ONTOLOGY-BASED FORMAL APPROACH FOR SAFETY AND SECURITY VERIFICATION OF INDUSTRIAL CONTROL SYSTEMS

by

Ramesh Neupane

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

August 2022

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Ramesh Neupane

Thesis Title: Ontology-based Formal Approach for Safety and Security Verification of Industrial Control Systems

Date of Final Oral Examination: 15 April 2022

The following individuals read and discussed the thesis submitted by student Ramesh Neupane, and they evaluated the student's presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Hoda Mehrpouyan, Ph.D.	Chair, Supervisory Committee
Bogdan Dit, Ph.D.	Member, Supervisory Committee
Craig Rieger, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Hoda Mehrpouyan, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

DEDICATION

This thesis is dedicated to my parents and my sisters for their endless love, support and encouragement.

ACKNOWLEDGMENT

I want to express my deepest gratitude to my supervisor Dr. Hoda Mehrpouyan for supporting and encouraging me throughout my graduate studies. Without her help and guidance, this thesis would not be possible.

I would also like to thank my committee members, Dr. Bogdan Dit and Dr. Craig Rieger (Idaho National Laboratory), for their constructive comments and feedback that have helped shape the scope of this thesis.

I am forever grateful to my parents, Sita Neupane and Ramkrishna Neupane, for their everlasting love and support during my academic journey. I am also thankful to my two beautiful sisters, Rajani and Rashila, for providing me with endless love.

The work of this thesis is based on work supported by the National Science Foundation Computer and Information Science and Engineering (CISE) division, award number 1846493 of the Secure and Trustworthy Cyberspace (SaTC) program: Formal TOols foR SafEty aNd Security of Industrial Control Systems (FORENSICS), so I would like to thank them for their support.

Last but not least, I want to thank my lab members, especially Sean O'Toole, Chidi Agbo, and Chibuzo Ukegbu, for their help, guidance, and support throughout my graduate studies.

ABSTRACT

Control logics, as part of the Industrial Control Systems (ICS), are used to control the physical processes of the critical infrastructures such as power plants, water, and gas distribution, etc. Most commonly, the Programmable Logic Controller (PLC) manages these processes through actuators based on information received from sensor readings. Any safety issues or cyberattacks on these systems may have catastrophic consequences on human lives and the environment. In an effort to improve the resilience and security of control logics, this thesis provides algorithms and tools to formally define the safety and security requirements w.r.t. the physical processes, and the industrial domain. Web Ontology Language (OWL) is utilized to create semantic relationships between the elements of industrial processes and knowledge mapping of the input and output of the control logic. Description Logic (DL) knowledge bases derived from OWL allow us to reason about the semantic security and safety concepts to ensure their consistency. Next, these formal specifications are translated to Timed Computational Tree Logic (TCTL) queries for the verification of the control logic modeled in UPPAAL as a network of timed automata (TA). In the second part of the thesis, boundary conditions are checked to perform a model verification. Boundary checking is essential in ICS because the sensor's readings and actuator's values need to be within the safe range to ensure secure ICS operation.

For the proof of concept, we have studied a part of an industrial chemical process

to implement our proposed approach. Experimental results in this work proved that the proposed method detects the inconsistencies in safety and security requirements and ensures that the input and output variables of the control logic are within a safe and secure range. The performance study of our implementations shows that the time grows linearly with the number of axioms in the ontology and the number of iterations in TA model simulations. Hence, the approach is scalable to have a practical implementation to help the technicians and engineers to create a safer and more secure control logic for ICS processes.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CTL Computational Tree Logic

DL Description Logic

FB Function Block

FBD Function Block Diagram

FOL First-Order Logic

FSA Finite State Automaton

ICS Industrial Control Systems

IT Information Technology

LTL Linear Temporal Logic

OT Operational Technology

OWL Web Ontology Language

PLC programmable logic controllers

SMV symbolic model verification

SWRL Semantic Web Rule Language

TA Timed Automaton

TCTL Timed Computational Tree Logic

UML Unified Modeling Language

PART I

CHAPTER 1:

INTRODUCTION

A host of industries and public utilities use Industrial Control Systems (ICS), such as programmable logic controllers (PLC), distributed control systems (DCS), supervisory control and data acquisition (SCADA), and safety instrumented systems (SIS) to monitor and control automation processes and their safe operations. With advances in information technology (IT), artificial intelligence (AI), and robotics, computer scientists and engineers continue to develop new ICS applications to improve quality and efficiency, which are also resulting in more commercial networks using the Internet to extend information sharing for further efficiency. Unfortunately, this elevates cybersecurity threats and the need for techniques and tools so system designers can identify and implement more robust security and safety requirements for these critical systems.

The goal of this research is to design and develop algorithms and tools to improve the safety and security of ICS, which are complex and highly interconnected software and hardware systems. These systems are considered critical and essential for the well-being of society. Hence, any type of issue or cybersecurity threat can result in significant destruction, affecting millions of people. Considering the seriousness of the consequences, it is essential to develop an understanding of how to integrate safety and security requirements into the control software. We answer this question by the

development of a knowledge base (KB) that can serve as a reference ontology for the requirements of the industrial control software. The formal language-based ontology inherently supports the automatic validation engine to check the consistencies and completeness of the specifications. This results in finding issues in the early stage of the requirement analysis, which is vital in the case of industrial applications because finding problems in a later stage of the development might have severe consequences.

Contradictory requirements between the safety/real-time properties and security needs of the system might cause several vulnerabilities in the system. Detecting such conflicts in early stages increases both the safety and security of the system. That is why there has been multiple research on identifying conflicts in different requirements [1, 2, 3]. The main approach of those researches is to use a common representation for all the requirements so that they can be translated into a formal language to be used in the validation process. After checking inconsistencies in the requirement specification, those requirements are to be verified with the implemented system. In the context of ICS, to check whether the system satisfies the safety and functional requirements, there are various testing methods [4, 5, 6, 7] that are in practice. Since we are dealing with the critical applications in ICS, formal verification approaches that are explored in [8, 9, 10, 10, 11, 12, 13, 14] are more reliable because it exhaustively performs systematic exploration of the mathematical model that represents the system.

For the formalization of the specifications and requirements, we use Description Logic (DL) as the mathematical formula, which is a fragment of first-order logic (FOL) and is part of the knowledge representation formalism family. We will build on the research [15] that has utilized DL for run time verification to specify the relationship between different attacks [16], exploring the domain of an specific attack [17], and in-

trusion detection [18]. Our contribution is to propose DL formalism to formulate the specification of entities and their relationships and design and develop an ontology that contains the essential information about the processes that the control logic is ought to manage, and its safety and security requirements. The built ontology acts as a knowledge base for the requirements verification. Two operational algorithms have been developed within this approach for defining safety/security concepts and consistent requirements to create an ICS ontology. Second, DL-reasoner is utilized to ensure that the requirements and specification rules are consistent before the verification of control logic is started. For the model checking process the control logic that is extracted from the PLC program and modeled as a Timed Automaton (TA), along with the requirements from the ontology, are translated into Timed Computational Tree Logic (TCTL), and tested to ensure safety and security properties hold.

In this research, a part of an industrial chemical process was taken to illustrate the approach. We built a relevant ontology through the detailed analysis of the chemical process using a popular ontology editor, Protégé [19]. The DL-reasoner Pellet [20] that is supported by the editor was utilized to check inconsistencies and completeness of the ontology. We were able to show that the OWL-DL [21] along with Semantic Web Rule Language (SWRL) [22] rules can be used for the formalization of the requirements and specification of the chemical process. We automatically transformed the rules expressed in SWRL into the TCTL queries using temporal operators such as eventually and globally. The TCTL formula is then input to the UPPAAL [23] model checker. In UPPAAL, we first model the control logic implemented in the PLC program with the required additional components for the logic as TA. UPPAAL is able to check the TCTL properties on a given TA model and provides execution

traces which helps in identifying issues in system logic. Experimental results on the chemical process show that the proposed method detects inconsistencies in the requirement specification with ontology and DL-reasoner. Moreover, we were able to show that the consistent requirements can be then used for further verification of the control logic with the help of UPPAAL model checker and conversion of TCTL queries into TCTL formulas.

To summarize, the aim of this thesis is to formalize the safety and security requirement specifications, to check the consistency between them and to exploit that formalized information for verifying control logic. The research questions of this thesis are as follows:

1. How to formalize and create the knowledge-base so that the inconsistencies in the safety and security requirements are automatically detected?

2. How to formally verify and validate control logic to make sure it satisfies its safety, security and functional requirements?

CHAPTER 2:

CHEMICAL PROCESS UNDER STUDY

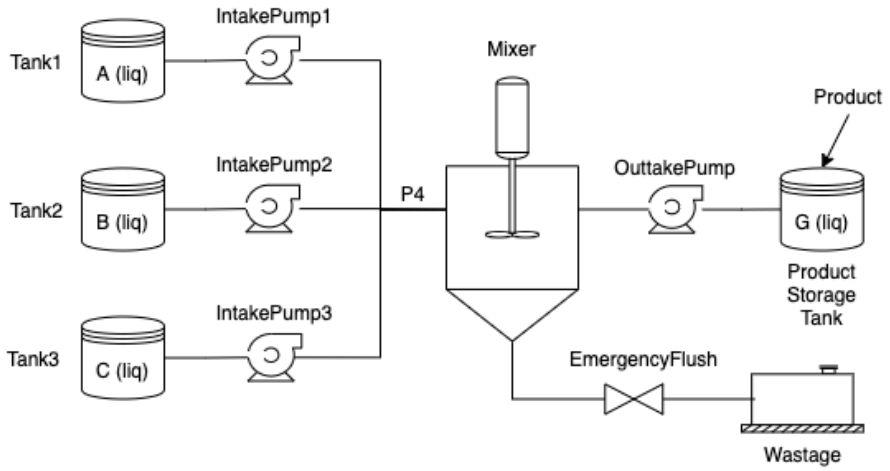


Figure 2.1: Chemical Process

For the proof of concept and throughout the paper, we will use the chemical process (CE) which is a partially modified version of industrial processes in water utilities. The selected process has four main components: storage tanks, intake and outtake pumps, a mixer and a flush as shown in Figure 2.1. The chemical process primarily involves an irreversible chemical reaction of three liquid reactants producing one liquid product:

$$A(liq) + B(liq) + 2C(liq) = G(liq)$$

In this process, the mixing reactor receives materials from three supply tanks

(Tank1, Tank2, and Tank3) through the pumps (*IntakePump1*, *IntakePump2*, *IntakePump3*). Each intake pump has a rate of 2 liters per second and the outtake pump has a rate of 3 liters per second. First, *IntakePump1* pumps 4 liters of chemical A from Tank1 to the Mixer. It is followed by the transfer of 4 liters of chemical B from Tank2 and 8 liters of chemical C from Tank3. The mixer starts to run immediately with *IntakePump2* and *IntakePump3*. The ingredients are then mixed for 4 seconds before being transferred to an idle product storage tank with the help of the pump *OuttakePump*. The user input *RunProcess* starts and pauses the chemical process, whereas the *EmergencyStop* switch resets the system and runs the emergency flush valve for 5 seconds.

For this process, the supply of chemicals for the chemical reaction must be maintained by filling the input tanks. Chemicals are supplied to the tank based on their usage, where a certain threshold is set to fill the tank, so that it does not run empty. A few of the safety constraints for chemical processes are discussed below along with its implications.

1. The mixer should not run if chemicals are not present
    - Running an empty mixer causes higher temperature in the mixer, which might eventually lead to an explosion
2. All three intake pumps should not run simultaneously
    - More flow-rate on pipe P4 might cause a pipe to burst
3. In the mixing reactor the chemical A should not be more than 6 liters
    - The reaction generates more heat, causing a breakdown of the mixer

4. Mixer and OuttakePump should not run at the same time

    - Useless final product

CHAPTER 3:

BACKGROUND AND RELATED WORKS

ICS has been isolated from external networks for years, so the verification and testing were more focused on the functional specifications and safety requirements. However, with the advent of Industry 3.0 and now 4.0 [24], the need for real-time data from Operational Technology (OT) has left us with no option other than connecting critical infrastructures to the Internet. With increased openness and complexity of the system, it has become more vulnerable to threats from both insider and external adversaries. The vulnerabilities reported in ICS have increased tenfold from 2010 to 2017 [25]. Not surprisingly, the number of attacks has also risen up significantly as discussed in the report [26]. As mentioned in the paper, the most prevalent weakness in the industry is the weak boundary protection between enterprise networks and ICS. After getting access to the system, attackers maliciously modify the values of the PLC variables to control and vandalize the infrastructure. Due to the rapid integration of IT and OT, there is a clear separation of knowledge in IT and OT systems [26]. In some of the literature, it is also mentioned that in certain aspects of the system, personnel from different domains view other parts of ICS as black boxes. So, this gap not only exists between IT and OT, but also inside the OT domains. Therefore, we can clearly say that there is a knowledge gap between Information Technology (IT) and OT although their cooperation is vital for a resilient ICS.

Due to this gap in IT security and OT operations, there could be conflicts in security and safety requirements and functional specifications. Thus, we need to have a mechanism to check consistency between the requirements and to verify that the control logic satisfies these requirements. The rest of this chapter provides a detailed review of different approaches applied in checking the consistency of the requirements and using those in the verification of control logic.

3.1 Formal Verification of Specifications

While verifying and validating control logic implemented in ICS, it is essential to have consistent and complete requirements. Consistent requirements are those safety and security requirements that are not in conflict with each other.

Consistency checking of requirement specifications is not a new idea. Heitmeyer et al. in [1], described an automatic consistency checking technique of requirement specifications. The method explained in the paper is specifically designed to check the consistency of the requirements expressed in Software Cost Reduction (SCR) tabular notation. The tabular details are modeled in Finite State Automaton (FSA) for further verification. Li et al. in [2] argued that Unified Modeling Language (UML) models are mostly used for consistency checking in software engineeing rather than SCR model, so, they proposed a formal logical way to check requirements modeled on UML. Although UML provides good support for specifying requirements, it does not inherently support logic inference for checking models. Also, some subsets of UML that support model checking suffer from undecidability in reasoning [27]. The ontology based approach on the other hand uses DL, that support logic inference for detecting conflicts between statements. Also, DL is a decidable fragment of the First-Order Logic (FOL), so it is decidable most of the time.

Not only in software engineering, consistency checking of specifications has been explored in other areas as well. Kamsu-Foguem et al. [3] proposed a conceptual graph-based framework for formally checking the compliance of construction buildings to verify that they satisfy certain standards or regulations. In this paper, the author modeled the building requirements and facts about building information in a conceptual graph using a graph-based visual tool called CoGui [28]. Using the tool, which utilizes subsumption relations between the conceptual graph for reasoning [29], the modeled graph is then reasoned and validated for compliance checking.

Although this approach of automatically checking the consistency of requirements specifications has been used in many areas, we did not find similar research that is focused on ICS. Therefore, we propose a new formal ontology-based method to check the consistency between requirements using the popular DL-based reasoner. For this approach, we need to build an ICS ontology with safety and security components. After building an ontology, it can be used not only for checking conflicts, but also for verifying control logic.

3.2 Verifying and Testing Control Logic

In this section, we first review the existing methodologies that are used to test the control logic in PLC software. Functional testing of control logic modules, with manually generated input test cases, is the technique most commonly used in industrial process control applications [4, 5]. Test cases are designed based on prior safety and security failure experiences, reports, and the tester's expertise [6]. While these types of test-case driven approaches are easy to implement, they lack complete testing of PLC logic and there is a high possibility of missing crucial program flaws that have not been discovered previously [4, 30]. In order to overcome this issue and to auto-

matically test the dynamic behavior of PLC, simulation-based tools are used.

Lee et al. developed a simulation-based testing environment [6] considering the characteristics of safety-critical PLC systems like scan-cycle, CPU architecture, and a memory map of PLC microprocessors. To precisely emulate the PLC software behavior and to check if the correct output is generated based on the specific program input and internal states of the program, the author introduced a software testbed that captures both the internal and external conditions of the PLC scan cycle. Another simulation-based tool, SIVAT [7] translates Function Block Diagram (FBD) into C and performs functional testing. Even though these simulation-based tools and simulators [31, 32] allow testing for specific scenarios in a controlled environment that specifies runtime contexts, it is not under their scope to test for unknown attacks and failures.

To overcome the weaknesses of manual and simulation-based testing, researchers have used a formal verification approach to increase the quality of the safety-critical PLC program by detecting more faults [8, 9, 10, 10, 11]. After the early effort of Palshikar and Nori [33] to use temporal logic to formally model the PLC program, many researchers have relied on formal methods for validation and verification of the PLC software.

Rausch and Krogh [12] presented an approach to translate the PLC program logic into symbolic model verification (SMV) modules and specifications into Computational Tree Logic (CTL), which is further verified using the SMV engine. Similarly, Soliman et al. in [13], translated the PLC Function Block (FB) program into the UPPAAL [23] TA model to formally verify the safety requirements formalized in temporal logic. This approach of abstractly making models of the system and exhaustively

checking it for safety issues is great if all the given requirements and specifications are complete and consistent. Moreover, the models created are more specific to the behavior of the controller program, and hence it fails to address the causal relationship of the program and other components in ICS. On the contrary, our research utilizes a knowledge base that provides the components involved in the ICS and the relationship between them. Thus, it ensures that the controller's program is in agreement with the other components of the system.

CHAPTER 4:

METHODOLOGY

This section provides a general overview and the motivation of our proposed methodology to verify that the PLC program logic satisfy the required safety and security properties. The overall outline of the methodology is as shown in Figure 4.1.

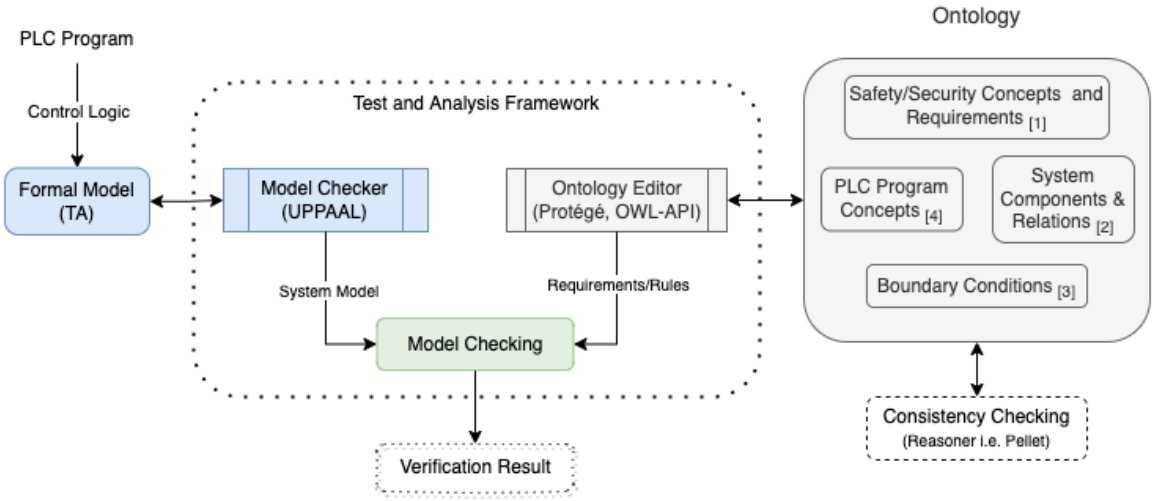


Figure 4.1: An overview of the proposed approach

In the first section of the methodology, an ontology that is based on DL is designed and developed. The proposed ontology is designed to include 1- safety/security concepts and requirements, 2- system components involved in the control logic, 3- several boundary values and conditions, and 4- PLC program concepts. With the help of ontology reasoner, we check for consistency and completeness of the requirements and

specifications.

In the second part of the proposed approach, the control logic and each of the components in the processes that are managed by the controller are modeled utilizing a formal modeling language. The translated model of the control logic is then used in the model checker.

Finally, having the formal model and ontology of the requirements, the verification is carried out. The requirements that are consistent and complete given by the ontology are translated into the model checker's requirements language, and then the model is checked against those requirements via model checking. The details of implementation are discussed further in this section.

Why Ontology? In many previous works, the requirements are translated into automata which are checked for consistency using model checking [34], or the requirements specifications are modeled in first-order logic and then verified using SMT-Solver for consistency [35]. These approaches do not scale to large sets of requirements and it also does not consider the system model in consideration. On the other hand, an ontology is scalable and considers the system model and specifications in the analysis. It is expressive enough to represent most of the ICS requirements. For the consistency checking, we don't have to transform it to other forms of logic because the reasoners like Hermit, Pellet, etc. are available for the DL based ontology.

Why Model Checking? Model checking [36] is a formal method, where a desired behavioural property is verified against a system model through a symbolic exploration of the system's state space. The main advantage of model checking is that unlike other approaches like logic and theorem proving, it doesn't require user's supervision and expertise in mathematical disciplines because it is fully automatic

[37]. In addition to that, if the model fails to satisfy the desired property, it helps in debugging by providing a counterexample, an execution of system's model violating the property [38].

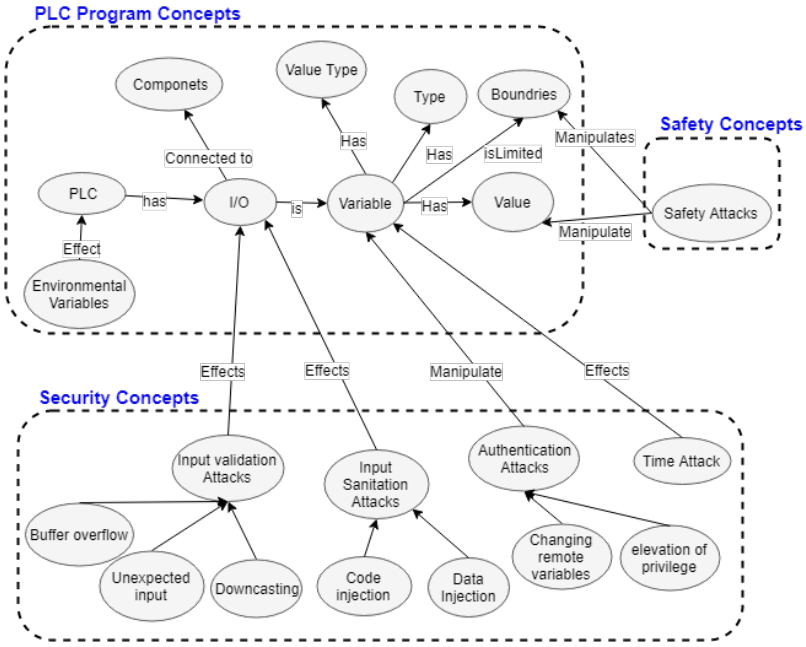


Figure 4.2: Safety, security and PLC program ontology

4.1 Ontology

In order to build the ontology in Figure 4.2, detailed information about the industrial processes that is under study is required. That domain knowledge is first formalized based on the formal language, Web Ontology Language (OWL)-DL [21] and then it is added to a knowledge base. OWL-DL has a rich tool set available for creating/editing an ontology while being able to check the consistencies of its components. In addition to that, it is also decidable with maximum expressiveness [39].

In OWL, class has a similar meaning as **concept** in Description Logic (DL) and property represents **role**. As depicted in Figure 4.3, the first step in implementing

Figure 4.3: Ontology building steps

	System Knowledge (Class Definition)	Formal Definition (DL)
1	Pump is a component	Pump $\sqsubseteq$ Component
2	Flush is a pump	Flush $\sqsubseteq$ Pump
3	IntakePump is a pump	IntakePump $\sqsubseteq$ Pump
4	PLC is a controller	PLC $\sqsubseteq$ Controller
5	Chemical is a material	Chemical $\sqsubseteq$ Material
6	IntakeTank and InputTank are same	IntakeTank $\equiv$ InputTank

Table 4.1: Hierarchical class definition of ICS components in DL

the ontology starts with defining the Safety, security, and control logic classes and their constraints and restrictions (Terminology Boxes (TBoxes)). Next, we create instances of those classes, which are also called Assertion Boxes (ABoxes). Thus, the knowledge base (KB) is basically a pair (T, A), where T is a $TBox$ and A is an $ABox$. The rest of this section explains the building blocks of mapping the control logic and the safety and security requirements of the industrial control process.

4.1.1 System Design and Processes Ontology

Using DL, components are defined as a hierarchical class with the help of the subsumption relation. For example, Table 4.1 shows the subsumption relations between the classes of a chemical process plant. Subsumption ($\sqsubseteq$) and Definition or Equivalence ($\equiv$) are two main relation operators that are used for defining classes.

Similarly, after defining the required components as a class, we need to define the relation and data properties of those classes. Relations describe the relationship between the components and the data property explains and assigns a literal value

	System Knowledge (Relation and Attribute)	Formal Definition (DL)
1	Pump pumps out material	pumpsOut(Pump, Material)
2	Tank contains material	containsMaterial(Tank, Material)
3	Tank has capacity of 20 liters	hasCapacity(Tank, 20)

Table 4.2: Relation and attribute definition of ICS components in DL

to an entity. The relationship can be formalized as adding them as a role that relates to two individuals, which is also called object properties in OWL. The properties of components, i.e., speed, power, etc., can be formally defined as data type properties. The data properties can be variables of data type int, float, etc. These variables can have values that are either constant, selectable, or continuous. If the value is $Constant(x)$ then a fixed value of x should be given. If the value is a $Selectable(x)$ then x is a set of values that specify possible selection options. If the value is a $Continuous(x)$ then x should specify the range that is acceptable. As we can see in Table 4.2, *pumpsOut* relation property describes the relationship between the Pump and the Material. Similarly, *hasCapacity* is a data property and it relates between the class tank and the literal integer value 20.

4.1.2 Safety and Security Ontology

To develop a safe system, we first need to analyze the hazards and the corresponding requirements at the system level. In our ontology, safety concepts are defined based on two groups of hazard analysis techniques; failure-based and system-based [40]. Failure-based methods focus on the effect of single component failures, i.e. fault tree analysis (FTA), whereas system-based methods are based on detailed analysis of the system design and the system parameters. Based on the hazard analysis, safety requirements are generated.

Safety requirements can be added to the ontology as constraints. Constraints are formulated as necessary and sufficient conditions for a member to belong to a class. In DL, these conditions are Boolean combinations of properties required for a class or relationships with other members and their properties. For example, the safety property input tank should only contain input materials.

$$InputTank(t1) \equiv Tank(t1) \sqcap \forall containsMaterial(t1, m1) \sqcap InputMaterial(m1)$$

which uses the class names *Tank* and *InputMaterial* and the object property *containsMaterial* as well as a class conjunction ($\sqcap$). We can also express cardinality constraints where we can limit the number of entities that belongs to a certain class. For example, a tank that can only contain one material can be formalized as follows:

$$Tank(t1) \sqcap \leq 1 \quad containsMaterial(t1, m1) \sqcap Material(m1)$$

The security ontology contains security concepts specific to ICS. The sources to these general concepts are the ICS security experts, published work, previous attack reports, etc. The general security concepts can be formally defined in DL as subsumption relations. For example, the concepts such as "Input validation attack is an attack" and "Buffer overflow is an input validation attack" can be formalized as follows.

$$InputValidationAttack \sqsubseteq Attack$$

$$BufferOverflow \sqsubseteq InputValidationAttack$$

Even more complex concepts like "Override switch can be remotely manipulated so it is an attack vector" can be formalized as

$$RemoteControlSwitch(s1) \equiv Switch(s1) \sqcap \forall hasRemoteControl(s1)$$
$$RemoteControlSwitch \sqsubseteq AttackVector$$
$$OverrideSwitch \sqsubseteq RemoteControlSwitch$$
$$OverrideSwitch \sqsubseteq AttackVector \quad (inferred)$$

where the fourth logical statement is inferred by the previous three statements.

4.1.3 Semantic Web Rule Language

Constraints and requirements in ontology can also be represented with another subset of predicate logic with efficient proof systems, called horn logic [41]. Horn logic rules also called horn-clauses are used in ontology to provide more dynamism to the DL based ontology. SWRL [22] is the rule language that includes an abstract syntax for horn-like rules.

For example, the following security requirement: *"The emergency stop switch can only be enabled by the authorized user. The user is authorized if the user is logged in and is in the allowed user group admin."* can be expressed as

$$isEnabled(EmergencyStop, true) \wedge isEnabledBy(EmergencyStop, user1)$$
$$\wedge isLoggedIn(user1, true) \wedge isAdmin(user1, true)$$

Similarly, the safety requirement *"If the chemical tank is empty, no intake pumps should run."* can be expressed as

$sensorReading(s1, 0) \wedge isEmpty(s1, true) \wedge isRunning(c1, false) \wedge IntakePump(c1)$

4.1.4 Reasoning in Ontology

Consistency Checking

An ontology is inconsistent if there exists an instance of either class or a property contradicts an axiom of the ontology. If there is a logical contradiction in an ontology, the ontology becomes meaningless, it's because we can derive any kind of statement from a set of logical axioms that contradicts each other [42]. Formally, an ontology is said to be inconsistent if an axiom in the ontology is unsatisfiable. For example: the assertions, {*EmergencyFlush: Pump, EmergencyFlush: Variable*} causes inconsistency in the ontology, because the name is considered unique in ontology. Hence, an ontology model, $M \nvDash (EmergencyFlush : Pump \wedge EmergencyFlush : Variable)$. Also, lets consider the clauses, "OuttakePump is running" i.e. s1 = isRunning(OuttakePump, 1) and "OuttakePump is not running" i.e. s2 = isRunning(OuttakePump, 0), hence $M \nvDash (s1 \wedge s2)$.

Specification Completeness

This can be achieved by checking the completeness of the ontology. The ontology is complete if every clause belongs to some statements and every instance of the lexical elements has related links to some statements. For example: The following security requirement *"To enable overwrite switch, the user should be logged in."* i.e. $LoggedInUser(user1) \wedge switchAccess(user1, true) \wedge enableSwitch(true, user1)$. This

cannot be complete if there are assertion $User(user1)$, and axiom $LoggedInUser \sqsubseteq User$ not present in the ontology.

4.2 Model Checking

In model checking, systems are modeled by finite-state machines, and properties are written in propositional temporal logic. The verification procedure is an exhaustive search of the state space of the design model. The model checking framework we are exploring is [43] in which the system descriptions are specified in TA, while the requirement specifications in TCTL formulas. The TCTL model-checking problem is P-SPACE complete [44]. In the following section, we give the formal definitions of TA and TCTL.

4.2.1 Timed Automaton

TA [43] [45] is an extension to a finite automaton with a finite set of real-valued variables called *clocks* to specify constraints of time between two events. If X = {x_1, x_2, ... , x_n} is a finite set of clocks, then a *clock valuation* is a mapping v: X $\in \mathbb{R}^X$ and $\phi(X)$ represents the set of formulas called clock constraints. For example, Figure 4.4 shows a basic timed automaton with clock variables x and y. One of the possible runs of this automaton is

$$(l_0,(0,0)) \xrightarrow{4.1} (l_0,(4.1,4.1)) \rightarrow (l_1,(4.1,0)) \xrightarrow{3.7} (l_1,(5.3,3.7)) \rightarrow (l_2,(5.3,3.7))$$

where (l_2, (5.3, 3.7)) means that the value of clock variable x is 5.3 and y is 3.7 at location l_2.

A TA is a tuple $A(L, L_0, \Sigma, X, I, E)$, where L is a finite set of locations, $L_0 \in L$

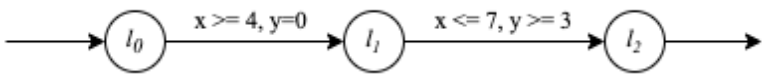


Figure 4.4: Timed Automaton

is a set of initial locations, Σ is a finite set of labels, X is a finite set of clocks, I is a mapping that labels each location l with some clock constraint in $\phi(X)$ and E is a finite set of edges of form e = (l, γ, α, x, l'), with l, $l' \in$ L the source and target states, γ is a conjunction of atomic constraints on X, called guard, α is a label for discrete actions or a time delay and $x \in X$ is a set of clocks to be reset upon crossing the edge.

Clock

A finite set of real clock variables used for the specification of quantitative time constraints which is associated with transitions. A clock is a variable that ranges over $\mathbb{R}$, the set of nonnegative reals. All clock variables in the timed automaton advance simultaneously.

States

A state of A is a pair of (l, v) where $l \in L$ is a discrete state i.e. location, and v is a set of all clock interpretations for X that satisfies $I(l)$. The initial state of A is s_0 = (l_0, 0).

Transitions

Let us consider a state $s = (l, v)$. The discrete transition of A, that is, $(l, v) \xrightarrow{\alpha} (l', v')$, occurs if the edge e = (l, γ, α, x, l'), such that $v \vDash \gamma$ and $v' = v[reset(e)]$, where

$reset(e)$ represents the clock assignment that maps all clocks in x to 0. The time transition of A i.e. $(l, v) \xrightarrow{\delta} (l, v + \delta)$, occurs if $(v + \delta \vDash I(l))$ and $\delta \in \mathbb{R}$. Time transition and discrete transition can also be written as $s \xrightarrow{\delta} s + \delta$ and $s + \delta \xrightarrow{\alpha} s'$ respectively.

Runs

A run of A from state s is a finite or infinite sequence r = $s_1 \xrightarrow{\delta_1} s_1 + \delta_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\delta_2} s_2 + \delta_2 \xrightarrow{\alpha_2}$ ...,, where $s_i = s$ for all $i = 1, 2, ...$, $s_i + \delta_i$ is time transition of s_i and $s_i + 1$ is the discrete transition of $s_i + \delta_i$.

Reachable States

A state s is reachable if there exists a finite run $s_0 \xrightarrow{\delta_0} s_0 + \delta_0 \ ... \ s_k \xrightarrow{\delta} s$, where s_0=(l_0 0) is the initial state and $k \in \mathbb{N}$.

4.2.2 Timed Computational Tree Logic

The finite state systems are modeled by labeled state transition graph called Kripke Structures. a Kripke structure is a triple M = (S, R, L), where, S is a set of states, $R \sqsubseteq S$ x S is a transition relation, and L: $S \rightarrow P(AP)$ is the set of atomic proposition (AP) true in each state. The structure can be unwound into an infinite tree with the initial state as the root. The path in M is an infinite sequence of states, $\pi = s_0, s_1,$ ... such that for $i \geq 0$, $(s_i, s_{i+1}) \in \mathbb{R}$.

In the design of the model checking tool, there are choices of the temporal language that are used to specify properties.The two types of temporal property specification languages are differentiated by their underlying model of time, Linear Temporal Logic (LTL) and CTL. In LTL, each moment in time has a unique possible future. i.e.

the operators provided are for describing the events along a single computational path. However, in branching time logic we can have a single moment of time split into multiple possible futures. TCTL is a timed extension of CTL logic which is branching-time logic, where the bound of a temporal operator is given as a pair of a lower bound and an upper bound.

TCTL formulas can be inductively defined via the following production rule.

$$\phi \coloneqq true \mid p \mid \neg p \mid \phi \mid \phi \vee \phi \mid E[\phi U_I \phi] \mid A[\phi U_I \phi]$$

where, p is a set of atomic formulas, A and E are the universal and existential path quantifiers, respectively. U(Until) is a temporal operator, and I represents any one of the relational operators ($=$, $<$, $\leq$). We can define many properties that uses "Always" ($\Box$ or G), or "Eventually" ($\Diamond$ or F) temporal operators based on the set of operators presented by the production rule. Given the finite state model M with an initial state s_0, the TCTL formula ϕ is satisfied by the model can be formally expressed as $(M, s_0) \vDash \phi$. For example: If p and q are the local atomic formula, then,

Invariance: $s_0 \vDash A\ G_{[2,3]}\ p$, implies that p is true for all possible path in future between the states s_{0+2} and s_{0+3}, shown in Figure 4.5a.

Bounded response time: $s_0 \vDash A\ G\ (p \rightarrow A\ F_{\leq 4}\ q)$, implies that for all paths, it is always a case that once p holds, q eventually holds within 4 time units, shown in Figure 4.5b.

4.3 Transformation to TCTL

There has been some work to translate specifications in ontology into the TCTL using the specification pattern system (SPS), which is a set of recurring patterns of func-

(a) $s_0 \vDash A\ G_{[2,3]}\ p$ **(b)** $A\ G\ (p \rightarrow A\ F_{\leq 4}\ q)$

Figure 4.5: TCTL Properties

tional and timing requirements [46]. In this work, for simplicity, we only consider the conversion of requirements represented in SWRL rules. The main reason for this conversion is that we need to verify the requirements using a model checker that supports temporal logic. The advantages of this conversion is that the requirements/properties that are generated for checking are consistent with the overall specifications and other requirements. It is important because model checking can be time and resource consuming, so, the properties we are checking should be consistent and complete which in this case is ensured by ontology.

Algorithm 1 shows the generation of temporal formula in TCTL. The input to these algorithms are the collections of the requirements and constraints from the ontology, i.e. horn clauses of DL concepts. For example: *"A maximum overflow of chemicals, can cause pipe burst. So, we need to always ensure that the material flowing through the pipe is less than the threshold"*, can be represented in horn-like clauses as

$$hasFlowrate(p1, fr) \wedge flowThreshold(p1, th) \wedge fr < th$$

where hasFlowrate and flowThreshold are data properties in the ontology that relates

pipe1(p1) with certain integer literal value *fr* and *th* respectively.

The algorithm first, split the clause to the collection of literals. Then, each literal is looked at into a mapping file that contains the relation between data/object properties in the ontology and the model template of the model checker (MapOntologyToModelChecker). For example: in the above example, the hasFlowRate(p1, fr) would be p1.flowrate=fr in the model checker. Then, the final task is to add temporal operators ($A, E, \Box, \Diamond$). In our case, for simplicity, we have limited the number of temporal operators to use, but the algorithm can be extended to support more temporal operators. The output of this algorithm is a set of TCTL formulas. i.e. E$\Diamond$ (p $\rightarrow$ q).

Algorithm 1 Translation of horn-like clauses to TCTL

```
1: Get all rules
2: for rule in rules(r_1, r_2, ..., r_n) do
3:     literals ← getLiterals(rule);
4:     q ← getHead(rule);
5:     for l in literals do
6:         pArr ← MapOntologyToModelChecker(l);
7:     end for
8:     p ← joinLiterals(pArr);
9:     tctl ← addTemporalOperators(tctl);
10:    tctls[0, 1, ..i] ← E◊ (p → q);
11: end for
```

CHAPTER 5:

CASE STUDY

For the proof of concept, we take an industrial OT process as we have discussed in chapter 2. We used Velocio PLC [47] to implement the control logic of the chemical process. Velocio uses ladder logic and flow chart language for programming. We use Protégé to create an ontology with essential classes, individuals, properties, and relationship axioms. Protégé uses OWL for creating an ontology. For the requirements and constraints, the SWRL rules are added to the ontology. A Pellet[20] reasoner that works with the ontology is used to infer knowledge based on the existing information on the ontology and to check the consistency of the ontology. The logic of the PLC program is modeled in TA of the UPPAAL model checker [23]. Then, the requirements from the ontology are translated into UPPAAL queries for further model verification.

The rest of this section provides a brief introduction on how we built the ontology for detecting the conflict in requirements, the UPPAAL model of the PLC program control logic, and the verification of the properties done using UPPAAL Verifier.

5.1 Ontology with Protégé and OWL-API

Due to the popularity of OWL in the semantic web, a lot of tools are available for editing and building an ontology. Protégé is one of the tools that is open-source and

	Information and Specifications	DL Equivalent (Ontology)
1	OuttakePump is a Pump	OuttakePump $\sqsubseteq$ Pump
2	Pump is a Component	Pump $\sqsubseteq$ Component
3	Mixer should run for 11 seconds	MixerO $\sqcap \exists$ shouldRunFor.11
4	IntakePump1 should run before Mixer	IntakePump1 $\sqsubseteq \exists$ shouldRunBefore.Mixer
5	IntakePump2 is running	IntakePump2 $\sqsubseteq$ isRunning.True

Table 5.1: Class and Property definitions

contains a reasoning and inference engine that supports the validation and verification of DL queries. To build an ontology, we used Protégé as a GUI tool and OWL-API [48] for automating ontology access, i.e update, delete. The OWL-API is a Java API implementation for creating, manipulating, and serialising OWL ontologies. To guide our ontology building process, we followed the guide [49] provided by Protégé.

In Protégé, we can create the hierarchical classes, define object properties and their relationship with individual classes, and add data properties that assign a literal value to class individuals. At first, we built an ontology of the components and the properties involved in the chemical process. Table 5.1 depicts examples of information that are used to build the ontology. In addition, the requirements and constraints are added through the SWRL queries in Protégé.

5.1.1 Safety and Security Ontology

To create the safety and security ontology, we browsed research articles and journals and noted some important security/safety concepts pertinent to PLC and ICS in general. The remaining part of this section provides the formalism of some of the examples of safety/security requirements and concepts we have used.

Security Requirements

If there are instantiated objects or variables in the PLC program, an attacker can gain access to the system with minimal effort by insertion [50]. Therefore, the security requirement for this case would be *"Do not leave unused variable in the PLC program."*. In the PLC program, each variable is either assigned to input/output components or used in the program for temporary operation, if the variable does not affect the control logic they are unused variables and are vulnerable to attack through immediate insertion. These types of inferences are automatically done by the DL-reasoner based on the already added information.

Hardcoded numeric values in PLC programs can be vulnerable to attack by allowing the number to be changed directly [50]. The security requirement for this case could be *"Avoid using hardcoded numeric values in PLC programs."*. These requirements are added to the ontology as follows:

$$hasValue(Variable, Value)$$

$$HardcodedValue \sqsubseteq Value$$

$$isVulnerableTo(HardcodedValue, InputValidationAttack)$$

$$InputValidationAttack \sqsubseteq SecurityAttack$$

Safety Requirements

One of the safety requirements for the chemical process is *"All three input tanks should not be running at the same time."*. The main reason is that P4 in Figure 2.1 has a threshold flow rate of 6 liters per second, so running all together will exceed the threshold, which could cause serious accidents by pipe burst. Another important

safety constraint is that if the mixer is running empty, it increases the temperature of the mixer abruptly, that might eventually lead to an explosion. This can be written as *"The mixer should not run if the chemicals are not present."*.

5.1.2 Detecting Inconsistencies

One of the main reasons for formalizing specifications and requirements in DL-based ontology is that it provides automatic classification and inconsistency checking. For example, in the chemical process, we have the following set of requirements: (1) *"Users must be logged in to enable/disable switch."* (security) (2) *"Login requires at least a minute to complete."* (specification) (3) *"In case of emergency, EmergencyStop switch should be enabled within 30 seconds."* (safety). Here, we can see that the requirements are clearly inconsistent because a loggedin user requires more than 30 seconds to disable the emergency stop switch, which needs to be disabled within 30 seconds. The above requirements are added to the ontology as:

$$LoggedInUser \sqsubseteq isLoggedIn\ value\ 'true'$$

$$canEnableSwitch(Domain : LoggedInUser, Range : LRSwitch)$$

$$timeRequiredToLogin(?u, ?t) \wedge t >= 60 \rightarrow LoggedInUser(?u)$$

$$LRSwitch(?s) \wedge enableSwitchWithin(?s, ?t1) \wedge LoggedInUser(?u) \wedge$$
$$timeRequiredToLogin(?u, ?t2) \wedge t1 <= t2 \rightarrow Violation(R1)$$

After adding these rules and specifications, the reasoner is synchronized in Protege to see if there are inconsistencies or violations. We can see (Figure 5.1) that there is a violation caused by the rules we have added. It can also be noticed that the reasoner

also provides the reason for the violation.

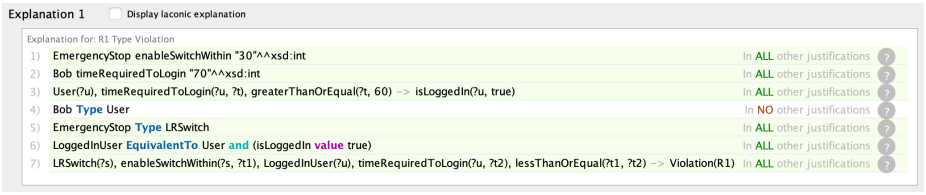
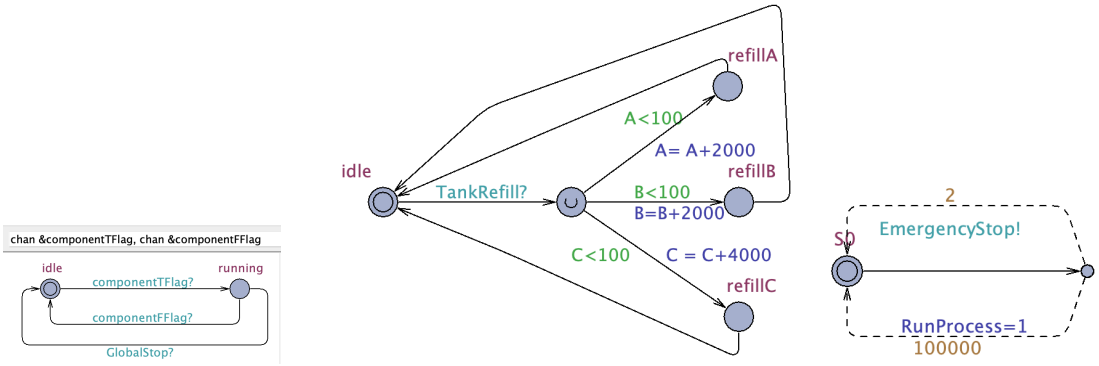


Figure 5.1: Checking requirement inconsistency (Protege)

5.2 Model Checking with UPPAAL

UPPAAL is a toolbox for verification of real-time systems. In UPPAAL, the system is modeled in the network of TA, while the requirements are formalized into UPPAAL's query language which is a subset of TCTL. UPPAAL extends timed automata with various other features like templates, synchronizations, urgent locations and expressions like guard, invariant [51].



(a) Component Model **(b) Tank Refill Model** **(c) Userinput**

Figure 5.2: UPPAAL model template

In our case study, i.e. chemical process, the intake pumps, outtake pump, mixer, and flush act in a same way as a timed automaton model. Thus, we created a template that is called component as shown in Fig. 5.2a. The component template

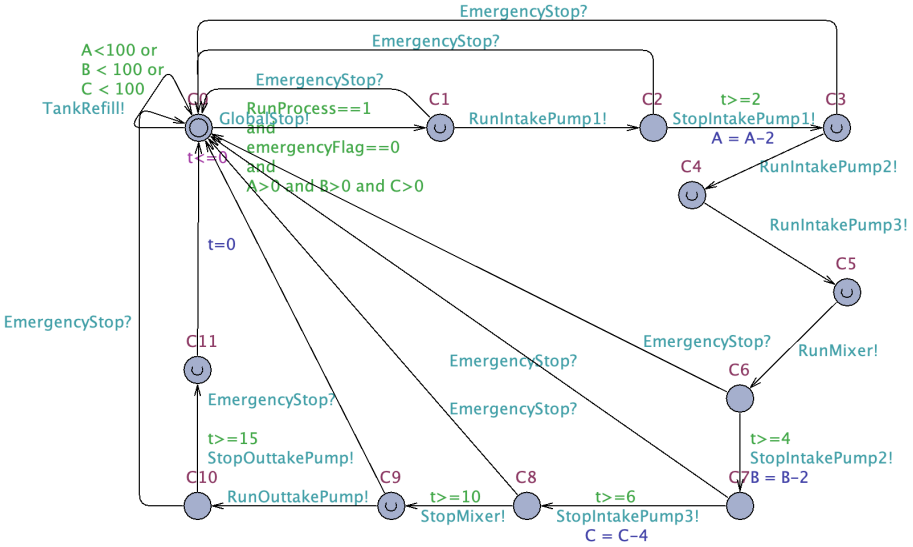


Figure 5.3: Overall chemical process

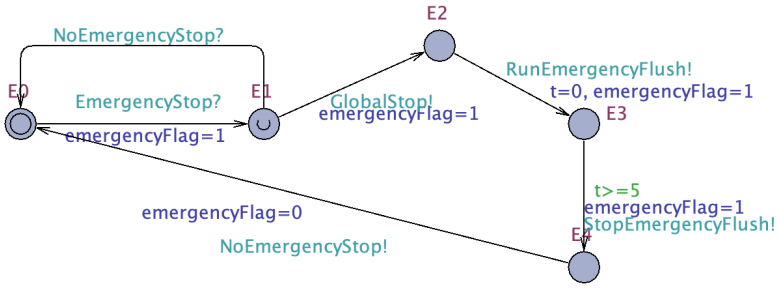


Figure 5.4: Emergency case

has two states {idle, running}. These states can be changed using the variables which is parameter to the template. This template can be initialized to create each component by creating an object in UPPAAL's system declarations as:

intakePump1 = Component(RunIntakePump1, StopIntakePump1),

intakePump2 = Component(RunIntakePump2, StopIntakePump2),

intakePump3 = Component(RunIntakePump3, StopIntakePump3),

mixer = Component(RunMixer, StopMixer),

$outtakePump = Component(RunOuttakePump,\ StopOuttakePump),$

$emergencyFlush = Component(RunEmergencyFlush,StopEmergencyFlush).$

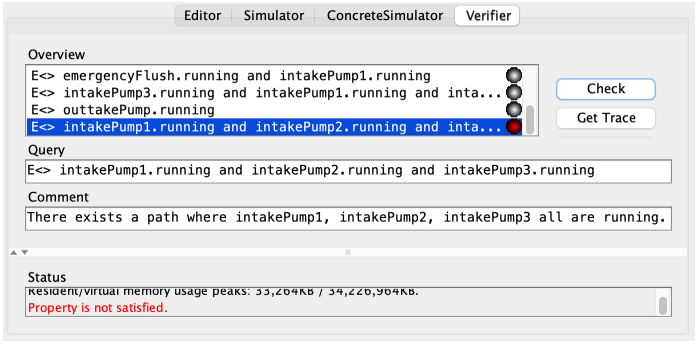


Figure 5.5: Requirement verification on UPPAAL (R2)

One of the specifications of the chemical process is that if the chemicals in all chemical tanks are less than 100 gallons, then it needs to be refilled soon. It is modeled in UPPAAL as shown in Figure 5.2b.

As we mentioned in the previous section, the requirement *"All three pumps should not run at the same time."* is formalized in SWRL rules as

$$isRunning(intakePump1, true) \wedge isRunning(intakePump2, true) \wedge$$
$$isRunning(intakePump3, true) \rightarrow Violation(R2)$$

If there is a case where these three pumps are running simultaneously, then its violation of the rule. This rule is translated to TCTL queries with the algorithm (1) implemented in Java. The translation gives us the TCTL formula for the UPPAAL model checker. The translated formula is

$$E <> \ intakePump1.running \ and \ intakePump2.running \ and \ intakePump3.running$$

This formula asks the model checker if there exists a path where all three pumps are running. We ran this query on our model in UPPAAL Verifier and this property is not satisfied (Figure 5.5). Similarly, following the SWRL rule,

$$containsAmount(Chemical\_A, ?am) \wedge am < 0 \wedge isRunning(intakepump1, true) \rightarrow Violation(R3)$$

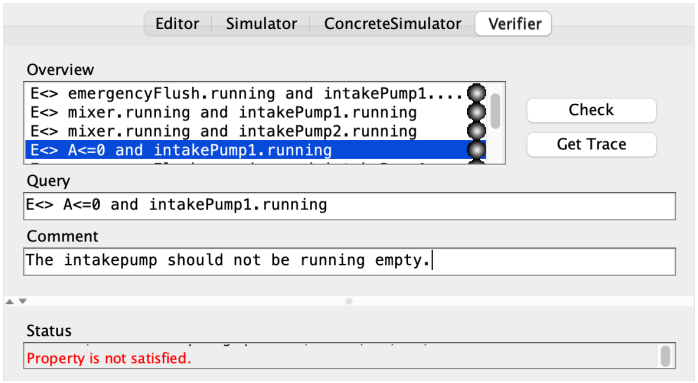


Figure 5.6: Requirement verification on UPPAAL (R3)

This rule says that the intakepump1 should not be running in an empty condition. That is, there should be some chemicals present in the corresponding input tank. This requirement is translated to TCTL query as

$$E <> \ \ A <= 0 \ and \ intakePump1.running$$

We checked this TCTL property to the model in UPPAAL (Figure 5.6). The result shows that the property is not satisfied, that means, there are no paths in the model execution, where the intakepump1 will be running empty. In this way, we can guarantee that the safety and security policies are not violated by the PLC control program. The policies that are being checked are ensured to be consistent and complete by the reasoner.

For this experiment, an ontology with a total of 313 logical axioms was built. The total time for checking consistency was on average 80 ms. The transformation of the five SWRL rules took 780 ms on average, and then the model checking took nearly 3 seconds for each of our specific requirements. This experiment was performed on 2.4 GHz Quad-Core Intel Core i5 MacBook Pro with 16 GB of RAM. The total time taken for the process increases as we increase the size of the ontology and the complications in the TA model. If we increase the size of the ontology with greater numbers of logical axioms, the reasoner will take more time for automatic classification and detecting inconsistencies. The time taken increases linearly with the axiom count as shown in the Figure 5.7. We can find a detailed empirical analysis by Dalwadi et al. on [52]. Similarly, model checking can also run into undefined states because of its state space explosion problem as the model becomes complicated.

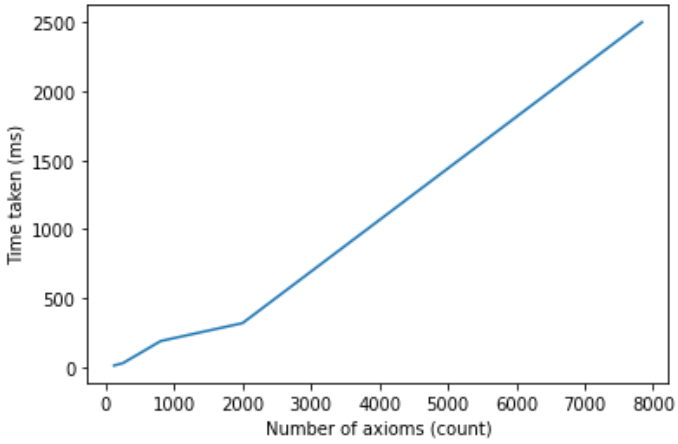


Figure 5.7: Axiom count vs Time taken

CHAPTER 6:

CONCLUSION AND FUTURE WORK

The first part of this thesis presented an approach to verify the PLC control logic using an ontology. The ontological knowledge base is built using OWL with the help of OWL-API. A DL-based reasoner is used to check consistency between the requirements added into the ontology. The created ontology is used to verify whether the control logic violates the requirements added in the ontology.

This research, however, is subject to several limitations. The first is that we need to extract the control logic from the PLC program and then that logic is translated into another language. Manual logic extraction and translation might help to identify several issues that were not evident before in the original PLC program. However, it is a lossy process, so, future work could be to automate this. The second limitation is that the algorithm presented in this paper to translate requirements in DL or SWRL rules to TCTL queries works only for specific cases. And finally, the third limitation is that while creating ontology and formalizing requirements, domain expert knowledge is required.

In the future work, we will extend the element of concept in DL to represent hazard scenarios, and security gaps that will result in loss, denial, and manipulation. Instances are elements belonging to a concept and roles are binary relations between two concepts. Based on the DL formalism, we will build a safety and security on-

tology, that is, a knowledge representation over a set of concepts within a domain and the relationship between them. In order to construct the ontology, we will utilize STPA-safesec [53], as a safety and security analysis technique that is an extension of Leveson's Systems-Theoretic Accident Model and Processes (STAMP) and System Theoretic Process Analysis (STPA) [54, 55] to map the safety and security concepts and roles. We will perform the STPA-Safesec analysis by: (1) defining the control layer concept, (2) identification of hazardous control actions (safety concept), (3) mapping the control and component layer (ontology), (4) modifying the safety and security constraints, (5) generating the hazard scenarios. In addition to have more enriched ontology, we are working on using the ontological knowledge base information like boundary values, relationship in symbolic model verification of UPPAAL.

PART II

CHAPTER 7:

REFERENCE KNOWLEDGE BASE IN LOGIC VERIFICATION

In the previous part, we discussed how the DL-based ontology, along with the reasoner, can be used to detect conflicts in the requirements and specifications of ICS. After checking the consistency, those requirements are used in model checking by translating the requirements from SWRL rules into TCTL queries. This part discusses how the built ontology can be utilized as a reference knowledge base in software verification.

7.1 Introduction

ICS is now more exposed to external attackers and threats [25] due to its rapid integration with the Internet. The few common vulnerabilities in the industrial sectors are Out-of-bounds write, Out-of-bounds read, and Improper Input Validation [10]. These vulnerabilities might result in loss of availability, integrity, and confidentiality in the critical systems where precision and control are vital for the safe operation of the processes. These vulnerabilities could be exploited as part of the control logic modification attack where the attacker tries to manipulate the sensors measurements and program variable values with some unexpected/out-of-bound values causing the controller to send out unsafe control commands. The remote manipulation can in-

clude changes in variables that go outside the bound causing the system to be in an unsafe state. These examples shows that if engineers, technicians, and risk management teams understand how the control logic and different subsystems, systems, and components relate to one another and how data is maintained and exchanged amongst many different domains in the industrial processes, they will be able to implement proper security and safety controls and prepare a precise risk evaluation/mitigation plan.

In order to provide the understanding and model the relationship that exists among data across different software domains, in software testing, bounds checking [56, 57] is common as a method of detecting whether a variable is within a safe and secure bounds for the correct operation of the system. Range checking [58, 59] is one of the bounds checking methods to verify whether the variable is within the safe range. These techniques are usually used during the compile time to result in safe code. However, there are not many tools and algorithms available for such testing in safety critical systems that are vulnerable to integrity attacks, i.e. sensor manipulation attacks, Human Machine Interface (HMI) attacks that will result lack of situational awareness. Control processes by nature are required to be highly resilient with a strong safety and security protection.

In this part of the thesis, we use DL [60] based ontology to describe knowledge regarding component boundaries, process dependencies, concepts of safety, and security. We propose a formal verification approach for checking the boundary conditions during the control logic verification. First, we simulated the formal model of the Programmable Logic Controller (PLC) program utilizing UPPAAL [61] and UPPAAL-API. Then, a boundary verification algorithm is designed and developed to

import the required information from the ontology.

The remainder of this chapter is organized as follows. Section 7.2 provides an overview of some background and related works. Section 7.3 summarizes our implementation of boundary verification and the formal definition of boundary conditions. Section 7.4 explains in detail the algorithm and preliminary concepts. Section 7.5 discusses the implementation results. Finally, Section 7.6 concludes the chapter with future work.

7.2 Related Works

Static and dynamic code analysis techniques are considered one of the important methods to detect security vulnerabilities in software design and development life-cycle [62, 63]. Static analysis examines the source code or binary without running the code, while dynamic analysis involves the actual execution of the software. Various tools and techniques are available for both analysis, however each have their pros and cons. While static code checking does not take into the consideration the dynamic nature of ICS data, it requires less resources and can be utilized at the early stages of the design and development of the code. Alvares et al. [64] proposed computational intelligence techniques such as a genetic algorithm (GA) to statically check the source code. In their work, the authors performed the boundary check of the program variables based on their type and size of the variables.

More specifically, these types of analysis have been used for intrusion detection in Cyber-Physical Systems (CPS) [65]. In this work, the authors utilized static timing analysis for bounds checking to detect unauthorized instructions in real-time CPS environments. Following the success of formal methods including (static analysis, theorem proving, and model checking) in verificaiton and validation of CPS, Zheng

et. al [66] has generated a report of the current state of the art verification practices in CPS. The authors concluded that "existing formal method techniques and simulation are, as yet, insufficient for supporting the development of entire general-purpose CPS".

One of the most challenging part of the verification and validation of CPS results from the dimensions of data interoperability or data fusion. We rely on data fusion to understand, monitor, and analyze many sensors, sophisticated control systems, and industrial processes in robotics, nuclear power plants, energy generation, distribution, and transportation. In 2013 an international standard Recommendation ITU-T X.1255 titled "Framework for discovery of identity management information" was approved to adopt a data model to provide a uniform concept to represent metadata records for purpose of interoperability across heterogeneous systems [67]. Following this recommendation, PLC-PROV [14] proposes the use of the provenance of the system to detect security and safety violations. Data provenance is a kind of metadata that describes the dependencies between data sets and processes. To create a data provenance graph, the input and output variables of the sensors and actuators with the timestamps are collected into the system execution traces. The traces obtained are fed into Curator, a data provenance management tool that creates a graph to show the flow of data from the sensor to the actuators through the PLC controller. The graph created in Curator is used to detect safety and security violations. The provenance graph is limited to the safety components that create the causal dependency graph between sensor inputs and actuator outputs through the PLC program based on the execution traces of the program. To the best of our knowledge, there has been limited work on the knowledge-base approach for automatic boundary checking

and formal verification of the PLC that contains the mapping of safety and security to components and processes.

7.3 Methodology

Figure 7.1 depicts an overview of the proposed method, which comprises two stages: 1-a DL-based ontology is developed to include safety/security policies, safety boundaries, and input and output mapping of the ICS system including PLC program. With the help of DL-based reasoner, the consistency and completeness of the requirements and specifications in the ontology is verified. 2- The PLC is modeled in formal modeling language where the variables and components are verified to ensure safety and security boundaries provided by the ontology are satisfied.

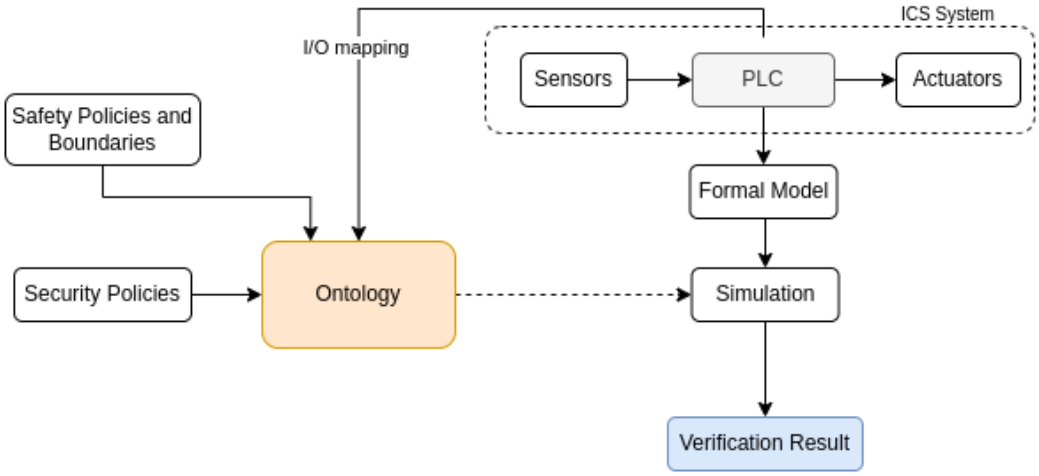


Figure 7.1: Overall Methodology

7.3.1 Ontology-based Verification and Validation

An ontology knowledge-base approach is used in software engineering domain as a knowledge management tool to improve the sharing of knowledge and learning practices in software testing [68]. The ontology presented in their work includes knowledge

of the software testing domain that contains concepts, properties, and their relationships. The knowledge base portal introduced in this paper consists of an experience-sharing portal, where the tester's experience is shared and added to the ontology. The reasoning layer reasons about the concepts and axioms to provide validation. Eventually, it is archived into the storage layer to retain the knowledge base. The knowledge retrieval layer makes use of SPARQL queries to import relevant information from the knowledge base. Similarly, ROoST: The Reference Ontology for Software Testing [69] provides guidelines for creating a reference ontology for Software Testing and for a detailed evaluation of the proposed testing technique. Although these guidelines are great for building an ontology of software testing concepts and properties as a reference for the software tester, they are not implemented and used in practice in software testing tools.

Algorithm 2 Boundary Verification on model simulation

```
initialize boundaries <list>
Get all ValueRange from ontology
for bVar in ValueRange(vr_1, vr_2, ..., vr_n) do
    boundaries ← [bVar.from, bVar.to]
end for
for Model simulation do
    for all variables do
        if variable not in boundary then
            violations ← state details (iteration no., transition, values)
        end if
    end for
end for
return violations
```

In this work, we perform control logic validation using the ontology-based reasoner. The process guarantees correctness of the boundary conditions of the components. The reasoner queries the logical statements in the ontology and generates flags if the

deduction and inferences lead to statements that form a tautology. In addition, a verification process is implemented to ensure that the control logic behaviour satisfies the safety and security requirements. In this case, the behavior of the PLC program should comply with the boundary conditions and the dependencies in the ontology. Algorithm 2 depicts the proposed algorithm for checking boundaries of the control logic variables and components. At first, the conditions that need to be checked are imported from the ontology. In particular, the value range of PLC variable and system components are imported. Then, the formal model of the PLC program and the system is simulated in order to check the conditions that are imported from the ontology.

7.3.2 Boundary Verification

To capture safety and security requirements, we need to start at the component level such as industrial controller, sensors, and actuators. The user defines the number of inputs and outputs to/from the control algorithm. The inputs are connected to sensors, switches, or other controller's outputs. The outputs are connected to actuators or other controllers' inputs. Each input and output can interact with the whole or part of a component. Consequently, each component is represented by multiple variables connected to the controller's inputs and outputs. Each component is defined as a set of variables that model the functionality of that component, i.e., pump. Each variable, i.e., speed, power, etc., has a type such as a float, boolean, etc., and a value type such as continuous, selectable, etc. Each component will be formalized as a DL concept. The variables can have values that are either constant, selectable, or continuous. If the value is a Constant(x) then a fixed value of x should be given. If the value is a Selectable(x) then x is a set of values that specify possible selection

options. If the value is a Continuous(x) then x should select the acceptable range.

Safety and security requirements in ICS is defined as constraints upon the input and output values of the controller logic. These constraints can be either limitation on variables or dependencies between variables. These constraints can also be defined at the component and type levels. For constraints at the component level, all instances of that component inherit that constraint. As for constraints at the type level, all variables of that type will have that constraint.

Boundaries

The boundary b_v^L is defined as possible safe values that a variable v can have at level $L = \{Component, Type, Instance\}$. The type of b is defined based on the variable that the boundary is defined upon. The value type of variable v defines whether the boundary b is a set or a range. For selectable and constant value types, b is defined as a set of values and for continuous value types, b is defined as a range $[b_1, b_2]$.

$$
\begin{aligned}
b_v^{Component} &:= \forall\ instances(I) \in Component(C),\ I.v \models b \\
b_v^{Type} &:= \forall\ variable(v)\ \text{, where } variable.type = Type,\ v \models b \\
b_v^{Instance} &:= \forall\ instances(I),\ I.v \models b
\end{aligned}
$$

For example, if intake pump 1 is an instance of the component intake pump. This can be represented in the program variable as RunIntakePump1. If the variable is a boolean type, then the boundary (b) for this case would be $\{0, 1\}$, i.e. $RunIntakePump1 \models b$.

To represent this boundary condition in an ontology, we first need to define a class called ValueRange. The value range will have starting and ending values, so we have two data properties rangeFrom and rangeTo that map the ValueRange instance to literal values. This is formalized in DL as follows:

$$ValueRange(BooleanRange), Variable(IntakePump1)$$

$$hasBoundary(IntakePump1, BooleanRange)$$

$$rangeFrom(BooleanRange, 0), rangeTo(BooleanRange, 1)$$

Figure 7.2 shows the graphical view of the boundary representation. Similarly, we can add more boundaries and dependencies mapping in the ontology.

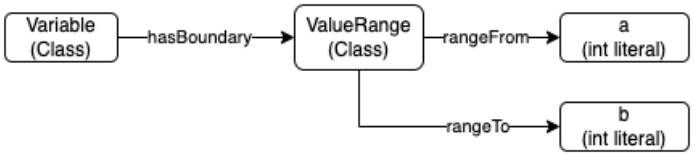


Figure 7.2: Boundary representation in ontology

Dependencies

In addition to the boundaries that describe the safe behavior of the components, it may also have dependencies on each other. For example, if the water level in the tank is maximum, then the pump should be off. This can be formalized in ontology as

$$waterLevel(Tank1, l1) \wedge l1 > levelThreshold \rightarrow maxWaterLevel(Tank1, `true`)$$

$$maxWaterLevel(Tank1, `true`) \rightarrow isRunning(Pump1, `false`)$$

where waterLevel is a property that maps the Tank class with integer values, levelThreshold is the maximum permissible water level for a tank, maxWaterLevel maps the status of the Tank and isRunning maps the running status of the Pump.

7.4 Case Study

7.4.1 Preliminaries

OWL-API

OWL is a computational logic-based language designed to represent rich and complex knowledge about things and the relationships between them. OWL-API [48] was originally developed in 2003 to support the creation and manipulation of the OWL ontology. OWL-API was designed to support the manipulation of the T-Box, that is, schema-level ontologies. It is implemented in Java and is available open source. One of the advantages of OWL-API is that we can use the in-memory ontology together with the ontology used in the database.

UPPAAL-API

UPPAAL-API [70] is an API for the UPPAAL model checker implemented in Java for the creation, manipulation, and simulation of UPPAAL models. The API reads the model written in XML files and creates an object of the UppaalSystem class provided by UPPAAL. It provides two essential packages: (1) engine contains main model checking engine as well as the model simulation engine and (2) model provides the classes, methods, and data structures that are required to represent the model of a system.

7.4.2 Ontology and UPPAAL Model Simulation Integration

For program logic testing, we simulate the behavior of the PLC program in a model checker. For the dynamic program simulation, we use UPPAAL simulator that is available with the UPPAAL model checker. To have more flexibility in analysis and to have automatic verification implementation, we used the API UPPAAL has provided.

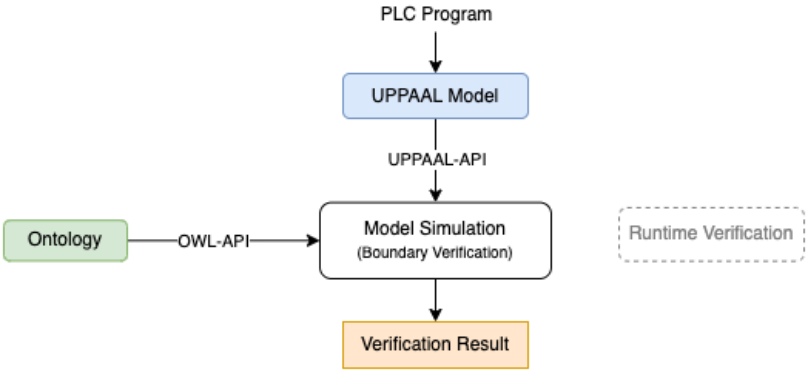


Figure 7.3: Overall Implementation

Figure 7.3 shows that overall implementation of our approach. OWL-API is used to access information such as boundary values from the ontology. UPPAAL-API is used to access and simulate the UPPAAL TA model. The boundary verification is performed while simulating the model.

Boundary Verification

Algorithm 2 shows how the boundary information is extracted from the ontology and used for logic testing in model simulation. First, all variables and components whose boundaries need to be tested are initialized in a list. After that, the ValueRange class from the ontology is accessed to assign the range for the variable in simulation.

Then, a model simulation is performed where in each simulation loop the variables are checked if it is within the boundary. If it gets out of scope, the simulation prints out warnings about the violation. This algorithm is implemented in Java and can be found on Github [71].

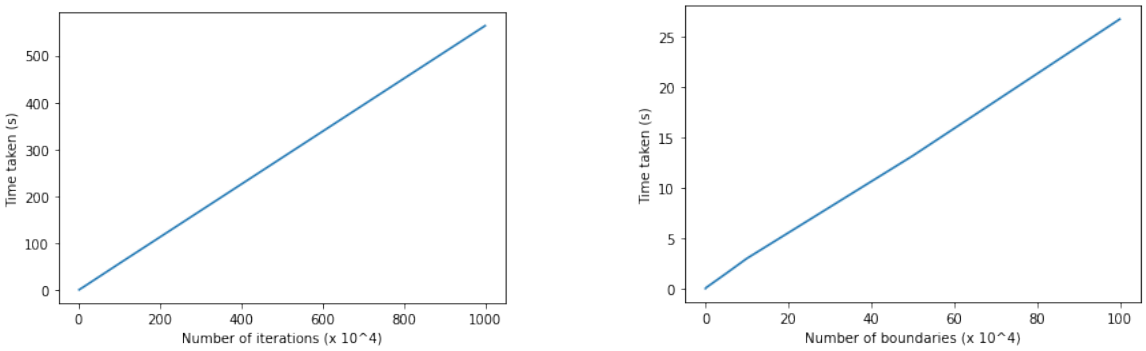
7.5 Results and Discussions

```
BoundaryA = [50, 1000]
BoundaryB = [50, 1000]
BoundaryC = [70, 1000]
13151, t ≥ 6, chemicalProcess: C8 → C9, mixer: running → idle, Warnings: C = 68,
13152, t ≥ 10, chemicalProcess: C9 → C10, outtakePump: idle → running, Warnings: C = 68,
13153, t ≥ 10, chemicalProcess: C10 → C11, outtakePump: running → idle, Warnings: C = 68,
13154, t ≥ 15, chemicalProcess: C11 → C0, Warnings: C = 68,
13155, t = 0, chemicalProcess: C0 → C0, fillTank: idle → RT, Warnings: C = 68,
13156, t = 0, fillTank: RT → refillC, Warnings: C = 68,
21044, t ≥ 6, chemicalProcess: C8 → C9, mixer: running → idle, Warnings: C = 68,
21045, t ≥ 10, chemicalProcess: C9 → C10, outtakePump: idle → running, Warnings: C = 68,
21046, t ≥ 10, chemicalProcess: C10 → C11, outtakePump: running → idle, Warnings: C = 68,
21047, t ≥ 15, chemicalProcess: C11 → C0, Warnings: C = 68,
21048, t = 0, chemicalProcess: C0 → C0, fillTank: idle → RT, Warnings: C = 68,
21049, t = 0, fillTank: RT → refillC, Warnings: C = 68,
35015, t ≥ 6, chemicalProcess: C8 → C9, mixer: running → idle, Warnings: C = 68,
35016, t ≥ 10, chemicalProcess: C9 → C10, outtakePump: idle → running, Warnings: C = 68,
35017, t ≥ 10, chemicalProcess: C10 → C11, outtakePump: running → idle, Warnings: C = 68,
35018, t ≥ 15, chemicalProcess: C11 → C0, Warnings: C = 68,
35019, t = 0, chemicalProcess: C0 → C0, fillTank: idle → RT, Warnings: C = 68,
35020, t = 0, chemicalProcess: C0 → C1, Warnings: C = 68,
35021, t = 0, fillTank: RT → refillC, Warnings: C = 68,
------------------------------------------------
```

Figure 7.4: Simulation Execution

Figure 7.4 shows the result of one of the simulation executions that we performed. We used the same UPPAAL model that was discussed in the case study section of Part I. We can see that the boundary violation for C starts at the program iteration 13151. At this time, the model transition of the chemicalprocess model is from C8 to C9 and the state of the mixer component changes from the running state to the idle state. The value of C is 68, which is outside the boundary of C, that is, $[70, 1000]$. We can also say that the violation continues for five more iterations. After that, the violation again occurred in the iterations 21044 and 35015 of the program.

From this execution example we can see that the model goes into an unexpected



(a) vs no. of simulation iterations **(b) vs no. of boundary values**

Figure 7.5: Timing Study for performance evaluation

state; that is, the system can have a quantity of chemicals outside the safety boundary. In this way, the control logic modeled in UPPAAL TA is verified against the boundaries present in the ontology.

In this experiment, we limited the number of iterations to 50,000 for the simulation of the UPPAAL model. We performed the timing study by changing the number of iterations and the number of boundary conditions shown in Figure 7.5a and Figure 7.5b. As seen in the graph, the time complexity is linear, which makes it fairly scalable.

7.6 Conclusion and Future Works

This paper proposed an ontology-based framework, containing safety and security knowledge of components and variables, for static boundary verification of PLC. These types of fast, lightweight formal verification tools and algorithms will help technicians and engineers to test their control logic at the early stages of the design and development to ensure safety and security properties are satisfied in the safety-critical processes.

At this point, only the boundary values of the variables are used in the testing to illustrate our method. In this work, we provide a foundational tool for the integration of the formal UPPAAL model checker and the OWL ontology. These tools can be further developed to enrich the available testing and verification toolchains.

REFERENCES

[1] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," ACM Trans. Softw. Eng. Methodol., vol. 5, pp. 231–261, July 1996.

[2] Xiaoshan Li, Zhiming Liu, and Jifeng He, "Consistency checking of UML requirements," in 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05), pp. 411–420, June 2005.

[3] B. Kamsu-Foguem, F. H. Abanda, M. B. Doumbouya, and J. F. Tchouanguem, "Graph-based ontology reasoning for formal verification of BREEAM rules," Cogn. Syst. Res., vol. 55, pp. 14–33, June 2019.

[4] B. F. Adiego, E. B. Viñuela, and A. Merezhin, "Testing & verification of PLC code for process control," Oct. 2013.

[5] J. Song, E. Jee, and D.-H. Bae, "FBDTester 2.0: Automated test sequence generation for FBD programs with internal memory states," Science of Computer Programming, vol. 163, pp. 115–137, Oct. 2018.

[6] S. H. Lee, S. J. Lee, J. Park, E.-C. Lee, and H. G. Kang, "Development of simulation-based testing environment for safety-critical software," 2018.

[7] S. Richter and J. U. Witiig, "Verification and validation process for safety I&C systems," Nucl. Plant J., vol. 21, no. 3, pp. 36–+, 2003.

[8] D. Darvas, I. Majzik, and E. B. Viñuela, "PLC program translation for verification purposes," Periodica Polytechnica Electrical Engineering and Computer Science, vol. 61, pp. 151–165, May 2017.

[9] S. Kottler, M. Khayamy, S. R. Hasan, and O. Elkeelany, "Formal verification of ladder logic programs using NuSMV," 2017.

[10] R. Sun, A. Mera, L. Lu, and D. Choffnes, "SoK: Attacks on industrial control logic and formal Verification-Based defenses," June 2020.

[11] O. Ljungkrantz, K. Åkesson, M. Fabian, and C. Yuan, "A formal specification language for PLC-based control logic," in 2010 8th IEEE International Conference on Industrial Informatics, pp. 1067–1072, July 2010.

[12] M. Rausch and B. H. Krogh, "Formal verification of PLC programs," 1998.

[13] D. Soliman and G. Frey, "Verification and validation of safety applications based on PLCopen safety function blocks using timed automata in uppaal," 2009.

[14] A. Al Farooq, E. Al-Shaer, T. Moyer, and K. Kant, "IoTC2: A formal method approach for detecting conflicts in large scale IoT systems," in 2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), pp. 442–447, Apr. 2019.

[15] F. Baader, A. Bauer, and M. Lippmann, "Runtime verification using a temporal description logic," in Frontiers of Combining Systems, pp. 149–164, Springer Berlin Heidelberg, 2009.

[16] J.-b. Gao, B.-w. Zhang, X.-h. Chen, and Z. Luo, "Ontology-based model of network and computer attacks for security assessment," Journal of Shanghai Jiaotong University (Science), vol. 18, no. 5, pp. 554–562, 2013.

[17] Y. Wu, R. A. Gandhi, and H. Siy, "Using semantic templates to study vulnerabilities recorded in large software repositories," in Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems, pp. 22–28, 2010.

[18] J. Undercoffer, A. Joshi, and J. Pinkston, "Modeling computer attacks: An ontology for intrusion detection," in International Workshop on Recent Advances in Intrusion Detection, pp. 113–135, Springer, 2003.

[19] H. Knublauch, M. Horridge, M. A. Musen, A. L. Rector, R. Stevens, N. Drummond, P. W. Lord, N. F. Noy, J. Seidenberg, and H. Wang, "The protege OWL experience," in OWLED, 2005.

[20] B. Parsia and E. Sirin, "Pellet: An owl dl reasoner," in Third international semantic web conference-poster, vol. 18, p. 13, Citeseer, 2004.

[21] D. L. McGuinness, F. Van Harmelen, and Others, "OWL web ontology language overview," W3C recommendation, vol. 10, no. 10, p. 2004, 2004.

[22] M. J. O'Connor and A. Das, "The SWRLTab: An extensible environment for working with SWRL rules in Protégé-OWL," Jan. 2006.

[23] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL—a tool suite for automatic verification of real-time systems," in International hybrid systems workshop, pp. 232–243, 1995.

[24] Y. Yin, K. E. Stecke, and D. Li, "The evolution of production systems from industry 2.0 through industry 4.0," Int. J. Prod. Res., vol. 56, pp. 848–861, Jan. 2018.

[25] K. E. Hemsley and D. R. E. Fisher, "History of industrial control system cyber incidents," tech. rep., Idaho National Laboratory, Dec. 2018.

[26] G. M. Makrakis, C. Kolias, G. Kambourakis, C. Rieger, and J. Benjamin, "Industrial and critical infrastructure security: Technical analysis of Real-Life security incidents," IEEE Access, vol. 9, pp. 165295–165325, 2021.

[27] M. Yu, Z. Wang, and X. Niu, "Verifying service choreography model based on description logic," Math. Probl. Eng., vol. 2016, Jan. 2016.

[28] J. F. Baget, M. Chein, M. Croitoru, and others, "Logical, graph based knowledge representation with CoGui," GAOC: Graphes et, 2010.

[29] M. Croitoru, Graph based knowledge representation and reasoning: Practical AI applications. PhD thesis, Université Montpellier 2, Nov. 2014.

[30] S. Guo, M. Wu, and C. Wang, "Symbolic execution of programmable logic controller code," in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, (New York, NY, USA), pp. 326–336, Association for Computing Machinery, Aug. 2017.

[31] S. C. Park, C. M. Park, G. Wang, J. Kwak, and S. Yeo, "PLCStudio: Simulation based PLC code verification," in 2008 Winter Simulation Conference, pp. 222–228, Dec. 2008.

[32] L.-J. Koo, C. M. Park, C. H. Lee, S. Park, and G.-N. Wang, "Simulation framework for the verification of PLC programs in automobile industries," Int. J. Prod. Res., vol. 49, pp. 4925–4943, Aug. 2011.

[33] G. Palshikar and K. V. Nori, "Formal specification of PLC programs using temporal logic," Dec. 1994.

[34] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso, "Model checking early requirements specifications in tropos," in Proceedings Fifth IEEE International Symposium on Requirements Engineering, pp. 174–181, Aug. 2001.

[35] P. Filipovikj, M. Nyberg, and G. Rodriguez-Navas, "Reassessing the pattern-based approach for formalizing requirements in the automotive domain," in 2014 IEEE 22nd International Requirements Engineering Conference (RE), pp. 444–450, Aug. 2014.

[36] E. M. Clarke, "Model checking," in Foundations of Software Technology and Theoretical Computer Science, pp. 54–56, Springer Berlin Heidelberg, 1997.

[37] N. Nourollahi, "Verification and TCTL model checking of Real-Time systems on timed automata and timed kripke structures, and inductive conversion issues in dense time,"

[38] H. Bel Mokadem, B. Bérard, V. Gourcuff, O. De Smet, and J. Roussel, "Verification of a timed multitask system with uppaal," IEEE Trans. Autom. Sci. Eng., vol. 7, pp. 921–932, Oct. 2010.

[39] X. Lin, H. Zhang, and M. Gu, "OntCheck: An Ontology-Driven static correctness

checking tool for Component-Based models," J. Appl. Math., vol. 2013, Apr. 2013.

[40] I. Friedberg, K. McLaughlin, P. Smith, D. Laverty, and S. Sezer, "STPA-SafeSec: Safety and security analysis for cyber-physical systems," Journal of Information Security and Applications, vol. 34, pp. 183–196, June 2017.

[41] S.-Å. Tärnlund, "Horn clause computability," BIT, vol. 17, pp. 215–226, June 1977.

[42] P. Haase and J. Völker, "Ontology learning and reasoning — dealing with uncertainty and inconsistency," in Lecture Notes in Computer Science, Lecture notes in computer science, pp. 366–384, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[43] R. Alur, C. Courcoubetis, and D. Dill, "Model-Checking in dense Real-Time," Inform. and Comput., vol. 104, pp. 2–34, May 1993.

[44] M. Kanovich, T. B. Kirigin, V. Nigam, A. Scedrov, and C. Talcott, "Discrete vs. dense times in the analysis of Cyber-Physical security protocols," in Principles of Security and Trust, pp. 259–279, Springer Berlin Heidelberg, 2015.

[45] R. Alur and D. L. Dill, "A theory of timed automata*," Theoretical Computer Science, vol. 126, pp. 183–235, 1994.

[46] N. Mahmud, C. Seceleanu, and O. Ljungkrantz, "Specification and semantic analysis of embedded systems requirements: From description logic to temporal logic," in Software Engineering and Formal Methods, pp. 332–348, Springer International Publishing, 2017.

[47] "Velocio.net." https://velocio.net/. Accessed: 2022-5-9.

[48] M. Horridge and S. Bechhofer, "The OWL API: A java API for OWL ontologies," Semant. Web, vol. 2, no. 1, pp. 11–21, 2011.

[49] "What is an ontology and why we need it." https://protege.stanford.edu/publications/ontology_development/ontology101-noy-mcguinness.html. Accessed: 2022-3-1.

[50] S. E. Valentine, PLC code vulnerabilities through SCADA systems. PhD thesis, University of South Carolina, 2013.

[51] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "Uppaal SMC tutorial," Int. J. Softw. Tools Technol. Trans., vol. 17, pp. 397–415, Aug. 2015.

[52] N. Dalwadi, B. Nagar, and A. Makwana, "Performance evaluation of semantic reasoners," in Proceedings of the 19th International Conference on Management of Data, pp. 109–112, 2013.

[53] I. Friedberg, K. McLaughlin, P. Smith, D. Laverty, and S. Sezer, "Stpa-safesec: Safety and security analysis for cyber-physical systems," Journal of information security and applications, vol. 34, pp. 183–196, 2017.

[54] N. Leveson, "A new accident model for engineering safer systems," Safety science, vol. 42, no. 4, pp. 237–270, 2004.

[55] N. G. Leveson, Engineering a safer world: Systems thinking applied to safety. The MIT Press, 2016.

[56] T. V. N. Nguyen and F. Irigoin, "Efficient and effective array bound checking," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 27, no. 3, pp. 527–570, 2005.

[57] N. Hasabnis, A. Misra, and R. Sekar, "Light-weight bounds checking," in Proceedings of the Tenth International Symposium on Code Generation and Optimization, pp. 135–144, 2012.

[58] V. Markstein, J. Cocke, and P. Markstein, "Optimization of range checking," in Proceedings of the 1982 SIGPLAN symposium on Compiler Construction, pp. 114–119, 1982.

[59] P. SPEET and E. Goroii, "Hardware support and code generation for dynamic range checking in c,"

[60] A.-Y. Turhan, "Introductions to description logics – a guided tour," 2013.

[61] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," International journal on software tools for technology transfer, vol. 1, no. 1, pp. 134–152, 1997.

[62] A. Aggarwal and P. Jalote, "Integrating static and dynamic analysis for detecting vulnerabilities," in 30th Annual International Computer Software and Applications Conference (COMPSAC'06), vol. 1, pp. 343–350, IEEE, 2006.

[63] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos, "Combining static and dynamic analysis for the detection of malicious documents," in Proceedings of the Fourth European Workshop on System Security, pp. 1–6, 2011.

[64] M. Alvares, T. Marwala, and F. B. de Lima Neto, "Applications of computational intelligence for static software checking against memory corruption vulnerabilities," in 2013 IEEE Symposium on Computational Intelligence in Cyber Security (CICS), pp. 59–66, Apr. 2013.

[65] C. Zimmer, B. Bhat, F. Mueller, and S. Mohan, "Time-based intrusion detection in cyber-physical systems," in Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems, pp. 109–118, 2010.

[66] X. Zheng, C. Julien, M. Kim, and S. Khurshid, "Perceptions on the state of the art in verification and validation in cyber-physical systems," IEEE Systems Journal, vol. 11, no. 4, pp. 2614–2627, 2015.

[67] E. R. Griffor, C. Greer, D. A. Wollman, M. J. Burns, et al., "Framework for cyber-physical systems: Volume 1, overview," 2017.

[68] S. Vasanthapriyan, J. Tian, D. Zhao, S. Xiong, and J. Xiang, "An ontology-based knowledge management system for software testing," in SEKE, pp. 230–235, 2017.

[69] É. F. d. Souza, R. d. A. Falbo, and N. L. Vijaykumar, "ROoST: Reference ontology on software testing," Appl. Ontol., vol. 12, pp. 59–90, Mar. 2017.

[70] "Java API :: UPPAAL documentation." `https://docs.uppaal.org/toolsandapi/javaapi/`. Accessed: 2022-3-31.

[71] R. Neupane, "testmerge: Research implementation." `https://github.com/codeezer/testMerge`.