D2.1 Specification of security formal models

Contract number:	957212
Project acronym:	VeriDevOps
Project title:	Automated Protection and Prevention to Meet Security Requirements in DevOps Environments
Delivery Date:	30/08/2021 (M11)
Coordinator:	ABO
Partners contributed:	MDH, MI, SOFT, ABO
Release Date:	30.11.2021
Version:	1.0
Abstract:	This deliverable discusses the formalisms used to specify security properties in the VeriDevOps project. As the state-of-the-art review demonstrated, there are several formalisms that may be applied depending on analysis techniques that are to be used to verify and enforce the security properties. Based on the expertise of the project partners and the requirements by the case studies, we suggest the use of four formalisms for security property specification: object-oriented concepts, linear temporal logic, timed

	computation tree logic, and UPPAAL timed automata. They have distinct strengths but serve the same purpose, ie., security. The goal of the document is to provide introductory information for those formalisms to the interested stakeholder in the context of VeriDevOps. Therefore in the document we briefly outline those four formalisms, give an overview of the major concepts and provide examples. We specify the references where more information should be found about those formalisms. The list of proposed formalisms may be updated later based on the interaction with other technical work packages and feedback from case study providers. Further deliverables, such as D2.2, will provide more information on the usage of these formalisms in various applications such as, for example, requirements patterns.
Status:	PU (Public)

Table of Contents

Executive Abstract

This deliverable discusses the formalisms used to specify security properties in the VeriDevOps project. As the state-of-the-art review demonstrated, there are several formalisms that may be applied depending on analysis techniques that are to be used to verify and enforce the security properties. Based on the expertise of the project partners and the requirements by the case studies, we suggest the use of four formalisms for security property specification: object-oriented concepts, linear temporal logic, timed computation tree logic, and UPPAAL timed automata. They serve different purposes and have to be used in complementary ways.

The goal of the document is to provide introductory information for those formalisms to the interested stakeholder in the context of VeriDevOps. Therefore in the document we briefly outline those four formalisms, give overview of the major concepts and provide examples. We specify the references where more information should be found about those formalisms.

The list of proposed formalisms may be updated later based on the interaction with other technical work packages and feedback from case study providers. Further deliverables, such as D2.2, will provide more information on the usage of these formalisms in various applications such as, for example, requirements patterns.

1. Introduction

In the VeriDevOps project [1], *Work package 2 - Automated generation of security requirements* investigates automatic extraction, formalization, and verification of the security requirements from natural language requirements, vulnerability databases and standards. The resulting information is used as input for WP4 - Prevention at development and WP3 - Reactive Protection at Operations.

The first task of WP2, T2.1: Security Formal Modelling investigates which formalisms would be suitable for the formalization of security requirements in the project. In order to be of practical use later on, security requirements need to be clearly specified, not only to avoid confusion and inconsistencies, but also to be easy to reason about and have them processed by tools for automatic verification, test generation, or runtime monitoring. The formalisms that we have chosen in this project and are going to be presented in this deliverable are selected so that they are expressive enough for the type of security requirements that we will use, are accompanied by tool support, and are suitable for the industrial case studies in the project.

A *security requirement* is a statement of needed security functionality that ensures one of many different security properties of software is being satisfied [2]. Security requirements are derived from industry standards (such as ISO 2700x etc, IEC62443 or common criteria documents[1]), applicable laws, and a history of past vulnerabilities. A security requirement typically addresses one or several *security criteria* or *goals*, such as Confidentiality, Integrity, Availability, and provides the basis for implementing security controls and policies. The characterization into three aspects is done by the Commission of the European Communities in their technical report [3]:

- *Confidentiality:* prevention of the unauthorized disclosure of information. The confidentiality represents the assurance that information is shared only among authorized persons or organizations. Breaches of confidentiality can occur when data is not handled in a manner adequate to safeguard the confidentiality of the information concerned. Such disclosure can take place by word of mouth, by printing, copying, e-mailing or creating documents and other data etc. The classification of the information should determine its confidentiality and hence the appropriate safeguards.
- *Integrity*: prevention of the unauthorized modification of information. The integrity represents the assurance that the information is authentic and complete, and that it can be relied upon to be sufficiently accurate for its purpose. The integrity of data is not only whether the data is correct, but whether it can be trusted and relied upon.
- *Availability*: prevention of the unauthorized withholding of information or resources. The availability represents the assurance that the systems responsible for delivering, storing and processing information are accessible when needed, by those who need them.

Other security properties (requirements), such as authenticity, non-repudiation or traceability can also be considered but are generally not regarded as primitive security properties. In general, users must have confidence in the security of the computer system that they use. That is why several criteria for evaluating security have been developed in several countries ([4], [5], [6] and [7]). To receive a high degree of assurance according to these criteria, a system must be formally specified and the proof that this specification meets the security requirements. This need has spawned a new field of research whose goal is to formally model the concept of security. The objectives of a security model are as follows:

- to express without any ambiguity the security requirements in a computer context,
- to offer the possibility to demonstrate that all requirements are met in the proposed model,
- to provide ways to justify that the model is correct (absence of incoherence or conflicts), and
- to afford methods to design and implement the target system.

Different taxonomies for security requirements have been proposed in literature [8]–[10] and they classify requirements into several categories, for instance authentication, authorization, integrity, immunity.

Similar to the general requirements specification process [11], the specification of security requirements requires rigorous methods to avoid inconsistencies, ambiguity and omissions. To that extent, several requirements engineering methods have been proposed [12] *Formal specification* of

[1] https://www.commoncriteriaportal.org/cc/

requirements is one of the techniques that addresses the above issues by capturing requirements into verifiable properties. However, although formal specification allows more precise requirements using a restricted syntax language with well-defined semantics based on mathematical concepts [13], its use could be time consuming and difficult to adopt by non-specialists. In VeriDevOps, we plan to increase the likelihood of adoption of formal requirement specifications, by proposing a set of methods and associated tool support that automate some of the steps of requirements formalization.

This deliverable presents an initial list of such formalisms, which will be updated later, based on the interaction with other technical work packages, and feedback from case-study providers. Concretely, at this phase of the project, we suggest four formalisms to be used: object-oriented concepts, linear temporal logic, timed computation tree logic, and UPPAAL timed automata. The formalized requirements will be later used by other work packages of the project, either for verification of the SUT specification, test generation, or monitoring, as depicted in Figure 1.

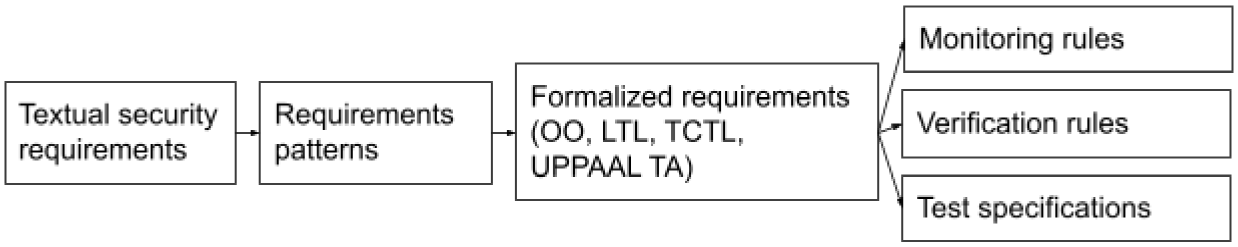

Textual security requirements
Requirements patterns
Formalized requirements (OO, LTL, TCTL, UPPAAL TA)
Monitoring rules
Verification rules
Test specifications

Figure 1. Conceptual approach on formalizing requirements in VeriDevops

The structure of this deliverable continues as follows. First, we briefly overview different formalisms used for formalizing security requirements, then present in more detail the selected formalisms, after which we present a set of tools provided by project partners or publicly available, which we plan to use in the project.

2. Formalisms used for Security Requirements

Many security models have been developed in the literature. The majority of them focused on the issue of information confidentiality (for example [14], [15], and [16]). The models that meet other security needs are less numerous. We can mainly cite Biba [17] and Clark-Wilson [18] models for integrity, and Yu-Gligor [19] and Millen [20] models for availability.

The common point between the three security goals -confidentiality, integrity and availability- is that the requirements they impose are generally deployed via security policies. A security policy is a set of rules that regulates the nature and the context of actions that can be performed within a system. It permits to govern the choices in a system's behavior [21]. *Authorization policies* are used to define what services or resources a subject (management agent, user or role) can access. *Obligation policies* are event-triggered condition-action rules that can be used to define adaptable management actions. These policies thus define the conditions for performing a wide range of management actions such as changing Quality of Service, when to perform storage server backups, register new users in a system, or install new software.

An example of well-known security policy is level-based security policy used generally by military organizations to protect critical information. Very soon, the Orange Book [4] proposed a classification of security policies into two main categories: mandatory control access policies (MAC policies), which include level-based security policies and Discretionary access control policies (DAC policies).

Security Policy-based management has become a promising solution for managing enterprise-wide networks and distributed systems. In practice, organizations define security policies to specify how their subjects can access their information (data and services). Basically, they regulate the information knowledge in the case of confidentiality, information changes and modifications in the case of the integrity and resources availability in the case of availability.

Other models for security policies based on the concept of role as RBAC [22], or task as TBAC [23] or team as TMAC [24] have also been proposed. Each of these models can respond to specific issues but are not sufficiently generic to express the different security needs that might arise in the security policy of an organization. More recently, Or-BAC [16] and Nomad [25] models have been proposed to overcome some limitations of previous models. Or-BAC can express security policies more dynamically including contextual safety rules. Whereas, Nomad [25] allows to express timed privileges on non atomic actions. It combines deontic and temporal logics and can describe conditional privileges and obligations with deadlines.

In addition, security management policies are needed to define the actions to be taken when security violations, such as a series of login failures occur for a particular user, or an attack on the system is detected. Furthermore, the heterogeneity of security mechanisms used to implement access control makes security management an important and difficult task.

Security policies define choices in behavior in terms of the conditions under which predefined operations or actions can be invoked rather than changing the functionality of the actual operations themselves. In today's web environments, security concerns tend to increase when code mechanisms are introduced to enable such adaptation, and so many researchers favor a more constrained form of rule-based policy adaptation. Large-scale systems may contain millions of users and resources. It is not practical to specify policies relating to individual entities instead, it must be possible to specify policies relating to groups of entities and also to nested groups such as sections within departments, within sites in different countries in an international organization. It is also useful to group the policies pertaining to the rights and duties of a role or position within an organization such as a network operator, nurse in a ward or mobile computing *visitor* in a hotel.

Security policies are derived from business goals, service level agreements or trust relationships within or between enterprises. The refinement of these abstract policies into policies relating to specific services and then into policies implementable (that we can implement) by specific devices supporting the service is not easy, and not amenable to automation. Although the technologies for building management systems and implementing security are available, work on the specification and deployment of policies is still scarce. The precise and explicit specification of implementable security requirements is important in order to achieve the organizational goals using currently available technologies.

2.1. Security Requirements Languages

Logic-based languages are used for the specification of security policies, as they have a well understood formalism, which is open to analysis. However they are not always directly translatable into efficient implementation and can be difficult to use. This section starts by presenting some of these specification languages classified by the type of logic used in their definition. We go on to describe languages like Role-Based Access Control (R-BAC), Organizational-Based Access Control (Or-BAC) and Nomad (Non atomic Actions and Deadlines model). Other languages are also briefly presented.

2.2. Logic-Based Languages

In the category of logic based languages, we discuss in the following two of the most used ones, first order logic and deontic logic, from which a selection of more concrete formalisms (e.g, linear temporal logic, timed computation tree logic) will be discussed in Section 3.2 as formalisms selected for this project.

2.2.1. First Order Logic

There are several examples that illustrate the application of first order logic to the specification of security policies. These include the logical notation introduced in [26] and the Role Definition Language (RDL) presented in [27] and RSL99 The RSL99 Language for Role-Based Separation [28]. Because all these approaches are based on the Role Based Access Control (R-BAC) model, we will defer a more detailed discussion to Section 2.2.3 where they can be considered together with other R-BAC specification techniques.

In addition to these R-BAC examples, there are some examples of the application of *Z* to defining security policies for a system. *Z* is a formal specification language that combines features of first order predicate logic with set theory [29]. In [30], the use of *Z* to specify and validate the security model for the NATO Air Command and Control System is described. The aim of this work was to develop a model for both mandatory and discretionary access controls based on the Bell-LaPadula approach mentioned previously.

One of the main problems encountered when using first order logic for policy specification arises when negation is used together with recursive rules. This leads to logic programs that cannot be decided and cannot be evaluated in a flounder-free manner [31]. Although it is possible to avoid the use of negation or recursion, this is not practical since it diminishes significantly the expressive power of the logical language.

2.2.2. Deontic Logic

Deontic logic was developed starting in the 1950s and revisited by Castaneda [31] in 1981 by extending modal logic with operators for permission, obligation and prohibition. Known as Standard Deontic Logic (SDL), traditionally it has been used in analyzing the structure of normative law and normative reasoning in law. Because SDL provides a means of analyzing and identifying ambiguities in sets of legal rules, there are many examples of the application of SDL to represent legislative documents [32] [33].

Typically, a deontic logic uses *OA* to mean it is obligatory that *A*, (or it ought to be (the case) that *A*), *PA* to mean it is permitted (or permissible) that *A*, and *FA* to to mean it is prohibited (or prohibitable) that *A*. The term deontic is derived from the ancient Greek déon, meaning, roughly, that which is binding or proper. In the following, we summarize the basic axioms of the SDL language:

- Autologies of propositional calculus
- O (p -> q) -> (Op -> Oq): If there is an obligation that *p* implies *q*, then an obligation to do *p* implies an obligation to do *q*.
- Op -> Pp: If there is an obligation to do *p* then *p* is permitted.
- Pp <-> Not(O) Not(p): Iff *p* is permitted then there is no obligation to not do *p*. In other words, iff *p* is permitted then there is no refrain policy with respect to *p*.
- Fp <-> Not(Pp): Iff *p* is forbidden then there is no permission to do *p*.
- p, (p -> q) / q (Modus Ponens): If we can show that *p* holds and that *q* is implied by *p*, then it is possible to infer that *q* must hold.
- p / Op (O-necessitation): If we can show that *p* holds, then it is possible to infer that the obligation to do p also holds.

Some of the earliest work using deontic logic for security policy representation can be found in [34]. The focus of this work was to develop a means of specifying confidentiality policies together with conditional norms. In [35], SDL is used to represent security policies with the aim of detecting conflicts in the policy specifications. This approach is based on translating the SDL representation into first order predicate logic before performing the necessary conflict detection analysis. Ortalo [36] describes a language to express security policies in information systems based on deontic logic. In his approach he accepts the axiom Pp <-> Not(O) Not(p) as a suitable definition of permission.

An inherent problem with the deontic logic approach is the existence of a number of paradoxes. For example, *Ross* paradox, i.e. if the system is obliged to perform the action send message, then it is obliged to perform the action send message or the action delete message. Although there is some work that offers resolutions to these paradoxes [37] [38], the existence of paradoxes can make it confusing to discuss policy specifications using deontic logic notations.

Additional logic-based languages are proposed in the literature. We can mention for example the authorization specification language (ASL) [39] which is an example of a stratified first order logic language for specifying access control policies. We can also mention the Barker model [40] that adopts an approach to express a range of access control policies using stratified clause-form logic, with emphasis on R-BAC policies.

2.3. R-BAC Language

Role-Based Access Control (R-BAC) [41] is an approach to restricting system access to authorized users. It is a newer alternative approach to *mandatory access control* (MAC) and *discretionary access control* (DAC). R-BAC is a policy neutral and flexible access control technology sufficiently powerful to simulate both DAC and MAC policies.

Within an organization, roles are created for various job functions. The permissions to perform certain operations are assigned to specific roles. System users are assigned particular roles, and through those role assignments acquire the permissions to perform particular system functions. Roles permit the grouping of a set of permissions related to a position in an organization such as finance director, network operator, ward-nurse or physician. This allows permissions to be defined in terms of the position rather than the person assigned to the permission, so policies do not have to be changed when people are reassigned to different positions within the organization.

Unlike Context-Based Access Control (C-BAC), R-BAC does not look at the message context (such as where the connection was started from).

Since users are not assigned permissions directly, but only acquire them through their role (or roles), management of individual user rights becomes a matter of simply assigning the appropriate roles to the user, which simplifies common operations such as adding a user, or changing a user's department.

R-BAC differs from access control lists (ACLs) used in traditional discretionary access control systems in that it assigns permissions to specific operations with meaning in the organization, rather than to low level data objects. For example, an access control list could be used to grant or deny write access to a particular system file, but it would not say in what ways that file could be changed. In an RBAC-based system an operation might be to create a `credit account' transaction in a financial application or to populate a `blood sugar level test' record in a medical application. The assignment of permission to perform a particular operation is meaningful, because the operations are fine grained and have meaning within the application.

Another motivation for R-BAC has been to reuse role specification by a form of inheritance whereby one role (often a superior in the organization) can inherit the rights of another role and thus avoid the need to repeat the specification of permissions.

A number of variations of R-BAC models have been developed, and several proposals have been presented to extend the model with the notion of relationships between the roles [42], as well as with the idea of a team, to allow for team-based access control where a set of related roles belonging to a team are activated simultaneously [24]. Chen and Sandhu [26] introduce a language based on set theory for specifying R-BAC state related constraints, which can be translated to a first-order predicate-logic language. They define an R-BAC system state as the collection of all the attribute sets describing roles, users, privileges, sessions as well as assignments of users to roles, permissions to roles and roles to sessions. They also define constraints as the specification of restrictions to R-BAC states, called invariants, as well as to state changes, called preconditions. They use this model to specify constraints for R-BAC in two ways: (i) by treating them as invariants that should hold at all times, and (ii) by treating them as preconditions for functions such as assigning a role to a user. They define a set of global functions to model all operations performed in an R-BAC system, and specify constraints which include: conflicting roles for some users, conflicting roles for sessions of some users, and prerequisite roles for some roles with respect to other users.

2.4. Or-BAC Language

Or-BAC [16] stands for Organization Based Access Control language. It is an access and usage control model based on first logic order that allows an organization to express its security policy including contextual rules. For this purpose, Or-BAC defines two abstraction layers. The first one is called abstract layer and describes a rule as a *role* having the permission, prohibition or obligation to perform an *activity* on a *view* in a given *context*. A *view* is a set of objects to which the same security rules apply. A *role* is a set of users with similar privileges and an *activity* considers a set of actions with similar properties. The second layer is the concrete one. It is derived from the abstract level and grants permission, prohibition or obligation to a *user* to perform an *action* on an *object*.

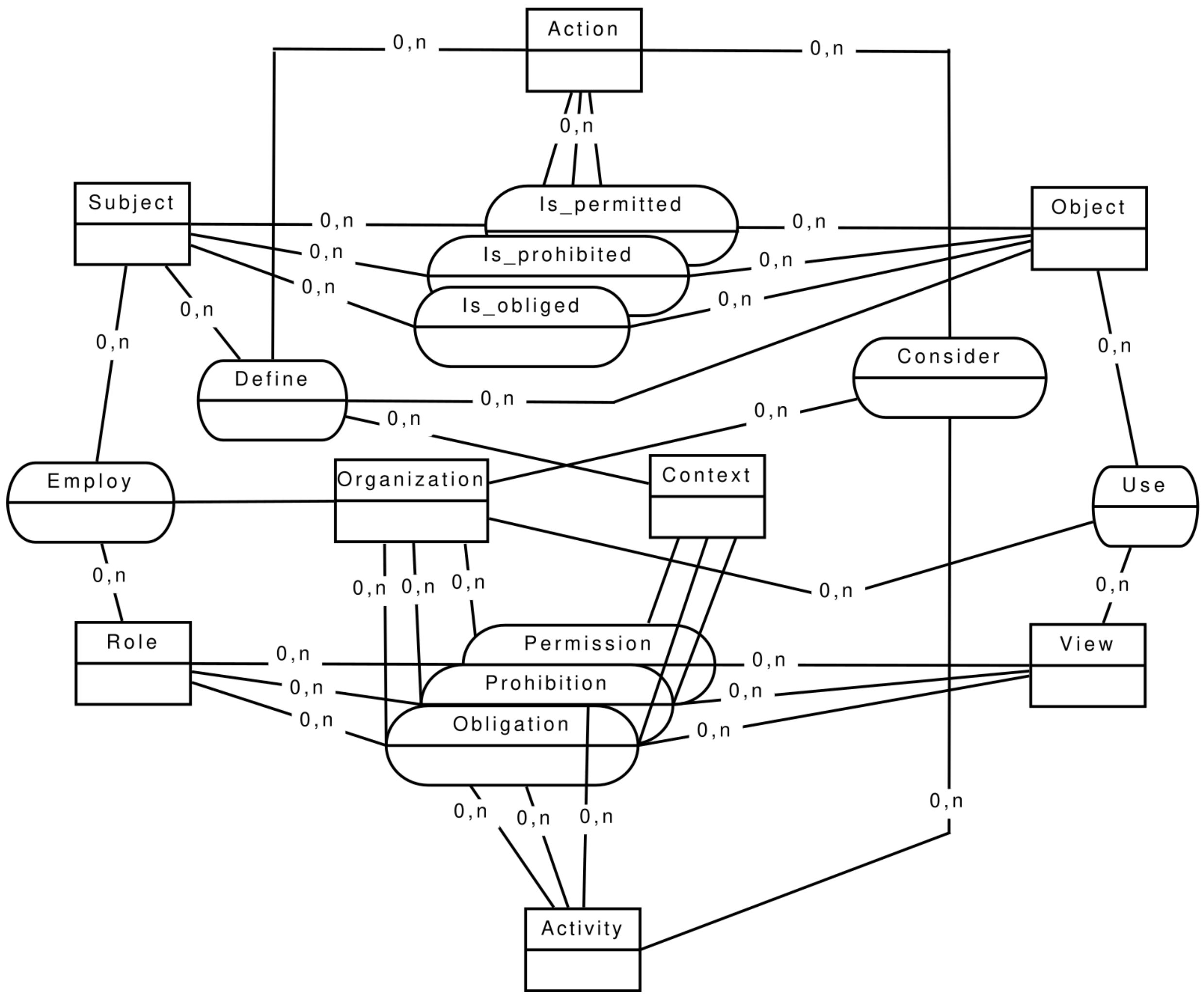


Figure 2. The Or-BAC model

A Security Rule in Or-BAC is a relation between organizations, roles (sets of subjects with similar properties), views (sets of objects that satisfy a common property), activities and context. It is defined as a *role* having the permission, prohibition or obligation to perform within an *organization* an *activity* on a *view* in a given *context*.

Thus, according to the Or-BAC syntax, a typical security rule has the following form:

- Obligation (S, R, A, V, C): this rule means that within the system S, the role R is obliged to perform the activity A targeting the objects of view V in the context C.
- Permission (S, R, A, V, C): this rule means that within the system S, the role R is permitted to perform the activity A targeting the objects of view V in the context C.
- Prohibition (S, R, A, V, C): this rule means that within the system S, the role R is prohibited to perform the activity A targeting the objects of view V in the context C.

In Or-BAC, we use contexts to express different types of extra conditions or constraints that control activation of rules expressed in the access control policy [43]:

- The temporal context that depends on the time at which the subject is requesting for an access to the system.
- The spatial context that depends on the subject location.
- The user-declared context that depends on the subject objective (or purpose).
- The prerequisite context that depends on characteristics that join the subject, the action and the object.
- The provisional context that depends on previous actions the subject has performed in the system.

Or-BAC also assumes that each organization manages some information system that stores and manages different types of information. To control activation, each information system must provide the information required to check that conditions associated with the context definition are satisfied or not. The following list gives the kind of information related to the contexts we have just mentioned.

- A global clock to check the temporal context.
- The subject environment and software and hardware architecture to check the spatial context.
- The subject purpose is to check the user-declared context.
- The system database to check the prerequisite context.
- An history of the action carried out, the check the provisional context.

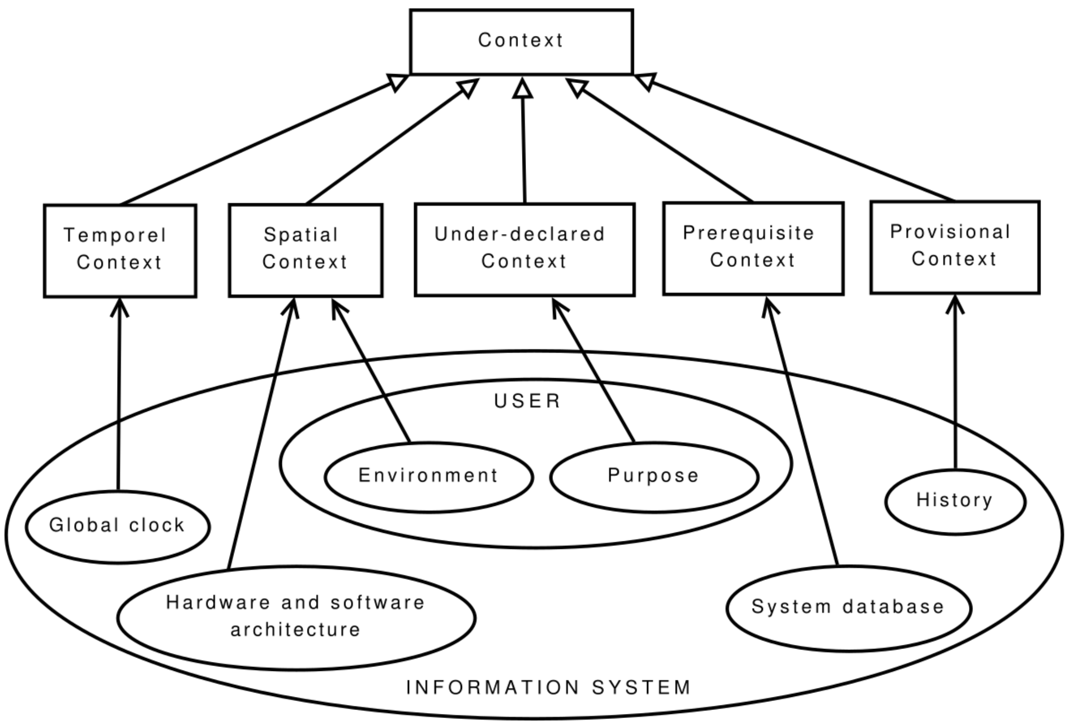


Figure 3: The Or-BAC Context

Figure 3 presents the correspondence between the contexts and the required data. If the information system does not provide some information in this list, then the corresponding context cannot be managed by the access control policy. The Or-BAC model has an associated tool called MotOrBAC [44] that was developed to help to design and implement security policies using the Or-BAC model. The current versions of this tool can design, upload and store security policies and simulate them. The policy simulation can be used to verify the consistency of a security policy. The tool can also detect potential conflicts and help the designer eliminate them.

2.5. Nomad Language

Nomad [25] stands for `Non Atomic Actions and Deadlines' model. It provides a way to describe permissions, prohibitions and obligations related to non-atomic actions within different contexts. Nomad allows the user to express privileges on non atomic actions. It combines deontic and temporal logics and can describe conditional[2] privileges and obligations with deadlines. The main advantages of Nomad model are to provide means to specify:

- Privileges (that is permission, prohibition or obligation) associated with non atomic actions.
- Conditional privileges, which are privileges only triggered when specific conditions are satisfied.
- Privileges that must be fulfilled before some specific deadlines.

Non-atomic action: If A and B are actions that can be performed within an organization Org, then (A;B) (which means "A followed by B") and (A&B) (which means "A in parallel with B") are non-atomic actions.

Formulae: If A is an action then start (A) (starting A), doing (A) (doing A) and done(A) (finishing A) are formulae.

Here are some properties on actions and formulae:

- If *alpha* and *beta* are formulas then *Not(alpha)*, *(alpha and beta)*, *(alpha or beta)*, *(alpha -> beta)* and *(alpha <-> beta)* are formulas.
- If *alpha* is a formulae then *+alpha* (in the next time, *alpha*) and *-alpha* (in the previous time, *alpha*) are formulas.
- If *alpha* is a formula then *O^d alpha* (*alpha* was true d units of time ago if *d <= 0*, *alpha* will be true after d units of time if *d >= 0*) is a formula too.
- If *alpha* is a formula then *O^{<d} alpha* (within d units of time ago, *alpha* was eventually true if *d <= 0*, alpha is eventually true within a delay of d units of time if *d>=0*) is a formula.
- *(alpha|gamma)* is a formula: in the context *gamma*, the formula *alpha* is true.

[2] Most privileges do not apply unconditionally. Thus, we need to model privileges that are only active in specific contexts.

Notice that using Nomad formalism, we deal with a discreet time. The choice of the unit of time can be very important and depends on the studied system.

2.6. Other Security Specification Approaches

Other security specification approaches are proposed in the literature. It is very difficult to cover all of them. For matter of space, we will mention briefly some of them:

- **SPL**: the security policy language (SPL) [45] is an event-driven policy language that supports access-control, history-based and obligation-based policies. SPL is implemented by an event monitor that, for each event, decides whether to allow, disallow or ignore the event. Events in SPL are synonymous with action calls on target objects, and can be queried to determine the subject who initiated the event, the target on which the event is called, and attribute values of the subject, target and the event itself.
- **XACML**: [46] is an XML specification for expressing policies for information access over the Internet and is being defined by the Organization for the Advancement of Structured Information Standards (OASIS) technical committee. The language permits access control rules to be defined for securely browsing XML documents that can update individual document elements. Similar to existing policy languages, XACML is used to specify a subject-target-action-condition oriented policy in the context of a particular XML document.
- **LaSCO**: LaSCO [47] is a graphical approach for specifying security constraints on objects, in which a policy consists of two parts: the domain (assumptions about the system) and the requirement (what is allowed assuming the domain is satisfied). Policies defined in LaSCO have the appearance of conditional statements used to express authorizations between objects in the system and are stated as policy graphs. A policy graph is an annotated directed graph where the annotations are domain and requirement predicates. Nodes in the policy graph represent the sort of objects described by the associated domain predicate. Collectively, the nodes, edges, and domain predicates form the domain of a policy graph.
- **TPL**: The trust policy language (TPL) by IBM [48] provides a clearer separation between the authentication of subjects based on certificates and the assignment of authorizations to those subjects which have been successfully authenticated. With TPL, the credentials result in a client being assigned to a role which specifies what the client is permitted to do, where a role is a group of entities that can represent specific organizational units (e.g. employees, managers, auditors). The assignment of access rights to roles is outside the scope of TPL; the philosophy of the work on TPL is to extend role-based access control mechanisms by mapping unknown users to well defined roles.
- **IETF policy model:** the Internet Engineering Task Force (IETF) policy model considers policies as rules that specify actions to be performed in response to defined conditions: if <condition(s)> then <action(s)>. The condition-part of the rule can be a simple or compound expression specified in either conjunctive or disjunctive normal form. The action-part of the rule can be a set of actions that must be executed when the conditions are true.

- **DMTF Policy Core Information:** The policy core information model (PCIM) [49] extends the common information model (CIM) [50] defined by the DMTF with classes to represent policy information. The CIM defines generic objects such as managed system elements, logical and physical elements, systems, service, users, etc, and provides abstractions and representations of the entities involved in a managed environment including their properties, operation and relationships. The information model defines how to represent managed objects and policies in a system but does not solve the problem of actually specifying policies.
- **PDL**: The policy description language (PDL) is an event-based language from Bell-Labs [51] in which they use the event-condition-action rule paradigm of active databases to define a policy as a function that maps a series of events into a set of actions. The language can be described as a real-time specialized production rule system to define policies. The syntax of PDL is simple and policies are described by a collection of two types of expressions: policy rules and policy defined event propositions. Policy rules are expressions of the form: event causes action if condition, which reads: If the event occurs under the condition the action is executed. Policy defined event propositions are expressions of the form: event triggers policy-defined-event if condition, which reads: If the event occurs under the condition, the policy-defined-event is triggered.
- Other Languages: others models for security specification exit, we can quote for example Open Distributed Programming Reference Model (ODP-RM), Law-Governed Interaction (LGI), Event-Trigger-Rules, Ponder language, Role Definition Language (RDL) and some extensions of R-BAC language like TR-BAC language. An overview of other approaches for formalizing security requirements has been presented in Deliverable D3.1 of the VeriDevOps project [52].

3. Formalisms for requirements specification adopted in VeriDevOps

Based on the initial project proposal, characteristics of the case studies, and expertise of the project participants, a set of formalisms has been selected for further experimentation in VeriDevOps. Their main goal is to enable specification, consistency checking, and verification of the security requirements in a semi-formal and formal context.

To illustrate how different formalisms can be used for specifying security requirements, we suggest as an example the following security requirement (SQ1):

“A user can insert the wrong password three times, after which the account is locked (immediately) in max one minute”.

3.1. Requirements as objects

Seamless Object-Oriented Requirements (SOORs) [53] is a requirements engineering approach that focuses on requirements verifiability and reusability. In SOORs, requirements are written as classes. A SOOR takes the form of a class that inherits from a SOOR template (SOORT), providing system-specific details through the object-oriented (OO) genericity and abstraction [54]. The main role of a SOORT is to encapsulate the verification complexity and make the template applicable across multiple systems. What remains to the specifier is to find the SOORT that most closely formalizes the behavioral pattern implied by the target requirement and then inherit from it, providing system-specific details through the OO genericity and abstraction. As a bonus, the resulting SOOR class can automatically produce a structured natural-language representation of the target requirement. As a result, the specifier will compare the initial, less formal representation with the automatically generated one. Our experiments have demonstrated that this side-by-side comparison may lead to switching the SOORT formalization pattern or reforming the initial requirement altogether. Applying the SOORs method to a well-known collection of verification-oriented specification patterns [55] has clearly demonstrated [56] that a general-purpose object-oriented programming language with contracts is powerful enough to make the identified patterns practically reusable.

In the frame of the VeriDevOps project we explore a concept of the Object-oriented Requirements and its extension in Java with the RQCODE method.

In the VeriDevOps project we chose a practical approach of implementing the SOOR in a convenient language like Java. We deliberately used a simplified structure that includes:

- a textual requirement in Natural Language,
- a method for reinforcement of a requirement,
- a method for checking that the requirement is implemented, for example with a test.

With this approach we would like to propose a lightweight method for formalizing requirements while adding support for reuse, traceability, patterns and automation.

To illustrate let us consider the following example from the FAGOR case study. Let us consider the following security requirement from the Security Technical Implementation Guide (STIG) for industrial personal computers (IPC) running Windows 10 operating system:

- Textual requirement: *V-63447: The system must be configured to audit Account Management - User Account Management failures.*
- *Reinforcement of the requirement*

FAGOR has provided the following enforcement script :

```
auditpol /set /subcategory:"{0CCE9235-69AE-11D9-BED3-505054503030}" /failure:enable
```

In the RQCODE, formalism this requirement can be represented as a class (see Figure 4) :

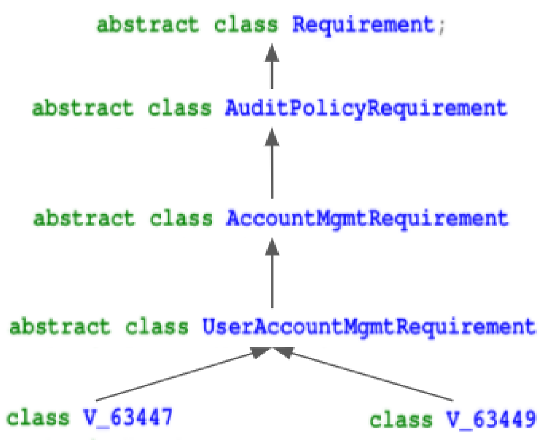

abstract class Requirement;
abstract class AuditPolicyRequirement
abstract class AccountMgmtRequirement
abstract class UserAccountMgmtRequirement
class V_63447
class V_63449

Figure 4. Requirements Hierarchy in Java

- a method for checking: The *V_63447* requirement class has a method toString that would give a textual STIG requirement as it was provided earlier. In addition it overrides the abstract method *enforce ()* by trying to execute the script. The requirement may have a method *check()* to verify that the policy has been effectively changed.

It is interesting to note that the V_63449 requirement is very similar except that it verifies a Success state for User Account Management. By using the generalization it is possible to provide a reusable form of the requirement implementation thus leveraging the existing implementation of the requirement.

This approach has been presented by the VeriDevOps partners at the IEEE Requirements Engineering Conference 2021 at the FORMREQ workshop in the paper entitled: "Security Requirements as Code: Example from VeriDevOps Project". Further on developed patterns will be covered in D2.2 deliverable.

3.2. Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) is a branch of Temporal Logic that describes how a system's static conditions change over time [57]. In LTL, one can encode formulae about the future of paths, e.g., a condition will eventually be true, a condition will be true until another fact becomes true, etc. It is a fragment of the more complex CTL, which additionally allows branching time and quantifiers. Subsequently, LTL is sometimes called propositional temporal logic, abbreviated PTL [58]. In terms of expressive power, LTL is a fragment of first-order logic [59]. LTL was first proposed for the formal verification of computer programs by Amir Pnueli in 1977.
LTL formulae denote properties that will be interpreted on each execution of a program. For each possible execution, satisfiability is checked on the run with no possibility of switching to another run

during checking. On the other hand, CTL semantics checks a formula on all possible runs and will try either all possible runs (A operator) or only one run (E operator) when facing a branch. In practice this means that some formulae of each language cannot be stated in the other language. For example, In CTL one can express that there is always a possibility that a state can be reached during a run, even if it is never actually reached (AG EF secured). LTL can only state that the "secured'' state is actually reached and not that it can be reached. On the other hand, the LTL formula ◊□s cannot be translated into CTL. This formula denotes the property of stability : in each execution of the program, s will finally be true until the end of the program (or forever if the program never stops). CTL can only provide a formula that is too strict or too permissive. We can craft two security requirements that fall into these examples of orthogonality of the two [60].

In this project, we are inspired by LTL to build a formalism to specify security properties denoting both good and bad behaviours (i.e., security requirements and malicious behaviours) that can happen in an industrial ICT system. LTL is extended with event deadlines to be able to raise alerts if these events do not occur after the timeout. The formalism is implemented and used in the Montimage Monitoring Tool called MMT.

MMT security properties formalism

The main objective of MMT security properties is to formally specify security goals and attack behaviors related to the application or protocol under test. The “MMT-Security property" model is inspired from LTL logic and can refer to two types of properties: “Security rules" and “Attacks" described as follows:

- A Security rule describes the expected behavior of the application or protocol under-test, whether it is functional or security-oriented. The non-respect of the MMT-Security property indicates an abnormal behavior, e.g. the access to a specific service must always be preceded by an authentication phase.
- An Attack describes a malicious behavior, whether it is an attack model, a vulnerability or misbehavior. Here, the respect of the MMT-Security property indicates the detection of abnormal behavior that might indicate the occurrence of an attack, e.g., a big number of requests from the same user in a limited period of time can be considered as a behavioral attack.

It must be noted that the events that we take into account within MMT-Security properties are related to observable system/network communications. In the case of a telecommunication network, they refer to traffic packets and flows. In other contexts, they can relate to any action stored in a server/database/software log file. In the following, we formally present the concepts of MMT-Security properties in the context of telecommunication networks. In the rest of this document the terms: “packet", “message" and “event" are used interchangeably.

The MMT-Security property model allows expressing complex security properties derived from security best practices and from domain-specific security requirements. These MMT-Security properties are described using an XML format to make interpretation easier for both humans and software.

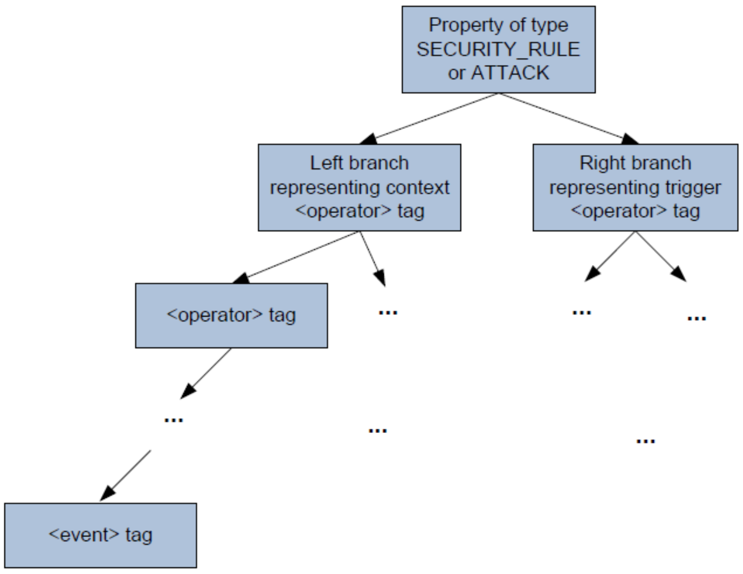

Property of type SECURITY_RULE or ATTACK
Left branch representing context <operator> tag
Right branch representing trigger <operator> tag
<operator> tag
<event> tag
...

Figure 5. MMT property structure

Each property begins with a <*property*> tag and ends with </property>. A property is a "general ordered tree" as shown in see Figure 5. The nodes of the property tree are: the property node (required), operator nodes (optional) and event nodes (required). The property node is forcibly the root node and the event nodes are forcibly leaf nodes. The left branch represents the context and the right branch represents the trigger. This means that the property is found valid when the trigger is found valid; and the trigger is checked only if the context is valid. In other words:

- If the context is verified and the trigger is not, then a property non-respect occurrence is detected.
  - In the case of a "security rule", this means that the context of the rule has been found and, since the trigger was not, we conclude that the "security rule" has been violated.
  - In the case of an "attack", this means that the context of an attack has occurred but the trace was attack free.
- If the context and the trigger are verified, then a property respect occurrence is detected.
  - In the case of a "security rule", this means that the context of the rule has been found, as well as the trigger. We conclude that the "security rule" has been respected.
  - In the case of an "attack", this means that the context of an attack has occurred, as well as the trigger. We conclude that the attack has been detected.

In the context of MMT, DPI (Deep Packet Inspection) and DFI (Deep Flow Inspection) are used to help detect and tackle harmful traffic and security threats; and, to throttle or block undesired behaviours. We define a set of security properties for network traffic, at both control and data levels, to detect interesting events. Indeed, based on the defined security properties, we register the attributes to be extracted from the inspected packets and flows. These attributes are of three types:

- Real attributes: They can be directly extracted from the inspected packet. They correspond to a protocol field value.
- Calculated attributes: They are calculated within a flow. Packets from the same flow are grouped and security/performance indicators are calculated (e.g., delays, jitter, packet loss rate) and made available for the security analysis engine.
- Meta attributes: These attributes are linked to each packet to describe capture information. The time of capture of each packet (timestamp attribute) is the main meta attribute in the current version of MMT.

The extracted attributes needed for security analysis can emanate from different data sources (probes and/or interfaces). This is managed in the MMT monitoring solution during the specification phase of the security properties. Indeed, the data sources identifiers are part of the meta-attributes that can be used to specify the relevant events for security analysis. Three architectures are taken into account in MMT:

- Local analysis: the collected traffic is analysed for security purposes in one probe that captures network traffic from one or several interfaces.
- Centralized analysis: the traffic capture is distributed, but the security analysis is centralized. All data sources send their collected traffic (filtered or not) to the same master server that correlates the traces (i.e., need to synchronize probes to perform this task).
- Distributed analysis: the traffic capture is distributed and the analysis is performed by all the probes that communicate together to share information. This analysis can be very interesting in some specific case studies like ad hoc networks.

We will use MI-LTL to formalize requirements as a set of rules that will be used to monitor the system under test at runtime. In the following example (Figure 6), we specify that an user's account is disabled after 3 failing attempts to authenticate to the system.

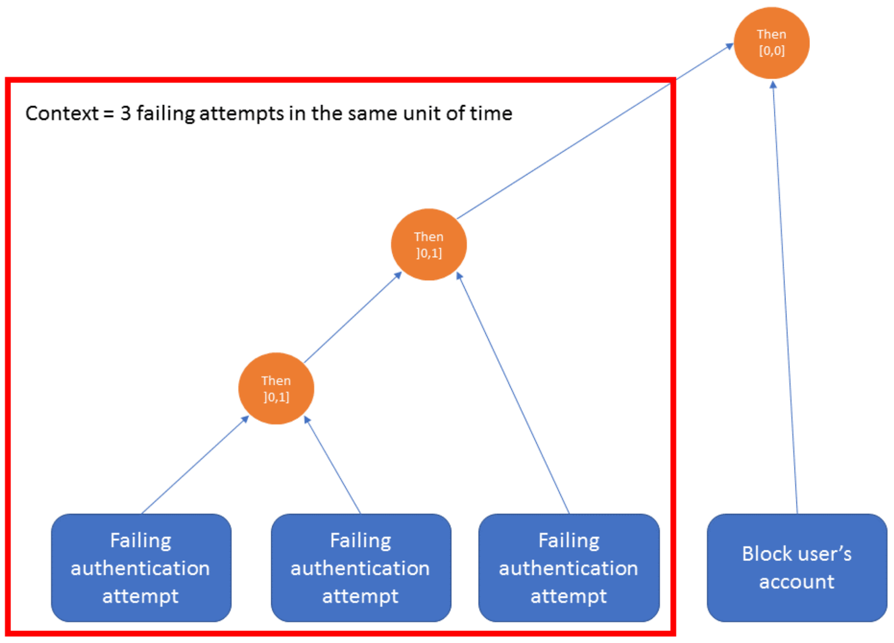

Context = 3 failing attempts in the same unit of time
Then
[0,0]
Then
]0,1]
Then
]0,1]
Failing authentication attempt
Failing authentication attempt
Failing authentication attempt
Block user's account

Figure 6: Security property in M-LTL : a user's account is blocked after three successive failing authentications.

MMT tool is a monitoring solution that allows checking different properties on extracted system traces. MMT can be used for online analysis of traces at runtime or as a complementary tool for a testing solution (at dev time) that allows to provide verdicts. It can also enhance dynamic or online testing. MMT is fully conceived and implemented by Montimage company and does not rely on any external library.

MMT relies on LTL logic and extends it by adding deadlines to system events in order to raise alerts if the deadline is not respected.

MMT inputs/outputs

- **Inputs:** Pre-recorded traces or traces in a streaming mode and a set of security properties to be analysed.
- **Output:** Verdicts and alerts

MMT will be used in both ABB and FAGOR use cases. Different traces are captured to be analyzed by MMT. We started by adding different plugins to MMT in order to extract relevant security attributes that can be correlated by security properties in order to detect malicious behaviors and anomalies.

For the ABB case study, we started analysing camera data containing 3 markers to determine the position of the cabin in a moving crane. Different safety properties have been specified, implemented and are under test. More Anomaly detection capabilities using AI/ML algorithms are under investigation. A link between MMT and other testing tools is also under investigation.

For the FAGOR case study, the first focus will be on the analysis of encrypted traffic, which is a challenging task from a research point of view. More scenarios are being identified at the current stage of the project.

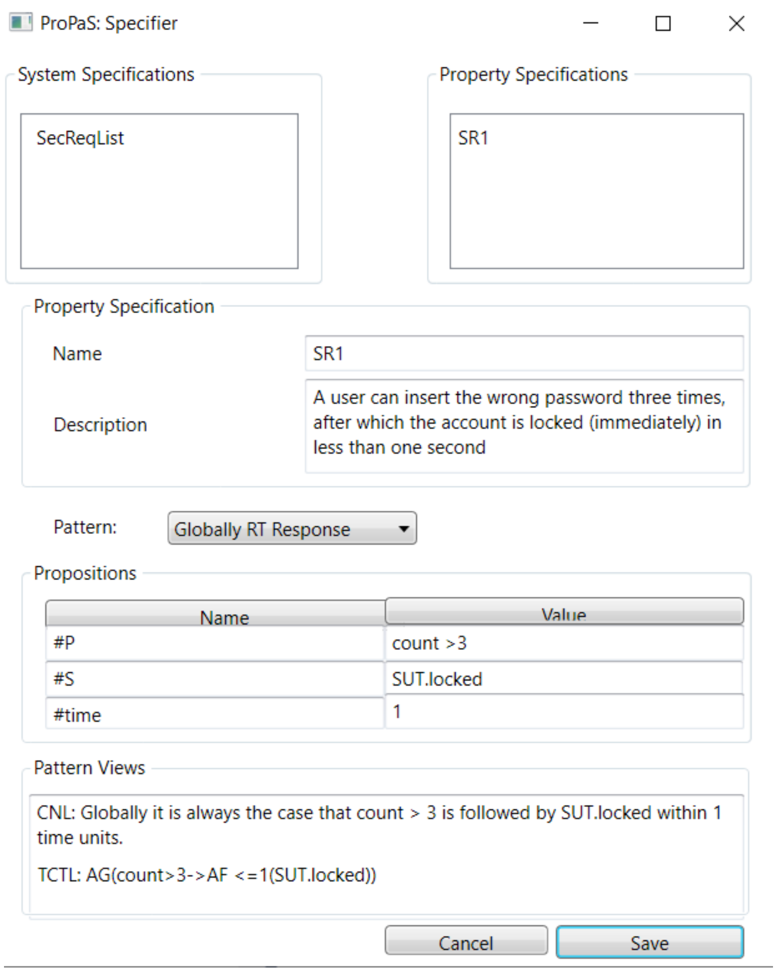
Notice that a root cause analysis engine is also being conceived to determine the origins of detected incidents to discard false positives and also apply relevant countermeasures to minimize the risk at runtime.

3.3. Timed Computation Tree logic (TCTL)

Computation tree logic (CTL) has been proposed by Emerson and Clarke [61] as a more efficient formalism for model-checking. In contrast to LTL where the model of time is linear, in CTL the time has a tree-like form. Timed Computation Tree logic (TCTL) has been proposed in [62] as an extension of CTL with quantitative temporal operators in order to enable model-checking of real-time systems. In TCTL, one can specify properties such as $\exists \Diamond_{<5}$, meaning "possibly within 5 time units" [62].

A TCTL property represents a specific encoding of a temporal logic property. The interpretation of a TCTL property is over a Kripke structure. Therefore, the syntax of TCTL consists of path quantifiers "All" (denoted as A) and "Exists" (marked as E), and path-specific temporal operators "G" (Globally, or for all states) and "F" (Future/Eventually). The universal path quantifier "A" stands for "all paths," while the existential quantifier "E" denotes that "there exists a path" from the set of all future paths starting from a given state s. We can divide all syntactic elements into two major categories: i) atomic propositions and ii) operators. The definition for atomic propositions is as usual. In contrast, the operators represent a union of the valid TCTL path-specific temporal (G, F, U, W), TCTL branch quantifiers (A, E), and the standard logical operators (and, or, =>, negation).

In VeriDevOps, we plan to specify both functional, non-functional and security requirements as TCTL properties. For instance, the TCTL query corresponding to our running example security requirement would be: *"A user can insert the wrong password three times, after which the account is locked (immediately) in less than one second" where p = wrong password three times, s = is locked and Time bound = in max 1 second. When using the* Globally, Real-time Response pattern (i.e., Globally, it is always the case that if P holds, then S eventually holds within T time units). The property becomes AG(*count >3 -> AF*$_{<=1}$*(SUT.locked)*). Such properties can be written manually or generated from requirement patterns using PROPAS (The PROperty PAttern Specification and Analysis) [63]. PROPAS is a tool set for automated pattern-based formalization of industrial critical requirements written in natural language. The system requirements specification is provided as a list of strings where each string encodes a system requirement and it generates the corresponding TCTL queries, as shown in Figure 7.



SecReqList

SR1

Property Specification

Name	SR1
Description	A user can insert the wrong password three times, after which the account is locked (immediately) in less than one second

Pattern: Globally RT Response

Propositions

Name	Value
#P	count >3
#S	SUT.locked
#time	1

Pattern Views

CNL: Globally it is always the case that count > 3 is followed by SUT.locked within 1 time units.

TCTL: AG(count>3->AF <=1(SUT.locked))

Figure 7. User interface of the ProPas tool.

In PROPAS, we use a pattern-based approach that uses predefined pattern-based templates [55] to aid requirements and test engineers in formulating security requirements. We use the ProPaS tool to specify these security requirements from textual descriptions of security threats and vulnerability scenarios. We use UPPAAL model checkers to generate test cases covering these requirements. The UPPAAL model checker uses a simplified version of TCTL and like the model, the requirement specification is expressed in a formal and well-defined language. While this formalism is appropriate for representing system properties, this is not that much used for capturing system behaviours. Timed-automata is a well established model that can be used for this purpose, since it can capture security requirements in a more explicit manner, using a set of richer constructs that actually show the internal behavior.

3.4. UPPAAL Timed Automata

Timed automata (TA) [62] is a formalism that allows modeling not only functional but also temporal properties of real-time systems and several model-checkers for TA have been developed in the last decades [64]. Among these, the UPPAAL model checker has been applied successfully to modeling, model-checking and testing real-time systems [65].

UPPAAL uses a version of TA extended with bounded integer variables and urgency [66], called UPPAAL TA. The nodes of the automata graph are called *locations* and directed vertices between locations are called *edges*. The state of an automaton consists of its current location and assignments to all variables, including clocks. Synchronous communication between processes is done by hand-shake synchronisation links that are called *channels*. A channel relates a pair of transitions in parallel processes where synchronized edges are labelled with symbols for input actions denoted *ch?* and output actions denoted *ch!*, where *ch* denotes the channel name. Asynchronous communication between processes is modelled using global variables accessible to all processes.

An example of a UPPAAL TA system specification implementing the previously discussed security requirement is presented in Figure 8. The environment automaton (*Env1*) provides inputs to the system under test (*SUT*) via the *input!* channel synchronization and the *pass* shared variable. The SUT will evaluate the received password and if three consecutive passwords have been received, the account will be locked within 1 time unit (*locked* location), and the locking notified back to the environment via the *lock* channel.

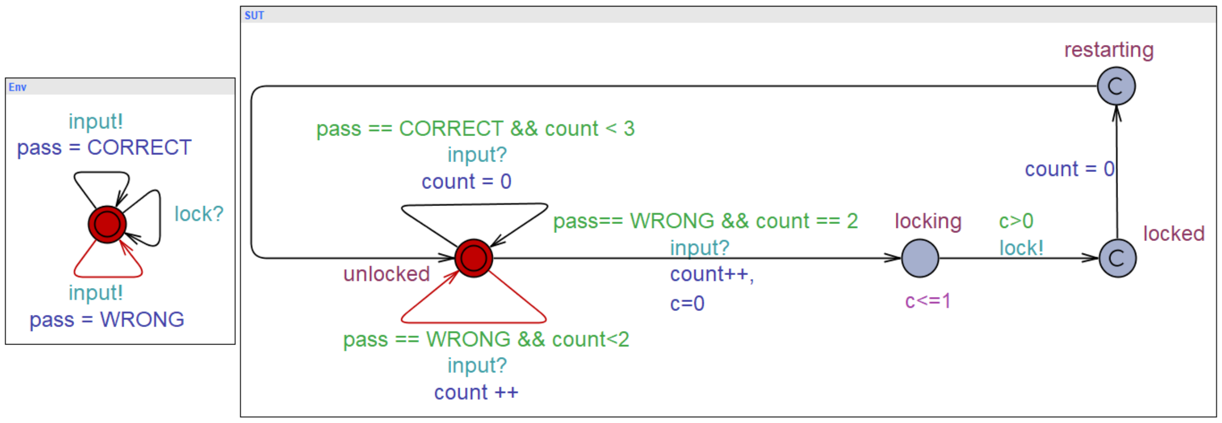


Figure 8. Example of UPPAAL model consisting of an environment automaton and a SUT automaton implementing the "three failed authentications" requirement

The correctness of the UPPAAL specification can be verified via a restricted set of TCTL. For instance, the following query verifies that the SUT implements the security requirement specified above.

A[] (count == 3 imply (SUT.locked || SUT.locking) && c<=1)

More complex verification queries are not always trivial to write due to the limitations of the UPPAAL TCTL, such as the nesting of path formulae [67]. In such cases, one can employ the so-called observer automata that would observe the same properties as the verification query.

An *observer automaton* is an automaton that follows the behavior of a system in order to check whether its behavior deviates from the expected one [68]. The observer encodes a logical property to be checked, and by being tightly composed with the system under observation, it observes relevant events coming from the latter. Special nodes, denoted as *reject nodes*, are included in the observer. During the synchronous execution of the observer against the system, a reachability graph is created. If during the execution one of the reject nodes is reached, then the property is not verified.

Depending on the level of intrusiveness in the system, the observers can be of two types [69]:

- *property observers who* do not modify the property of the system
- *restriction observers,* which allow expressing constraints on the system, i.e. to limit its behavior when a violation of the property is observed.

An example of an observer for the TCTL query above is given in Figure 9. As one can observe, the observer has a REJECT location, which all be reached when the property is not satisfied. One should note that the observer automaton must be tightly integrated with both the environment and the SUT models and has to have access to their shared variables in order to observe the state of the SUT. For that purpose, an adapter is required to capture and relay Environment<->SUT synchronizations to the observer. For instance, the input sent by the environment (via the *ainput* channel, is forwarded first to the Observer via the *oinput* channel, and to the SUT via the *input* channel. In a similar way, the output from the SUT, sent via the the *lock* channel to the Adapter is forwarded to the Observer via the *olock* channel and to the environment via the *alock* channel.

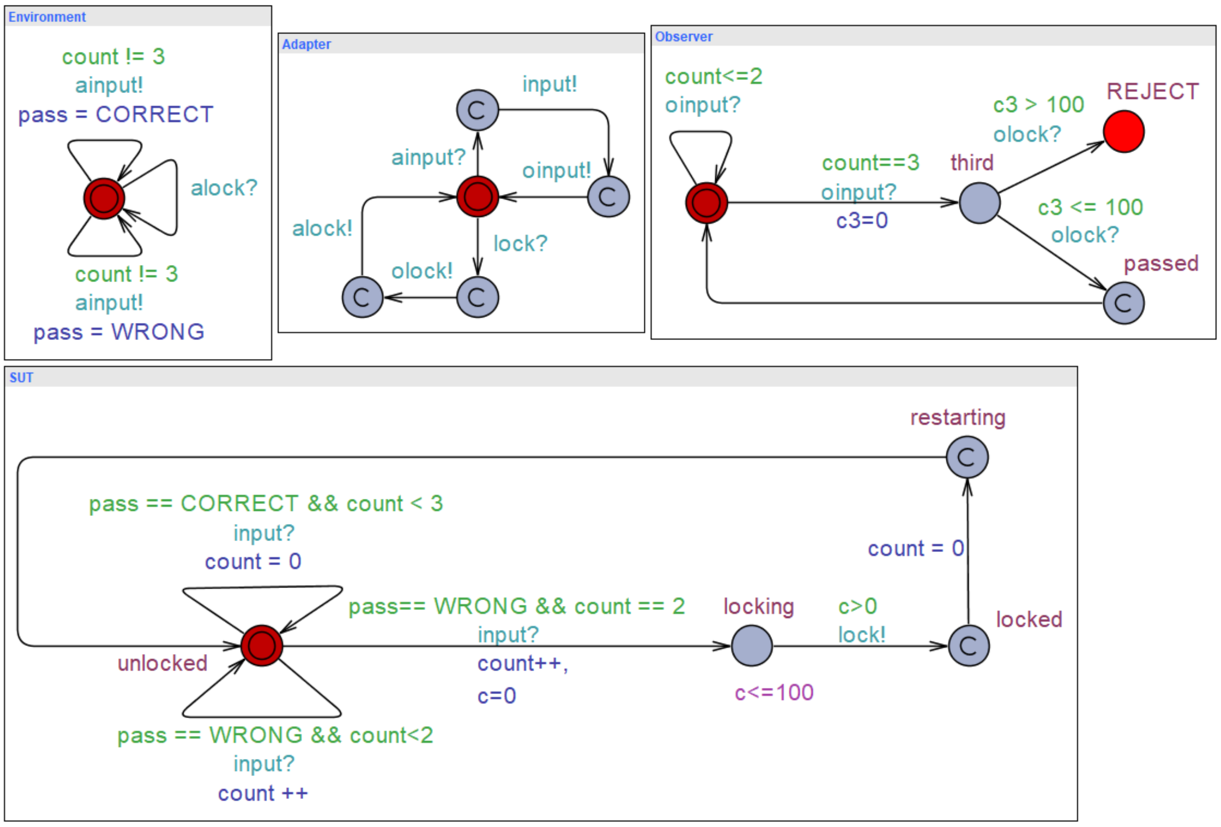


Figure 9. Example of UPPAAL TA implementing an observer, an adapter and an environment for the SUT in Figure 6.

Several observer automata, monitoring different security properties, can be attached to the SUT by extending the adapter automaton.

In VeriDevOps we investigate the use of observer automata to complement the verification, monitoring and test generation of security properties. To that extent, we will define: a) a set of mappings from CTL syntax to TA fragments and b) a set of composition rules for these mappings into complex queries. The approach will be accompanied by associated tool support for automation.

4. Conclusions

This deliverable D2.1 listed four formalisms that are proposed to be used in the context of the VeriDevOps to formalize security properties. The discussion focused on the relevance of these formalisms accompanied by small examples on how they are going to be applied in the project. Each of these formalisms provide different levels of expressiveness and rigorousness, and they come with the associated tool support. The associated tool support for using these formalisms will be presented in detail in future deliverables of the project.

References

[1] A. Sadovykh *et al.*, “VeriDevOps: Automated Protection and Prevention to Meet Security Requirements in DevOps,” in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, Feb. 2021, pp. 1330–1333.

[2] OWASP® Foundation, “OWASP Top Ten Proactive Controls 2018 - Security Requirements,” *OWASP projects*. https://owasp.org/www-project-proactive-controls/v3/en/c1-security-requirements (accessed Oct. 26, 2021).

[3] Jahl, “The information technology security evaluation criteria,” in *Proceedings - 13th International Conference on Software Engineering*, Jan. 1991, vol. 0, p. 306,307,308,309,310,311,312.

[4] “Trusted Computer Systems Evaluation Criteria,” Department of Defense, 1983.

[5] “Critères Communs des Technologies de l’Information. Partie 1 : Introduction et modèle général,” Jan. 2004.

[6] “Critères Communs des Technologies de l’Information. Partie CCIMB-99-032, Partie 2 : Exigences Fonctionnelles de Sècuritè,” Jan. 2004.

[7] “Critères Communs des Technologies de l’Information. Partie CCIMB-99-032, Partie 3 : Exigences d'Assurance de Sècuritè,” Jan. 2004.

[8] D. Firesmith, “A Taxonomy of Security-Related Requirements.” May 2005. [Online]. Available: https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=30108

[9] M. E. Calderón C., “A Taxonomy of Software Security Requirements,” *Revista Avances en Sistemas e Informática*, vol. 4, no. 3, pp. 47–56, 2007.

[10] D. Firesmith, “Engineering Security Requirements,” *Journal of Object Technology*, vol. 2, no. 1, pp. 53–68, January-February 2003.

[11] I. F. Alexander and R. Stevens, *Writing Better Requirements*. Pearson Education, 2002.

[12] B. Fabian, S. Gürses, M. Heisel, T. Santen, and H. Schmidt, “A comparison of security requirements engineering methods,” *Requirements Engineering*, vol. 15, no. 1. pp. 7–40, 2010. doi: 10.1007/s00766-009-0092-x.

[13] I. Rodhe, *Overview of Formal Methods in Software Engineering*. Totalförsvarets forskningsinstitut (FOI), 2015.

[14] D. E. Bell and L. J. LaPadula, “Secure Computer Systems: Mathematical Foundations.” 1973. [Online]. Available: citeseer.ist.psu.edu/548063.html

[15] J. A. Goguen and J. Meseguer, “Unwinding and Inference Control,” in *1984 IEEE Symposium on Security and Privacy*, Apr. 1984, vol. 00, pp. 75–75.

[16] A. Abou El Kalam *et al.*, “Organization Based Access Control,” Jun. 2003.

[17] K. J. Biba, MITRE CORP BEDFORD MA., and United States. Air Force. Systems Command. Electronic Systems Division, *Integrity Considerations for Secure Computer Systems*. 1977.

[18] D. D. Clark and D. R. Wilson, “A Comparison of Commercial and Military Computer Security Policies,” *Sociol. Perspect.*, p. 184, 1987.

[19] C.-F. Yu and V. D. Gligor, “A Specification and Verification Method for Preventing Denial of Service,” *IEEE Trans. Software Eng.*, vol. 16, no. 6, pp. 581–592, 1990.

[20] J. K. Millen, “A resource allocation model for denial of service,” presented at the 1992 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, USA, 2003. doi: 10.1109/risp.1992.213265.

[21] M. Sloman, “Policy driven management for distributed systems,” *Journal of Network and Systems Management*, vol. 2, no. 4, pp. 333–360, Dec. 1994.
[22] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, “Proposed NIST standard for role-based access control,” *ACM Trans. Inf. Syst. Secur.*, vol. 4, no. 3, pp. 224–274, Aug. 2001.
[23] R. K. Thomas and R. S. Sandhu, “Task-Based Authorization Controls (TBAC): A Family of Models for Active and Enterprise-Oriented Authorization Management,” in *DBSec*, 1997, pp. 166–181.
[24] R. K. Thomas, “Team-based access control (TMAC): a primitive for applying role-based access controls in collaborative environments,” in *Proceedings of the second ACM workshop on Role-based access control*, Fairfax, Virginia, USA, Nov. 1997, pp. 13–19.
[25] F. Cuppens, N. Cuppens-Boulahia, and T. Sans, “Nomad: a security model with non atomic actions and deadlines,” in *18th IEEE Computer Security Foundations Workshop (CSFW’05)*, Jun. 2005, pp. 186–196.
[26] F. Chen and R. S. Sandhu, “Constraints for role-based access control,” in *Proceedings of the first ACM Workshop on Role-based access control*, Gaithersburg, Maryland, USA, Dec. 1996, p. 14–es.
[27] R. J. Hayton, J. M. Bacon, and K. Moody, “Access control in an open distributed environment,” in *Proceedings. 1998 IEEE Symposium on Security and Privacy (Cat. No.98CB36186)*, Oakland, CA, USA, 2002, pp. 3–14.
[28] G.-J. Ahn and R. Sandhu, “The RSL99 language for role-based separation of duty constraints,” in *Proceedings of the fourth ACM workshop on Role-based access control - RBAC ’99*, Fairfax, Virginia, United States, 1999, pp. 43–54.
[29] J. M. Spivey, “An Introduction to Z and Formal Specifications,” *J. Software Engineering*, Jan. 1989.
[30] A. Boswell, “Specification and validation of a security policy model,” *IEEE Trans. Software Eng.*, vol. 21, no. 2, pp. 63–68, Feb. 1995.
[31] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, “Complexity and expressive power of logic programming,” *ACM Comput. Surv.*, vol. 33, no. 3, pp. 374–425, Sep. 2001.
[32] M. J. Sergot, F. Sadri, R. A. Kowalski, F. Kriwaczek, P. Hammond, and H. T. Cory, “The British Nationality Act as a logic program,” *Commun. ACM*, vol. 29, no. 5, pp. 370–386, May 1986.
[33] A. J. I. Jones and M. Sergot, “A Formal Characterisation of Institutionalised Power,” *Log. J. IGPL*, vol. 4, no. 3, pp. 427–443, Jun. 1996.
[34] J. Glasgow, G. Macewen, and P. Panangaden, “A Logic for Reasoning About Security,” *ACM Trans. Comput. Syst.*, vol. 10, no. 3, pp. 226–264, 1992.
[35] L. Cholvy and F. Cuppens, “Analyzing Consistency of Security Policies,” in *IEEE Symposium on Security and Privacy*, 1997, pp. 103–112.
[36] R. Ortalo, “A Flexible Method for Information System Security Policy Specification,” in *ESORICS*, 1998, pp. 67–84.
[37] H. Prakken and M. Sergot, “Contrary-to-duty obligations,” *Studia Logica*, vol. 57, no. 1, pp. 91–115, Jul. 1996.
[38] H. Prakken, “Dialectical Proof Theory for Defeasible Argumentation with Defeasible Priorities (Preliminary Report),” in *ModelAge Workshop*, 1997, pp. 202–215.
[39] S. Jajodia, P. Samarati, and V. S. Subrahmanian, “A logical language for expressing authorizations,” presented at the Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097), Oakland, CA, USA, 2002. doi: 10.1109/secpri.1997.601312.
[40] S. Barker, “Information Security: A Logic Based Approach,” in *ICEIS*, 2000, pp. 9–14.
[41] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, “Role-Based Access Control Models,” *IEEE Computer*, vol. 29, no. 2, pp. 38–47, 1996.
[42] J. Barkley, K. Beznosov, and J. Uppal, “Supporting relationships in access control using role based access control,” in *Proceedings of the fourth ACM workshop on Role-based access control*, Fairfax,

Virginia, USA, Oct. 1999, pp. 55–65.
[43] F. Cuppens and A. Miege, “Modelling contexts in the Or-BAC model,” presented at the 19th Annual Computer Security Applications Conference, 2003., Las Vegas, Nevada, USA, 2004. doi: 10.1109/csac.2003.1254346.
[44] F. Cuppens, N. Cuppens-Boulahia, and C. Coma, “MotOrBAC : Un Outil d’Administration et de Simulation de Politiques de Securite,” 2006.
[45] C. Ribeiro, A. Zuquete, P. Ferreira, and P. Guedes, “SPL: An Access Control Language for Security Policies with Complex Constraints,” 2001.
[46] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah, “First experiences using XACML for access control in distributed systems,” in *Proceedings of the 2003 ACM workshop on XML security*, Fairfax, Virginia, Oct. 2003, pp. 25–37.
[47] J. A. Hoagland, R. Pandey, and K. N. Levitt, “Security Policy Specification Using a Graphical Approach,” *arXiv [cs.CR]*, Sep. 30, 1998. [Online]. Available: http://arxiv.org/abs/cs/9809124
[48] A. Herzberg, Y. Mass, J. Mihaeli, D. Naor, and Y. Ravid, “Access control meets public key infrastructure, or: assigning roles to strangers,” presented at the 2000 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 2002. doi: 10.1109/secpri.2000.848442.
[49] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen, “Policy Core Information Model -- Version 1 Specification,” 2001. [Online]. Available: http://computer.org/proceedings/hicss/0001/00013/00013016abs.htm
[50] R. Podmore, D. Becker, R. Fairchild, and M. Robinson, “Common information model-a developer’s perspective,” in *Proceedings of the 32nd Annual Hawaii International Conference on Systems Sciences. 1999. HICSS-32. Abstracts and CD-ROM of Full Papers*, Jan. 1999, vol. Track3, p. 6 pp.–.
[51] J. Lobo, R. Bhatia, and S. A. Naqvi, “A Policy Description Language,” in *AAAI/IAAI*, 1999, pp. 291–298.
[52] VeriDevOps Project Consortium, “Deliverable D1.3 State of the art.” Mar. 31, 2021. [Online]. Available: https://veridevops.eu/state-of-the-art.pdf
[53] A. Naumchev, “Seamless Object-Oriented Requirements,” in *2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*, 2019, pp. 0743–0748.
[54] B. Meyer, M. Bertrand, and R. Milner, *Object-oriented Software Construction*. Prentice Hall, 1988.
[55] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, “Patterns in property specifications for finite-state verification,” *Proceedings of the 21st international conference on Software engineering - ICSE ’99*. 1999. doi: 10.1145/302405.302672.
[56] A. Naumchev, *Exigences Orientées Objets Dans Un Cycle de Vie Continu*. 2019.
[57] R. Sherman, A. Pnueli, and UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFORMATION SCIENCES INST., *Model Checking for Linear Temporal Logic: An Efficient Implementation*. 1990.
[58] A. Kurucz, F. Wolter, M. Zakharyaschev, and D. M. Gabbay, *Many-Dimensional Modal Logics: Theory and Applications*. Elsevier, 2003.
[59] J. A. W. Kamp, *Tense Logic and the Theory of Linear Order*. 1968.
[60] R. Fagin, Y. Moses, J. Y. Halpern, and M. Y. Vardi, *Reasoning About Knowledge*. MIT Press, 2003.
[61] E. Allen Emerson and E. E. Clarke, *Characterizing Correctness Properties of Parallel Programs Using Fixpoints*. 1980.
[62] R. Alur, C. Courcoubetis, and D. Dill, “Model-checking for real-time systems,” *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*. doi: 10.1109/lics.1990.113766.
[63] P. Filipovikj, G. Rodriguez-Navas, M. Nyberg, and C. Seceleanu, “Automated SMT-based consistency checking of industrial critical requirements,” *SIGAPP Appl. Comput. Rev.*, vol. 17, no. 4, pp. 15–28, Jan. 2018.

[64] F. ul Muram, H. Tran, and U. Zdun, “Systematic Review of Software Behavioral Model Consistency Checking,” *ACM Comput. Surv.*, vol. 50, no. 2, Apr. 2017, doi: 10.1145/3037755.
[65] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, “Testing Real-time Embedded Software Using UPPAAL-TRON: An Industrial Case Study,” in *Proceedings of the 5th ACM International Conference on Embedded Software*, Jersey City, NJ, USA, 2005, pp. 299–306.
[66] G. Behrmann, A. David, and K. G. Larsen, “A Tutorial on \sc Uppaal,” in *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, Sep. 2004, pp. 200–236.
[67] G. Behrmann, A. David, and K. Larsen, “A Tutorial on Uppaal,” in *Formal Methods for the Design of Real-Time Systems*, vol. 3185, M. Bernardo and F. Corradini, Eds. Springer Berlin Heidelberg, 2004, pp. 200–236.
[68] N. Halbwachs, F. Lagnier, and P. Raymond, “Synchronous Observers and the Verification of Reactive Systems,” in *Algebraic Methodology and Software Technology (AMAST’93)*, 1994, pp. 83–96.
[69] P. Dhaussy, J. C. Roger, H. Bonnin, E. Saves, J. Honore, and J. Lohman, “Experimentation of timed observers for avionics models validation,” Jan. 2006. Accessed: Oct. 12, 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02270437