D2.2 Specification of patterns for security requirements

Contract number:	957212
Project acronym:	VeriDevOps
Project title:	Automated Protection and Prevention to Meet Security Requirements in DevOps Environments
Delivery Date:	30/11/2021 (M13)
Coordinator:	SOFT
Partners contributed:	All
Release Date:	02.12.2021
Version:	1
Revision:	01
Abstract:	Based on the body of knowledge for requirements patterns, this deliverable elaborates specifics of the patterns for security formal properties patterns.
Status:	PU (Public)

Table of Contents

Executive Abstract

The current deliverable briefly presents three security requirements patterns approaches that are developed in the context of the VeriDevOps project. We discuss security requirements boiler plates that are intermediate forms between requirements in natural language, constrained natural language and formal representations in various formal languages. The boiler plates requirements have been presented in the context of security and addressed in several research papers. Another important dimension is temporal properties of security requirements. Thus we describe temporal patterns and show examples of their applicability in a security context. Finally we introduce the approach for formalizing requirements patterns in a form of object-oriented code which can be parameterized and reused as any classes, while providing means to integrate various representations for requirements and call corresponding verification means.

The document focuses on three tools by the VeriDevOps partners - ReSA[1] and PROPAS[2] - but also others by MDH and RQCODE[3] by SOFT. They should be used in combination since they provide complimentary services. ReSA checks requirements for consistency, PROPAS helps to add temporal properties, while RQCODE may integrate various representations and verification means for security requirements.

The deliverable presents the current work in progress for security requirements formalization and provides several examples for the VeriDevOps case studies by FAGOR and ABB. The methods and tools described in the deliverable will be further elaborated and released in subsequent deliverables of WP2.

1. Introduction

In VeriDevOps, *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. The proposed formalisms to be used are selected so that they are both expressive enough for the security requirements that we are going to use and suitable for the industrial case studies in the project. The security requirements patterns are essential for the automation of generating formal requirements from the textual sources in natural language. Thus, this deliverable contributes to the further deliverables on requirements generation.

[1] https://github.com/nasmdh/ReSA-Tool-0.0
[2] https://github.com/predragf/propas
[3] https://github.com/anaumchev/VDO-Patterns

This deliverable is structured as follows. Section 2 gives an overview of general concepts and methods applied for security requirements patterns definition and application. Section 3 provides links and briefly describes the concrete methods and tools implemented in the VeriDevOps project. Section 4 provides several examples of application of security requirements patterns in the context of case studies from VeriDevOps. Section 5 gives concluding remarks.

2. Natural Language and Patterns Specification Approaches

Most industrial system requirements are specified in natural language, Natural language is de facto requirements specification language. This is a way of expressing requirements in an intuitive and expressive way. Nevertheless, this means that these can be inherently ambiguous and too flexible. Despite this acknowledged situation, current specification methods and tools [1], [2], [3] lack adequate support to represent requirements in a more structured way and analyze the consistency of high-level natural language requirements, in order to improve the quality of their specification. There is evidence [4], [5] suggesting the following problems that can appear in natural language requirement documents:

- Ambiguity (a term or expression that has a couple or more different definitions).
- Vagueness (loss of accuracy, composition and detail).
- Complexity (composite requirements including mixed sub-clauses and various interrelated descriptions).
- Omission (missing requirements, especially conditions to control undesired behavior).
- Duplication (duplication of elements that represent the equivalent requirement).
- Wordiness (use of an excessive amount of information).
- Inappropriate implementation (descriptions of how the system should be created, rather than what it should do).
- Untestability (conditions that cannot be verified if the system is realized).

There are different obstacles that this research does not examine, including contradictory requirements and absent traceability connections. There are two main motivations for their exclusion: firstly, certain obstacles are not unique to natural language specifications texts, and secondly, there are no circumstances of these difficulties in various current studies from industry [4], [5].

Since engineers are very knowledgeable about disciplines like electronics, mechanics, hydrodynamics, microelectronics, mechatronics, etc., they are not so experienced in computer science and discrete computation. Training each engineer on defining requirements with for instance, temporal logic, would demand a lot of time and would be extremely expensive to be worthwhile. Moreover, it is not only the engineers who are required to understand the requirements. Different stakeholders at various levels of the business, e.g. customer assistance or support service, must prepare the requirements and verify them according to their particular demands.

Several researchers understand that this gap can be filled with the aid of software that will support the engineers in the method of automated conversion of requirements from natural language into temporal logics [6]–[8]. This method is usually called the *formalization of requirements*. There are some compelling tools on the market implementing this functionality [9], [10], but they are not generally accepted by practitioners. More experimentation is still required to learn how automated requirements will suit an organization's development process. An example of semi-formal specification languages are patterns (also known as *templates* or *boilerplates*) that can be in the form:

if <condition>, <System> shall do <Action> within <Time><Unit>.

The advantage of using these patterns relates to reuse, reduction of ambiguity, improvement of comprehensibility. In the next subsections we will outline some of the approaches for semi-formalization of requirements using patterns we will use in VeriDevOps.

2.1. Given-When-Then (GWT) Patterns

The Given-When-Then[4] was proposed by Dan North as part of behavior-driven development (BDD). Given-When-Then (GWT) is a semi-structured approach to writing requirements and test specifications more closely related to testing and test cases. It receives its name from the three applied conditions, starting with the words given, when and then (as shown in Figure 2.1). Given explains the preconditions and primary state before starting a test and provides for any pre-test settings that may happen. When is representing activities conducted by a user throughout a test. Then explains the consequence emerging from operations exercised in the when clause. Although Given-When-Then is a good way to describe interactions, state and behaviour, it is not a straightforward way to describe data, calculations and timing in a security context.

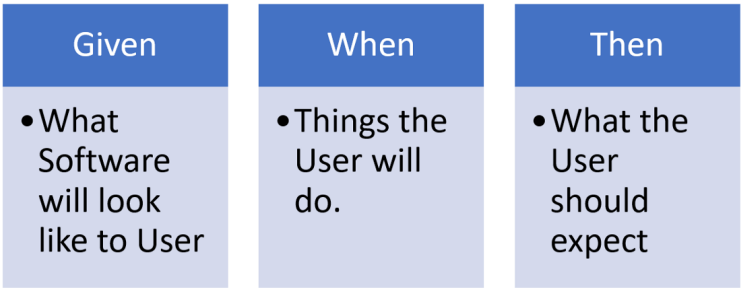

Figure 2.1. Given-When-Then Structure

2.2. Ontology-Based Patterns

One way to represent requirements is using Ontology technologies that are commonly used in various application domains now. Because concepts, relationships, and categorizations in the physical environment can be described in an ontology, this can be used as information resources, especially in a

[4] https://martinfowler.com/bliki/GivenWhenThen.html

specific utilization area. Moreover, different classes of semantic processing can be performed in requirements analysis without rigorous natural language processing (NLP) techniques using such ontology.

One of the most famous definitions of "ontology" is "formal, explicit specification of a shared conceptualization." For example, each requirement statement can be evaluated based on the principles of atomic elements of an application, and ontology is adopted as such knowledge (as shown in Figure 2.2). The edges shown in Figure 2.2 refer to conceptual relations, describing the main functionality of an automotive function, by a node-link diagram. The entities having oval shape are concepts of the ontology, except Driver, which is a concrete:

- System - refers to any physical or logical entity that can process an input and return an output.
- InPara - refers to input parameters, or properties of System.
- OutPara - refers to output parameters of System
- State - refers to operational modes of System
- Event - refers to internal or external occurrence that needs to be handled by System
- ActOnSys, ActOnPara, ActOnDriver - refer to (action) verbs that precede System, InPara/Outpara, Driver, respectively, in a requirement specification.

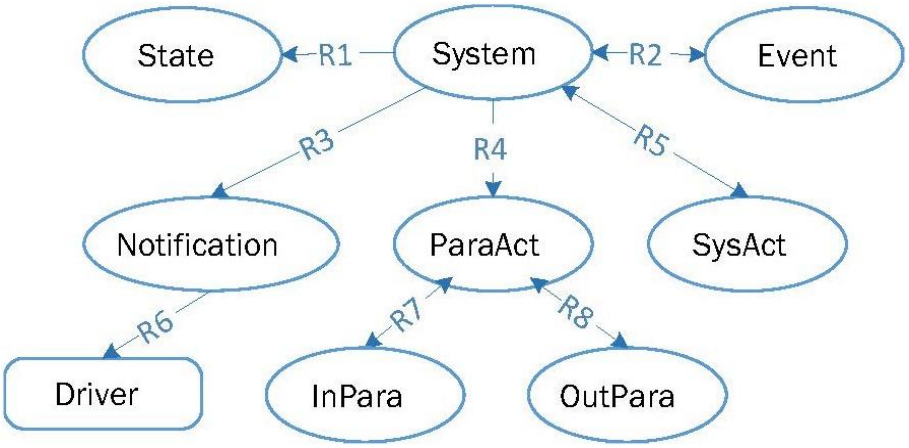

Figure 2.2. Example of an Ontology

The language (i) renders natural language terms (words, phrases), and syntax, (ii) uses an ontology that defines concepts and syntactic rules of the specification, and (iii) uses boilerplates to structure the specification.

2.3. TEARS Patterns

Some researchers from Rolls-Royce [4] proposed 20 years ago another way by adopting a certain generic requirements syntax. Having an uncomplicated composition pushes the requirement writer to

isolate the circumstances in which the requirement can be invoked (*preconditions*), the event that starts the condition (*trigger*), and the required system function (*system response*). Both preconditions and triggers are elective, depending on the requirement class.

The generic requirement syntax is specialised into five types of requirements - Ubiquitous, Event-driven, Unwanted behaviours, State-driven[5]:

- Ubiquitous requirements: The <system name> shall <system response>
- Event driven requirements: When <trigger>, the <system name> shall <system response>
- Unwanted behavior: If <trigger>, then the <system name> shall <system response>
- State driven requirements: While <precondition(s)>, the <system name> shall <system response>

A *ubiquitous element* has no preconditions or triggers. It is not requested by an event recognized by the component or in reply to a set system state but is permanently activated. The typical structure of a ubiquitous requirement can take the following form: “The control system shall prevent engine Overspeed”. This is a system behavior that needs to be working each time; therefore, this is a ubiquitous requirement.

A *state-driven requirement* is activated while the system is in a defined state. The keyword *While* is utilized to indicate state-driven conditions. The standard form of a state-driven requirement can look as follows: “While the system is working, the control system shall support combustible motor stream over certain threshold”. The system response is needed in all circumstances while the operation is in the defined state.

T-EARS, short for, Timed - Easy Approach to Requirements Syntax [11] was introduced as an easy way to use specification language for expressing timed passive test cases and has been continuously improved since then. The results of these studies on T-EARS show that the modified notation has a number of advantages over the use of unconstrained natural language. Even so, there are problems with this. This needs to be adapted to your domain, since concepts like time should be taken into account. The problems of ambiguity, vagueness and wordiness should be reduced, but not eliminated with this way of representing requirements.

2.4. RQCODE Formalization of Requirements Expressed in Natural Language

RQCODE [12] is a concept for the Object-Oriented (OO) representation of requirements and translation of requirements expressed in natural language into tests that helps to verify the conformance of the system to the requirements. The requirements in natural language are stored within one *class* with

[5] All these types of requirements can contain optional features and can be composed into more complex requirements.

tests that help to verify the conformance. This approach can be generalized to allow all representations (GWT, TEARS and etc.) of the requirement and verification means for conformance to be stored together so that various stakeholders may have a better understanding of the requirements.

This representation enables to apply the concepts of inheritance and encapsulation for implementing OO patterns to reuse and extend the requirements. For instance, the boiler plates may be implemented as methods that accept the parameters to be added to boiler plates to instantiate the requirements. Once a pattern is applied, the requirements engineer obtains access to all possible representations of the resulting requirement, including a structured natural language representation (to perform e.g. roundtrip engineering for validating the original NL requirement). Moreover, the representations in other languages will also be updated, e.g. formal verification properties, and tests. The pattern may also include graphical representations for requirements e.g. using UML. This approach gives a requirements engineer means to compare and analyse requirements based on these various representations. The embedded verification means make it possible to check the conformity or reinforce the requirements. The class-based approach makes it possible to implement another representation at any moment catering to arbitrary categories of stakeholders.

2.5. Real-Time Specification Patterns

Another solution relates to the *specification patterns system (SPS)* [11], [13] which is an approach proposed to facilitate the formal specification of system properties for practitioners who are not experts in formal methods. It is based on the premise that the systems' specs are outlined within reoccurring solutions, from which a set of patterns can be obtained and collected for later reuse. Each pattern is described by behavior that it captures and the program execution called a scope, within which the operation must operate. The patterns are represented as a sequence of literal and nonliteral terminals. The nonliteral terminals can be Boolean compositions that represent system properties or integer values that capture timing information. The rest of the pattern is made of literal terminals, describing the pattern's fixed part and cannot be changed;the original SPS catalog proposed by Dwyer et al. [14] is compiled by analyzing more than 500 examples of property specifications from various domains. It contains 13 qualitative patterns, which for easier navigation are divided into two categories: order and occurrence, expressed in various types of temporal logic (a subset is shown in Figure 2.3).

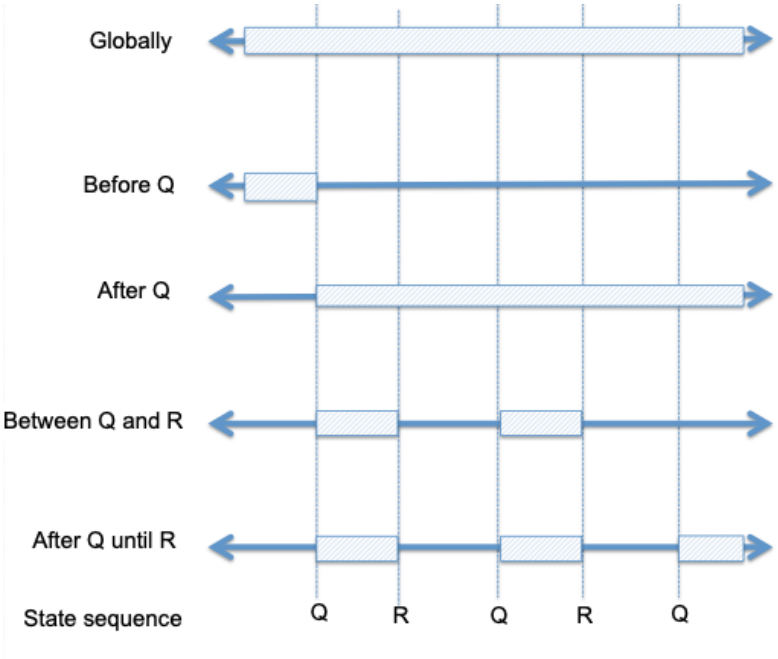

Figure 2.3. Real-time Specification Pattern System

Konrad and Chen [15] have consequently extended the specification patterns catalog with a new kind of patterns called real-time patterns to specify real-time systems properties. Therefore, the new and extended catalog was named *Real-time Specification Pattern System* (RTSPS). Besides introducing the new collection of patterns, Konrad and Cheng proposed an additional encoding of the specification patterns in *controlled natural language* (CNL). The proposed purpose of the CNL description is to allow various stakeholders that are not instructed in formal methods to understand and evaluate the system specification declared via patterns.
In the following, we provide examples of temporal requirement patterns.

Global universality pattern and its timed version

The global universality pattern takes the following form: "*Globally, it is always the case that P holds.*" Its timed version sets the minimal time period during which the desired property should be observed: *"Globally, it is always the case that P holds for at least T time units."* This and other universality patterns are available in different notations[6].

[6] https://people.cs.ksu.edu/~dwyer/SPAT/universality.html

Global real time response pattern (timed)

The global real time response pattern takes the following form: *"Globally, it is always the case that if P holds, the S eventually holds within T time units."* This and other response patterns are available online[7].

After-until universality pattern and its timed version

The after-until universality pattern takes the following form: *"After Q, it is always the case that P holds until R holds."* Its timed version depends on two deadlines: *"After Q holds for T time units, it is always the case that P holds until R holds for at most T1 time units."* This universality pattern is available[8].

2.6. Security Patterns as Boilerplates

Several authors have focused specifically on ontologies and requirements boilerplates targeting security. The use of ontologies provides the necessary background knowledge, and domain knowledge that is required to identify security threats, and recommend appropriate countermeasures, while the requirements boilerplates provide a reusable template for writing requirements in a consistent way in order to eliminate ambiguity.

Firesmith et al. [16] identified four different types of defence against security threats, which can be used to assign specific security threats to the types of defence actions to counter them. These are:

- Prevention of malicious harm, security incidents, security threats and security risks.
- Detection of malicious attack, security incidents, security threats, and security risks.
- Reaction to detected malicious attack.
- Adaptation of system to avoid or minimize the negative consequences of the malicious harm, security incidents, security threats and security risks. This could also be in terms of recovery of the system from attacks.

The following examples by Daramola et al. [17], could be the basis for some generic security boilerplates in VeriDevOps, e.g.:

- Security Boilerplate 1: The <system> should [prevent | detect] at least <percentage> of <harm | incident | threat | risk>
- Security Boilerplate 2: Upon detection of <harm | incident | threat | risk> the system shall <action>
- Security Boilerplate 3: ...of attacks with maximum duration <time unit>
- Security Boilerplate 4: ...made by attackers with profile <attacker profile>
- Security Boilerplate 5: ...at least <percentage> of the time

[7] https://matthewbdwyer.github.io/psp/patterns/response.html
[8] https://people.cs.ksu.edu/~dwyer/SPAT/universality.htm

3. Implementation of Security Requirements Patterns

3.1. ReSA Tool for Structured Requirements Specification

The ReSA toolchain is an Eclipse-based implementation of the RESA requirements specification language. The toolchain supports contextual content completion and text validation features. Furthermore, it seamlessly integrates a function for checking the logical consistency of requirements using the Z3 SMT[9] solver (as shown in Figure 3.1). The toolchain also supports specifying requirements at different levels of software development, using appropriate concepts valid at a specific level of abstraction. This tool has been specialized for EAST-ADL[10], with respect to the vehicle, analysis, and design level of abstraction, but it can be extended to any other architectural or block diagram descriptions. The Graphical User Interface of the toolchain is shown in Figure 3.2, displaying demo projects, both EAST-ADL generic specification, as well as EAST-ADL abstraction level aware specification. The RESA open-source toolchain is available for download from Github[11].

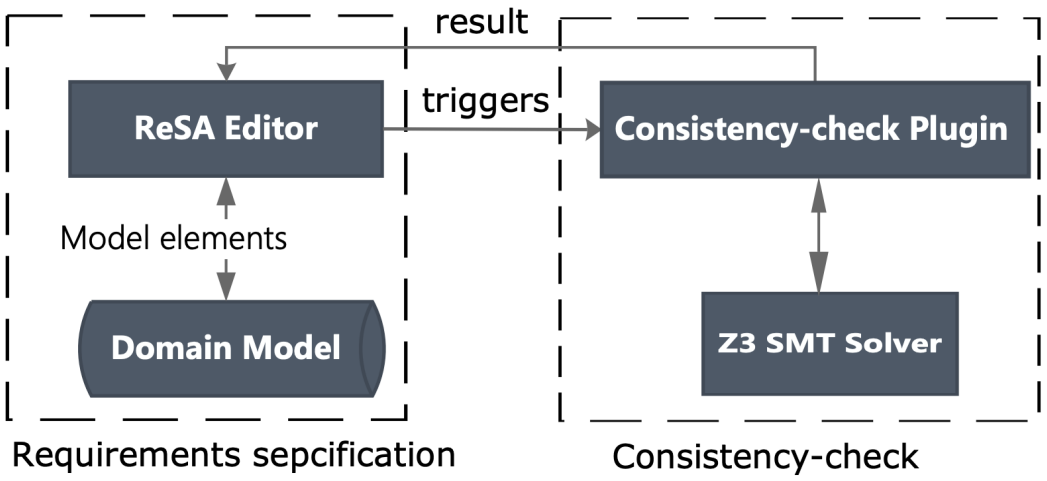

Figure 3.1. The ReSA tool chain architecture.

Figure 3.1 shows the architecture of the ReSA toolchain. It consists of requirements specification and consistency checking of requirements. The specification part is basically the ReSA specification editor (a.k.a. Resa App), and a domain model. During writing requirements specifications, domain elements can be accessed from the domain model, but also model elements can be populated during specification. Such an approach allows the consistent use of terms among different requirements engineers, reduces typographic errors, and maintains a knowledge base for later system refinements.

[9] https://github.com/Z3Prover/z3
[10] https://www.east-adl.info/
[11] https://github.com/nasmdh/ReSA-Tool-0.0.git

The consistency checking part consists of a consistency checking plugin that calls the Z3 SMT solver. The result of the consistency checking is returned to the editor's perspective.

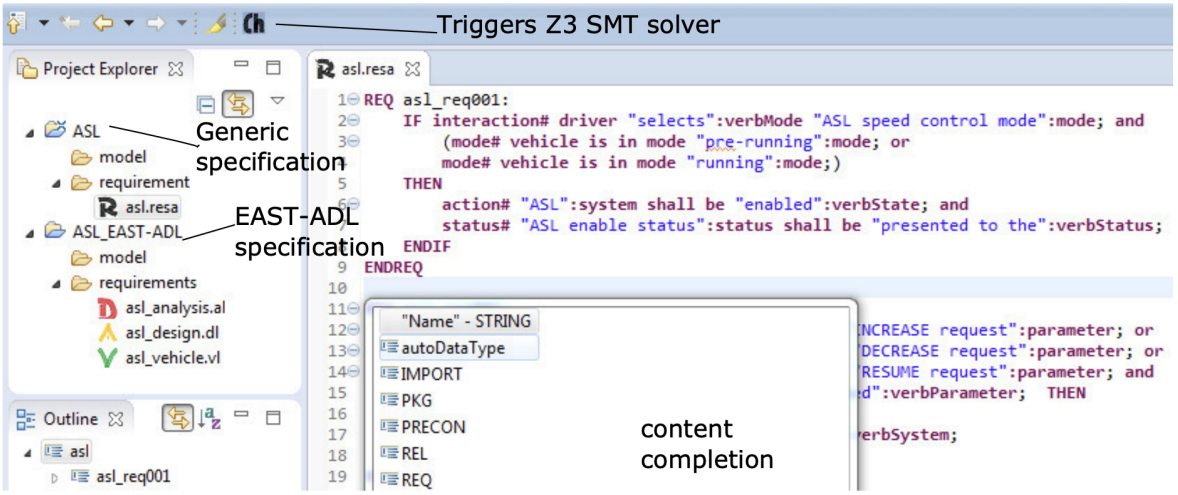

Figure 3.2. The ReSA tool chain user interface.

ReSA will be used in both ABB and FAGOR use cases. We use a requirement specification structured using boiler plates and check their logical consistency. In the ABB use case, we use the ReSA tool to specify requirements (functional and security related) from textual descriptions and the Function Block Diagram[12] structural and architectural model.

ReSA Specification of Requirements: The framework consists of the Hierarchical Grammar, the ReSA Application, and the System Model. The Hierarchical Grammar is composed of a generic grammar and grammar definitions for each EAST-ADL abstraction level. This grammar defines the generic rules of constructing requirements specifications in automotive systems. It uses automotive concepts to typeset domain elements and action verbs associated with instances of concepts. The grammar defines the syntax of the boilerplates and the requirements specification that is built from the boilerplates. The following context-free grammar is a snippet of the ReSA language from which one can build patterns:

- <specification> |= <simple> | <complex> | <nested-complex>…
- <simple> |= <mainClause>.
- <complex> |= <subClause> , <mainClause>.
- <mainClause> |= <primitiveClause> | ...
- <primitiveClause> |= <sub> <verb> <obj> | ....

[12] For more details on Function Block Diagrams we refer the reader to: https://help.codesys.com/api-content/2/codesys/3.5.14.0/en/_cds_f_programming_language_fbd_ld_il/

For the detailed grammar specification of the language, the reader is invited to refer to [18] or the RESA documentation[13]. For example the following requirements can be specified using RESA:

- System shall send notification:status every 200ms
- if User selects control:mode and system in running mode then system shall be enabled within 200ms
- After system is enabled, if Button is pressed, system shall be activated
- If user selects control:mode then system shall be disabled.

In addition, the Security Pattern Boilerplates mentioned in section 2 are a natural fit to the ReSA tool.

3.2. PROPAS Patterns

In PROPAS, we use a pattern-based approach that uses predefined pattern-based templates [19] 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. Requirements can be written manually or generated from requirement patterns using PROPAS (The PROperty PAttern Specification and Analysis) [20]. PROPAS is a tool set for automated pattern-based formalization of industrial critical requirements written in natural language. A screenshot of this tool is shown in Figure 3.3.

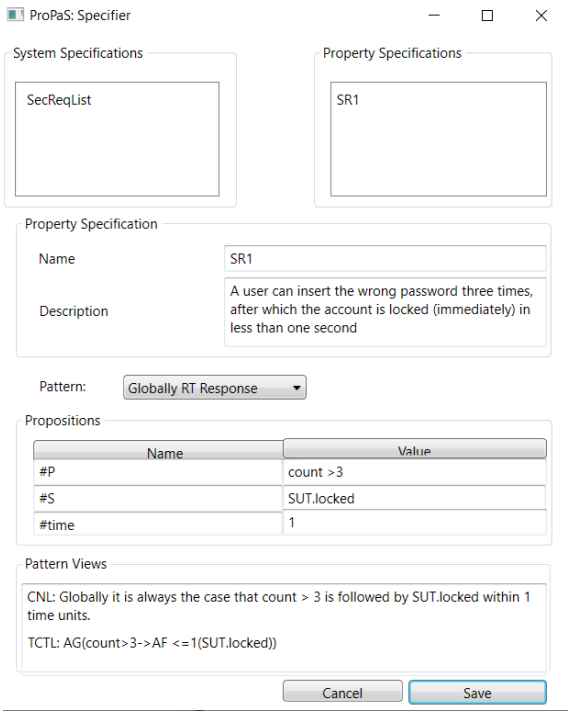

Figure 3.3. User interface of the ProPas tool.

[13] https://bitbucket.org/nasmdh/resa/src/master/

P1: Globally, Universally: *Globally, it is always the case that P holds.*

P2: Timed Globally, Universally: *Globally, it is always the case that if P held for T time units, then S holds.*

P3: Globally, Real-time Response: *Globally, it is always the case that if P holds, the S eventually holds within T time units.*

P4: After Q Until R Universally P: *After Q, it is always the case case that P holds until R holds.*

P5: Timed After Q Timed Until R Universally P: *After Q holds for T time units, it is always the case that P holds until R holds or at most T_1 time units.*

Figure 3.4. Real-time Specification Pattern Systems in PROPAS.

In VeriDevOps work, we use the subset of real-time specification patterns shown in Figure 3.4 and mentioned earlier in Section 2. These enable various stakeholders who are not trained in formal methods to read and interpret the system specification expressed via patterns.

3.3. RQCODE: Object Oriented Requirements as Code

This section describes an additional approach to the formal specification of security pattern based on the Software Object-Oriented Requirements.

3.3.1. STIG Rules as Security Requirements Patterns

This section explains the RQCODE (ReQuirements as CODE) approach to specifying requirements and their patterns, applied to the collection of **Security Technical Implementation Guide (STIG)** rules as per request by case study partner FAGOR.

To demonstrate the approach we will illustrate with an example of specific STIG rules for systems run by the Windows 10 operating system. These rules are provided in the Windows 10 Security Technical Implementation Guide. For many systems, STIG rules come scripts for verifying the conformance and enforcing the rules. For Windows 10, FAGOR developed PowerShell scripts that check conformance of systems to these rules. For some rules, such checks are complemented with scripts that enforce conformance to the respective rules.

STIG Rules in RQCODE

We had analyzed the STIG rules given from FAGOR and found subgroups of rules that look very similar – both in their textual descriptions and in the PowerShell scripts (where applicable) that check and enforce conformance to these rules. This is bad, because a decision to modify one rule from such a

subgroup would require synchronizing the change with all the similar rules, and this process is prone to errors. We decided to apply the object-oriented software construction process to remove the repetition. The below class tree depicts a subset of the resulting collection of classes:

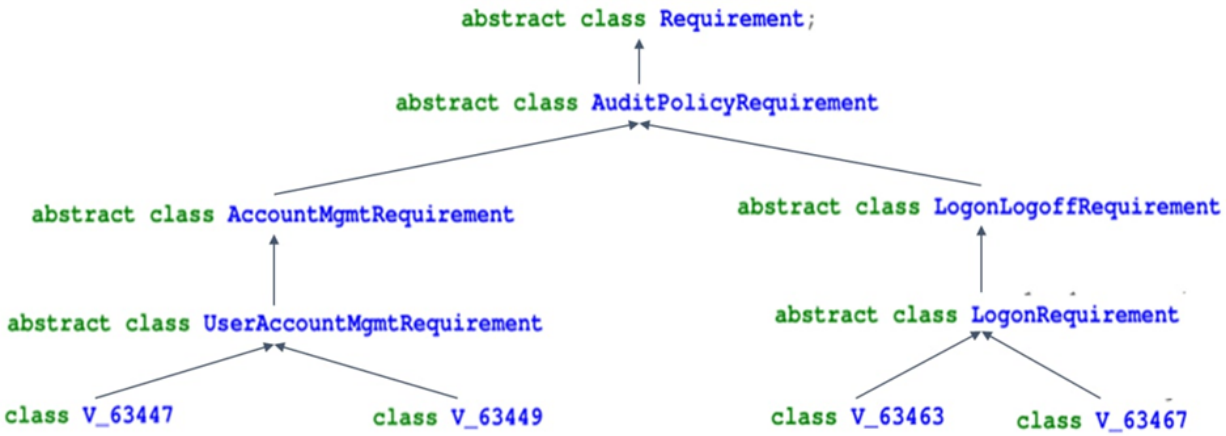

Figure 3.5. RQCODE patterns for STIG rules.

The leaves of the tree correspond to actual STIG rules. The abstract classes encode the commonalities shared by their descendant classes. Each class implementing a STIG rule features up to three public methods:

- toString(), which prints out the textual representation of the STIG rule;
- check(), which performs conformance-checking of the target system against the rule;
- enforce(), which enforces conformance of the target system against the rule.

3.3.2. Temporal Security Patterns in RQCODE

This section enumerates patterns for specifying security requirements as temporal and timed properties.

When applying the RQCODE approach to the Temporal and real-time security requirement patterns, we wanted to take advantage of the already implemented STIG rules from FAGOR. In particular, we wanted the checks inside the STIGs implementations to be reusable as the properties the temporal patterns talk about. The resulting implementation can be found on GitHub: https://github.com/anaumchev/VDO-Patterns/tree/master/src/patterns/temporal.

Checkable interface

All RQCODE implementations of the STIG rules implement *Checkable* interface that offers a method returning the result of checking the target system against the corresponding rule:

```
public interface Checkable {
    enum CheckStatus {
        PASS, FAIL, INCOMPLETE
    }

    CheckStatus check();
}
```

Abstract monitoring loop

To verify a temporal real time property means to monitor the target system over some period to see if it meets the desired property. This is why all RQCODE implementations of these patterns are descendants of *MonitoringLoop* class that implements the *Checkable* interface:

```
public abstract class MonitoringLoop implements Checkable {
    protected int boundary = 0;
    final private int variant(int i) {
        if (boundary > 0) { return i - 1; }
        return i;
    }
    protected int sleepMilliseconds() { return 1000; }
    protected boolean invariant() { return true; }
    protected boolean precondition() { return true; }
    protected boolean postcondition() { return true; }
    protected boolean exitCondition() { return false; }
    final public Checkable.CheckStatus check() {
        while(!precondition()) {
            try {
                Thread.sleep (sleepMilliseconds());
            } catch (InterruptedException e) {
                return Checkable.CheckStatus.INCOMPLETE;
            }
        }
        for (int i = boundary; i >= 0 && !exitCondition(); i = variant(i))
{
            if (!invariant()) {
                return Checkable.CheckStatus.FAIL;
            }
            try {
                Thread.sleep (sleepMilliseconds());
            } catch (InterruptedException e) {
                return Checkable.CheckStatus.INCOMPLETE;
            }
        }
        if(!postcondition()) {
            return Checkable.CheckStatus.FAIL;
        }
        return Checkable.CheckStatus.PASS;
    }

    abstract public String TCTL();

}
```

The *check()* method contains the monitoring loop itself. Its implementation cannot be redefined but it is expressed in terms of conceptual methods that can be redefined in descendant classes:

- *boundary* limits the number of time units to execute the loop; 0 means no bound;
- *sleepMilliseconds()* controls the length of a time unit in milliseconds;
- *invariant()* must hold on every step of the loop;
- the loop waits for *precondition()* to hold before it start checking desired properties;
- *postcondition()* is checked after the loop terminates;
- *exitCondition()* specifies the exit condition of the loop.

Security requirements may not only be monitored but also model checked. The RQCODE requirements may contain several representations for the same requirement, for example, including one in the TCTL. The *TCTL()* abstract method of the MonitoringLoop class forces concrete classes to implement TCTL representation of the pattern. The *variant(int i)* method implements bounded (un)execution of the monitoring loop: if the value of boundary is 0 then variant returns the value of its argument unchanged; otherwise, it returns the decremented value of the argument. If the input value does not change, the i >= 0 part of the continuation condition of the loop will remain true forever, and the loop will only exit if *exitCondition()* becomes true.

Example: Global Universality pattern

The following class implements the Global Universality pattern:

```
public class GlobalUniversality extends MonitoringLoop {
```

The class contains a Checkable field p that captures the condition expected to hold forever:

```
public GlobalUniversality(Checkable p) {
    this.p = p;
}
protected Checkable p;
```

The class overrides the invariant() method so that it only returns true if p passes the check:

```
@Override
public boolean invariant() {
    return (p.check() == CheckStatus.PASS);
}
```

This check() method of MonitoringLoop will repeatedly check this condition inside the loop.

This is how the TCTL representation of the pattern is constructed:

```
public String TCTL() {

      String pStr;

      if (p instanceof MonitoringLoop) {

      pStr = ((MonitoringLoop) p).TCTL();

      } else {

      pStr = p.getClass().getSimpleName();

      }
```

```
        return "AG (" + pStr + ")";

}
```

If p itself implements *MonitoringLoop*, we embed its TCTL representation into the generic TCTL scheme of the pattern; otherwise, we embed the name of the dynamic type of p.

We do not discuss how the textual representation is computed inside the *toString()* method because its implementation is straightforward.

Timed versions of existing patterns

The timed version of the global universality pattern extends the non-timed version:

```
public class GlobalUniversalityTimed extends GlobalUniversality {
```

We update the constructor so that it can also bound the execution of the monitoring loop:

```
public GlobalUniversalityTimed(Checkable p, int boundary) {

        super(p);

        this.boundary = boundary;

}
```

The *TCTL()* and *toString()* methods are updated accordingly in a straightforward way.

4. Application to the VeriDevOps Project

4.1. Application to PLC-Level Security Vulnerabilities

In VDO, our approach starts with a combination of requirements patterns that can be used for functional, non-functional and security monitoring, verification and testing. Therefore we aim to start with textual requirements, use NLP and/or human/tool interaction to get to instantiated requirements patterns, formalize these if possible and then use them in the later stages of development and operations.

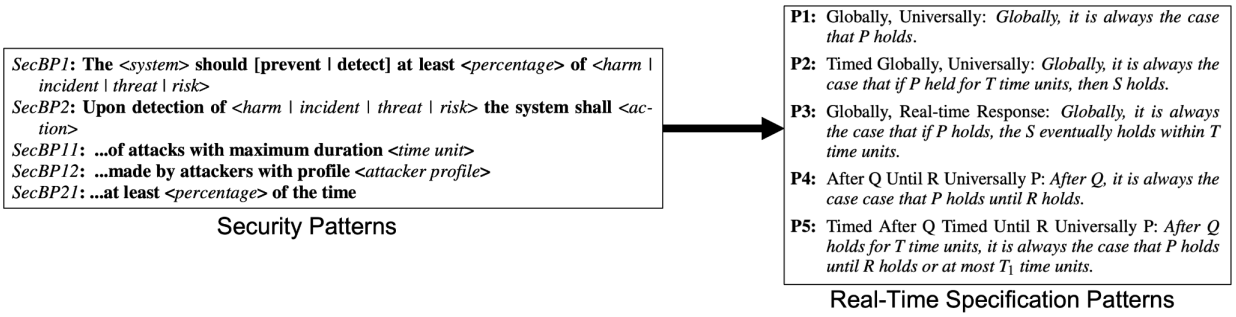

Figure 4.1. Going from Security Patterns to Real-Time Specification Patterns.

Programmable Logic Controllers (PLCs) are widely used in real-time software for many types of software systems including nuclear plants and train systems. A PLC is an integrated embedded system that contains a processor, a memory, and a communication bus. These systems are programmed using IEC 61131-3 programming languages. The PLC security threats landscape is complex, spanning different layers such as: people, application layer, operating system, communications, policies, hardware, software/logic, etc. PLCs interact with their surrounding components such as sensors, actuators, human machine interfaces, via networks and subnetworks. Thus they can be easily exposed to attacks, including the control logic attacks. Such attacks are performed in order to modify the control logic of the PLC either through program payload/code modification and/or program input manipulation.

Since PLC programs can carry within their own threats and vulnerabilities that can be exploited by hackers or regular users, vulnerabilities are arising from the way the PLC code is written or designed. Some examples of PLC-level security vulnerabilities are:

- Overflow: occurs when a parameter length of input or output does not match what the PLC is expecting. That usually occurs because of programmers or when a malicious attack manipulates parameters.
- Hard coded values: in certain situations, using hard coded parameters in instructions like comparative blocks could increase vulnerabilities. Parameters used in the comparison blocks can be manipulated by users, hackers, or malicious code.
- Infinite loops, nested timers, etc.
- Fatal Faults: trigger certain faults that could cause PLC programs to crash.

As an example of a security attack, we can devise one based on a certain security requirement related to the application level of a PLC as follows. The PLC is implemented in one of the existing specification languages and contains timing information. This time period can be changed by a malicious user via a timer preset. If the attacker modified the timer present to be greater than a certain value then the system could halt its operation due to alarm output being sent too late. This security and safety failure can affect such systems. In the context of the PLC use case, any data entry points may be exploited by malicious users. Users may exploit these points of entry, directly or indirectly, to modify control data. In PROPAS, RESA, GWT and TEARS patterns we can systematize PLC-level attack methodologies related to program source code. At the source code level, the code injection or modification has to be stealthy, in

a way that no observable changes would be introduced to the major functionality of the program, or masked as novice programmer mistakes. In other words, the attacks could be disguised as unintentional bad coding practices.

4.2. RQCODE Example for FAGOR Case

In the context of the VeriDevOps project FAGOR provided a case study to verify the application of STIG rules for Industrial PC running Windows 10 operating system. In this section we will see some examples of applying the RQCODE patterns. The RQCODE has been presented in Section 3. This example combines the temporal patterns with STIG patterns to monitor it continuously.

The following line creates an object from the RQCODE class implementing the V_63449 STIG[14] rule:

```
V_63449 v_63449 = new V_63449();
```

We then can construct a timed global universality RQCODE requirement from v_63349 since this rule should be permanently enforced:

```
GlobalUniversalityTimed globalUniversality_V_63449 =

      new GlobalUniversalityTimed(v_63449, Integer.MAX_VALUE);
```

Property V_63449 is expected to globally hold for at least Integer.MAX_VALUE time units.

Let us now create another object from another STIG rule's RQCODE implementation:

```
V_63467 v_63467 = new V_63467();
```

Now we have v_63449 and v_63467 at our disposal and can construct an RQCODE requirement from the timed global response pattern:

```
GlobalResponseTimed globalResponseTimed_V_63449_V_63447 =

      new GlobalResponseTimed(v_63449, v_63467, Integer.MAX_VALUE);
```

The resulting requirement expresses the following: *"globally, whenever v_63449 holds, v_63467 should hold within Integer.MAX_VALUE time units".*

But it is not only possible to construct pattern-based RQCODE requirements from "primitive" implementers of the Checkable interface – the "building blocks" may by other pattern-based requirements:

```
GlobalResponseTimed globalResponseTimedComposite =
```

[14] https://www.stigviewer.com/stig/windows_10/2016-06-08/finding/V-63449

```
        new GlobalResponseTimed(globalUniversality_V_63449,

                globalResponseTimed_V_63449_V_63447, Integer.MAX_VALUE);
```

We can print out textual representations of the constructed objects:

```
System.out.println(globalResponseTimedComposite);
```

The output is too large to discuss it here but the code can be downloaded from the repository and executed locally.

We can also print out a TCTL representation:

```
System.out.println(globalResponseTimedComposite.TCTL());
```

The previous line will produce the following output:

AG((AG >=2147483647 (V_63449)) ==> (AF <= 2147483647 (AG((V_63449) ==> (AF <= 2147483647 (V_63467))))))

Finally, it is possible to monitor RQCODE requirements by calling check():

```
System.out.println(globalResponseTimedComposite.check());
```

It will launch the monitoring loop and check whether the property captured by the object holds.

5. Conclusions

In this document we have briefly discussed the methods and tools proposed in VeriDevOps for security requirements, patterns definition and application. The approaches are complementary. ReSA is focusing on natural language boilerplates and consistency checking, while PROPAS concentrates on temporal properties. RQCODE may effectively be used as integration means to merge several approaches. The requirements representations and verification means can be easily integrated in RQCODE concept that represents the requirements as objects in Java.

In the document we have shown different examples of security requirements applied to the examples from VeriDevOps case studies. In the following stages of the project, we will further apply these methods and tools to the case studies for evaluation.

We also have identified a number of directions for improvement of the current methods and tools. ReSA currently focuses on the automotive domain and needs to be adapted to new domains such as industrial control. New security oriented boilerplates mentioned in Section 2 have to be experimented in the ReSA context. The PROPAS is a great means as an intermediate representation bridging the requirements in natural language through constrained natural language to the temporal requirements.

PROPAS has to be further improved to accept inputs from VeriDevOps Task 2.1 on Natural Language Processing for requirements extraction and formalization.

RQCODE, in its turn, is an nascent stage we intend its possible development in the following directions:

- Implement generation of Gherkin representation to support behavior-driven development (BDD). Gherkin is the notation used to express BDD-style properties in natural language.
- Implement concurrent monitoring. Sometimes temporal properties say something like “if a globally holds, then b will also hold at some point”. In this case we cannot check a and only then check b – both have to be checked in parallel, and the results should be evaluated after both of the checks terminate.
- Train a natural language processing based model to guess RQCODE patterns based on natural language inputs.

References

[1] S. Farfeleder, T. Moser, A. Krall, T. Stalhane, H. Zojer, and C. Panis, “DODT: Increasing requirements formalism using domain ontologies for improved embedded systems development,” *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*. 2011. doi: 10.1109/ddecs.2011.5783092.

[2] R. Bloem *et al.*, “RATSY – A New Requirements Analysis Tool with Synthesis,” *Computer Aided Verification*. pp. 425–429, 2010. doi: 10.1007/978-3-642-14295-6_37.

[3] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj, “SCR*: A Toolset for Specifying and Analyzing Software Requirements.” 1998. doi: 10.21236/ada465334.

[4] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak, “Easy Approach to Requirements Syntax (EARS),” *2009 17th IEEE International Requirements Engineering Conference*. 2009. doi: 10.1109/re.2009.9.

[5] T. Tahvonen and E. Uusitalo, “Easy Approach to Requirements Syntax in Nuclear Power Plant Safety Design,” *2018 1st International Workshop on Easy Approach to Requirements Syntax (EARS)*. 2018. doi: 10.1109/ears.2018.00006.

[6] R. L. Cobleigh, G. S. Avrunin, and L. A. Clarke, “User guidance for creating precise and accessible property specifications,” *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering - SIGSOFT ’06/FSE-14*. 2006. doi: 10.1145/1181775.1181801.

[7] S. Konrad and B. H. C. Cheng, “Facilitating the construction of specification pattern-based properties,” *13th IEEE International Conference on Requirements Engineering (RE’05)*. 2005. doi: 10.1109/re.2005.29.

[8] S. P. Overmyer, L. Benoit, and R. Owen, “Conceptual modeling through linguistic analysis using LIDA,” *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. doi: 10.1109/icse.2001.919113.

[9] G. Fanmuy, A. Fraga, and J. Llorens, “Requirements Verification in the Industry,” *Complex Systems Design & Management*. pp. 145–160, 2012. doi: 10.1007/978-3-642-25203-7_10.

[10] A. Ferrari, F. Mazzanti, D. Basile, M. H. ter Beek, and A. Fantechi, “Comparing formal tools for system design,” *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020. doi: 10.1145/3377811.3380373.

[11] D. Flemstrom, H. Jonsson, E. P. Enoiu, and W. Afzal, “Industrial Scale Passive Testing with T-EARS,” *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2021. doi: 10.1109/icst49551.2021.00047.

[12] K. Ismaeel, A. Naumchev, A. Sadovykh, D. Truscan, E. P. Enoiu, and C. Seceleanu, “Security Requirements as Code: Example from VeriDevOps Project,” *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)*. 2021. doi: 10.1109/rew53955.2021.00063.

[13] 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.

[14] G. S. Avrunin, J. C. Corbett, and M. B. Dwyer, “Benchmarking finite-state verifiers,” *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4. p. 317, 2000. doi: 10.1007/s100090050038.

[15] S. Konrad and B. H. C. Cheng, “Real-time specification patterns,” *Proceedings. 27th International*

Conference on Software Engineering, 2005. ICSE 2005. doi: 10.1109/icse.2005.1553580.
[16] D. Firesmith, “Engineering Security Requirements,” *The Journal of Object Technology*, vol. 2, no. 1. p. 53, 2003. doi: 10.5381/jot.2003.2.1.c6.
[17] O. Daramola, G. Sindre, and T. Stalhane, “Pattern-based security requirements specification using ontologies and boilerplates,” *2012 Second IEEE International Workshop on Requirements Patterns (RePa)*. 2012. doi: 10.1109/repa.2012.6359973.
[18] N. Mahmud, C. Seceleanu, and O. Ljungkrantz, “ReSA: An ontology-based requirement specification language tailored to automotive systems,” *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2015. doi: 10.1109/sies.2015.7185035.
[19] A. Hoffmann, “A Pattern-based approach for analysing requirements in socio-technical systems engineering,” *2012 20th IEEE International Requirements Engineering Conference (RE)*. 2012. doi: 10.1109/re.2012.6345834.
[20] P. Filipovikj, *Automated Approaches for Formal Verification of Embedded Systems Artifacts*. 2019.