D1.3

Modelling framework description

Project number:	731453
Project acronym:	VESSEDIA
Project title:	Verification engineering of safety and security critical dynamic industrial applications
Start date of the project:	1st January, 2017
Duration:	36 months
Programme:	H2020-DS-2016-2017

Deliverable type:	Report
Deliverable reference number:	DS-01-731453 / D1.3 / 1.0
Work package contributing to the deliverable:	WP 1
Due date:	Jun 2018– M18
Actual submission date:	29th June, 2018

Responsible organisation:	CEA
Editor:	Shuai Li
Dissemination level:	PU
Revision:	1.0

Abstract:	This document specifies the modelling framework for secure software development within the VESSEDIA project. It also proposes the usage of the framework within a software/security co-engineering method. Tool implementation specifications for D1.4 are given. Early examples are shown on a simple ping protocol.
Keywords:	Architecture framework, Architecture Description Language, ISO 42010, UML, SysML, ACSL, xLIA, Software/security co-engineering

The project VESSEDIA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731453.

Editor

Shuai Li (CEA)

Contributors (ordered according to beneficiary numbers)

Shuai Li (CEA)

Boutheina Bannour (CEA)

Imen Boudhiba (CEA)

Vincent Lorenzo (CEA)

Disclaimer

Executive Summary

In this document, the VESSEDIA modelling framework is proposed for the development of secure software. The goal of the modelling framework is to bridge the gap between high-level textual requirements in an architecture model and low-level properties in the code.

The particularity of the VESSEDIA modelling framework is that it does not propose new ADLs or risk assessment methodologies. It rather aggregates existing approaches and uses generic enough artefacts to be extended with new approaches in the future.

The VESSEDIA modelling framework is based on three major parts: the SecSoftML architecture description language (ADL), the soft/security co-engineering method that uses and transforms the elements of SecSoftML, and finally implementation with VESSEDIA tools.

SecSoftML is an ADL whose specification was done with respect to the ISO 42010 standard for architecture description languages. SecSoftML aims to answer the classical software development and security concerns of the software engineer and security analyst.

SecSoftML is used within the software/security co-engineering method proposed in this document. The method has the advantage of parallelizing the classical software development process with the security analysis process. The goal is to avoid as much as possible work product dependencies and blocking among the two domains and tasks. A certain number of transformation and analysis steps will exploit the models and artefacts produced during steps of the method.

An implementation for SecSoftML is proposed with Papyrus UML profiles, diagrams, Xtext editors, and viewpoints developed within the architecture framework model of Papyrus. Implementation of the transformations and analyses in the co-engineering method is proposed with existing tools like the DIVERSITY symbolic execution engine, the Papyrus Software Designer code generator, and the Frama-C static code analysis. The necessary extensions for these tools were specified in this document. Implementation will be done in task T1.3 for deliverable D1.4.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

The multiplication of stakeholders, with different non-orthogonal concerns, has increased the complexity of system development. In the IoT domain, with more and more intrusive systems, safety and security concerns have become crucial. Because of the criticality of such systems, software engineers are prone to follow security methodologies and guidelines. The increased complexity of systems, and their new constraints, imposes to R&D teams, of different industries, to adopt new methods and their associated tools.

Context. Within VESSEDIA, IoT use-cases are developed with focus given to the safety and security requirements and properties. In VESSEDIA, static code analysis is one of the preconized technique that is used to verify the properties at code level. In such an approach, software and security properties are expressed as annotations in the code, and are verified with a static code analysis tool (e.g. Frama-C [Krichner15], Verifast [Jacobs11]). Such properties can express functional properties at code level, or non-functional properties like the absence of buffer overflows.

However, in order to reason about system correctness, it is important to have a global view of it. Indeed, source code level properties only provide a local view that only allows a partial understanding of the system. Global security requirements are often specified at the system modelling level. Such requirements must be verified at code level. The challenge is to provide a modelling framework to bridge the gap between local properties, associated to code, with global system requirements, associated with system architecture and the knowledge of an appropriate abstraction of its behaviours. The modelling framework must thus not only focus on the expressivity of the modelling languages but also methodological issues like traceability and conformance of requirements at different levels of abstraction.

Considering the context of VESSEDIA use-cases, the modelling framework must respect the following requirements:

- ➔ The modelling framework must answer common software development concerns like requirement specification, architecture modelling (structural and behavioural), and code implementation.
- ➔ The modelling framework must answer safety/security concerns like requirement specification, risk assessment, and analysis.
- ➔ The modelling framework must allow to deal with requirements expressed in a human textual language.
- ➔ The modelling framework must allow to deal with architectures expressed in a graphical language.
- ➔ The modelling framework must allow to deal with properties expressed in textual source code annotations (e.g. ANSI/ISO C Specification Language (ACSL) [ACSLSpec]).
- ➔ The modelling framework must propose a generic method that can be adapted to specific methodologies and tools.
- ➔ The modelling framework must be instantiable with VESSEDIA tools, i.e., Papyrus modeller [Gerard07], DIVERSITY [Arnaud16] symbolic execution engine, Frama-C [Krichner15] static C code analyser.

The goal of such requirements is to ensure that the modelling framework can bridge the gap between high-level textual requirements in an architecture model and low-level properties annotating the code.

The rest of this document focuses on security aspects of the VESSEDIA project considering how they are intrinsically related to software designs and implementations that are verified in VESSEDIA. For source code, this document focuses on the C language and its related specification languages.

Related approaches. For the time being, there are many methodologies that address the concern of building a secure system. For example, in the case of risk management and risk assessment methodologies, the European Union Agency for Network and Information Security (ENISA) has generated an inventory of 17 methods in [ENISAMethods] and 18 tools in [ENISATools]. For example, EBIOS [EBIOS] is a method proposed in 1995 to assess and handle risks related to information system security.

These methodologies do not provide a language aimed by the VESSEDIA modelling framework, as language definition is out of their scope. They propose methodological requirements and recommendations that can be instantiated for a particular method used in a company. Therefore their contributions are not incompatible with the VESSEDIA modelling framework which proposes languages and methods.

In terms of Domain-Specific Modelling Languages (DSML) that support the design and security analysis of a system, Unified Modelling Language-based languages have explored security requirement modelling and risk assessment. UML is a graphical software modelling language with a certain number of diagrams dedicated to structure and behaviour modelling. Through its profile mechanism, UML can be extended for domain-specific needs. A UML profile has stereotypes, with attributes, that extend UML meta-model elements. Once a profile is applied on a UML model, its stereotypes can be applied on UML elements to give them new semantics.

Attack Modelling Language (AtML) [Bannour14] is a UML profile that proposes to model architecture vulnerabilities and attack scenarios with re-usable patterns of attacks from the threats. Most of the modelling work is done at the interaction level, using UML sequence diagrams to assess the risks.

The AtML approach does not use static code analysis as a mean to verify the modelled security requirements. Furthermore, it does not support behavioural modelling of the architecture. However, as we shall see, AtML can be integrated into the VESSEDIA modelling framework.

Perhaps the approach that *a priori* suits the most the needs of VESSEDIA is System Modelling Language for Security (SysML-Sec) [Roudier15]. SysML-Sec is an extension to the System Modelling Language (SysML) [SysMLSpec] profile, itself an extension of UML for system engineering purposes. The language proposes a taxonomy of security requirements, and annotations for the hardware and software design, with the goal of verifying security, safety, and performance in a co-engineering framework. Verification is based on formal modelling and model-checking (e.g. UPPAAL and ProVerif).

However, SysML-Sec does not suit all requirements of the VESSEDIA modelling framework because of the need to have low level enough languages that can bridge the gap with the source code. Indeed, SysML-Sec is based on SysML which suits system engineering rather than software implementation concerns. Furthermore, the analysis methods to use in conjunction with SysML-Sec are based on model-checking while in VESSEDIA static code analysis is performed to verify that the implementation of the system indeed respects its high level security requirements.

Contributions. To face the challenges of VESSEDIA, we propose to use previous experience and expertise in system and software design by Model-Driven Engineering (MDE) methods and tools. MDE focuses on description and exploitation of domain-specific models to separate concerns and promote representation of knowledge in a particular domain rather than implementation details. Model transformation [Sendall03] is crucial to MDE. Model transformation takes some input model conform to a meta-model, does some computations, and produces some output model conform to a meta-model.

In this document a modelling framework is proposed for the design of secure software systems. Considering the high number of security methodologies and DSMLs available, proposing a new methodology or new DSML in VESSEDIA is not justified. Instead existing DSMLs will be aggregated together and used in collaboration to answer different software/security concerns and VESSEDIA use-case requirements. Aggregated languages shall also be generic enough for them to be extended with specific DSMLs proposed in the future. A generic software/security

co-engineering method is also proposed to use the packaged languages within a same framework. The steps of the generic method can be mapped to existing methodology activities.

In terms of tooling, this document proposes an Integrated Development Environment (IDE) that implements the modelling framework with existing tools such as the Papyrus modeller and model transformation tools [Pham16], the DIVERSITY symbolic execution engine, and the Frama-C static code analyser.

The aggregation of DSMLs, the proposed generic co-engineering method, and tooling are the core of the VESSEDIA modelling framework. It is thus composed of the following contributions:

- The specification of an Architecture Description Language (ADL) called Secure Software Modelling Language (SecSoftML) that is conform to ISO 42010 [ISO42010], which proposes a conceptual model for architecture description. This document shows how SecSoftML can be conform to an ADL in the ISO 42010 sense, while being an aggregation of existing ADLs.
- A generic software/security co-engineering method, with modelling, transformation, and analysis steps. Elements of SecSoftML are used in the modelling steps and to represent analysis results.
- A specification of the implementation of the modelling framework based on tools of VESSEDIA: Papyrus, DIVERSITY, and Frama-C.

The rest of the paper is structured as follows. Chapter 2 describes the ISO 42010 standard on which the ADL of the VESSEDIA modelling framework is built upon. Chapter 3 describes the ADL conform to ISO 42010. Chapter 4 presents the software/security co-engineering method. Chapter 5 gives and implementation specification for deliverable D1.4 produced by task T1.3. Chapter 6 shows a case-study to illustrate the whole approach, with an early implementation of the modelling framework. Chapter 7 concludes with some ideas for future works.

Chapter 2 ISO 42010

In this chapter, the ISO 42010 standard for architecture description is explained. In particular, we focus on the concept of architecture description language, given in the standard. Indeed, in VESSEDIA an architecture description language is specified and used within its modelling framework.

ISO 42010 addresses the creation, analysis and sustainment of architectures of systems through the use of architecture descriptions. An architecture description, as defined by the ISO 42010 standard, is a “work product used to express an architecture”. The goal of the standard is to homogenise architecture description by defining standard terms and providing a conceptual foundation for expressing and exploiting architecture descriptions.

The standard specifies the required concepts of an architecture description. Such concepts are introduced to codify conventions and common practices of architecture description. Based on the concepts of architecture description specified in the standard itself, it establishes conceptual models architecture description languages shown in Figure 1. Note that as it is a conceptual model, it cannot be taken as a formally well-defined and consistent meta-model. Nevertheless, we will try to be conform to this conceptual model in the definition of our architecture description languages used within the VESSEDIA modelling framework.

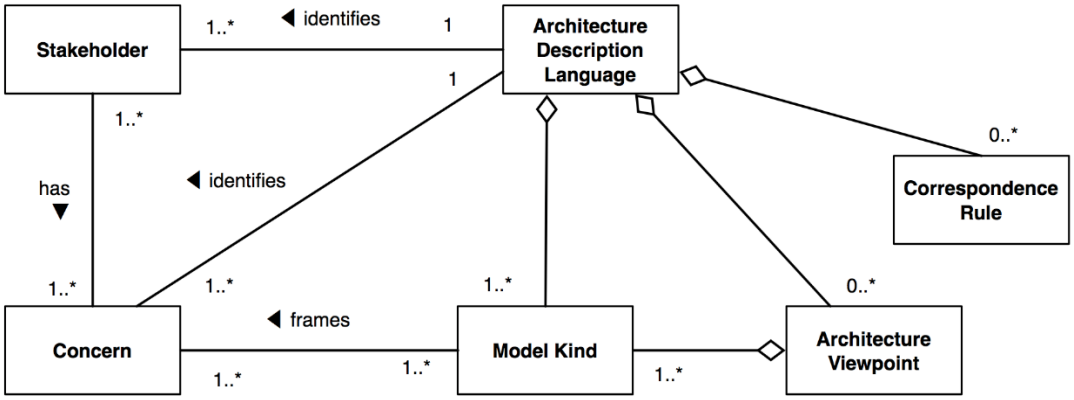

Figure 1: ISO 42010 conceptual model of architecture description

In Figure 1, the conceptual model has a number of concepts that are defined in the standard as follows:

- **Stakeholder**: Stakeholders are individuals, groups or organizations holding Concerns for the System of Interest. Examples of stakeholders: client, owner, user, consumer, supplier, designer, maintainer, auditor, CEO, certification authority, architect.
- **Concern**: A Concern is any interest in the system. The term derives from the phrase “separation of concerns” as originally coined by Edsger Dijkstra. Examples of concerns: (system) purpose, functionality, structure, behaviour, cost, supportability, safety, interoperability.
- **Architecture Viewpoint**: An Architecture Viewpoint is a set of conventions for constructing, interpreting, using and analysing one type of Architecture View. A viewpoint includes Model Kinds, viewpoint languages and notations, modelling methods and analytic techniques to frame a specific set of Concerns. Examples of viewpoints: operational, systems, technical, logical, deployment, process, information.
- **Model Kind**: A Model Kind defines the conventions for one type of Architecture Model.

- **Architecture Description Language (ADL)**: An ADL is *any form of expression* for use in architecture descriptions. An ADL might include a single Model Kind, a single viewpoint or multiple viewpoints. Examples of ADLs: Rapide, SysML, ArchiMate, ACME, xADL.
- **Correspondence Rule**: Correspondence Rules enforce relations within an Architecture Description or between Architecture Descriptions.

Using the conceptual model of Figure 1, the VESSEDIA modelling framework will be specified in the next two chapters. In Chapter 3, all concepts except correspondence rule will be specified for the ADL of the VESSEDIA modelling framework. The correspondence rules will be specified in Chapter 4, which focus on relations and transformations between the model kinds.

Chapter 3 Secure Software Modelling Language

The ADL used in the VESSEDIA modelling framework is called SecSoftML. Contrary to classical ADLs, the one in VESSEDIA is actually an aggregation of existing subsets of modelling and specification languages that answer the different concerns of the stakeholders.

In the following sections, the concepts of stakeholders, concerns, model kinds, and viewpoints, from ISO 42010, are used to specify SecSoftML, the VESSEDIA modelling framework.

3.1 Stakeholders and concerns identified in SecSoftML

The stakeholders and concerns, within the context of VESSEDIA, have been identified with a very practical approach, by working on the development of the use-cases, e.g., the 6LowPan sensors network.

Figure 2 shows the stakeholders and concerns of secure software development in VESSEDIA that need to be answered by SecSoftML. The figure is made with the UML use-case diagram conventions.

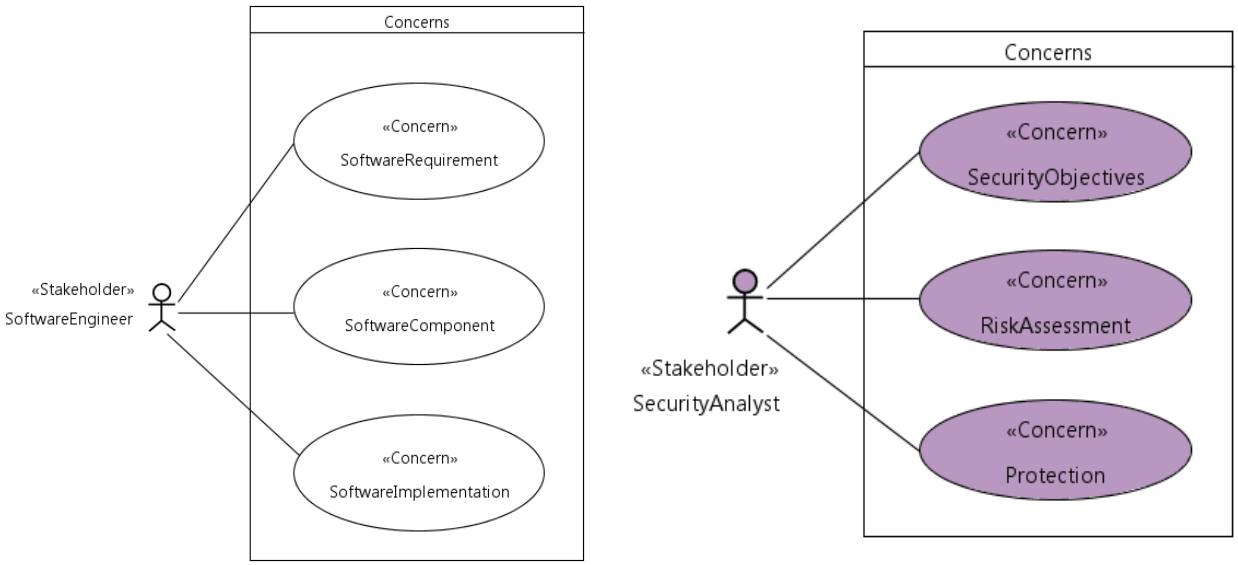

Figure 2: Stakeholders and concerns of secure software development in VESSEDIA

To simplify this report, there are two identified stakeholders: the software engineer and the security analyst. Obviously these two stakeholders can be specialized according to company development methods. Within VESSEDIA, the two stakeholders are defined as follows:

- **Software engineer**: the software engineer's role is to develop all required functionalities. As such, this stakeholder covers all system development life-cycle phases from textual description of requirements to executable runtime. The stakeholder makes sure the functional requirements and properties are met so the system can accomplish its use-cases.
- **Security analyst**: the security analyst's role is to secure the developed product. As such, this stakeholder brings security expertise inputs in all system development life-cycle phases, starting at the requirements specification phase. The security analyst makes sure non-functional security properties are met so the system can prevent foreseen attacks.

The software engineer has the following concerns:

- **Software requirement**: This concern consists in defining the software functional requirements which are the needs, identified by the client or by the software engineers, that the system must fulfil. Such requirements are expressed in a more or less abstract manner, i.e., from textual descriptions to refined models of use-cases and interactions.
- **Software component**: This concern deals with the definition, assembly, and composition of functional software components necessary for the system to meet its requirements. The interfaces of components and possible collaborations between components are of utmost interest. Verification of composability is recommended.
- **Software implementation**: This concern is the realization of the software components so they are implementable. The implementation details, functions and their behaviours, variables and their types and values, communication protocol realizations, etc…, are of interest. Verification of correct functional behaviour is recommended.

The software analyst has the following concerns:

- **Security objectives**: This concern consists in defining the security requirements of the software system to develop. Security requirements are defined in parallel with functional requirements.
- **Risk assessment**: Risk assessment in security engineering consists in identifying the assets, threats, and vulnerabilities of the system. The risk is computed from all three factors.
- **Protection**: This concern aims at preventing attacks by specifying and implementing countermeasures. A countermeasure can be a new watchdog component in the system, or simply be security properties on software elements that must be respected during the implementation.

The concerns identified in this section must be framed by some viewpoints, and their model kinds, within SecSoftML. Such architecture description elements are described in the next section.

3.2 Viewpoints and model kinds of SecSoftML

As a reminder, SecSoftML is an aggregation of subsets languages instead of a classical ADL. In order to achieve this objective of not only aggregating languages but their subsets necessary for our concerns, SecSoftML aggregates the representations of the languages as model kinds in the ISO 42010 sense.

A representation of a language is a particular syntax of the language, e.g., it can be a particular graphical diagram of that language (e.g., UML state-machine diagram), a tabular representation (e.g., SysML Requirements table) or the simple textual form of the language (e.g., ACSL text). Representations of languages already constrain the subset semantic elements of language that can be represented. Therefore, by aggregating only representations, we already choose the subset of a language that are used in our ADL.

Since model kinds are the atomic unit of aggregation in our ADL, in the next sections, the model kinds in SecSoftML are first described, followed by its viewpoints. For each viewpoint, the list of model kinds it aggregates will be given, as well as the rationale behind the viewpoint.

3.2.1 Model kinds of SecSoftML

As a reminder, model kinds in SecSoftML are representations of the different languages aggregated by the ADL. In the following sections, we first present an overview of all model kinds and their framed concerns. Afterwards, each of the following sections will describe the model kinds of a language used in SecSoftML. The goal of this document is not to give the entire

specification of each model kind, but rather an overview of elements expressible in a model kind that are of interest to SectSoftML, with respect to the identified concerns.

3.2.1.1 Summary of model kinds and framed concerns

Figure 3 sums up the model kinds of SecSoftML and the concerns they frame.

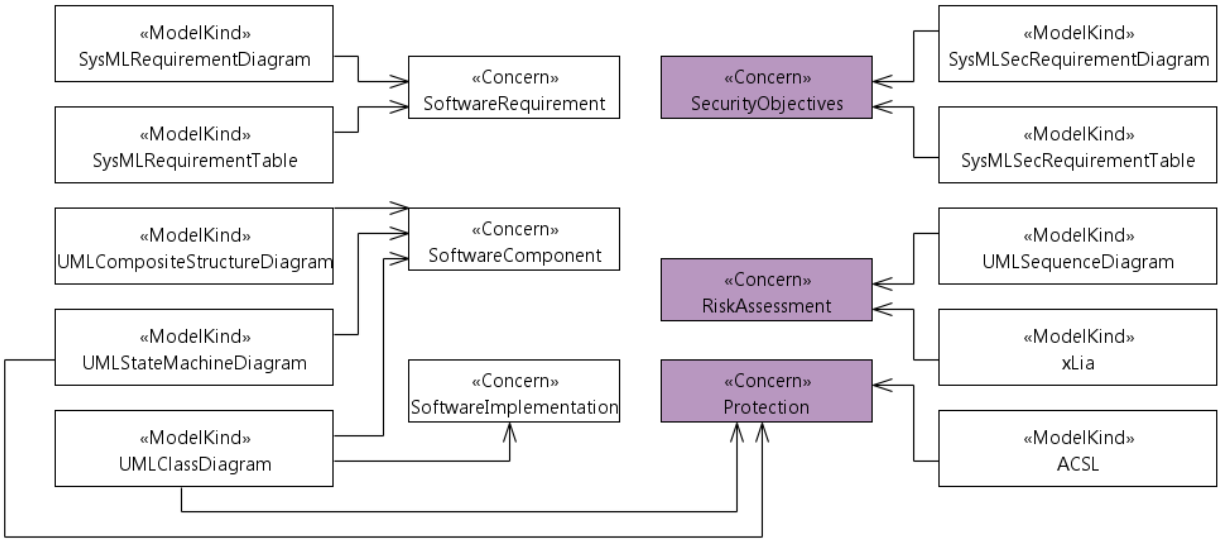

Figure 3: Model kinds of SecSoftML and their framed concerns

The next sections give more details on the model kinds in Figure 3.

3.2.1.2 SysML requirement model kinds

SysML is a generic language for system engineering, currently standardized by the OMG. SysML is a specialization of UML for system specification, analysis, design, verification and validation. SysML takes a subset of UML structural diagrams and specializes them for requirements, allocations, blocks modelling, and parametric modelling, in order to have a consistent and smaller sized language. For behavioural modelling, SysML imports UML behavioural diagrams.

In the case of SecSoftML, only the SysML requirement model kinds are of interest. According to the SysML specification, a requirement specifies a capability or condition that must (or should) be satisfied. A requirement may specify a function that a system must perform or a performance condition a system must achieve. SysML requirement model kinds frame the software requirement concern of the software engineer.

3.2.1.2.1 SysML requirement diagram

In SysML requirement diagrams, the textual requirements are graphically modelled as classes stereotyped <<Requirement>>. Each <<Requirement>> has an id and a text description. Requirements can be packaged together within packages or other requirements to create some hierarchical decomposition.

The graphical model is also ideal to represent traceability relationships between requirements themselves and between requirements and other model elements. Such traceability relationships are:

- Copy of a requirement by another requirement
- Derivation of a requirement by another requirement
- Satisfaction of a requirement by a model element
- Refinement of a requirement by a model element (usually a behavioural model element)

- Verification of a requirement by a model element (usually a model element representing some test case)

3.2.1.2.2 SysML requirement table

Requirements and relationships can also be represented in tabular views. In SysML requirement tables, the modelled requirements, and their packaging, are represented in a hierarchical tabular view. This is to ease the analysis of the requirements decomposition.

3.2.1.2.3 SysML allocation table

In SysML allocation tables, the traceability relationships are represented in a tabular view. This is to ease the analysis of, e.g., requirement satisfactions by the modelled solution.

3.2.1.3 SysML-Sec requirement model kinds

SysML-Sec is an extension of SysML to design safe and secure systems. Within SysML-Sec, the security requirements model kinds are of interest to SecSoftML. Indeed, SysML-Sec has specific stereotypes for security-related requirements (e.g., confidentiality, authenticity, integrity). Such security requirements will be represented in SysML-Sec requirement diagrams and SysML-Sec requirement tables. These model kinds are similar to those of SysML requirement, except they are specialized to represent SysML-Sec requirements and traceability relationships for SysML-Sec requirements.

The SysML-Sec requirement model kinds frame the security requirement concern of the security analyst.

3.2.1.4 UML model kinds

UML [UMLSpec] is a generic language originally proposed for software engineering. Historically, the goal of UML was to unify different languages and tools used in the software domain. UML has a number of diagrams to fulfil the needs of requirement, analysis, design, verification, and validation in software development. These diagrams are graphical representations of the model elements. Therefore the diagrams are the model kinds to aggregate in SecSoftML.

3.2.1.4.1 UML sequence diagram

Among model kinds of UML, some are useful for requirement refinement. The UML sequence diagram represents interactions between elements of the model. Such elements can be components of the solution or external actors. Elements are represented as chronologically ordered lifelines and they exchange messages among them. When a message is received by an element, some operation (function) may be executed.

Messages and executed operations may be subject to constraints expressed in a domain-specific textual language.

One of the main usage of the sequence diagram is to refine textually written requirements, modelled in SysML requirement diagrams. Indeed, the sequence diagram formalizes the behaviours that are textually described and add constraints that are expressed in a domain-specific textual language.

This model kind frames the risk assessment and protection concerns of the security analyst.

3.2.1.4.2 UML composite structure diagram

Some model kinds are dedicated to the structural specification of the software system. The UML composite structure diagram represents components as UML structured classes, and their internal structure. Classes have ports, typed by interfaces, and parts that may be connected with connectors. This diagram therefore shows how the components are composed and assembled, and how functionalities are delegated. The goal of this kind of diagram is to focus on the collaborations between components.

This model kind frames the software component concern of the software engineer.

3.2.1.4.3 UML class diagram

Some model kinds are dedicated to detailed structural specification in SecSoftML, focusing on implementation issues to refine the more abstract components. The UML class diagram represents classes, their attributes (variables), and their operations (functions). Implementation specific properties (e.g., programming language specificities) may also be specified as annotations on these elements.

The elements may also have constraints expressed in a domain-specific textual language. Such constraints may express, e.g., security properties on operations that must be respected by the implementation. In other cases, the constraints may express security properties on the whole program described by the modelled architecture.

This model kind frames the software component and software implementation concerns of the software engineer. It also frames the protection concern of the security analyst since it is used to represent elements that will have annotated security properties.

3.2.1.4.4 UML state machine diagram

Some model kinds are necessary for behavioural specification in SecSoftML. The UML state machine diagram represents behaviours of components in an automata formalism. Classes have states connected by transitions. Internal and external events trigger the transitions and therefore changes of state. Operations (functions) and atomic behaviours may be executed in states and in transitions.

The elements may also have constraints expressed in a domain-specific textual language. Such constraints may express, e.g., security properties on states, transitions, and executed operations and atomic behaviours that must be respected by the implementation.

This model kind frames the software component concern of the software engineer. It also frames the protection concern of the security analyst since it is used to represent elements that will have annotated security properties.

3.2.1.5 ACSL model kinds

ACSL allows to formally specify the properties of a C program, in order to be able to formally verify that the implementation respects these properties.

The most important ACSL concept is the function contract. A function contract for a C function `f` is a set of requirements over the arguments of `f` and/or a set of properties that are ensured at the end of the function. The formula that expresses the requirements is called a pre-condition, whereas the formula that expresses the properties ensured when `f` returns is a post-condition. Together, these conditions form a contract between `f` and its callers: each caller must guarantee that the pre-condition holds before calling `f`. In exchange, `f` guarantees that the post-condition holds when it returns.

In VESSEDIA, ACSL relational properties on a program have also been developed. For example a relational property can express the order in which a function should execute with respect to another function. Such properties would allow users to specify the allowed execution paths of a program.

For a full list of first-order logic expressions, predicates, axioms, etc… that can be written in ACSL, the reader is invited to refer to the ACSL specification [ACSLSpec].

The model kind of ACSL is the ACSL specification which is expressed as textual comments. In terms of integration with UML, constraints can be expressed in ACSL to represent properties that the implementation must respect.

The ACSL specification frames the security protection concern.

3.2.1.6 xLIA model kinds

xLIA (executable Language for Interaction and Architecture) [Arnaud16] is the pivot language of the DIVERSITY tool introducing a set of communication and execution primitives allowing one to encode a wide class of dynamic model semantics, e.g., hierarchical timed communicating

Symbolic Transition Systems (STS), UML/SysML (UML state machine diagram, sequence diagrams), Specification and Description Language (SDL), and abstractions of hybrid systems. The root entity in an xLIA model is a so-called system. A system is an executable entity that can be atomic (as STS), compositional, or hierarchical.

xLIA supports many communication and execution schemes which allows the specification of reference system models which can be used to capture high level system concerns (e.g., sub-systems can communicate asynchronously (over FIFO) with interleaving scheduling. More operators exist in xLIA such as sequencing, parallel, choice.

We focus on a subset of DIVERSITY that we will be used in VESSEDIA to formalize semantics of specialized UML interactions with symbolic data and function call constraints (by model translation of UML interactions to xLIA models). This subset of xLIA allows the specification of (1) STS where transitions are composed of a source and a target control state, (2) sequence of instructions such as guards built from state variables, (3) some communication actions (receptions of values stored on state variables or emissions of values through some ports), (4) variable updates denoted by classical assignments, and (5) black-box function calls with explicit parameter data. The latter constitutes an extension of xLIA done in VESSEDIA to support STS enriched with function calls. xLIA is endowed with symbolic execution mechanisms which allows to compute and reason about semantics of models in an efficient manner using a symbolic representation of the state-space. On the basis of this symbolic representation, algorithms for ACSL program relational properties inference, for called functions, are being developed in DIVERSITY (T3.1).

The model kind of xLIA is the xLIA specification which is expressed in text. It frames the risk assessment concern.

In following section, the model kinds described so far will be aggregated among viewpoints of SecSoftML.

3.2.2 Viewpoints of SecSoftML

In SecSoftML there are three viewpoints dedicated to requirements analysis, security analysis, and software design. Each viewpoint aggregates the model kinds described in Section 3.2 in order to provide a single context dedicated to one concern, or several concerns if synergy among concerns is recommended. Each of the following sections describes a viewpoint and the rationale behind it.

3.2.2.1 Requirements analysis viewpoint

The requirements analysis viewpoint is used for requirements modelling and organization. As such, the model kinds that frame both the software requirements and security objectives are aggregated within the requirements analysis viewpoint. Note that model kinds that frame software requirements and security objectives are both aggregated within a same viewpoint because it is usually necessary to have a holistic view of all system requirements for co-engineering purposes.

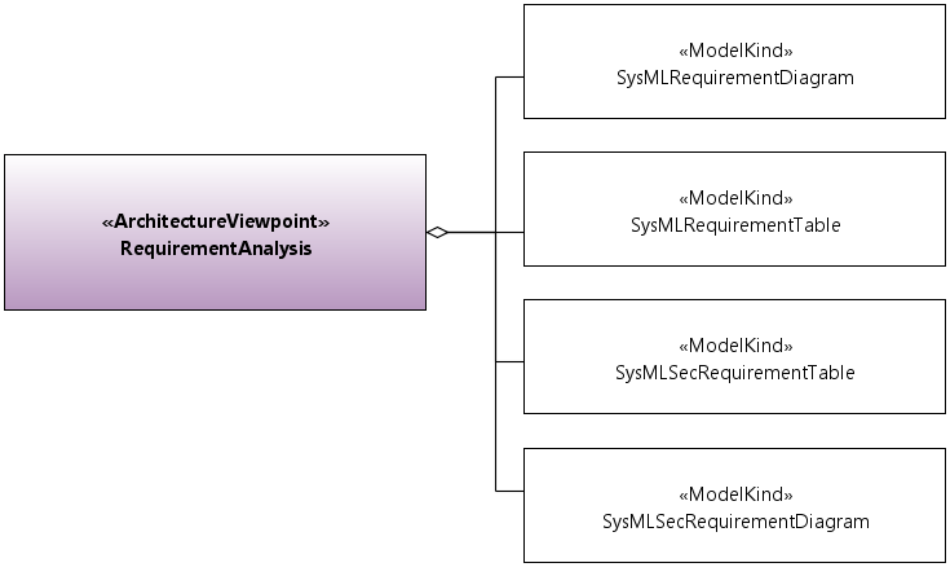

Figure 4: Requirement analysis viewpoint

Figure 4 shows the requirement analysis viewpoint with the model kinds it aggregates.

3.2.2.2 Security analysis viewpoint

The security analysis viewpoint is used to assess assets, threats, and vulnerabilities. As such, the model kinds that frame the risk assessment and protection concerns are aggregated. The risk assessment is done in xLIA-annotated UML sequence diagrams where it is possible to represent the assets, threats, and vulnerabilities through attack scenarios. Protection is done by specifying security properties as ACSL annotations in UML class diagrams, UML state machine diagrams, and inferring security properties as ACSL from UML sequence diagrams.

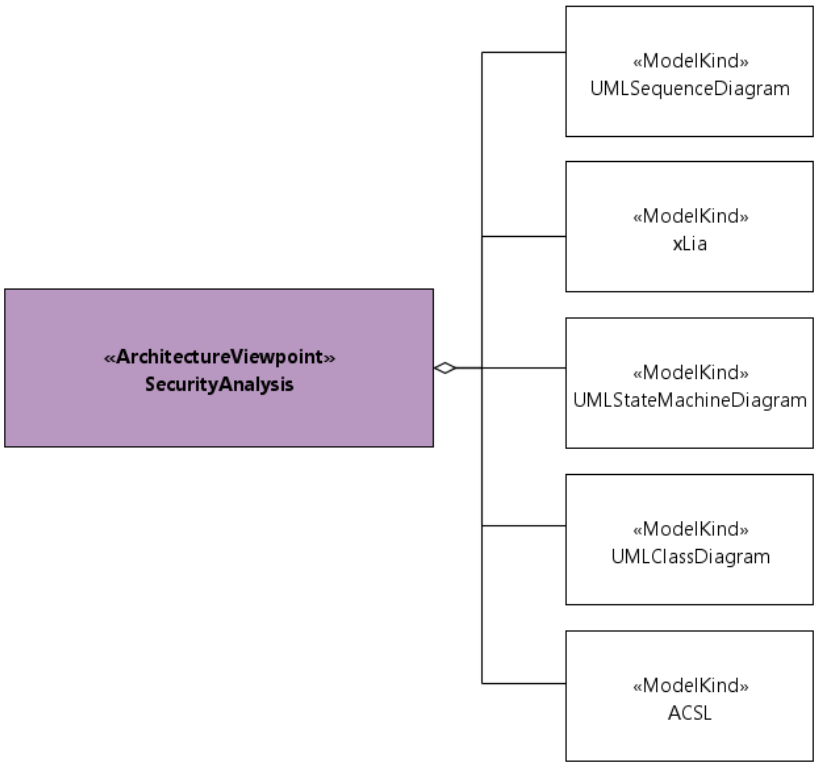

Figure 5: Security analysis viewpoint

Figure 5 shows the security analysis viewpoint with the model kinds it aggregates.

3.2.2.3 Software design viewpoint

The software design viewpoint is used for classical software development steps. As such, the model kinds that frame the software component and software implementation concerns are aggregated.

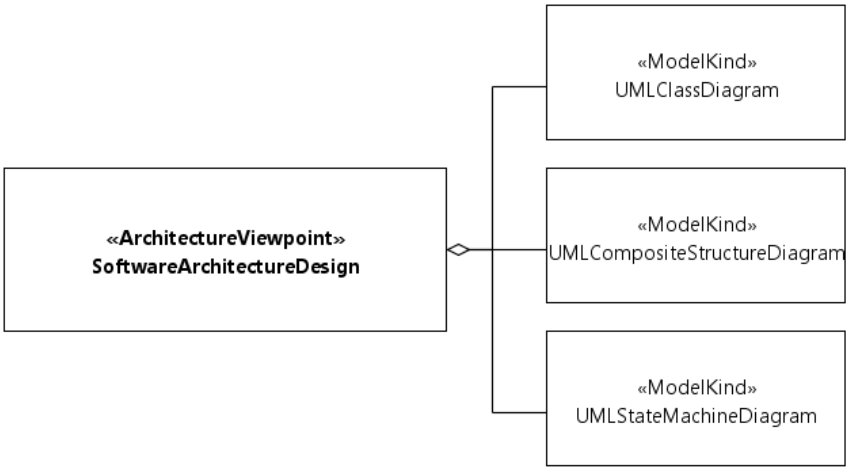

Figure 6: Software design viewpoint

Figure 6 shows the software design viewpoint with the model kinds it aggregates.

The next chapter relates the model kinds presented in this chapter, through a software/security co-engineering method.

Chapter 4 Software/security co-engineering method

In traditional security and safety methodologies, there may exist work-product dependencies between steps of a method. Such dependencies can slow down the software development. Therefore, an innovative and efficient way to implement such methodologies recommendations is to parallelize steps when possible.

The method is based on a generic compositional approach where traditional software development steps are done in parallel to security analysis steps, before they converge at synchronization points. Since the method is generic, its steps can be mapped to existing methodologies.

In the next sections, first the co-engineering method is described. Afterwards its steps are mapped to model kinds of viewpoints of SecSoftML described in Section 3.2. Finally, the transformations and relationships between model kinds, within the proposed co-engineering method, will be described as correspondence rules of ISO 42010.

4.1 Compositional approach for software/security co-engineering

The proposed co-engineering method, for the VESSEDIA modelling framework, is shown in Figure 7.

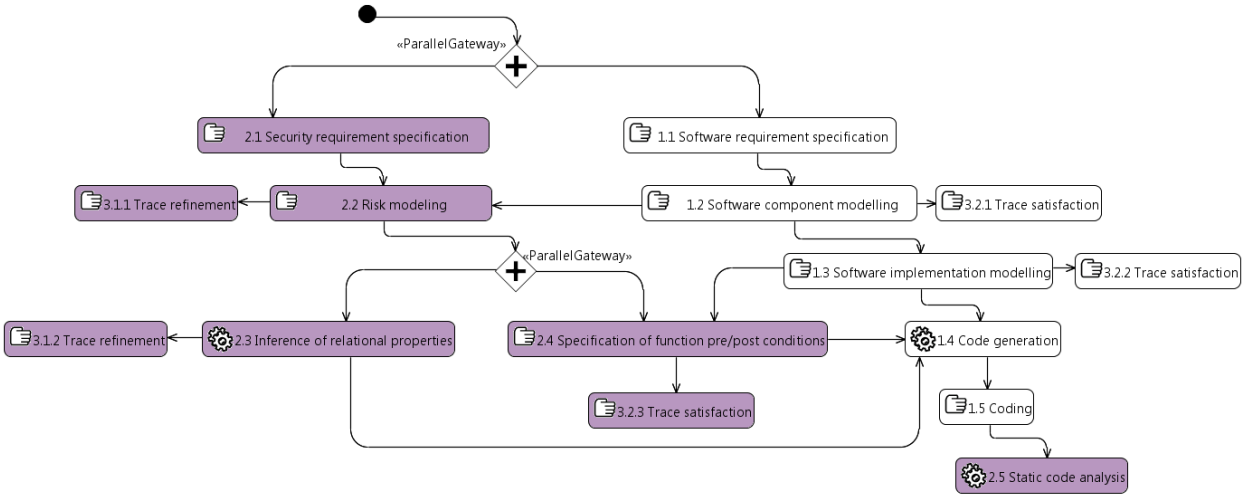

Figure 7: Software/security co-engineering method

In Figure 7 there are two processes running in parallel, the software development process highlighted in white, and the security process highlighted in violet. The advantage of such an approach is that security steps do not block software development steps.

The proposed co-engineering method contains a certain number of steps. Steps with a hand icon are modelling and specification steps. Steps with a cog icon are transformation and analysis steps. Arrows indicate dependencies between steps, i.e., a step must wait for all of its input to be available before it can start.

The steps in the software development process are the following:

- **1.1 Software requirement specification**: the software functional requirements are specified textually and modelled as behaviours refining the textual requirements.
- **1.2 Software component modelling**: the software components are modelled.
- **1.3 Software implementation modelling**: the modelled software components are refined with low level implementation details, and eventually decomposed in low level implementation classes.
- **1.4 Code generation**: from the implementation model, code is generated. This generated code is a skeleton that must be completed later on. The generated code also contains ACSL annotations that correspond to ACSL-expressed properties in the model, which are the results of steps 2.3 and 2.4, in the security process, described below.
- **1.5 Coding**: in this step, the generated skeleton is completed with, e.g., function bodies. Since the ACSL annotations are generated, they must be respected by the manually written code.

The steps in the security process are the following:

- **2.1 Security requirement specification**: the security requirements are specified textually.
- **2.2 Risk modelling**: the security requirements are refined as scenarios, including attack scenarios. The assets, threats, and vulnerabilities are identified during the definition of such scenarios to manage the risk.
- **2.3 Inference of relational properties**: from the scenarios it is possible to infer program relational properties. An example of a relational property is, e.g., a chain of functions that must execute and only execute at runtime.
- **2.4 Specification of function pre/post conditions**: pre/post conditions of functions may also be manually specified as ACSL annotations.
- **2.5 Static code analysis**: the completed code is analysed to verify that it respects the properties written as ACSL annotations. In particular, the properties for security need to be verified since their conformance represents protections.

Finally there are a certain number of traceability steps whose purposes are all the same:

- **3.1.X Trace refinement**: when any requirement is refined, the refining element must be traced back to the original requirement.
- **3.2.X Trace satisfaction**: when any requirement is satisfied by a design element or analysis result, the satisfying artefact must be traced back to the requirement.

In the next section, the steps described here will be mapped to existing model kinds of viewpoints when appropriate.

4.2 Mapping of steps to model kinds of viewpoints

The software/security co-engineering method has several modelling steps. Other steps produce specifications. Such steps will use or represent their results in the model kinds of the viewpoints of SecSoftML defined in Section 3.2. Table 1 shows the mapping of the steps to model kinds of viewpoints.

Step	Viewpoint	Model kind
1.1 Software requirement specification	Requirements analysis	SysML requirement diagram, SysML requirement table
1.2 Software component modelling	Software design	UML composite structure diagram, UML state machine diagram
1.3 Software implementation modelling	Software design	UML class diagram
2.1 Security requirement specification	Requirements analysis	SysML-Sec requirement diagram, SysML-Sec requirement table
2.2 Risk modelling	Security analysis	UML sequence diagram, xLIA specification
2.3 Inference of rational properties for security	Security analysis	ACSL specification
2.4 Specification of function pre/post conditions for security	Security analysis	UML class diagram, ACSL specification
3.1 Trace refinement	Requirements analysis	SysML requirement diagram, SysML allocation table, SysML-Sec requirement diagram
3.2 Trace satisfaction	Requirements analysis	SysML requirement diagram, SysML allocation table, SysML-Sec requirement diagram

Table 1: Mapping of co-engineering steps to model kinds of viewpoints

Note that the software/security co-engineering method, and its mapping to proposed security viewpoints and model kinds, can be easily adapted to safety concerns. The only difference is the viewpoints and model kinds which need to be those framing safety concerns. The co-engineering method itself is generic enough so it does not need adaptation.

The next chapter gives an instantiation of the software/security co-engineering method, through its implementation with tools.

Chapter 5 Implementation specification

In order to provide a streamlined experience for the modelling framework of VESSEDIA, several of its major parts must be implemented as tools and packaged: SecSoftML, transformations and analyses, packaging in an IDE.

The role of this chapter is to provide an implementation specification for the modelling framework and therefore identify existing tools that can be re-used and others that must be developed in VESSEDIA. The next sections propose implementation specifications for the different mentioned parts of the VESSEDIA modelling framework.

5.1 SecSoftML implementation

The implementation of SecSoftML is based on the implementation of modellers for its model kinds and viewpoints. The following sections first give an overview of the technology enablers that let us implement SecSoftML. Afterwards, we specify the implementation of the model kinds. Finally, the specifications of the viewpoints implementations are given.

5.1.1 Papyrus and Xtext

The majority of the ADL is implemented with Papyrus and Xtext. The following sections give an overview of these tools and explain how they are customizable for the needs of SecSoftML.

5.1.1.1 Papyrus

Papyrus is an open-source UML modeller that implements the whole UML language specification and its representations. Papyrus also provides a framework to implement new UML profiles and a customization of Papyrus modelling UI for the profile. Such is the case for SysML, whose whole language specification and representations are implemented in Papyrus.

Figure 8 shows the Papyrus modelling environment with its different UI elements.

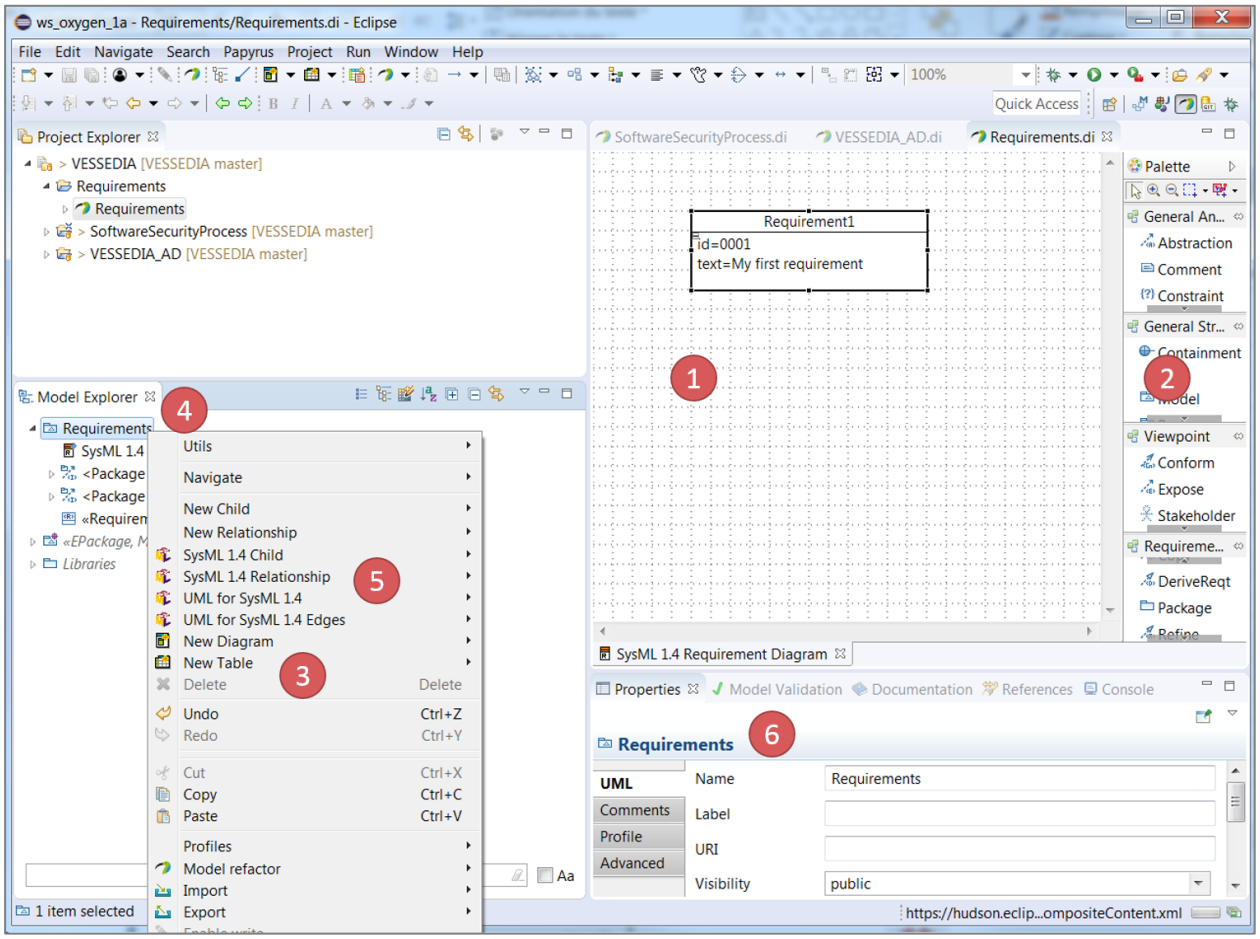

Figure 8: Papyrus modelling environment, example of UML state machine diagram

In Figure 8, there are UI elements for the edition of the diagram. In the following, when a diagram must be implemented, the following UI elements must be implemented:

- 1) CSS-enabled diagram editor: the editor shows the diagram representing elements of the model. The editor enables applying CSS style sheets to customize visualisation.
- 2) Diagram palette: the diagram palette contains tools to create model elements in the diagram.
- 3) New diagram/table: the new diagram/table menu allows to create diagrams specific to the language.

In Figure 8, there are also UI elements for the edition of the model itself. In the following, when a profile must be implemented, the following UI elements must be implemented:

- 4) Model explorer: the model explorer shows a hierarchical view of the model elements that are not graphically represented in diagrams.
- 5) New child: the new child menu allows to directly create model elements of a language without using diagrams.
- 6) Properties view: the properties view of a language allows to edit the properties of an element selected in a diagram or the model explorer.

Papyrus is not only a graphical modeller but it also supports textual modelling through Xtext, which is described in the next section.

5.1.1.2 Xtext

Xtext [Xtext] is a framework to write domain-specific textual languages. By defining the syntax and grammar of a textual language, and visual specifications for the text editor, the Xtext framework generates a text editor in which coloured expressions can be written in the particular

domain-specific textual language. Papyrus supports integration of Xtext editors to edit textual expressions.

For example in Figure 9, an UML element is specified textually, instead of graphically, with a UML Xtext editor integrated in a Papyrus diagram.

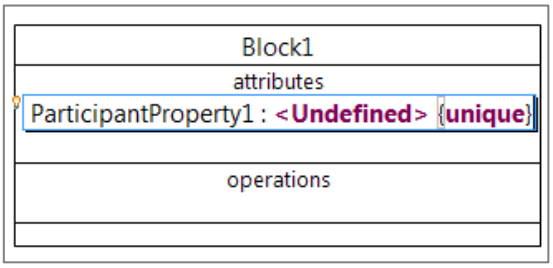

Figure 9: Xtext integration in Papyrus

The next section shows how Papyrus and Xtext were combined to implement SecSoftML model kinds.

5.1.2 SecSoftML model kinds implementation

The model kinds of SecSoftML are in majority implemented as UML profiles with representations (diagrams, tables) in Papyrus.

5.1.2.1 SysML requirement model kinds implementation

The SysML requirement profile is already implemented in Papyrus. Its requirement diagram, table, and allocation table are already implemented. All UI elements are also implemented. Therefore the Papyrus implementation of SysML will be re-used as is.

5.1.2.2 SysML-Sec requirement model kinds implementation

The SysML-Sec requirement profile must be implemented in Papyrus. UI elements to edit the model in SysML-Sec requirement must be implemented (see Section 5.1.1.1). The SysML-Sec requirement diagram and table must be implemented with their UI elements (see Section 5.1.1.1). The SysML allocation table will be re-used.

5.1.2.3 UML model kinds implementation

UML and all of its diagrams required by SecSoftML are already implemented in Papyrus. They will therefore be re-used.

5.1.2.4 ACSL model kinds implementation

ACSL must be implemented as an Xtext editor to be integrated into Papyrus. The ACSL Xtext editor is attached to any menu where ACSL must be expressed, e.g., menus where constraints are written. The ACSL Xtext editor must also show coloured text of the ACSL expression.

Figure 10 gives an example of the early work-in-progress ACSL Xtext editor integrated in the Papyrus model explorer.

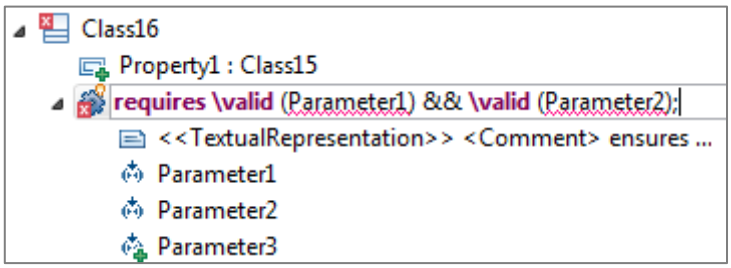

Figure 10: Early work-in-progress ACSL Xtext editor

5.1.2.5 xLIA model kinds implementation

xLIA is implemented as an Xtext editor and a Properties view editor, both integrated into Papyrus. Therefore the editors will be re-used.

Once all model kinds are implemented, they must be aggregated within the implemented viewpoints of SecSoftML, described in the next section.

5.1.3 SecSoftML viewpoints implementation

Papyrus has an architecture framework to develop ADLs that are ISO 42010 compliant. The framework is implemented as an Ecore meta-model [EMF] in Eclipse. The developer can then implement the ADL by modelling it in the dedicated Eclipe Modelling Framework (EMF) [EMF] modelling editor. SecSoftML and its viewpoints, as specified in Chapter 3, must be implemented with this Papyrus architecture framework.

The model must contain the stakeholders, the concerns, the ADL, and its aggregated viewpoints. The model kinds are also modelled as references to existing diagrams.

Figure 11 shows an early modelling of SecSoftML with the framework. For example, it contains the requirement analysis viewpoint, which itself refers to SysML requirement model kinds.

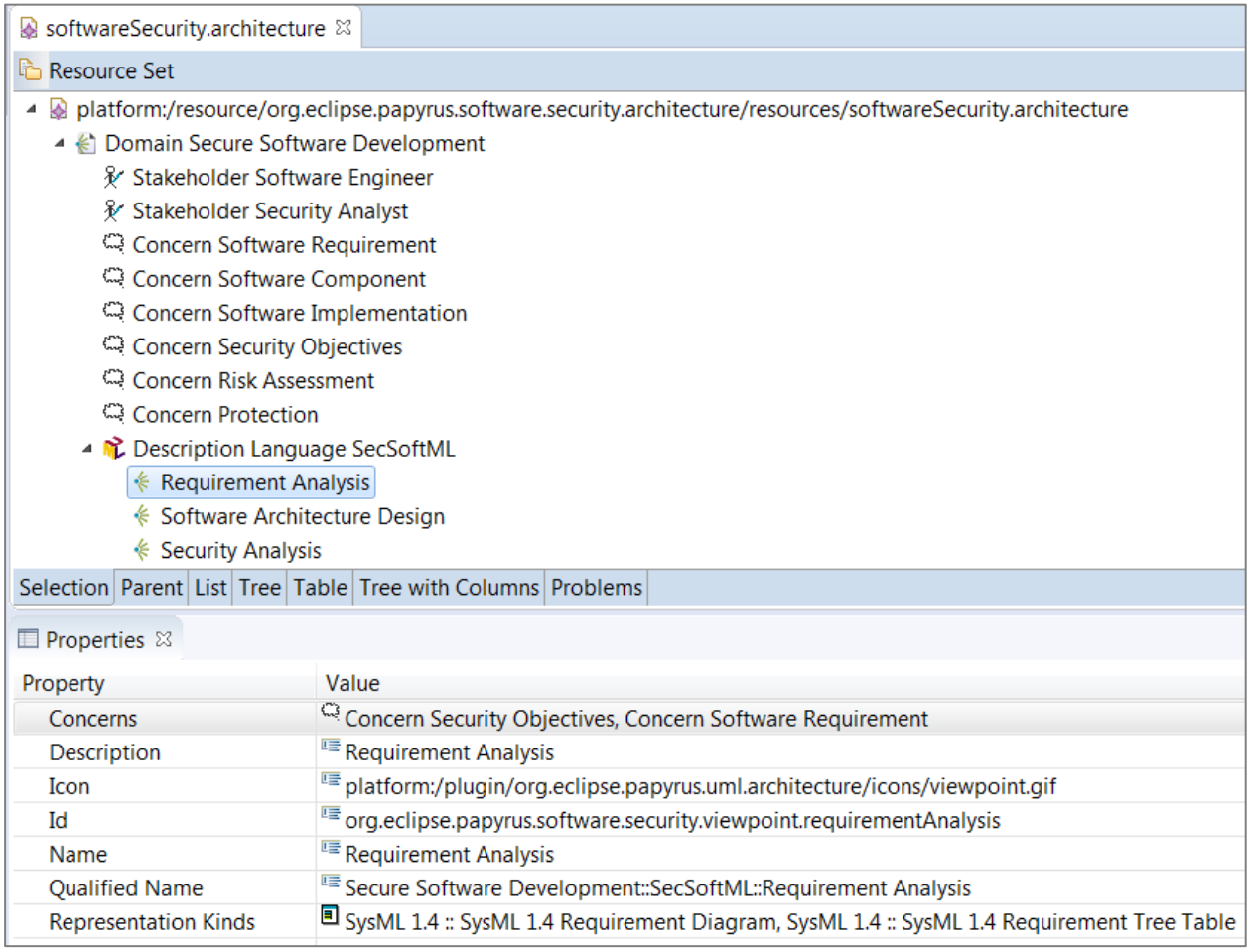

Figure 11: Papyrus architecture framework model

In the current architecture framework of Papyrus, only diagram-implemented model kinds (e.g., UML, SysML, and SysML-Sec) can be referenced. Therefore Xtext-implemented model kinds

(e.g , xLIA, ACSL) will not be referenced by their respective viewpoint. However this only means that the Xtext-implemented model kinds will be available in all viewpoints.

After implementation of SecSoftML, several model transformation and analysis tools must be implemented for the co-engineering method where model kinds of viewpoints of SecSoftML are used.

5.2 Transformations and analyses implementation

In the software/security co-engineering method, described in Section 4.1, there are a certain number of steps dedicated to model transformation and analysis. For the implementation of these steps, existing Papyrus-based MDE tools, and existing Frama-C-based static code analysers, will be re-used. The implementation of these steps, and what needs to be modified/added in the tools, will be specified in the following sections.

5.2.1 Code generation with Papyrus Software Designer

In step 1.4, code with properties in ACSL must be generated. For code generation, we propose to use existing Papyrus Software Designer code generator frameworks to develop new generators necessary for the modelling framework of VESSEDIA.

Papyrus Software Designer is a plugin for Papyrus which generates code in different programming languages from UML-based models. Currently the tool supports C++ code generation for software components and state machines modelled in UML.

For VESSEDIA, a C code generator for UML structured classes and state machines must be developed. The new and existing code generators must also be updated to consider constraints expressed in ACSL, and generate the equivalent ACSL property in the code.

5.2.2 ACSL program relational properties inference with DIVERSITY

DIVERSITY is model analysis tool based on symbolic execution. DIVERSITY is extensible allowing customizing the basic symbolic treatments to implement specific Formal Analysis Modules (FAM) (e.g., Model-based Testing (MBT), algorithms, exploration strategies and heuristics). DIVERSITY is connected with SMT solvers, e.g., CVC4, Z3 and YICES which can be easily used to implement new FAMs.

In fact, DIVERSITY provides "hooks" into the basic symbolic execution algorithm which customize the construction of the symbolic tree. These "hooks" allow FAMs – by implementing specific functions – to pre-process or post-process current reached symbolic states and to manage the queue of the remaining symbolic states to be processed. This mechanism provides developers with extension mechanisms to instrument and specialize the traditional symbolic execution algorithm without having to re-implement the basic symbolic treatments.

The ACSL program relational properties contract inference FAM (inference FAM in short) must be developed in VESSEDIA using those facilities. As glimpsed previously, DIVERSITY provides a pivot language called xLIA. UML interactions, with symbolic data and function call constraints, are translated to xLIA. This way, we endow the interactions with symbolic semantics used to infer ACSL program relational properties. Specific xLIA patterns and extensions must be developed in VESSEDIA to support in particular black-box functions calls. The inference FAM must take as input such models together with a specific user coverage goal in order to compute ACSL program relational properties contracts. The inference FAM must post process the symbolic states/paths and compute, by another FAM, the “reachability heuristics”. This latter FAM must compute a set of symbolic states targets of some paths satisfying a coverage goal such as covering (successively) some transitions, states, I/O actions, function calls, or satisfying logical formulas on the model state variables. Hence the inference FAM is meant to be used intertwined with the reachability heuristics in order to infer contracts for a cooperation of function

calls which implement specific safe high level system scenarios. More details on the inference FAM can be found in deliverable D3.1.

5.2.3 Static code analysis with Frama-C

In step 2.3, static code analysis must be performed on the ACSL-annotated C code. We propose to use the existing Frama-C static code analyse to develop new analysis.

The ACSL++ specification language for C++ is described in deliverable D.2.3. The relational properties analysis is specified in D.3.1.

A new Frama-C launcher can be developed to be integrated within the same IDE as the transformation tools.

The next section describes how the modellers of SecSoftML, and the transformation and analysis tools are packaged, are packaged within a same IDE.

5.3 Integrated development environment

The SecSoftML modellers and the transformation and analysis tools are to be packaged within an IDE. To build the IDE, we propose to use the Eclipse Rich Client Platform (RCP) framework.

Eclipse [EclipseIDE] is a modular IDE application. Eclipse RCP supports reusing components of the Eclipse platform to build stand-alone applications based on the same technology as the Eclipse IDE. The Eclipse platform is used as a basis to create feature-rich stand-alone applications.

For the VESSEDIA modelling framework, the IDE must obviously contain the developed SecSoftML modellers described in Section 5.1, and the developed transformation and analysis tools described in Section 5.2. For such features, the following dependencies must be packaged in the IDE: Papyrus, Papyrus Software Designer, Xtext, DIVERSITY, Eclipse CDT (C/C++ tools).

Chapter 6 VESSEDIA modelling framework applied on a ping-pong use-case

In this chapter, we show how an early implementation of the VESSEDIA modelling framework is used to design a simple use-case called “ping-pong”. We will also model and verify its requirements at architecture model and code levels. In particular, model kinds and steps related to software component modelling, risk modelling, and inference of program relational properties will be shown.

In the next sections, first we mainly present our usage of the UML sequence diagram, depicting interactions, including opaque function calls. Since we are using sequence diagrams for specification purposes of the context in which functions are called, we have introduced primitives to control (and hence reason about) exchanged data and function parameters in an abstract manner. We use as well other kinds of UML diagrams: the class diagram allows to define for each system component in an interaction its computation variables, and functions signatures. Also, we use the composite structure diagram which gives a static view of the interconnected system components using connectors. The modelling concepts are introduced in this chapter by means of a toy example: a simple “ping-pong” protocol.

We then explain the attribution to such models of an operational semantics by translation into xLIA, the entry language of the Diversity tool. This will enable the usage of the DIVERSITY formal analysis module developed in T3.1 (D3.1) which infers contracts for a cooperation of function calls that implement specific critical interactions captured by the UML sequence diagram. These contracts are then translated as program relational properties expressed in ACSL.

6.1 Software component modelling of ping-pong use-case

The software components of the ping-pong use-case are modelled in Figure 12 with a composite structure diagram, focusing on their collaborations, and in Figure 13 with a class diagram, focusing on their interfaces.

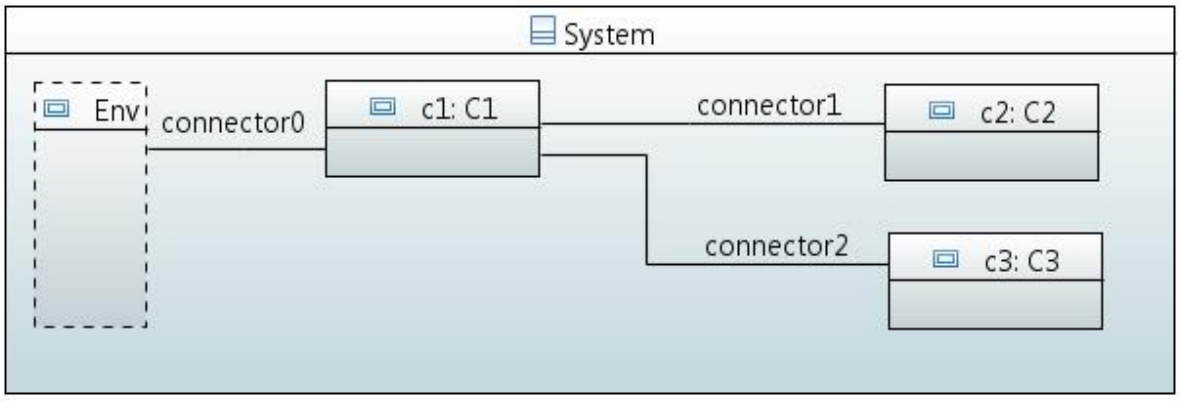

Figure 12: Composite structure diagram representing ping-pong components

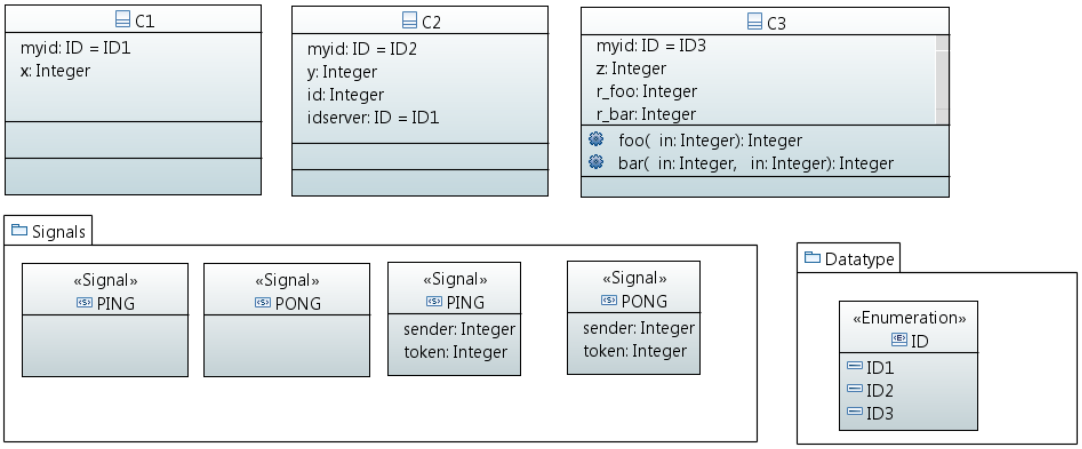

Figure 13: Class diagram representing ping-pong components

Parts c1, c2 and c3 are represent the components C1, C2 and C3 respectively. Note that variables are defined over common data types: e.g. the variable myid of Class C1 is an integer denoting the identity of C1 in the network which is set to the identifier ID1 by default (ID1 together with ID2 and ID3 is a literal member of the enumeration ID). The composite structure diagram shows two connectors joining c1 respectively to c2 and c3. The connectors specify the communication channels used to exchange messages between components or the environment represented in the composite structure diagram as distinguished part in dashed line. Signals, e.g., PING, PONG, etc…, are used to model asynchronous communication between components. Their attributes represent data exchanged between components. In our case, data is handled in an abstract manner as first order structures using variables, functions and predicates together with associated terms and formulas. The latter are built using the usual logical connectives. We distinguish a subset of functions in the component signature which represent the targeted user programs for which contract will be inferred in order to verify their actual code. They are treated in an abstract manner, i.e., as a black box functions without their internal behaviour modelled. Two of such functions are defined in C3, foo() and bar() together with their parameters(‘s types).

The next section shows how requirements of these components are modelled as interactions in a sequence diagram.

6.2 Risk modelling of ping-pong use-case

Consider the UML sequence diagram of Figure 14 showing interactions between the three components previously modelled. The interaction elements are annotated with constraints expressed in xLIA. This model shows the required ping-pong protocol behaviour that must be respected by the communicating components.

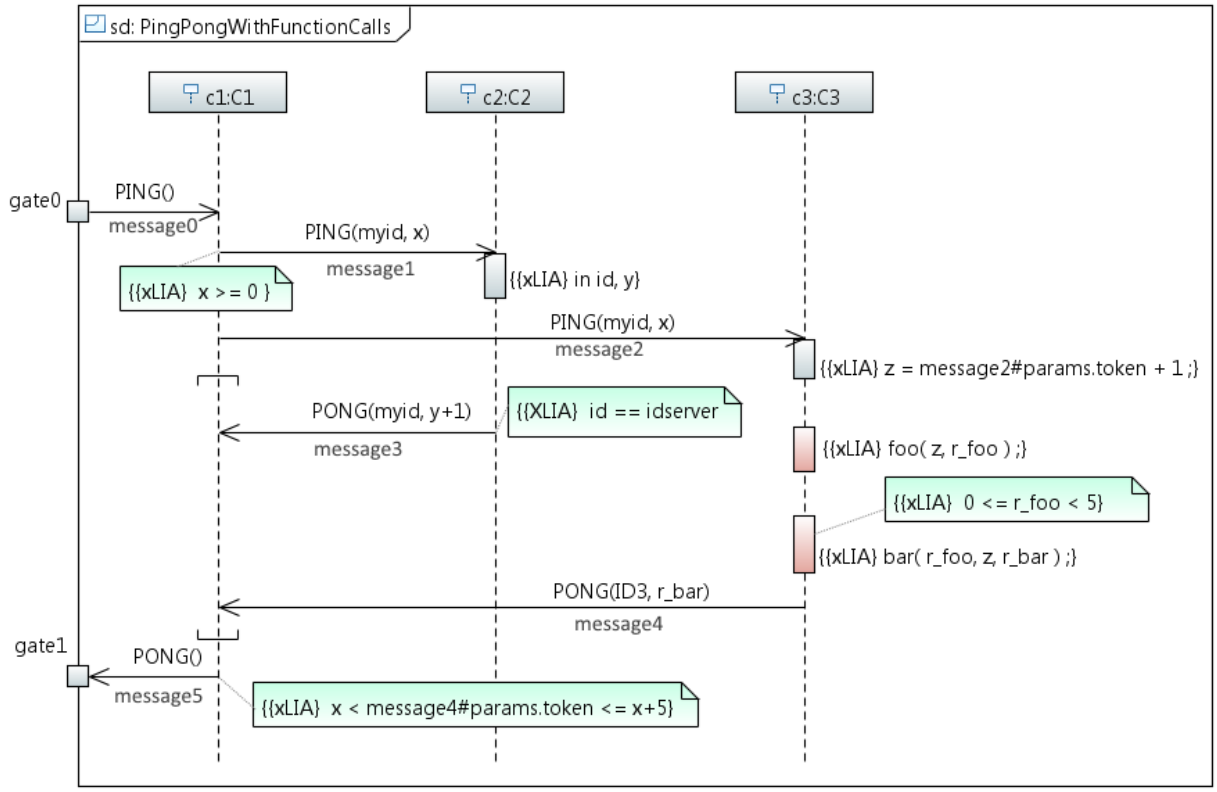

Figure 14: Interaction between components of ping-pong

Components c1, c2, and c3 are associated each with a vertical lifeline where time evolves from top to bottom. Communications are based on asynchronous passing of messages message0 to message5, represented by arrows connecting the lifelines. The (communication) events in the sequence diagrams are partially ordered. Thus, receptions of message1 and message2 respectively by components c2 and c3 may come in any order.

Locally, events along a given lifeline come in sequence. However, the co-region operator allows to specify arbitrarily ordered events as for receptions of message3 and message4 by component c1. The co-region operator delimits an area on the lifeline between square brackets where these receptions are situated.

The connector architecture given in the UML composite structure diagram allows component c1 to emit PING signals by sending the messages message1 and message2 respectively to components c2 and c3.

A single message may convey multiple pieces of data as an abstraction of all kinds of signal parameter values that can be exchanged between components and represented by abstract terms. For example, message3 is the response of component c2 by sending the PONG(myid, y+1) signal where “myid” and “y+1” are terms representing respectively the identity of component c2 and the variable “y” being incremented. Here “y” is a kind of nonce associating a response to its issued request thus ensuring that both messages are newly made.

When a message is received, reception variables may be specified: e.g. upon the reception of message1 the values of “myid” and “x” are stored respective in the variables “y” and “id”. In case no reception variables are specified theses values will be stored in an implicit structure “message2.params” of the same typed as PING, i.e., having hence two integer attributes message2.params.sender and message2.params.token which are handled as usual variables as well. Note that handling variables as first-order-structures which can be used in computations, or as parameters to signals and function calls is a particular feature of our modelling framework which is not clearly well stated in the UML norm.

Consequently, we will see next that iterations (as UML sequence diagram) can hence be associated unambiguous semantics using symbolic execution. We take into consideration more

complex interactions combined with the following operators: the loop operator defines repetitive interaction, the alt operator defines alternative interactions, and the strict operator defines a strict sequencing of interactions which enforces the executions of all message exchanges in every interaction before any message of the following interaction. The first models of the 6LowPAN use-case, discussed in D3.1, make use of such combining operators.

The next section shows how to infer program relational properties from the interaction model presented so far.

6.3 Inference of program relational properties of ping-pong use-case

The inference of program relational properties is a two steps process where first the requirements expressed as interactions in UML sequence diagrams are translated to xLIA automatas in DIVERSITY. The tool then computes contracts which are then translated to ACSL-expressed program relational properties.

The next sections first present the translation of the interactions to xLIA automatas and computation of contracts. Afterwards the translation to ACSL program relational properties is given.

6.3.1 Translation of UML interactions to xLIA

We overview the operational semantics of sequence diagrams computed by a model transformation. The transformation takes as input a specialized UML Interaction and produces its equivalent set of STS communicating over FIFO buffers in xLIA format. An excerpt of the generated xLIA is given in Figure 15.

```
@xlia< system , 1.0 >:
system< and > PingPongWithFunctionCalls {
@property:
…
type PONG struct {
var string signature;
var integer sender;
var integer token;
…
signal message4( PONG );
signal message5;
@composite:
  machine C1#c1Lifeline {…}
  machine C2#c2Lifeline {…}
  machine C3#c3Lifeline {
   @property:
     public var ID myid = ID.ID3;
     public var integer z;
     public var integer r_foo;
     public var integer r_bar;
   @routine:
     macro routine foo(integer x, return integer y) {}
     macro routine bar(integer a, integer b, return integer c) {}
  @behavior:
    statemachine< or > c3Lifeline {
      @region:
        state BhExec#BehaviorExecSpec1 {
          transition BehaviorExecSpec1 {
            guard(  0 <= r_foo < 5 );
            bar( r_foo, z, r_bar ) ;
            AllCallsStack <=< currentCall;
          } --> MsgOcc#message4Send;
        }
        state MsgOcc#message4Send {
          transition tr_message4Send {
            output message4( { "PONG", ID3 , r_bar } ) --> C1#c1Lifeline;
          } --> final_c3Lifeline;
        }
    }
@com:
  …
  route<fifo> [ message4];
  route<fifo> [ message5];
}
```

Figure 15: xLIA generated from UML interaction of ping-pong components

Each UML lifeline's component is translated into an STS and is able to communicate with each other using input-output communication actions. Messages in the UML Interaction represent data exchanged between lifelines. As we consider asynchronous messages: i.e., the sender lifeline of a message is not blocked while the message has not been received. We capture this by routing each message when it is sent in dedicated FIFO buffer. Note that routing all messages between the same two lifelines in the same FIFO does not allow message overtake which is allowed in sequence diagrams in order to capture a wide range of system protocols and communication mechanisms. About data, xLIA provides seemingly different concepts from UML signals, datatypes, enumerations that we have used to capture their usage in UML: xLIA concepts are handled as first-order structures. Finally black box functions such as foo() and bar() are translated into xLIA routines without their internal behaviour modelled (not effect on input parameters), yet each time they are called their formal input parameters and return values are stored in a call stack. This allows the application of specific symbolic treatments to their calls unlike usual functions with explicit behaviour.

The next section gives the translation of xLIA to program relational properties in ACSL.

6.3.2 Translation to ACSL-expressed program relational properties

As explained in the previous section, we use xLIA enriched with data variables to abstractly denote system states (we call them data variables) and we keep track of black box function calls (stored in the call stack) in order to associate them with specific symbolic treatments which is the extension proposed in the scope of VESSEDIA. The resulting xLIA is associated with efficient semantics computation using symbolic execution techniques. Symbolic execution main principle is to reason about all the possible executions of the model by studying how the assignments of its variables evolve when transitions are executed. In practice the variables are assigned with formal parameters. Besides return values of function calls are represented as well in a symbolic manner using dedicated formal parameters. Then constraints on those formal parameters are computed in order to characterize the effect on the global executions of the guards, instructions and function calls occurring in transitions. Those constraints are called Path Conditions, PC in short and will be processed together with the accumulated function calls to infer the relational properties (more details on the inference process can be found in D3.1). This an example of a PC computed for the ping-pong example:

$$x1 \geq 0 \wedge 0 \leq r\_foo1 < 5 \wedge x1 < r\_bar1 \leq x1+5$$

and the accumulated function calls for PC is:

$$\{ (\text{“foo”}, x1+1, r\_foo1), (\text{“bar”}, r\_foo1, x1+1, r\_bar1) \}$$

where x1, r_foo1, r_bar1 are the formal parameters introduced by the symbolic execution.

Finally we give in Figure 16 the inferred relational property based on both information expressed in the format acceptable by Frama-C (ASCL/RPP plugin).

```
/* @ relational

\forall   int x1;
\callset(
    \call(foo, x1+1, id1) ,
    \call(bar, \callresult(id1), x1+1, id2)
)
=>
(x1  >= 0  =>
         ( \callresult(id1) >= 0  && \callresult(id1) <5
                =>
         (\callresult(id2) > x1 && \callresult(id2) <= x1 +5)));
*/
```

Figure 16: Inferred program relational property in ACSL for ping-pong components interaction

Chapter 7 Summary and conclusion

In this document, the VESSEDIA modelling framework was proposed for the development of secure software. The goal of the modelling framework is to bridge the gap between high-level textual requirements in an architecture model and low-level properties annotating the code.

The particularity of the VESSEDIA modelling framework is that it does not propose new ADLs or risk assessment methodologies. It rather aggregates existing approaches and uses generic enough artefacts to be extended with new approaches in the future.

The VESSEDIA modelling framework is based on three major parts: the SecSoftML architecture description language, the soft/security co-engineering method that uses and transforms the elements of SecSoftML, and finally implementation with VESSEDIA tools.

SecSoftML is an ADL whose specification was done with respect to the ISO 42010 standard for architecture description languages. SecSoftML aims to answer the classical software development and security concerns of the software engineer and security analyst. It contains a number of viewpoints for requirements analysis, security analysis, and software design. The viewpoints will aggregate model kinds which are diagrams and tables of SysML, SysML-Sec, and UML. They also aggregate textual specification languages such as ACSL and xLIA.

SecSoftML's viewpoints and model kinds are used within the software/security co-engineering method proposed in this document. The method has the advantage of parallelizing the classical software development process with the security analysis process. The goal is to avoid as much as possible work product dependencies and blocking among the two domains and tasks. A certain number of transformation and analysis steps will exploit the models and artefacts produced during steps of the method. Relational ACSL-expressed properties are inferred from interactions in UML sequence diagrams. ACSL-expressed pre/post-conditions of functions and function behaviour properties are generated from ACSL-expressed constraints on elements in UML class and state machine diagrams. Final program code, with ACSL-expressed properties, are analysed with Frama-C.

An implementation for SecSoftML was proposed with Papyrus UML profiles, diagrams, Xtext editors, and viewpoints developed within the architecture framework model of Papyrus. Implementation of the transformations and analyses in the co-engineering method was proposed with existing tools like DIVERSITY for ACSL-expressed program relational properties inference, Papyrus Software Designer for ACSL-annotated C code generation, and Frama-C for static code analysis. The necessary extensions for these tools were specified in this document. All implementation works will be done within task T1.3 for deliverable D1.4.

An early implementation of the VESSEDIA modelling framework was applied for a ping-pong use-case. The use-case is a simple ping and acknowledgement ("pong") protocol. The use-case's components were modelled in UML composite structure and class diagrams. The required protocol behaviour was modelled as an interaction in a UML sequence diagram. Program relational properties were then inferred with a two steps process where first the interactions are translated to xLIA automatas so DIVERSITY can compute program relational properties contracts. The computed contracts are then translated to ACSL-expressed program relational properties. The properties need to be respected by the implementation to guarantee the ping-pong protocol's required behaviour.

In the future, we would like to integrate in SecSoftML other textual constraint specification languages like OCL [Warmer98] and Verifast [Jacobs11] for Java. The UML diagrams in SecSoftML are also generic so they can be specialized with profiles adding new taxonomy to simplify the modelling. For example we would like to explore the FormalML profile developed in VESSEDIA for formal properties modelling, and the AtML profile for attack modelling.

Chapter 8 List of abbreviations

Abbreviation	Translation
ADL	Architecture Description Language
ACSL	Ansi C Specification Language
AtML	Attack Modelling Language
CSS	Cascading Style Sheets
DSML	Domain-Specific Modelling Language
EMF	Eclipse Modelling Framework
ENISA	European Union Agency for Network and Information Security
FAM	Formal Analysis Module
IDE	Integrated Development Environment
MBT	Model-Based Testing
MDE	Model-Driven Engineering
SDL	Specification and Description Language
SecSoftML	Secure Software Modelling Language
STS	Symbolic Transition System
SysML	System Modelling Language
SysML-Sec	System Modelling Language for Security
UML	Unified Modelling Language
xLIA	executable Language for Interaction and Architecture

Chapter 9 Bibliography

[Arnaud16] M. Arnaud, B. Bannour, A. Lapitre. "An illustrative use case of the DIVERSITY platform based on UML interaction scenarios". Electronic Notes in Theoretical Computer Science, vol. 320, pp 21-34, Febraury 2016.

[Bannour14] B. Bannour, J. Escobedo, C. Gaston, P. Le Gall, G. Pedroza. "Security weaknesses detection by symbolic analysis of scenarios". Proceedings of APSEC, 2014.

[EBIOS] https://www.enisa.europa.eu/topics/threat-risk-management/risk-management/current-risk/risk-management-inventory/rm-ra-methods/m_ebios.html

[EclipseIDE] https://www.eclipse.org/

[EMF] https://www.eclipse.org/modeling/emf/

[ENISAMethods] https://www.enisa.europa.eu/topics/threat-risk-management/risk-management/current-risk/risk-management-inventory/rm-ra-methods

[ENISATools] https://www.enisa.europa.eu/topics/threat-risk-management/risk-management/current-risk/risk-management-inventory/rm-ra-tools

[Gerard07] S. Gérard, C. Dumoulin, P. Tessier, B. Selic. "Papyrus: a UML2 tool for domain-specific language modeling". Proceedings of MBEERTS, 2007.

[ISO42010] https://www.iso.org/standard/50508.html

[Jacobs11] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, F. Piessens. "Verifast: a powerful, sound, predictable, fast verified for C and Java". Proceedings of NASA Formal Methods, 2011.

[Kirchner15] F. Kirchner, N. Kosmatov, V. Provosto, J. Signoles, B. Yakobowski. "Frama-C: a software analysis perspective". Formal Aspects of Computing, vol. 27:3, pp 573-609, May 2015.

[Pham16] V.C. Pham, S. Li, A. Radermacher, S. Gérard, C. Mraidha. "Fostering software architect and programmer collaboration". Proceedings of ICECCS, 2016.

[Roudier15] Y. Roudier, L. Apvrille. "SysML-Sec: A model driven approach for designing safe and secure systems". Proceedings of MODELSWARD, 2015.

[Sendall03] S. Sendall, W. Kozaczynski. "Model transformation: the heart and soul of model-driven software development". IEEE Software, vol. 20:5, pp 42-45, September 2003.

[SysMLSpec] https://www.omg.org/spec/SysML/1.4/PDF

[UMLSpec] https://www.omg.org/spec/UML/2.5.1/PDF

[Warmer98] J.B. Warmer, A.G. Kleppe. "The Object Constraint Language: Precise Modeling With Uml". Addison-Wesley Object Technology Series, 1998.

[Xtext] https://www.eclipse.org/Xtext/