrated with a practical case study. For critical applications, users should understand the limitations of this validation and consider additional verification if deemed necessary. Future work could explore formal

verification of the algorithms developed in this thesis and their implementation in ODF. This would provide stronger correctness guarantees.

6.4 Performance Evaluation

The performance of ODF is currently uncharacterized. The case study was conducted on a small model, where individual queries were evaluated in under 200 milliseconds. However, it remains unclear how ODF will scale with larger models and more complex queries. Performance testing and optimization are therefore important directions for future work.

A standard approach to performance evaluation is to use a benchmark suite. While benchmark sets exist for traditional fault and attack trees [40], none are available for the new DOG formalism. A practical alternative is to generate a collection of synthetic DOG models with varying size and complexity.

A potential tool for this task is Grammarinator [16], which generates inputs based on a context-free grammar. Grammarinator is designed for fuzz testing but offers features that are useful for generating valid DOGs. It allows weighting grammar rules to control the distribution of generated structures. It also supports constraints and the reuse of generated elements. These features are necessary to ensure the generated DOGs are well-formed. For example, an event can only be attached to an object that has been defined. Similarly, a condition on an event can only reference object properties of objects that participate in that event. A purely random generator would not be able to satisfy such constraints.

A key challenge is controlling the structural properties of the generated models. The performance of query evaluation may be influenced by several factors. These include the number of shared subtrees, the distribution of `AND` to `OR` gates, the number of events an object participates in, the complexity of conditions, and many others. Indeed, related research on evaluating Boolean Fault tree Logic (BFL) formulae on fault trees suggests that the structure of the fault tree has a greater impact on performance than its size [43]. Therefore, a benchmark suite should contain a wide variety of DOGs with different structural characteristics.

It is uncertain whether Grammarinator provides sufficient control to generate such a diverse set of models. If it proves too restrictive, a custom model generator may need to be developed. This would ensure that the performance of ODF can be evaluated across a comprehensive range of DOGs.

Chapter 7

Conclusion

This thesis presented the design and implementation of a model checker for DOGLog, the custom logic of the WATCHDOG framework for object-oriented risk assessment. The work addressed the practical challenge of analyzing object-oriented DisruptiOn Graphs (DOGs) through the development of a complete toolchain that translates DOGLog queries into Binary Decision Diagrams (BDDs) for efficient verification.

7.1 Answers to Research Questions

RQ1: How can model checking algorithms be designed and implemented for the three layers of DOGLog? We developed model checking algorithms for DOGLog's three-layer logic. BDDs are at the core of our approach. DOGLog formulae are translated into BDD representations, before the algorithms we developed use them for analysis. Specifically, these BDDs enable satisfiability checking and the computation of minimal risk scenarios for layer 1 formulae. For layer 2, the algorithms use BDDs to calculate the probability of formulae. In layer 3, BDDs are used for individual element analysis such as identifying the most risky events or actions for an object, while Multi-Terminal BDDs (MTBDDs) are used to determine total risk exposure and optimal configurations by aggregating risk contributions from all nodes in which an object participates. Furthermore, we implemented a tool named ODF that includes a parser for a custom domain-specific language to define DOGs and to write DOGLog formulae, as well as the implementation of a model checker that processes DOG models and evaluates DOGLog queries. The implementation supports the full DOGLog syntax, enabling the modeling and analysis of object participation and parthood relations, probability calculations, and risk aggregation functions. The tool provides a command-line interface for practical use.

RQ2: How can DOGLog be used for performing risk analysis on a complex system, and what conclusions can be drawn from the application of DOGLog to such a system? Through a comprehensive case study based on a cyber-physical pipeline system, we demonstrated the practical utility of our implementation. The object-oriented approach of WATCHDOG, supported by this implementation, enables focused analysis on specific system components. This allows analysts to examine risk from multiple perspectives within the same model. This capability proved valuable for understanding how different objects are affected by various attack and failure scenarios. The case study demonstrated how risk modeling can be an iterative process, where formal analysis helps refine the model. Formal analysis with DOGLog can uncover subtle modeling errors that might be missed in traditional approaches. Specifically, unexpected results concerning

pipeline risk sensitivity in our analysis helped identify a misinterpretation of dependencies in the original BDMP model. This highlighted how DOGLog analysis can also serve as a validation step for risk models. Specific conclusions from the case study included the identification of optimal system configurations to minimize risk, the quantification of risk levels for critical objects, and the uncovering of non-obvious interactions and dependencies between safety and security measures.

RQ3: What are the current limitations of the WATCHDOG framework and its associated logic DOGLog, and what are the most promising opportunities for future enhancements to improve its expressiveness and usability? As a result of the case study, we identified and discussed several potential enhancements to the WATCHDOG framework. These include support for nodes as parameters in layer 3 functions, direct probability calculation queries, variables and arithmetic operations, and conditional probabilities and impacts based on object configurations. Further research could focus on extending the framework to handle dynamic models for systems where timing is critical, as DOGs are currently static. Additionally, the object-oriented paradigm of WATCHDOG could potentially be applied to other risk analysis formalisms beyond fault and attack trees. Future work could explore formal semantics to bridge WATCHDOG with diverse existing approaches.

Bibliography

[1] H. Abdo et al. "A Safety/Security Risk Analysis Approach of Industrial Control Systems: A Cyber Bowtie – Combining New Version of Attack Tree with Bowtie Analysis". In: *Computers & Security* 72 (Jan. 1, 2018), pp. 175–195. ISSN: 0167-4048. DOI: 10.1016/j.cose.2017.09.004. URL: https://www.sciencedirect.com/science/article/pii/S0167404817301931 (visited on 05/23/2025).

[2] Akers. "Binary Decision Diagrams". In: *IEEE Transactions on Computers* C-27.6 (June 1978), pp. 509–516. ISSN: 1557-9956. DOI: 10.1109/TC.1978.1675141. URL: https://ieeexplore.ieee.org/document/1675141 (visited on 12/12/2024).

[3] Israel Barragan santiago and Jean-Marc Faure. "FROM FAULT TREE ANALYSIS TO MODEL CHECKING OF LOGIC CONTROLLERS". In: *IFAC Proceedings Volumes*. 16th IFAC World Congress 38.1 (Jan. 1, 2005), pp. 86–91. ISSN: 1474-6670. DOI: 10.3182/20050703-6-CZ-1902.01439. URL: https://www.sciencedirect.com/science/article/pii/S1474667016374511 (visited on 01/24/2025).

[4] Hichem Boudali, Pepijn Crouzen, and Marielle Stoelinga. "Dynamic Fault Tree Analysis Using Input/Output Interactive Markov Chains". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07). June 2007, pp. 708–717. DOI: 10.1109/DSN.2007.37. URL: https://ieeexplore.ieee.org/document/4273022 (visited on 01/25/2025).

[5] Marc Bouissou and Jean-Louis Bon. "A New Formalism That Combines Advantages of Fault-Trees and Markov Models: Boolean Logic Driven Markov Processes". In: *Reliability Engineering & System Safety* 82.2 (Nov. 1, 2003), pp. 149–163. ISSN: 0951-8320. DOI: 10.1016/S0951-8320(03)00143-1. URL: https://www.sciencedirect.com/science/article/pii/S0951832003001431 (visited on 05/13/2025).

[6] Edwin Bourget et al. "Probabilistic Event Graph to Model Safety and Security for Diagnosis Purposes". In: *Data and Applications Security and Privacy XXXII*. Ed. by Florian Kerschbaum and Stefano Paraboschi. Cham: Springer International Publishing, 2018, pp. 38–47. ISBN: 978-3-319-95729-6. DOI: 10.1007/978-3-319-95729-6_3.

[7] Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* C-35.8 (Aug. 1986), pp. 677–691. ISSN: 1557-9956. DOI: 10.1109/TC.1986.1676819. URL: https://ieeexplore.ieee.org/document/1676819 (visited on 12/10/2024).

[8] Carlos E. Budde and Mariëlle Stoelinga. "Efficient Algorithms for Quantitative Attack Tree Analysis". In: *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 2021 IEEE 34th Computer Security Foundations Symposium (CSF). June 2021, pp. 1–15. DOI: 10.1109/CSF51468.2021.00041. URL: https://ieeexplore.ieee.org/document/9505179 (visited on 11/20/2024).

[9] *Built-in Types*. Python documentation. URL: https://docs.python.org/3/library/stdtypes.html (visited on 05/21/2025).

[10] E.M. Clarke et al. "Spectral Transforms for Large Boolean Functions with Applications to Technology Mapping". In: *Formal Methods in System Design* 10.2 (Apr. 1, 1997), pp. 137–148. ISSN: 1572-8102. DOI: 10.1023/A:1008695706493. URL: https://doi.org/10.1023/A:1008695706493 (visited on 06/17/2025).

[11] *Cudd: CUDD Documentation*. Jan. 27, 2018. URL: https://web.archive.org/web/20180127051756/http://vlsi.colorado.edu/~fabio/CUDD/html/index.html (visited on 05/21/2025).

[12] Adrien Derock, Patrick Hebrard, and Frédérique Vallée. "Convergence of the Latest Standards Addressing Safety and Security for Information Technology". In: *ERTS 2010 Proceedings*. Toulouse, France, May 2010. URL: https://hal.science/hal-02267717 (visited on 01/25/2025).

[13] David Peter Eames and Jonathan Moffett. "The Integration of Safety and Security Requirements". In: *Computer Safety, Reliability and Security*. International Conference on Computer Safety, Reliability, and Security. Springer, Berlin, Heidelberg, 1999, pp. 468–480. ISBN: 978-3-540-48249-9. DOI: 10.1007/3-540-48249-0_40. URL: https://link.springer.com/chapter/10.1007/3-540-48249-0_40 (visited on 01/25/2025).

[14] M. Fujita, P.C. McGeer, and J.C.-Y. Yang. "Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation". In: *Formal Methods in System Design* 10.2 (Apr. 1, 1997), pp. 149–169. ISSN: 1572-8102. DOI: 10.1023/A:1008647823331. URL: https://doi.org/10.1023/A:1008647823331 (visited on 06/17/2025).

[15] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function Using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[16] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. "Grammarinator: A Grammar-Based Open Source Fuzzer". In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, Nov. 5, 2018, pp. 45–48. ISBN: 978-1-4503-6053-1. DOI: 10.1145/3278186.3278193. URL: https://dl.acm.org/doi/10.1145/3278186.3278193 (visited on 06/17/2025).

[17] Nils Husung et al. "OxiDD: A Safe, Concurrent, Modular, and Performant Decision Diagram Framework in Rust". In: *Proceedings of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'24)*. 2024. DOI: 10.1007/978-3-031-57256-2_13.

[18] Joost-Pieter Katoen and Mariëlle Stoelinga. "Boosting Fault Tree Analysis by Formal Methods". In: *ModelEd, TestEd, TrustEd*. Springer, Cham, 2017, pp. 368–389. ISBN: 978-3-319-68270-9. DOI: 10.1007/978-3-319-68270-9_19. URL: https://link.springer.com/chapter/10.1007/978-3-319-68270-9_19 (visited on 01/24/2025).

[19] Siwar Kriaa et al. "A Survey of Approaches Combining Safety and Security for Industrial Control Systems". In: *Reliability Engineering & System Safety* 139 (July 1, 2015), pp. 156–178. ISSN: 0951-8320. DOI: 10.1016/j.ress.2015.02.008. URL: https://www.sciencedirect.com/science/article/pii/S0951832015000538 (visited on 01/24/2025).

[20] Siwar Kriaa et al. “Safety and Security Interactions Modeling Using the BDMP Formalism: Case Study of a Pipeline”. In: *Computer Safety, Reliability, and Security*. Ed. by Andrea Bondavalli and Felicita Di Giandomenico. Cham: Springer International Publishing, 2014, pp. 326–341. ISBN: 978-3-319-10506-2. DOI: 10.1007/978-3-319-10506-2_22.

[21] Rajesh Kumar and Mariëlle Stoelinga. “Quantitative Security and Safety Analysis with Attack-Fault Trees”. In: *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*. 2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE). Jan. 2017, pp. 25–32. DOI: 10.1109/HASE.2017.12. URL: https://ieeexplore.ieee.org/document/7911867 (visited on 01/24/2025).

[22] *Lark-Parser/Lark*. Lark - Parsing Library & Toolkit, May 20, 2025. URL: https://github.com/lark-parser/lark (visited on 05/21/2025).

[23] C. Y. Lee. “Representation of Switching Circuits by Binary-Decision Programs”. In: *The Bell System Technical Journal* 38.4 (July 1959), pp. 985–999. ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1959.tb01585.x. URL: https://ieeexplore.ieee.org/document/6768525 (visited on 12/12/2024).

[24] Milan Lopuhaä-Zwakenberg, Carlos E. Budde, and Mariëlle Stoelinga. “Efficient and Generic Algorithms for Quantitative Attack Tree Analysis”. In: *IEEE Transactions on Dependable and Secure Computing* 20.5 (Sept. 2023), pp. 4169–4187. ISSN: 1941-0018. DOI: 10.1109/TDSC.2022.3215752. URL: https://ieeexplore.ieee.org/document/9925106 (visited on 11/19/2024).

[25] Igor Nai Fovino, Marcelo Masera, and Alessio De Cian. “Integrating Cyber Attacks within Fault Trees”. In: *Reliability Engineering & System Safety*. ESREL 2007, the 18th European Safety and Reliability Conference 94.9 (Sept. 1, 2009), pp. 1394–1402. ISSN: 0951-8320. DOI: 10.1016/j.ress.2009.02.020. URL: https://www.sciencedirect.com/science/article/pii/S0951832009000337 (visited on 01/25/2025).

[26] Stefano M. Nicoletti, E. Moritz Hahn, and Mariëlle Stoelinga. “BFL: A Logic to Reason about Fault Trees”. In: *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). June 2022, pp. 441–452. DOI: 10.1109/DSN53405.2022.00051. URL: https://ieeexplore.ieee.org/document/9833769 (visited on 11/26/2024).

[27] Stefano M. Nicoletti et al. *ATM: A Logic for Quantitative Security Properties on Attack Trees*. May 17, 2024. DOI: 10.48550/arXiv.2309.09231. arXiv: 2309.09231. URL: http://arxiv.org/abs/2309.09231 (visited on 11/19/2024). Pre-published.

[28] Stefano M. Nicoletti et al. *DODGE: Ontology-Aware Risk Assessment via Object-Oriented Disruption Graphs*. Dec. 18, 2024. DOI: 10.48550/arXiv.2412.13964. arXiv: 2412.13964 [cs]. URL: http://arxiv.org/abs/2412.13964 (visited on 01/12/2025). Pre-published.

[29] Stefano M. Nicoletti et al. “Model-Based Joint Analysis of Safety and Security:Survey and Identification of Gaps”. In: *Computer Science Review* 50 (Nov. 1, 2023), p. 100597. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2023.100597. URL: https://www.sciencedirect.com/science/article/pii/S1574013723000643 (visited on 01/26/2025).

[30] Stefano M. Nicoletti et al. "PFL: A Probabilistic Logic for Fault Trees". In: *Formal Methods*. Ed. by Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker. Cham: Springer International Publishing, 2023, pp. 199–221. ISBN: 978-3-031-27481-7. DOI: 10.1007/978-3-031-27481-7_13.

[31] Stefano M. Nicoletti et al. "WATCHDOG: An Ontology-aWare Risk AssessmenT approaCH via Object-Oriented DisruptiOn Graphs". In: *Advanced Information Systems Engineering*. Ed. by John Krogstie et al. Cham: Springer Nature Switzerland, 2025, pp. 314–331. ISBN: 978-3-031-94571-7. DOI: 10.1007/978-3-031-94571-7_18.

[32] Dan S. Nielsen. *The Cause/Consequence Diagram Method as a Basis for Quantitative Accident Analysis*. Report 87-550-0084-3. Roskile, Denmark: Risø National Laboratory, 1971. URL: https://orbit.dtu.dk/en/publications/the-causeconsequence-diagram-method-as-a-basis-for-quantitative-a.

[33] Ludovic Piètre-Cambacédès and Marc Bouissou. "Attack and Defense Modeling with BDMP". In: *Computer Network Security*. Ed. by Igor Kotenko and Victor Skormin. Berlin, Heidelberg: Springer, 2010, pp. 86–101. ISBN: 978-3-642-14706-7. DOI: 10.1007/978-3-642-14706-7_7.

[34] Ludovic Piètre-Cambacédès and Marc Bouissou. "Beyond Attack Trees: Dynamic Security Modeling with Boolean Logic Driven Markov Processes (BDMP)". In: *2010 European Dependable Computing Conference*. 2010 European Dependable Computing Conference. Apr. 2010, pp. 199–208. DOI: 10.1109/EDCC.2010.32. URL: https://ieeexplore.ieee.org/document/5474179 (visited on 05/13/2025).

[35] Ludovic Piètre-Cambacédès and Marc Bouissou. "Modeling Safety and Security Interdependencies with BDMP (Boolean Logic Driven Markov Processes)". In: *2010 IEEE International Conference on Systems, Man and Cybernetics*. 2010 IEEE International Conference on Systems, Man and Cybernetics. Oct. 2010, pp. 2852–2861. DOI: 10.1109/ICSMC.2010.5641922. URL: https://ieeexplore.ieee.org/document/5641922 (visited on 05/13/2025).

[36] Chunling ZHU Quan JIANG. "Qualitative analysis for state/event fault trees using formal model checking". In: *Journal of Systems Engineering and Electronics* 30.5 (Oct. 9, 2019), pp. 959–973. ISSN: 1004-4132. DOI: 10.21629/JSEE.2019.05.13. URL: https://www.jseepub.com/CN/abstract/abstract7037.shtml (visited on 01/24/2025).

[37] Antoine Rauzy. "New Algorithms for Fault Trees Analysis". In: *Reliability Engineering & System Safety* 40.3 (Jan. 1, 1993), pp. 203–211. ISSN: 0951-8320. DOI: 10.1016/0951-8320(93)90060-C. URL: https://www.sciencedirect.com/science/article/pii/095183209390060C (visited on 11/26/2024).

[38] R. Rudell. "Dynamic Variable Ordering for Ordered Binary Decision Diagrams". In: *The Best of ICCAD: 20 Years of Excellence in Computer-Aided Design*. Ed. by Andreas Kuehlmann. Boston, MA: Springer US, 2003, pp. 51–63. ISBN: 978-1-4615-0292-0. DOI: 10.1007/978-1-4615-0292-0_5. URL: https://doi.org/10.1007/978-1-4615-0292-0_5 (visited on 05/21/2025).

[39] Enno Ruijters and Mariëlle Stoelinga. "Fault Tree Analysis: A Survey of the State-of-the-Art in Modeling, Analysis and Tools". In: *Computer Science Review* 15–16 (Feb. 1, 2015), pp. 29–62. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2015.03.001. URL: https://www.sciencedirect.com/science/article/pii/S1574013715000027 (visited on 11/19/2024).

[40] Enno J. J. Ruijters et al. “FFORT: A Benchmark Suite for Fault Tree Analysis”. In: *ESREL 2019: Proceedings of the 29th European Safety and Reliability Conference*. 29th European Safety and Reliability Conference, ESREL 2019. Research Publishing, 2019, pp. 878–885. DOI: 10.3850/978-981-11-2724-3_0641-cd. URL: https://research.utwente.nl/en/publications/ffort-a-benchmark-suite-for-fault-tree-analysis (visited on 06/17/2025).

[41] Caz Saaltink. *CazSaa/Dd*. Mar. 7, 2025. URL: https://github.com/CazSaa/dd (visited on 05/21/2025).

[42] Caz Saaltink. *ODF: Object-oriented Disruption Framework*. Version v1.0.0. Zenodo, June 26, 2025. DOI: 10.5281/zenodo.15744808. URL: https://zenodo.org/records/15744808 (visited on 06/26/2025).

[43] Caz Saaltink et al. “Solving Queries for Boolean Fault Tree Logic via Quantified SAT”. In: *Proceedings of the 9th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems*. FTSCS ’23: 9th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems. Cascais Portugal: ACM, Oct. 18, 2023, pp. 48–59. ISBN: 979-8-4007-0398-0. DOI: 10.1145/3623503.3623535. URL: https://dl.acm.org/doi/10.1145/3623503.3623535 (visited on 11/26/2024).

[44] Giedre Sabaliauskaite and Aditya P. Mathur. “Aligning Cyber-Physical System Safety and Security”. In: *Complex Systems Design & Management Asia*. Springer, Cham, 2015, pp. 41–53. ISBN: 978-3-319-12544-2. DOI: 10.1007/978-3-319-12544-2_4. URL: https://link.springer.com/chapter/10.1007/978-3-319-12544-2_4 (visited on 01/25/2025).

[45] Tiago Prince Sales et al. “The Common Ontology of Value and Risk”. In: *Conceptual Modeling*. Ed. by Juan C. Trujillo et al. Cham: Springer International Publishing, 2018, pp. 121–135. ISBN: 978-3-030-00847-5. DOI: 10.1007/978-3-030-00847-5_11.

[46] B. Schneier. “Attack Trees”. In: *Dr. Dobb’s Journal* (Dec. 1999). URL: https://www.schneier.com/academic/archives/1999/12/attack_trees.html (visited on 12/12/2024).

[47] Mu Sun et al. “Addressing Safety and Security Contradictions in Cyber-Physical Systems”. In: (). URL: https://seclab.illinois.edu/addressing-safety-and-security-contradictions-in-cyber-physical-systems.

[48] Andreas Thums and Gerhard Schellhorn. “Model Checking FTA”. In: *FME 2003: Formal Methods*. International Symposium of Formal Methods Europe. Springer, Berlin, Heidelberg, 2003, pp. 739–757. ISBN: 978-3-540-45236-2. DOI: 10.1007/978-3-540-45236-2_40. URL: https://link.springer.com/chapter/10.1007/978-3-540-45236-2_40 (visited on 01/25/2025).

[49] *Tulip-Control/Dd*. Temporal Logic Planning (TuLiP) toolbox, Apr. 23, 2025. URL: https://github.com/tulip-control/dd (visited on 05/21/2025).

[50] Weiwei Zhou et al. “Integrating Security Factors into Fault Tree Analysis: A Safety and Security Co-Analysis Approach for AADL Models”. In: *International Conference on Computer Application and Information Security (ICCAIS 2023)*. International Conference on Computer Application and Information Security (ICCAIS 2023). Vol. 13090. SPIE, Apr. 8, 2024, pp. 219–225. DOI: 10.1117/12.3025564. URL: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/13090/130900X/Integrating-security-factors-into-fault-tree-analysis--a-safety/10.1117/12.3025564.short (visited on 12/12/2024).

Appendix A

Declarations

A.1 Use of AI

During the implementation of ODF, GitHub Copilot code completions were enabled. GitHub Copilot was occasionally used to help write unit tests. During the preparation of the thesis, the author used Claude 3.5 Sonnet and Gemini 2.5 Pro in order to help rewrite sentences and create a more coherent text. After using these tools, the author reviewed and edited the content as needed and takes full responsibility for the content of the work.