MBTA: A Model-Based Threat Analysis Approach for Software Architectures

Anas Motii(✉)

Mohammed VI Polytechnic University, Ben Guerir, Morocco
anas.motii@um6p.ma

Abstract. In the last decade, several efforts have been achieved to integrate security in the Software Development Life-cycle (SDL). Analyzing software architecture in order to identify threats is an essential step in secure software development processes. However, performing this task manually can result in identifying false positives. It is thus time-consuming and error-prone. Therefore, there is a need for automated tool support to perform this task. Existing efforts are limited to specific, predefined security properties or threats that are checked either manually or using limited toolsets. In this paper, we present a general and constructive model-based approach for threat analysis. We employ domain-specific modeling language techniques to develop a set of modeling languages that enable the specification of the software architecture structure. We used the Object Constraint Language (OCL) for the purposes of precise specification and verification of security threats as properties of a modeled system. To validate our work, we explore a set of representative threats in the context of SCADA systems.

Keywords: Model-Based · OCL · Security · Threat Analysis · Software architecture

1 Introduction

Our society has become more dependent on software-intensive systems, such as Information and Communication Technologies (ICTs) systems, not only in safety-critical areas but also in areas such as finance, medical information management, and systems using web applications. The complexity of such systems during their design comes from the involvement of trans-disciplinary concerns. In addition, security experts, practitioners and researchers from different international organizations, associations, and academia have agreed that security should be treated in the early stages of the software and systems development life-cycle [8]. Otherwise, security vulnerabilities are more likely to be introduced in various stages and the cost of protecting them becomes increasingly more important. In this context, the use and application of security mechanisms through the life-cycle process would be easier if designers and developers had security guidelines during development. Architecture threat analysis is the process of identifying

J. Guiochet et al. (Eds.): SAFECOMP 2023, LNCS 14181, pp. 121–134, 2023.
https://doi.org/10.1007/978-3-031-40923-3_10

threats to an architecture. It is very useful when it comes to detecting threats at early stages. Reported vulnerabilities show that architecture design weaknesses represent half of the total vulnerabilities of a system. Several efforts have been done to assist threat identification [3]. However, the complexity of systems requires automated tool support.

This work is part of a more general process devoted to incremental pattern-based modeling and safety and security analysis for correct-by-construction systems design. In previous works, an approach and its tool support to support Security, Dependability and Resource Trade-offs using Pattern-based Development and Model-driven Engineering have been presented [5]. In this paper, a Model-Based Threat Analysis approach for software architecture and its tool support is introduced in order to allow automatic threat detection based on the Object Constraint Language (OCL). The remainder of the paper is organized as follows. Section 2 presents the main steps of the MBTA approach. In Sect. 3, the MDE framework supporting MBSPI is presented. The threat formalization process using OCL is explained step by step. Section 4, MBTA is assessed over a SCADA (Supervisory Control and Data Acquisition) system case study. Section 5 identifies related work tackling software architecture threat analysis. Finally, Sect. 6, concludes and sums up the contributions.

2 MBTA Approach

The approach depicted in Fig. 1 allows the analysis of software architectures in order to detect existing threats based on formalization. The first step consists of formalizing threats using OCL[1] from existing threat classification references **(step 0)**. Then, the software architecture model is passed to the analysis module **(step 1)** which outputs existing threats.

Specifying threats is based on experience in the security domain thus this activity should be done by security experts. Once formalized, these threats are stored in a knowledge base. Inputs are existing threat classification references: OWASP[2], STRIDE [6], Common Attack Pattern Enumeration and Classification (CAPEC)[3], Common Weakness Enumeration (CWE)[4]. These references describe informally a set of threats. Each threat has a signature. This signature specifies the conditions in which a threat can occur. Thus it defines the threats according to a certain scenario. However, the threats are described informally, and thus applying them manually is error-prone and time-consuming. The considered threats are discussed below. This is neither a comprehensive nor a complete list but two well-known categories which have been used in OWASP's top 10:

[1] https://www.omg.org/spec/OCL/.
[2] https://owasp.org/www-project-top-ten/.
[3] https://capec.mitre.org/.
[4] https://cwe.mitre.org/.

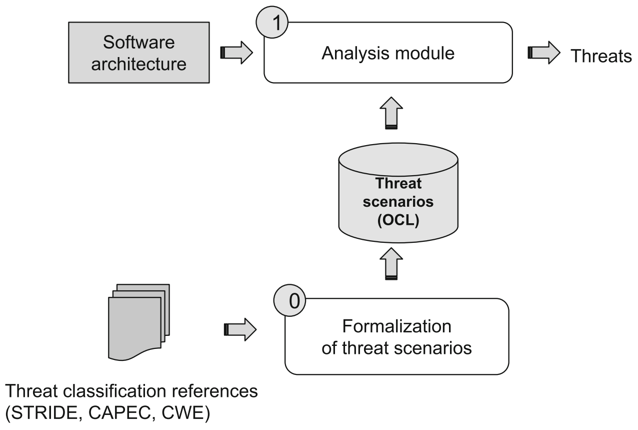
Software architecture
1
Analysis module
Threats
Threat scenarios (OCL)
0
Formalization of threat scenarios
Threat classification references (STRIDE, CAPEC, CWE)

Fig. 1. Threat Analysis Process

- **Man-In-The-Middle (MITM)**: is responsible for relaying or altering messages between two parties. The signature of this threat is the lack and/or weakness of encryption and Authenticity mechanisms.
- **Injection**: is responsible for passing malicious inputs to gain higher privileges, alter data, or crash the system. The signature of this threat is the lack and/or weakness of input validation and the secure development of an application.

OCL is a formal language used to describe rules on UML models. We have used OCL to formalize the aforementioned threats as invariants. The analysis allows the detection of threats over the architecture. If an invariant is violated then the corresponding threat is relevant. In order to evaluate the formalized threats, the precision metric is used. It measures the soundness of the results. A high precision rate means that the detected threats contain more True Positives (TP) i.e., valid results than False Positives (FP) i.e., false results. It is computed as follows:

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

3 Model-Driven Development

We now present the MDE framework supporting the previous approach and we detail its construction from the system architecture and security perspectives. For the system architecture aspects, we used a UML-Like[5] modeling language to describe software architecture using the component-port-connector fashion. The security perspective which consists of three iterations introduces additional architectural elements and uses OCL for the specification and analysis of the security threats.

[5] https://www.omg.org/spec/UML/.

3.1 Modeling the Architecture: ComponentUML

In the context of Component-Based Development (CBD), the UML profile "ComponentUML" in Fig. 2 has been defined in order to model the application. The need to define this profile occurred during OCL formalization using OCL. The OCL rules were difficult when using UML because concepts that were not relevant appeared. Hence this profile was used for a matter of simplification. The UML profile has been defined based on the following concepts: Structured-Classifiers, Messages and Deployments from UML.

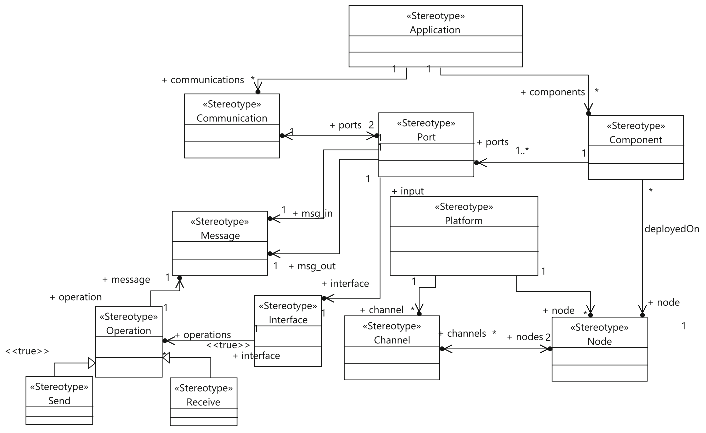

Fig. 2. UML profile for component-based software architectures

Working Example: Metamodel Instantiation. Figure 3 shows the software architecture of a three-tier web application[6]. The architecture consists of three component types: *Page*, *Webapp* and *Database*. Each component is associated with ports, interfaces, data types and messages, accordingly. For instance, a component *webpage* of type Page uses a Port Client Server for the communication with component *webapp* of type *Webapp*. For that, component *webapp* uses a port *Port Server Client*. The comments in blue show the different messages: *m1* to model the request sent to the application, *m2* to model the response from the application, *m3* to model the request sent to the database and m4 to model the response from the database. From the deployment perspective, the underlying platform consists of three nodes: *Browser* hosting *webpage*, *Server* (exposed to Internet) hosting *wepapp* and Back to host database. The software architecture model for the web application has been made intentionally not secure to test the OCL constraints. In fact, the model does not contain any sort of security

[6] https://www.ibm.com/topics/three-tier-architecture.

mechanisms: encryption and input validation. Hence it is vulnerable to injection and MITM attacks. The objective is to detect: one injection and one MITM threat.

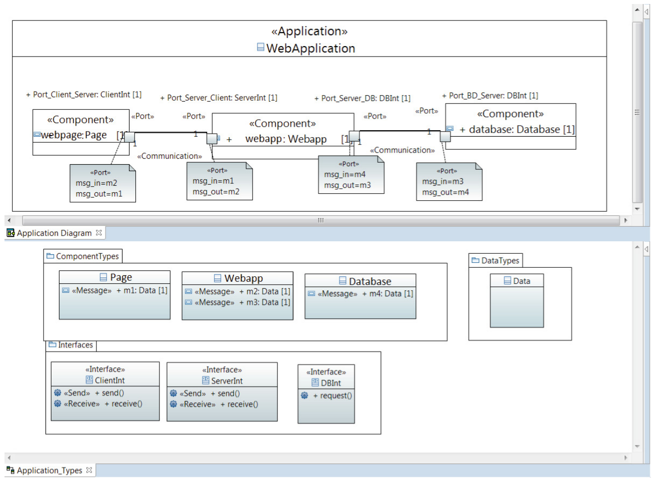

Fig. 3. Web application software architecture model and types (Color figure online)

3.2 Modeling the Security Solutions as Security Patterns

As introduced in Sect. 1, our work is part of a general approach to building secure software at high-level design stage using patterns (PBSE). We developed a UML profile called *SepmUML*, as depicted in Fig. 4 using UML notations (not all classes and attributes are shown on the diagram to avoid cluttering). *SepmUML* contains the necessary stereotypes for modeling a security pattern in UML environments (stereotypes in white). The solution of the security pattern is modeled using ComponentUML (stereotypes in grey). In addition pattern integration-related concepts (stereotypes in blue). The specification of the UML profile is out of scope this paper and is detailed in [5].

In the context of this work, during the formalization process, we considered the following security mechanisms:

- **Firewall**: This mechanism is responsible for input validation.
- **Encryptor**: encrypts transmitted messages using a key.
- **Decryptor**: decrypts received messages using a key.
- **Signer**: produces for each message a signature that guarantees the authenticity and integrity of the message. It is sent together with the message.
- **Verifier**: verifies the integrity and authenticity of the message via its accompanied signature.

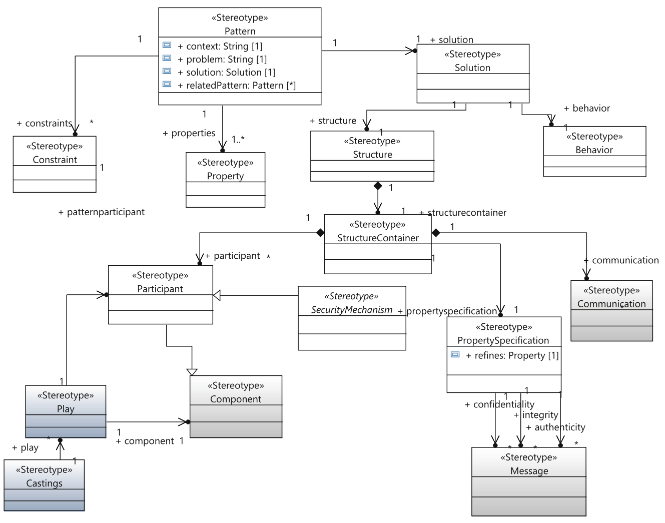

Fig. 4. SepmUML UML profile (Color figure online)

3.3 Formalizing the Threats Using OCL

The main objective of this work is to analyze software architectures allowing the detection of threats according to formalized threats. In this section, the integration-related detecting threats are described. As we shall see, this has required additional concepts in ComponentUML in Fig. 5. At each iteration, threats are formalized using OCL. The evaluation through the precision rate (TP and FP) is measured in the context of the working example from Sect. 3.1. *Iteration 1* starts with the initial ComponentUML. Man-In-The-Middle (MITM) and Injection threats are formalized. *Iteration 2* adds the concept of trust level to ComponentUML. Last but not least, *Iteration 3* adds the concept of port kind is added i.e., if the port is external (public) or internal (private).

Iteration 1. Man-In-The-Middle (MITM) and injection threats are formalized using OCL. MITM exploits the lack of encryption and integrity protection mechanisms. Injection threats exploit the lack of input validation. In Listing 1.1 and Listing 1.2 are given the OCL constraints of MITM and injection, respectively.

```
Context Application inv Man-In-The-Middle_v1
self.components->select(c1 |
self.components->exists(c2 |
not(c1.oclIsKindOf(PatternProfile::SecurityMechanism))
and
not(c2.oclIsKindOf(PatternProfile::SecurityMechanism))
and
/* if c1 and c2 are different*/
c1._'<>'(c2)
and
/* if c1 and c2 are deployed in different nodes*/
(c1.nodlater>'(c2.node) and c1.node.channels->exists(ch | c2.node.
    channels->includes(ch))
and
/* c1 and c2 communicate */
(c1.ports->exists(inp| c2.ports->exists(inpt2 | inpt2.communication
      = inp.communication)))
 and

/* The security mechanisms exist: encryptor, decryptor, signer and
      verifier */
(self.components->select(enc |  self.components->exists(dec , mac1
      | self.components->exists(mac2|

enc.oclIsKindOf(PatternProfile::Encryptor) and  dec.oclIsKindOf(
      PatternProfile::Decryptor) and mac1.oclIsKindOf(PatternProfile
      ::Signer) and mac2.oclIsKindOf(PatternProfile::Verifier)
and
(enc.node = c1.node) and (dec.node = c2.node) and (mac1.node = c1.
      node) and (mac2.node = c2.node)
))))))->size()
```

Listing 1.1. Man-In-The-Middle (MITM) threat formalized using OCL

The constraint in Listing 1.1 explores the application model via the stereotypes applied on them. For a given application, components are parsed and only those that are not security mechanisms are checked. For each two different components $c1$ and $c2$ deployed on different nodes and which can communicate. The constraint checks if the following security mechanisms (described in Sect. 3.2): “encryptor”, “decryptor”, “signer” and “verifier” exist and are deployed in the same nodes as $c1$ and $c2$. The second constraint is commented on Listing 1.2.

```
Context Application inv Injection_v1
self.components->select(c1 |
/* Firewall exists and is deployed on the same node as c1*/
self.components->select(firewall | firewall.oclIsKindOf(Firewall)
      and firewall.node = c1.node
))->size()
```

Listing 1.2. Injection threat formalized using OCL

Results. Table 1 gives the number of threats, TPs, FPs. The actual version of the threats has a Precision of 60%. The precision rate indicates that 40% are FPs.

Table 1. Number of detected threats, TPs and FPs (iteration 1)

Threat Category	Detected	TP	FP
Man-In-The-Middle	2	1	1
Injection	3	2	1
Total	5	3	2

After investigating, three injection threats were detected for the three components: web page, web application and DBMS. The third one is an FP because an injection threat is more likely to happen when components are exposed. Hence, in iteration 2 a new concept is added in iteration 2: “port type”.

Two MITM threats were detected: (1) between the browser and the web application and (2) between the web application and DBMS. The second one is an FP because a MITM is more likely to happen on “untrusted” zone. Hence, in iteration 3, the concept of “Trust Level” is added.

Iteration 2. In this iteration, the concept of trust level is added to ComponentUML in order to check if components are in a trusted network zone or not. Figure 5 shows ComponentUML model with the *TrustLevel* enumeration with two literals *trusted* and *untrusted*. The application model is modified and considers that DBMS node is in a “trusted” node while the web page and web application are in an “untrusted” node. In addition, the constraint does not only check for the existence of security mechanisms but that they are correctly used. This is done by verifying that the transmitted messages between two components are encrypted/unecrypted and signed/verified. The application model in Fig. 3 is modified. In this version the web page and application are considered in “untrusted” node while the DBMS is in an “untrusted” node. In addition, the OCL in Listing 1.1 constraint is modified. Listing 1.3 gives an extract of the OCL constraint. The lines that have already explained have been removed intentionally for matter of simplicity.

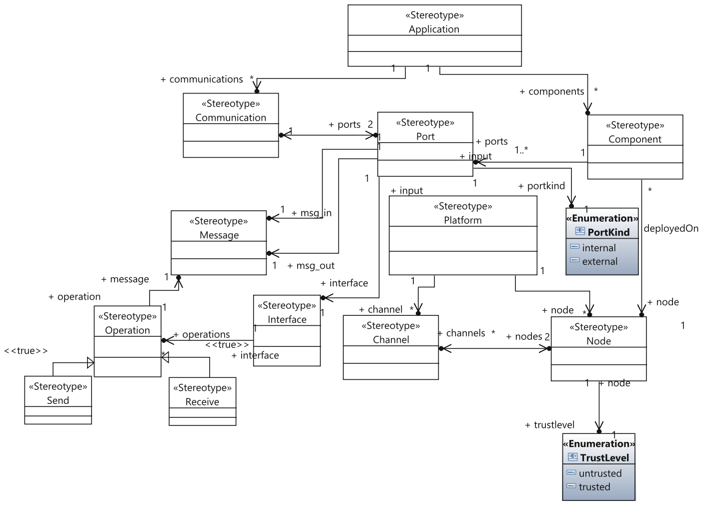

Fig. 5. Augmented ComponentUML model

```
Context Application inv Man-In-The-Middle_v2
self.components->select(c1 |
self.components->exists(c2 |
/* c1 or c2 are deployed in an untrusted node */
(c1.node.trustlevel= TrustLevel::untrusted or c2.node.trustlevel =
    TrustLevel::untrusted )
and
```

```
 7 /* c1 and c2 two components communicating deployed in different
       nodes
 8 [...]
 9 and
10 /* The security mechanisms exist: encryptor, decryptor, signer and
       verifier and Mechanisms are connected to components */
11 [...]
12 /* The mechanisms are called correctly */
13 c1.ports->select(inp_c1_c2 | inp_c1_c2.msg_out._'<>'(null) and c2.
       ports->exists(inpt2 | inpt2.communication = inp_c1_c2.
       communication))->forAll(inp_c1_c2 | mac1.ports->exists(sign_in
       |c1.ports->exists(c1_inp| c1_inp.communication = sign_in.
       communication)
14  and
15  (enc.ports->exists(enc_in | c1.ports->exists(c1_signorEnc |
16 -- case 1: message flow encrypt and then sign
17      (c1_signorEnc.communication = enc_in.communication  and
             sign_in.msg_out = inp_c1_c2.msg_out and sign_in.msg_in =
             enc_in.msg_out and enc_in.msg_in = c1_signorEnc.msg_out)
18     or
19 -- case 2: message flow sign and then encrypt
20     [...]
```

Listing 1.3. Man-In-The-Middle (MITM) threat version 2 formalized using OCL

The constraint explores the application model via the stereotypes applied on them. For each two different components c1 and c2, the constraint checks if they are deployed in an "untrusted" node (lines 5–7). It checks also if they are connected to the aforementioned security mechanisms (line 5–7). Then, it checks if they are calling the security mechanisms and that they are correctly used in message flows (lines 10–20). Two cases have been identified:

- The message sent from c1 to c2 is encrypted then signed
- The message sent from c1 to c2 is signed then encrypted

Results. Table 2 shows the results after checking the new version of the OCL constraints over the working example. The results show that three threats have been detected and are TPs and only one is an FP. Hence, the precision have increased to 75%.

Table 2. Number of detected threats, TPs and FPs (iteration 2)

Threat Category	Detected	TP	FP
Man-In-The-Middle	1	1	0
Injection	3	2	1
Total	4	3	1

Iteration 3. In this iteration, the concept of port kind is added. Figure 5 shows ComponentUML with the *PortKind* enumeration with two literals *external* (public ports) and *internal* (private ports). The application model in Fig. 3 is modified. In this version, the web page and application ports are considered "external" while the DBMS port is "internal". In addition, Listing 1.2 is modified.

Listing 1.4 gives an extract of the new version of the OCL constraint. The lines that have already been explained have been removed intentionally for a matter of simplicity. Only line 3 was kept and considers external ports.

```
1 Context Application inv Injection_v2
2 /*Publicly accessible port*/
3 self.components->select(c1 | c1.ports->exists(public_port | (
      public_port.portkind = PortKind::external)
4 and
5 /* Checks if Firewall exists and is connected to component c1 and
      message flow is correct*/
6 [...]
7 )->size()
```

Listing 1.4. Injection threat formalized using OCL

Results. As depicted in Table 3, the third version of the threats formalized with OCL has a precision of 100%. Of course, this is specific to the working example that has been presented which is a very simple example. In addition, the results are specific to the threats that have been considered.

Table 3. Number of detected threats, TPs and FPs (iteration 3)

Threat Category	Detected	TP	FP
Man-In-The-Middle	1	1	0
Injection v2	2	2	0
Total	3	3	0

4 Case Study: SCADA System

This section assesses the feasibility of the contributions of our work through the modeling and analysis of a SCADA (Supervisory Control And Data Acquisition) system. SCADA system applications are different from classical ITs (i.e., web applications) and have strong security requirements.

4.1 Description and Modeling

SCADA systems are meant to continuously control, monitor processes and acquire field information. In our experiment, we consider an adapted and simplified version of SCADA used in the context of smart grids [9]. In this context, the controlled process is power distribution. The control center consists of a control and a corporate network. The corporate network provides the operator with a Human-Machine Interface (HMI) that allows access to system data, SCADA servers, and databases that store operational and financial information. The SCADA server controls and gathers field information from geographically distributed substations or Remote Terminal Units (RTUs). The software components perform the following functions: (1) Perform control, (2) Poll Data, (3) System Start-up/shutdown, (4) Adjust Parameter Settings, (5) Log Field Data, (6) Archive Data, (7) Trigger Alarm, (8) Perform Trending: Select Parameters, Display Parameters, Zooming, Scrolling. Figure 6 depicts the software architecture model. In addition, ports, interfaces, data types, and transmitted messages are specified to provide a more detailed model of the application. The platform is also modeled to specify the relationship between components and nodes.

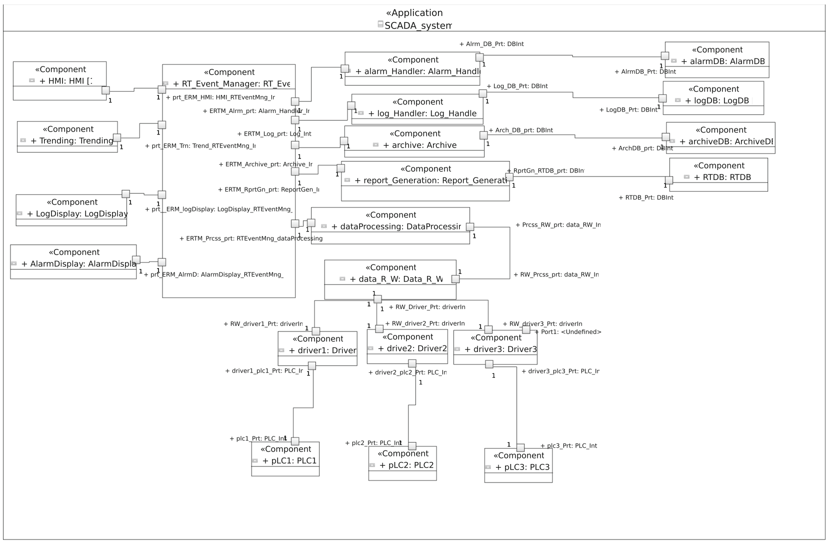
«Application
SCADA_system
«Component + HMI: HMI
«Component + RT_Event_Manager: RT_Eve
«Component + Trending: Trending
«Component + LogDisplay: LogDisplay
«Component + AlarmDisplay: AlarmDispla
«Component + alarm_Handler: Alarm_Handl
«Component + log_Handler: Log_Handle
«Component + archive: Archive
«Component + report_Generation: Report_Generati
«Component + dataProcessing: DataProcessir
«Component + data_R_W: Data_R_W
«Component + alarmDB: AlarmDB
«Component + logDB: LogDB
«Component + archiveDB: ArchiveDI
«Component + RTDB: RTDB
«Component + driver1: Driver
«Component + drive2: Driver2
«Component + driver3: Driver3
«Component + pLC1: PLC1
«Component + pLC2: PLC2
«Component + pLC3: PLC3
+ Alrm_DB_Prt: DBInt
+ AlrmDB_Prt: DBInt
+ Log_DB_Prt: DBInt
+ LogDB_Prt: DBInt
+ Arch_DB_prt: DBInt
+ ArchDB_prt: DBInt
+ RprtGn_RTDB_prt: DBInt
+ RTDB_Prt: DBInt
+ prt_ERM_HMI: HMI_RTEventMng_Ir
+ ERTM_Alrm_prt: Alarm_Handler_Ir
+ ERTM_Log_prt: Log_Int
+ prt_ERM_Trn: Trend_RTEventMng_I
+ ERTM_Archive_prt: Archive_Ir
+ ERTM_RprtGn_prt: ReportGen_I
+ prt_ERM_logDisplay: LogDisplay_RTEventMng_
+ ERTM_Prcss_prt: RTEventMng_dataProcessing
+ prt_ERM_AlrmD: AlarmDisplay_RTEventMng_
+ Prcss_RW_prt: data_RW_In
+ RW_Prcss_prt: data_RW_In
+ RW_Driver_Prt: driverIn
+ RW_driver1_Prt: driverIn
+ RW_driver2_Prt: driverIn
+ RW_driver3_Prt: driverIn
+ Port1: <Undefined>
+ driver1_plc1_Prt: PLC_Ir
+ driver2_plc2_Prt: PLC_Ir
+ driver3_plc3_Prt: PLC_Ir
+ plc1_Prt: PLC_Int
+ plc2_Prt: PLC_Int
+ plc3_Prt: PLC_Int

Fig. 6. SCADA software architecture model

4.2 Comparison of MBTA and ASTORIA

To assess MBTA, The obtained results are compared to the work of [9]. The latter proposes a framework named ASTORIA for attack scenario simulation for smart grid systems. The selection of the framework was motivated by the fact that ASTORIA is a simulation framework whereas ours is a formal verification-like framework. The ASTORIA team has simulated attack scenarios and evaluated their impact on the smart grid system to discover existing threats. In addition to the two threats presented previously, two more threats were formalized: Tampering and Denial of Service. Their formalization was omitted for simplification purposes. However, we give a brief explanation. *Denial of Service (DoS)* can make the system resources unavailable for authorized users. The signature of this threat is the lack or weakness of Firewall, Authentication, and Authorization mechanisms. *Tampering* is responsible for altering data at rest or in transit. The signature of this threat is the lack or weakness of Authenticity mechanisms.

Results. Table 4 presents the results obtained with the ASTORIA framework and "MBTA". For each asset, we conclude that all the detected threats are TPs. In addition, MBTA detected at the level of RTUs and communication new threats i.e., Tampering and Injection. FNs, i.e., threats that were not detected are due to different reasons. Some attack scenarios were simply not formalized or out of scope of our framework. For instance, Phishing is an attack scenario that attempts to obtain sensitive information such as credentials, and credit card details by using emails. This attack exploits social engineering which is out of scope of the study. Some attack scenarios are of the same kind or are pre-attacks of some formalized threats. For instance, replay attacks are a kind of Man-In-The-Middle attacks where the attacker maliciously or fraudulently repeats or delays a valid data transmission. Ping sweeps are generally used to check if a node is alive or dead. Some attack scenarios are at a lower stage (implementation) such as malicious software. In fact, we deal with software architecture analysis and not with code analysis.

Table 4. Threat Analysis results comparison

Assets	ASTORIA [9]	MBTA
Control Center	**Injection**	**Injection**
	Denial of Service	**Denial of Service**
	Malware, Phishing, Port scanning, Replay	
RTU	**Denial of Service**	**Denial of Service**
	Malware, Phishing, Port scanning, Replay	**Injection**
Communication	**Man-In-The-Middle**	**Man-In-The-Middle**
	Sniffing, Eavesdropping, Denial of Service, Replay	**Tampering**

Discussion. After analyzing the case study and the conducted assessment, some lacks in the current version of MBTA have been identified and are left for future work. Threat analysis can be generalized by replacing ComponentUML with OMG standards for Component-Based Development particularly UCM[7]. The second step is to construct a library of helpers to easily formalize threats. The specification of threats was done using OCL. OCL is a general language for constraining UML models. The goal is to enable security experts to contribute to the threat knowledge-base, who are not necessarily familiar with OCL and with less effort. In this context, we can inspect DSMLs for specifying these rules and then study mappings towards OCL.

5 Related Work

Security architecture assessment approaches can be categorized into two groups: scenario-based and property-based approaches. *Scenario-based Analysis.* focuses

[7] https://www.omg.org/spec/UCM.

on modeling security scenarios and then analyzing the architecture with regard to these scenarios. In literature, most of these works [1,2,7] have limitations in formalizing scenarios, in reusing and extending them, in automatizing the verification process and they also lack tool support. Recently Maidl et al. [4] have proposed a model-based threat modeling approach for Cyber-Physical Systems. It is based on a two-dimensional taxonomy that links system components and relevant attacks. The formalization language is OCL. The tool helps in prefiltering relevant attack actions and their documentation. In [7], the authors propose a framework for detecting architectural flaws in a code and introduce SCORIA as a formalization language. It starts by generating a graph describing a run-time architecture using static analysis. Then they assign security properties to the graph of objects. The constraints in this approach are highly dependent on the application and are not generic or reusable. The aim of "MBTA" is to foster reuse. In [2], the authors present a framework for detecting flaws in the code. The formalization language is OCL. The code is first transformed in STRIDE Data Flow Diagrams (DFDs) using static analysis. Then based on a 'best practice' repository where threat patterns are stored, an automatic check is performed to detect the threats and security measures that may be applied as annotations to DFDs to mitigate these threats. *Property-Based Analysis.* focuses on formalizing security properties to assess a software architecture. They defined a set of modularity properties used for analyzing the architecture. Table 5, compares "MBTA" to the aforementioned ones mainly: Almorsy et al. [1], Vanciu et al. [7] and Berger et al. [2] according to the following criteria: (C1) Foster reuse of the formalized threats, (C2) Verify that the architecture has the right security mechanisms, (C3) Verify that these security mechanisms are used correctly, and (C4) and Have a list of well-known threats.

Table 5. Positioning of the contribution with regards to other approaches

Approaches	(C1)	(C2)	(C3)	(C4)
MBTA	✓	✓	✓	✓
Maidl et al. [4]	✓	×	✓	✓
Almorsy et al. [1]	✓	✓	×	✓
Vanciu et al. [7]	×	✓	✓	✓
Berger et al. [2]	✓	✓	×	✓

6 Conclusion

In this paper, a model-based threat analysis for software architecture "MBTA" has been introduced. The contribution of this work is twofold. First, the approach enables detailed exploration of the software architecture. The formalized threats allow not only the verification of the existence of security mechanisms but also the verification of their correct usage. The second aspect is that the threats are reusable and extensible. OCL has been used to formalize Injection and Man-In-The-Middle threats. The formalization process has been explained through three iterations. For each iteration, the precision is evaluated. The formalized threats are not application dependent. They can be further extended if a threat exploits new vulnerabilities and weaknesses. The next step of this work consists of defining a correct-by-construction pattern-based security engineering process. It aims to provide the correct-by-construction integration of security patterns into an application while offering a certain degree of liberty to the designer using it. In order to be able to validate the integration, a formal specification of the pattern must be constructed, i.e., its properties, constraints, and related validation artifacts, as input to the pattern-based development process. Here, the concepts behind the formalized threats will be used and combined with patterns, to integrate security solutions in the application model and perform a security analysis within other types of threats.

References

1. Almorsy, M., Grundy, J., Ibrahim, A.S.: Automated software architecture security risk analysis using formalized signatures. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE 2013, pp. 662–671. IEEE Press (2013)
2. Berger, B.J., Sohr, K., Koschke, R.: Extracting and analyzing the implemented security architecture of business applications. In: 2013 17th European Conference on Software Maintenance and Reengineering, pp. 285–294. IEEE (2013)
3. Fernandez, E.B., Yoshioka, N., Washizaki, H.: Modeling misuse patterns. In: 2009 International Conference on Availability, Reliability and Security, pp. 566–571 (2009)
4. Maidl, M., Münz, G., Seltzsam, S., Wagner, M., Wirtz, R., Heisel, M.: Model-based threat modeling for cyber-physical systems: a computer-aided approach. In: van Sinderen, M., Maciaszek, L.A., Fill, H.-G. (eds.) ICSOFT 2020. CCIS, vol. 1447, pp. 158–183. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-83007-6_8
5. Motii, A., Hamid, B., Lanusse, A., Bruel, J.M.: Guiding the selection of security patterns for real-time systems. In: 21st International Conference on Engineering of Complex Computer Systems, ICECCS 2016, pp. 155–164. IEEE (2016)
6. Shostack, A.: Experiences threat modeling at Microsoft. In: Proceedings of the Workshop on Modeling Security, vol. 413, pp. 5:1–5:12. CEUR-WS.org (2008)
7. Vanciu, R., Abi-Antoun, M.: Finding architectural flaws using constraints. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 334–344 (2013)
8. Weiss, M., Mouratidis, H.: Selecting security patterns that fulfill security requirements. In: 2008 16th IEEE International Requirements Engineering, RE 2008, September 2008, pp. 169–172 (2008)
9. Wermann, A.G., Bortolozzo, M.C., da Silva, E.G., Schaeffer-Filho, A., Gaspary, L.P., Barcellos, M.: ASTORIA: a framework for attack simulation and evaluation in smart grids. In: NOMS 2016–2016 IEEE/IFIP Network Operations and Management Symposium, April 2016, pp. 273–280 (2016)