AN ONTOLOGY-BASED FRAMEWORK FOR FORMAL VERIFICATION OF SAFETY AND SECURITY PROPERTIES OF CONTROL LOGICS

Ramesh Neupane
Department of Computer Science
Boise State University
Boise, USA
rameshneupane@u.boisestate.edu

Hoda Mehrpouyan
Department of Computer Science
Boise State University
Boise, USA
hodamehrpouyan@boisestate.edu

2022 14th International Conference on Electronics, Computers and Artificial Intelligence (ECAI) | 978-1-6654-9535-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/ECAI54874.2022.9847508

***Abstract*—Any safety issues or cyber attacks on an Industrial Control Systems (ICS) may have catastrophic consequences on human lives and the environment. Hence, it is imperative to have resilient tools and mechanisms to protect ICS. To verify the safety and security of the control logic, complete and consistent specifications should be defined to guide the testing process. Second, it is vital to ensure that those requirements are met by the program control algorithm.**

In this paper, we proposed an approach to formally define the system specifications, safety, and security requirements to build an ontology that is used further to verify the control logic of the PLC software. The use of ontology allowed us to reason about semantic concepts, check the consistency of concepts, and extract specifications by inference. For the proof of concept, we studied part of an industrial chemical process to implement the proposed approach.

The experimental results in this work showed that the proposed approach detects inconsistencies in the formally defined requirements and is capable of verifying the correctness and completeness of the control logic. The tools and algorithms designed and developed as part of this work will help technicians and engineers create safer and more secure control logic for ICS processes.

I. 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.

The goal of this paper is to design and develop algorithms and tools to improve the safety and security of Industrial Control Systems (ICS), which are complex and highly interconnected software and hardware systems. These systems are considered critical and essential for the well-being of society. Therefore, any type of issue or cybersecurity threat can result in significant destruction, affecting millions of people. Considering the severity of the consequences, it is essential to develop an understanding of how to integrate safety and security requirements into control software. We answer this question by developing 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 called reasoner to verify the consistencies and completeness of the specifications.

The contradictory nature of the safety/real-time properties and security needs of the system might cause several vulnerabilities in the system. Detecting such conflicts early increases both the safety and security of the system [1], [2].

Our contribution is to design and develop an ontology that contains the essential information about the processes that the control logic ought to manage and its safety and security requirements. The ontology acts as a knowledge base for the verification of requirements. Two operational algorithms have been developed within this approach to define safety/security concepts and consistent requirements to create an ICS ontology. Second, a 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, which is extracted from the PLC program and modeled as a Timed Automaton (TA), along with the requirements from the ontology that are translated into Timed Computational Tree Logic (TCTL), is tested to ensure safety and security properties hold.

As part of this work, the design and development process of the ontology is explained and implemented by performing a detailed analysis of the chemical process using a popular ontology editor, Protégé [3]. The DL-reasoner Pellet [4] was utilized to check the consistency and completeness of the safety and security requirements in the ontology. We were able to show that OWL-DL [5] along with the Semantic Web Rule Language (SWRL) rules [6] can be used for the formalization of the requirements and specification of the chemical process. We transform the rules expressed in SWRL into TCTL queries.

This material is based upon work supported by the National Science Foundation Computer and Information Science and Engineering (CISE) devision, award number 1846493 of the Secure and Trustworthy Cyberspace (SaTC) program: Formal TOols foR SafEty aNd. Security of Industrial Control Systems (FORENSICS).

978-1-5386-5541-2/18/$31.00 ©2022 IEEE

The rest of the paper is organized in the following way. Section 2 gives an overview of the related works; Section 3 describes the industrial process we are using to illustrate our ideas and methodology; Section 4 provides the overall methodology including the design and development of the ontology followed by the model-checking approach; Section 5 presents the case study process to illustrate our methodology with our observation; and finally Section 6 concludes the paper in addition to our future work.

II. Related Work

Consistency checking of requirement specifications is not a new idea [7]. Other domains in addition to software engineering have studied consistency check of specifications, for example Kamsu-Foguem et al. [8] proposed a conceptual graph-based framework for formally checking the compliance of construction buildings to verify that they satisfy certain standards or regulations.

Functional testing of control logic modules with manually generated input test cases is the most commonly used technique in industrial process control applications [9], [10]. Test cases are designed based on previous safety and security failure experiences, reports, and the expertise of the tester [11]. 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 [9], [12].

To our knowledge, PLC-PROV [13] is the only work that utilizes the provenance of the system data to detect security and safety violations. However, the proposed provenance graph is limited to safety components that create the causal dependency chart between sensor inputs and actuator outputs through the PLC program based on program execution traces. In our proposed approach, we are designing and developing an ontology-based knowledge base that contains the relationship between both the security and safety components of the PLC program elements. Furthermore, the ontology design is based on the prior incident records. Thus, we are able to increase testing coverage to both security and safety components, as well as overall industrial processes.

III. Chemical Process Under Study

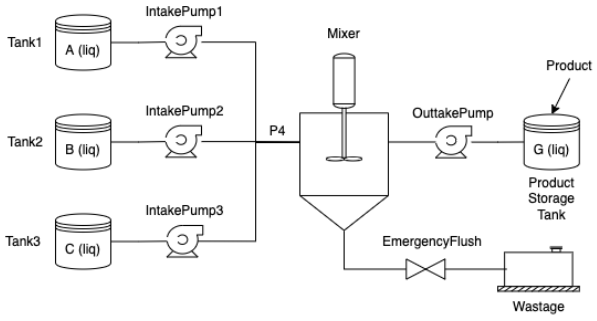


Fig. 1: Chemical Process

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 outflow pumps, a mixer and a flush as shown in Figure 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 empty product storage tank with the help of the pump *OuttakePump*. The user input *RunProcess* starts and stops the chemical process, whereas the *EmergencyStop* switch resets the system and runs the emergency flush valve for 5 seconds.

IV. Methodology

This section provides an overview and the motivation of our proposed methodology to verify that the logic of the PLC program satisfies the required safety and security properties. The general outline of the methodology is shown in Figure 2.

In the first section of the methodology, an ontology based on DL is designed and developed as a mathematical foundation. The proposed ontology is designed to include 1 safety / security concepts and requirements, 2 system components involved in the control logic, 3 boundary values and conditions, and 4 PLC program concepts. With the help of an ontology reasoner, we check the 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 using 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, 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 in Sections 5 and 6.

A. Ontology

In order to build the proposed ontology in Figure 3, detailed information is required about the industrial processes that are under study. Domain knowledge is first formalized based on the formal language, OWL-DL [5], and then added to a knowledge base. OWL-DL is a rich tool set available for creating/editing an ontology while being able to check the

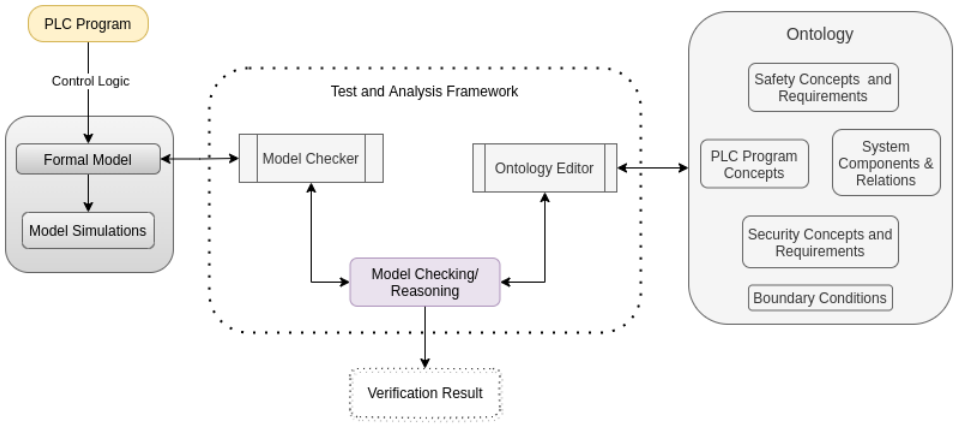


Fig. 2: Overall Methodology

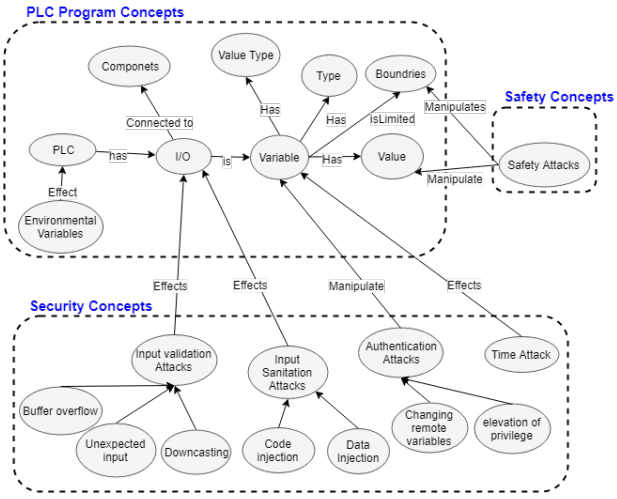


Fig. 3: PLC concepts, safety and security ontology

consistency of its components. In addition to that, it is also decidable with maximum expressiveness [14].

In OWL, the class has a similar meaning as **concept** in Description Logic (DL), and the property represents **role**. The first step in implementing 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 is also called Assertion Boxes (ABoxes). Thus, the knowledge base (KB) is a pair (T, A), where T is a $TBox$ and A is a $ABox$. 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 and how ontology provides consistency and completeness checking via reasoning.

1) System and Processes Ontology: Using DL, components are defined as a hierarchical class with the help of a subsumption relation. For example, Table I 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 to define 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 explain and assign a literal value to an entity. The relationship can be formalized by adding them as a role that relates 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 acceptable range. For example, 'Pumps pumps out chemicals' is represented as $pumpsOut(Pump, Chemical)$, where the property of the relationship *pumpsOut* describes the relationship between the Pump and the chemical. Similarly, *"The tank has a capacity of 2500 gallons."* is represented as $hasCapacity(Tank, 2500)$, where *hasCapacity* is a data property and is related between the class "tank" and the literal integer value "2500".

	Class Definition	Formal Definition (DL)
1	Pump is a component.	Pump $\sqsubseteq$ Component
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 I: Hierarchical class definition of ICS components

2) Safety and Security Ontology: To develop a safe system, we first need to analyze the hazards and 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 [15].

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.

$$InputTank(t1) \equiv Tank(t1) \sqcap \forall containsMaterial(t1, m1) \sqcap InputMaterial(m1) \quad (1)$$

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 belong to a certain class. For example, a tank can contain only one material and can be formalized as

$$Tank(t1) \sqcap \leq 1\, containsMaterial(t1, m1) \sqcap Material(m1) \quad (2)$$

Similarly, the security ontology contains security concepts specific to ICS. Sources of these general concepts are the ICS security experts, known vulnerabilities, previous reported attacks, etc. The general security concepts could be formally defined in DL as subsumption relations. For example, the "input validation attack is an attack" can be formalized as $InputValidationAttack \sqsubseteq Attack$. Similarly, the concept *"Buffer overflow is an input validation attack"* can be formalized as $BufferOverflow \sqsubseteq InputValidationAttack$. Compound concepts such as "Overwrite switch can be remotely manipulated so it is considered an attack vector" and are formalized as a collection of logical statements,

$$RemoteControlSwitch(s1) \equiv Switch(s1) \sqcap \forall hasRemoteControl(s1) \quad (3)$$

$$RemoteControlSwitch \sqsubseteq AttackVector \quad (4)$$

$$OverwriteSwitch \sqsubseteq RemoteControlSwitch \quad (5)$$

From the previous three statements, the logical statement

$$OverwriteSwitch \sqsubseteq AttackVector \quad (6)$$

could be inferred. Constraints and requirements in ontology can also be represented with another subset of predicate logic with efficient proof systems, known as horn logic [16]. Horn logic rules, also called horn-clauses, are used in ontology to provide more dynamism to the DL based ontology. Semantic Web Rule Language (SWRL) [6] is the rule language that includes an abstract syntax for horn-like rules. For example: if we have a requirement, "Overwrite switch is an attack vector, so, the user should be logged in to enable those switch". Since we have already shown how to formalize the first part of the requirement in the previous paragraph, the second part can be formalized in SWRL rules as $OverwriteSwitch(?switch) \wedge hasEnabled(?user, ?switch) \wedge LoggedInUser(?user)$.

B. Reasoning in Ontology

1) Consistency Checking: An ontology is inconsistent if there exists an instance of either class or a property that contradicts the axioms of the ontology. If there is a logical contradiction in an ontology, the ontology becomes meaningless, since any form of statement could be derive from a set of logical axioms that contradict each other [17]. Formally, an ontology is said to be inconsistent if an axiom in the ontology is unsatisfiable. For example, assertions such as EmergencyFlush: Pump, EmergencyFlush: Variable causes inconsistency in the ontology, because the name is considered unique in the ontology. Hence, an ontology model,

$$M \nvDash (EmergencyFlush : Pump \wedge EmergencyFlush : Variable). \quad (7)$$

Moreover, lets consider the clauses, "OuttakePump is running" i.e. s1 = isRunning(OuttakePump, 1) and "OuttakePump is not running" i.e. s2 = isRunning(OuttakePump, 0),

$$M \nvDash (s1 \wedge s2). \quad (8)$$

2) 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 if 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."* that is, $LoggedInUser(user1) \wedge switchAccess(user1, true) \wedge enableSwitch(true, user1)$. This cannot be complete if there are assertions $User(user1)$, and axiom $LoggedInUser \sqsubseteq User$ not present in the ontology.

C. 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 [18] in which system descriptions are specified in Timed Automaton (TA), while requirements are expressed based on Timed Computational Tree Logic (TCTL) formulas. The TCTL model checking problem is P-SPACE complete [19]. In the following section, we give the formal definitions of the TA and TCTL.

1) Timed Automaton: TA [18] is an extension of a finite automaton with a finite set of real-valued variables called *clocks* to specify constraints of time between two events. If X = $\{x_1, x_2, \ldots, x_n\}$ is a finite set of clocks, then a *clock valuation* is a mapping v: X $\in \mathbb{R}^X$ and $\phi(X)$ represent the set of formulas called clock constraints. For example, Figure 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 the clock variable x is 5.3 and y is 3.7 at the location l_2.

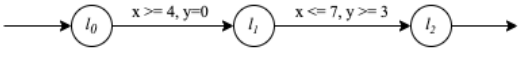


Fig. 4: Timed Automaton

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

2) Timed Computational Tree Logic: TCTL is a timed extension of the 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 defined inductively via the following production rule.

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

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 use "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 and can be formally expressed as (M, s_0) $\vDash \phi$. For example: If p and q are the local atomic formulas, then

Invariance: $s_0 \vDash A\ G_{[2,3]}\ p$, implies that p is true for all possible future paths between states s_{0+2} and s_{0+3}, shown in Figure 5a.

The 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 5b.

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

Fig. 5: TCTL Properties

3) Transformation to TCTL: There has been some work in translating specifications in ontology into TCTL using the Specification Pattern System (SPS), which is a set of recurring patterns of functional and time requirements [20]. In this work, for simplicity, we only consider the conversion of requirements represented in the SWRL rules. The main reason for this conversion is that we need to verify the requirements using a model checker that only supports temporal logic. The advantages of this conversion are 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 the ontology and the reasoner.

Algorithm 1 shows the generation of a temporal formula in TCTL. The inputs to these algorithms are the collections of 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"*, this 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 splits the clause into a collection of literals. Then, each literal is looked at in a mapping file that contains the relation between data/object properties in 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

```
Get all rules
for rule in rules(r_1, r_2, ..., r_n) do
    literals ← getLiterals(rule);
    q ← getHead(rule);
    for l in literals do
        pArr ← MapOntologyToModelChecker(l);
    end for
    p ← joinLiterals(pArr);
    tctl ← addTemporalOperators(tctl);
    tctls[0, 1, ..i] ← E◊ (p → q);
end for
```

V. Case Study

For the proof of concept, we take an industrial operational technology (OT) process as we have discussed in Section 3. We used Velocio PLC 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 the Web Ontology Language (OWL) to create an ontology. For the requirements and constraints, the SWRL rules are added to the ontology. Pellet [4] reasoner that works with the ontology is used to infer knowledge based on the existing information on the ontology and to verify the consistency of the ontology. The logic of the PLC program is modeled in timed automata of the UPPAAL model checker. 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 to detect the conflict in requirements, the UPPAAL model of the PLC program control logic, and the verification of the properties done using the UPPAAL Verifier.

A. Ontology with Protégé and OWL-API

Due to the popularity of OWL in the semantic web, many tools are available to edit and build an ontology. Prot'eg'e is one of the tools that is open source and contains a reasoning and inference engine that supports the validation and verification of DL queries. To build an ontology, we used Prot'eg'e as a GUI tool and OWL-API for automating ontology access, i.e., update, and delete. OWL-API is a Java API implementation that can be used to create, manipulate, and serialize OWL ontologies.

In Prot'eg'e, we can create 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 properties involved in the chemical process. Table II shows examples of information that are used to build the ontology. In addition, the requirements and constraints are added through the SWRL queries in Protégé.

1) Safety and Security Ontology: To create the safety and security ontology, we review previous work and identify some important security / safety concepts pertinent to PLC and ICS in general. The remainder 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 immediately insertion [21]. Therefore, the security requirement for this case would be *"Do not leave unused variable in the PLC program."*. In PLC program, each variable is either assigned to input/output components or used in the program for temporary program operation, if the variable does not affect on the control logic, they are unused variables and are vulnerable to attack. These types of inference are automatically done by the DL-reasoner based on the already added information.

Similarly, hard-coded numeric values in PLC programs can be vulnerable to attack by allowing the number to be changed directly [21]. The security requirement for this case could be *"Avoid using hardcoded numeric values in PLC programs."*. This requirement is 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 1 has a threshold flow rate of 6 liters per second, so running all together will exceed the threshold, which might cause a serious pipe burst accident. Another important safety constraint is that if the mixer is running empty, it abruptly increases the temperature of the mixer, which could eventually lead to an explosion. This can be written as *"The mixer should not run if the chemicals are not present."*.

Detecting Inconsistencies: One of the main reasons for formalizing specifications and requirements in the DL-based ontology is that it provides automatic classification and inconsistency verification. 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 the logged-in 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:

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

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

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

$$LRSwitch(?s) \wedge enableSwitchWithin(?s,?t1) \wedge LoggedInUser(?u)$$

$$\wedge timeRequiredToLogin(?u,?t2) \wedge t1 <= t2 \longrightarrow Violation(R1)$$

After adding these rules and specifications, the reasoner is synchronized in Protégé to see if there are inconsistencies or violations. We can see (Figure 6) that there is a violation caused by the rules we have added. We can notice that the reasoner also provides the reason for the violation.

Fig. 6: Checking requirement inconsistency (Protégé)

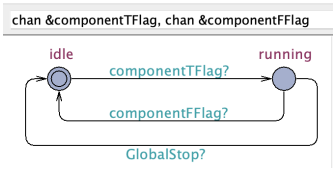
B. Model Checking with UPPAAL

UPPAAL is a toolbox for real-time system verification. 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, such as templates, synchronizations, urgent locations, and expressions such as guard and invariant [22].

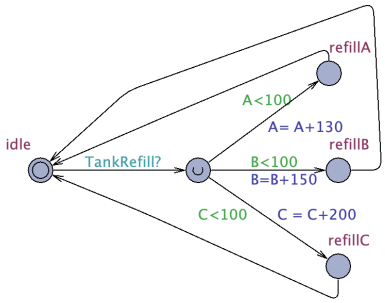
In our case study, that is, in a chemical process, the intake pumps, outtake pump, mixer, and flush can be modeled as a similar timed automaton model. Thus, we created a template that is called component as shown in Figure 7a. The component template has two states {idle, running}. These states can be changed using the variables that are parameters of 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)*. One of

	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

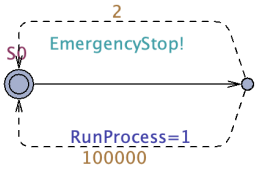
TABLE II: Class and Property definitions



(a) Component Model



(b) Tank Refill Model



(c) User Input Model

Fig. 7: UPPAAL model template

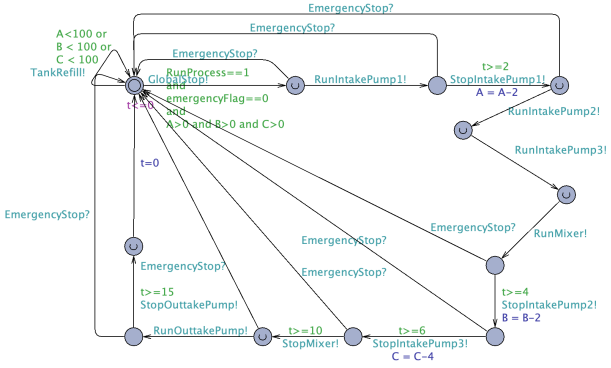


Fig. 8: Overall chemical process (UPPAAL)

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 7b.

As mentioned in the previous section, the requirement *"All three pumps should not run at the same time."* is formalized in the SWRL rules as isRunning(intakePump1, true) $\wedge$ isRunning(intakePump2, true) $\wedge$ isRunning(intakePump3,

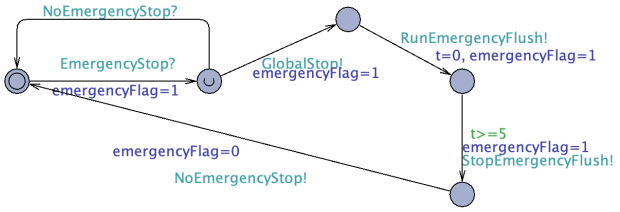


Fig. 9: Emergency case (UPPAAL)

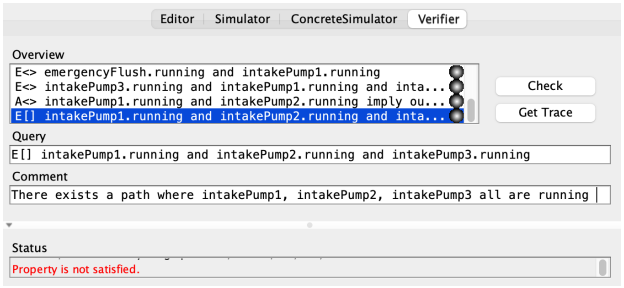


Fig. 10: Requirement verification on UPPAAL (R2)

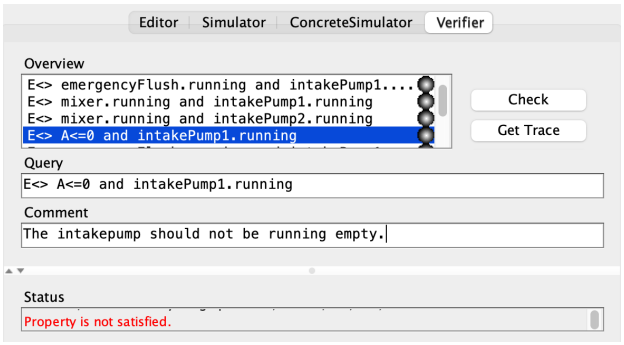


Fig. 11: Requirement verification on UPPAAL (R3)

true) $\rightarrow$ Violation(R2). If there is a case where these three pumps are running simultaneously, then this is a violation of the rule. This rule is translated into TCTL queries with the algorithm (1) implemented in Java. The translation gives us the TCTL formula as 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 10).

Similarly, the SWRL rule $containsAmount(Chemical\_A, ?am) \wedge am < 0 \wedge isRunning(intakepump1, true) \rightarrow Violation(R3)$, explains that the intakepump1 should not run in an empty condition. There should be some chemicals

in the corresponding input tank. This requirement is translated into the TCTL query as $E <> A <= 0\ and\ intakePump1.running$. The model in UPPAAL is then checked (Figure 11). The property is not satisfied, that means there are no path in the state model, where the intakepump1 will be running in an empty condition. 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 to check 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 carried out on a 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 a larger number of logical axioms, the reasoner will take more time for automatic classification and detecting inconsistencies. The time taken increases linearly with the increase in the number of axioms. We can find a detailed empirical analysis by Dalwadi et al. on [23]. Similarly, model checking can also run into undefined states because of its state-space explosion problem as the model becomes complicated.

VI. Conclusion and Future Work

This paper 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 the consistency between the requirements added to the ontology. The created ontology is then used to verify the control logic whether it violates the requirements or not.

Future work will include automatic translation of control logic into formal language. Furthermore, the algorithm presented in this paper to translate requirements from DL or SWRL rules to TCTL queries only works for some limited rules and temporal operators, i.e. SWRL query and E <>. In our future work, we will extend the element of *concept* 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.

References

[1] S. Toole, C. Sewell, and H. Mehrpouyan, "Iot security and safety testing toolkits for water distribution systems," in *2021 8th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pp. 1–8, 2021.

[2] S. Sapkota, A. K. M. N. Mehdy, S. Reese, and H. Mehrpouyan, "Falcon: Framework for anomaly detection in industrial control systems," *Electronics*, vol. 9, no. 8, 2020.

[3] 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.

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

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

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

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.