UMLsec4UML2 – Adopting UMLsec to Support UML2*

Using UMLsec4UML2 for the Specification of Architectural Security Patterns

Holger Schmidt and Jan Jürjens

In this paper, we present an approach to adopt UMLsec, which is defined for UML 1.5, to support the current UML version 2.3. The new profile *UMLsec4UML2* is technically constructed as a UML *profile diagram*, which is equipped with a number of *integrity conditions* expressed using *OCL*. Consequently, the UMLsec4UML2-profile can be loaded in any Eclipse-based EMF- and MDT-compatible UML editing tool to develop and analyze different kinds of security models. The OCL constraints replace the static checks of the tool support for the old UMLsec defined for UML 1.5. Thus, the UMLsec4UML2-profile not only provides the whole expresiveness of UML2.3 for security modeling, it also brings considerably more freedom in selecting a basic UML editing tool, and it integrates modeling and analyzing security models.

Since UML2.3 comprises new diagram types, as well as new model elements and new semantics of diagram types already contained in UML1.5, we consider a number of these changes in detail. More specifically, we consider *composite structure* and *sequence diagrams* with respect to modeling security properties according to the original version of UMLsec. The goal is to use UMLsec4UML2 to specify *architectural security patterns*.

1 Introduction

The main goal of the work presented in this paper builds upon open research questions developed in the authors' PhD thesis (Schmidt, 2010). There, a pattern- and component-based method to develop secure software is presented. This method is named *Security Engineering Process using Patterns* or short SEPP, and it focuses on the early phases of software development, i.e., requirements analysis, specification, and architectural design.

In (Schmidt, 2010, Part III), we presented several patterns for security sub-architectures, which are denoted using different UML (Unified Modeling Language) (UML Revision Task Force, 2010) diagrams to represent structural as well as behavioral views on the sub-architectures. Against this background, the work presented in this paper should serve to achieve the following main goals:

- Express the patterns for security sub-architectures presented in (Schmidt, 2010, Part III) using UMLsec
- Since the patterns for security sub-architectures are expressed using UML2.3 (e.g., structural views are expressed using composite structure diagrams), adopt UMLsec to support UML2.3

The rest of the paper is organized as follows: we introduce some background information about UML and UMLsec in Section 2, and we present an example of a security-critical software development problem in Section 3, which we will use in the following sections to demonstrate the techniques presented in this paper. We present the UMLsec4UML2-profile in Section 4. In Section 5, we present an approach to specify architectural security patterns that makes use of the UMLsec4UML2-profile and a diagram type new to UML2.3, i.e., composite structure diagrams. In Section 6, we give a summary.

*Partially supported by the EU project "Security Engineering for Lifelong Evolvable Systems (Secure Change)" (ICT-FET-231101)

2 Background

We first present an overview of the UMLsec-profile for UML1.5 in Section 2.1. Second, we compare the current UML version 2.3 with version 1.5 in Section 2.2.

2.1 Previous Version of UMLsec for UML1.5

UMLsec (Jürjens, 2005) is a profile for UML1.5 to develop and analyze security models. UMLsec considers confidentiality and integrity properties depicted in different UML diagrams using *stereotypes*, *tags*, and *constraints*. Stereotypes give a specific meaning to the elements of a UML diagram they are attached to, and they are represented by labels surrounded by double angle brackets. Constraints are associated with the stereotypes. The set of stereotypes predefined by UMLsec comprises on the one hand structural syntactic conditions, and on the other hand semantic conditions on behavior models. The stereotypes available in UMLsec are summarized in (Jürjens, 2005, p. 51). A tag or tagged value is a name-value pair in curly brackets associating data with elements in a UML diagram. The tags available in UMLsec are summarized in (Jürjens, 2005, p. 52).

As an example, we discuss the stereotype `<<critical>>`, which can be attached to objects or subsystem instances containing data that is security-critical. Using tags such as secrecy and/or integrity, the stereotype is specified in more detail. The values of the tag secrecy are the names of attributes or message parameters of the object to be kept confidential. The values of the integrity tag are pairs (v;E) where v is a variable of the object whose integrity should be protected and E is the set of acceptable expressions that may be assigned to v. The expression E can be left out, which means that the value v must remain unchanged completely. Constraints based on these tags are enforced by the stereotype `<<data security>>` that labels subsystems containing `<<critical>>` objects. The tag adversary can be used to describe attackers of a certain strength such as the default strength (Jürjens, 2005, pp. 57 ff.).

As already mentioned above, the stereotypes' constraints can be grouped into structural syntactic conditions (see Figure 1 for an overview) and semantic conditions on behavior models (see Figure 2 for an overview). Note that these sets of stereotypes are not disjoint.

Secure software design according to UMLsec usually starts by first applying and verifying stereotypes that constrain the structural design. Then, once the structural properties are successfully verified, one proceeds by applying and verifying stereotypes that constrain the behavioral design models. The verification is supported by a tool suite available online via `http://www.umlsec.de/`, which basically reflects the mentioned groups of stereotypes:

Static checks are used for the verification of stereotypes that constitute structural syntactic conditions on different UML diagrams.

Permission analysis is used for the verification of stereotypes that express RBAC (Role-Based Access Control) (Ferraiolo & Kuhn, 1992) constraints on behavior models such as activity and sequence diagrams.

Integration with external verification tools by generating input data for these tools (e.g., SPASS theorem prover (*SPASS Theorem Prover 3.5*, 2010)) and receiving result data from them. Here, the semantic behavioral conditions are verified using the external tools.

Code generation for some diagram types, e.g., class diagrams.

2.2 Changes in UML2.3 Compared to UML1.5

In this section, we briefly present changes in the current UML version 2.3 compared to the version 1.5, which is supported by UMLsec.

Structure Diagrams

Class diagram already contained in UML1.5; semantics and metamodel partly changed, e.g., association ends, association classes, etc.

Component diagram already contained in UML1.5; the component construct gains capabilities of composite structure diagrams, e.g., ports and connectors

Stereotype	**Diagram Type**	**Model Element**	**Constraints**
fair exchange	use case diagram	package	can be refined only by an activity diagram that is stereotyped `<<fair exchange>>`
rbac	activity diagram	package	enforces role-based access control; see (Jürjens, 2005, pp. 55 ff.) for details
Internet	deployment diagram	communication path	Internet connection; conflicts with other UMLsec stereotypes for communication paths; see (Jürjens, 2005, pp. 56 ff.) for details
encrypted	deployment diagram	communication path	encrypted connection; conflicts with other UMLsec stereotypes for communication paths; see (Jürjens, 2005, pp. 56 ff.) for details
LAN	deployment diagram	communication path, node	LAN (local area network) connection; conflicts with other UMLsec stereotypes for communication paths or nodes; see (Jürjens, 2005, pp. 56 ff.) for details
wire	deployment diagram	communication path	wire; conflicts with other UMLsec stereotypes for nodes; see (Jürjens, 2005, pp. 56 ff.) for details
smart card	deployment diagram	node	smart card node; conflicts with other UMLsec stereotypes for nodes; see (Jürjens, 2005, pp. 56 ff.) for details
POS device	deployment diagram	node	POS (point-of-sales) device; conflicts with other UMLsec stereotypes for nodes; see (Jürjens, 2005, pp. 56 ff.) for details
issuer node	deployment diagram	node	issuer node; conflicts with other UMLsec stereotypes for nodes; see (Jürjens, 2005, pp. 56 ff.) for details
secrecy	class diagram, component diagram, deployment diagram	dependency	requires secrecy; see (Jürjens, 2005, pp. 59 ff.) for details
integrity	class diagram, component diagram, deployment diagram	dependency	requires integrity; see (Jürjens, 2005, pp. 59 ff.) for details
high	class diagram, component diagram, deployment diagram	dependency	requires high sensitivity; see (Jürjens, 2005, pp. 59 ff.) for details
critical	class diagram, component diagram, deployment diagram	class, component, interface	critical class, component, or interface; see (Jürjens, 2005, pp. 58 ff.) for details
secure links	deployment diagram	package	enforces secure communication links; used to include malicious environment; see (Jürjens, 2005, p. 59) for details
secure dependency	class diagram, component diagram	package	structural data security; see (Jürjens, 2005, pp. 59 ff.) for details
guarded access	???	package	access control using guard classes; see (Jürjens, 2005, pp. 65 ff.) for details
guarded	class diagram	class	guarded class; see (Jürjens, 2005, p. 66) for details

Figure 1: UMLsec Stereotypes with Syntactic Structural Constraints

Stereotype	Diagram Type	Model Element	Constraints
fair exchange	activity diagram	package	see (Jürjens, 2005, pp. 53 ff.) for details
provable	activity diagram	package	see (Jürjens, 2005, p. 55) for details
data security	at least one structure and one behavior diagram	package	basic data security requirements; see (Jürjens, 2005, pp. 60 ff.) for details
no down-flow	state machine diagram, sequence diagram	package	information flow condition; see (Jürjens, 2005, pp. 64 ff.) for details
no up-flow	state machine diagram, sequence diagram	package	information flow condition; see (Jürjens, 2005, pp. 64 ff.) for details

Figure 2: UMLsec Stereotypes with Semantic Behavioral Constraints

Composite structure diagram *new diagram type* to describe component structures

Deployment diagram already contained in UML1.5; e.g., nodes can now contain any element that can be included in a package (not only components)

Object diagram already contained in UML1.5;

Package diagram *new diagram type* to depict how a system is split up into logical groupings by showing the dependencies among these groupings

Profile diagram *new diagram type* to define UML profiles using standard extension mechanisms

Behavior Diagrams

Activity diagram already contained in UML1.5

- semantics changed and now similar to Petri-nets
- swim-lanes, called activity partitions, that describe what specific classes or subsystems do
- no longer emphasizes transitions; more concerned with token flow along activity edges

State machine diagram already contained in UML1.5

Use case diagram already contained in UML1.5; new model elements are actor generalization and direction of interaction; stereotype `<<used>>` renamed to `<<include>>`

Interaction Diagrams

Interaction diagrams, a subset of behavior diagrams, emphasize the flow of control and data among the different parts in the system being modeled:

Communication diagram named *collaboration diagram* in UML1.5

Interaction overview diagram *new diagram type* that represents an activity diagram in which the nodes represent interaction diagrams;

- Top-level description of the main flow of interactions
- Basically an activity diagram, where nodes can be refined using other interaction diagram types

Sequence diagram already contained in UML1.5; e.g., new model elements are combined fragments

Timing diagram *new diagram type* that represents an interaction diagram, where the focus is on timing constraints

Moreover, the definition and usage of stereotypes has changed in UML2.3. In contrast to UML1.5, a tagged value is always attached to a specific stereotype.

3 Running Example

We partly demonstrate the results presented in this paper using the example of a *password manager*. This case study is taken from (Deshmukh, Kawana, & Erkulla, 2010) and (Schmidt, 2010, Part IV), where additional material such as the complete results from requirements analysis, the corresponding specification, and the architectural design can be found.

The password manager is a client-server-based application. The *password manager client* is displayed graphically to a user. It allows a user to create a *master user account* by filling in a *registration form*, which requests a *master username* and a *master password*. On submit, the master user account data is transmitted to the *password manager server*, where it is stored in a *database*.

Once the master user account is successfully created, a user can anytime login to the password manager by entering his/her master username and master password. After successful login, a user can manage his/her *personal user account*. A personal user account consists of a *personal username*, a *personal password* and a *personal user account description*. A user can *add, view, modify and delete* his/her personal user accounts.

After login, a user should be able to view his/her existing personal username(s) (if any). From the displayed personal usernames a user can select any of the personal username to view the personal password and personal user account description.

A user should fill in an *add form* to add a personal user account, where a user is requested for the personal username, the personal password and the personal user account description. On submit, the personal user account is transmitted to the password manager server, where it is stored in a database and associated to his/her master user account.

A user can modify the existing personal user accounts by using a *modify form*. On submit, the modified data is transmitted to the password manager server, where the changes are made in the database.

A user can delete the existing personal user account by selecting any of his/her personal username. On select the request is sent to the web-based password manager server, where the selected personal user account is deleted from the database.

Finally, the user can log out from the password manager.

4 A UMLsec-Profile for UML2

In this section, we first explain how profiles can be constructed, defined, and used according to UML2.3 in Section 4.1. Second, we describe the UMLsec4UML2-profile in Section 4.2, and finally, we present OCL integrity conditions for this profile in Section 4.3. Note that the complete UMLsec4UML2-profile, all examples shown in this paper, as well as additional material are available online via `http://www.umlsec.de/umlsec4uml2.html`.

4.1 Technical Background

We give in this section an overview of the technical background for the construction (Section 4.1.1), definition, and usage (Section 4.1.2) of UML2.3 profiles.

4.1.1 UML Profile Construction

The *Eclipse Modeling Framework* (EMF) (Steinberg, Budinsky, Paternostro, & Merks, 2009) is a framework for modeling and code generation. It unifies different representation forms of models, i.e., Java (*SUN Java 6 Standard Edition*, 2010), XML (*XML - Extensible Markup Language*, 2010), UML (UML Revision Task Force, 2010), by enabling seamless transformations between these notations. The *ECore* model is used to represent EMF models. Hence, the ECore model is a metamodel for EMF models. In fact, it is a meta-metamodel, since it is an EMF model, too. Eclipse (*Eclipse - An Open Development*

Platform, 2010) and Eclipse-based applications support the ECore model and hence, EMF models. The *Model Development Tools* (MDT) (*Model Development Tools Project (MDT)*, 2010) project introduces implementation of industry standard metamodels such as UML2.3 (UML Revision Task Force, 2010), OCL2.0 (UML Revision Task Force, 2006), and BPMN2.0 (Object Management Group (OMG), 2009), and it provides sample tools for developing models based on those metamodels. Finally, Eclipse in combination with the EMF-based implementations of the UML2.3 metamodel allows the construction of UML profiles. For the work presented in this paper, the Eclipse-based UML editing tool *Papyrus UML* (*Papyrus UML 1.12*, 2010) is used. It is available as an Eclipse-plugin, and it is free and open-source. Alternative and compatible editing tools are, e.g., Topcased (*Topcased 3.4.1*, 2010), Eclipse, and MagicDraw UML (*MagicDraw UML 16.8*, 2010). Note that Papyrus UML is now part of the Eclipse MDT project. There, a new state-of-the-art UML editing tool is currently developed based on Papyrus UML, Topcased, and MOSKitt (*MOSKitt – MOdeling Software Kitt*, 2010).

The basis for a UML2.3 profile is a profile diagram, which defines extensions to the UML2.3 reference metamodel. This way, it is possible to adapt the metamodel to a specific platform or domain. Metaclasses from the reference metamodel are extended via stereotypes, which are defined as profile parts. Stereotypes may have properties, which are referred to as tag definitions. Moreover, a profile diagram can be enriched with formal constraints in OCL (see Section 4.3 for details). These constraints allow to check the validity and consistency of a model created using the profile.

4.1.2 UML Profile Definition and Usage

Before a UML profile can be used, it has to be defined. Since a UML profile represents an extension of the reference UML2.3 metamodel, in order for the specified extensions to appear as though they are part of the UML2.3 metamodel, they need to be defined at the meta-metamodel (i.e., Ecore) level. This is done automatically when storing a UML profile in Papyrus UML.

A UML2.3 profile can be used to create corresponding models using any Eclipse-based editing tool that supports EMF/ECore and MDT. If the used profile is enriched with OCL constraints, and the used editing tool supports the verification of OCL constraints (as it is the case for Papyrus UML, Eclipse, MagicDraw UML, etc.), then models created using this profile can be verified automatically with respect to the OCL constraints.

4.2 UMLsec4UML2

We present a UML2.3-compatible profile UMLsec4UML2 that adopts the UML1.5-compatible profile UMLsec (Jürjens, 2005). It provides the model elements as defined for the UMLsec profile developed for UML1.5, and it allows to check if a model is valid and consistent with respect to the UMLsec4UML2-profile. Using the UMLsec4UML2-profile comprises several benefits:

- availability of the whole *expressiveness of UML2.3* for security modeling including *new diagram types of UML2.3*
- *freedom* to select from a number of compatible UML editing tools
- partly *integration of modeling and analysis*, since static checks are executed directly within the UML editing tool
- *improved feedback for debugging models* provided by evaluation of OCL constraints
- *improved maintainability*, since UMLsec4UML2 is developed as a UML2.3 package diagram, which can be evolved easily. This also applies to the OCL constraints included in the profile.
- opens the way to develop *additional tool support* "à la maison", i.e., as Eclipse-plugins

UMLsec4UML2 is constructed using the Papyrus UML editing tool (*Papyrus UML 1.12*, 2010) as a UML profile diagram, which is part of UML2.3. A profile diagram operates at the metamodel level to show stereotypes as classes with the `<<stereotype>>` stereotype, and profiles as packages with the `<<profile>>` stereotype. The extension relation (solid line with closed, filled arrowhead) indicates what metamodel element a given stereotype is extending. Figure 3 shows a part of the package diagram that represents the UMLsec4UML2-profile.

The workflow for using the UMLsec4UML2-profile is illustrated in Figure 4. The workflow is incremental and iterative in nature, and it consists of the following steps:

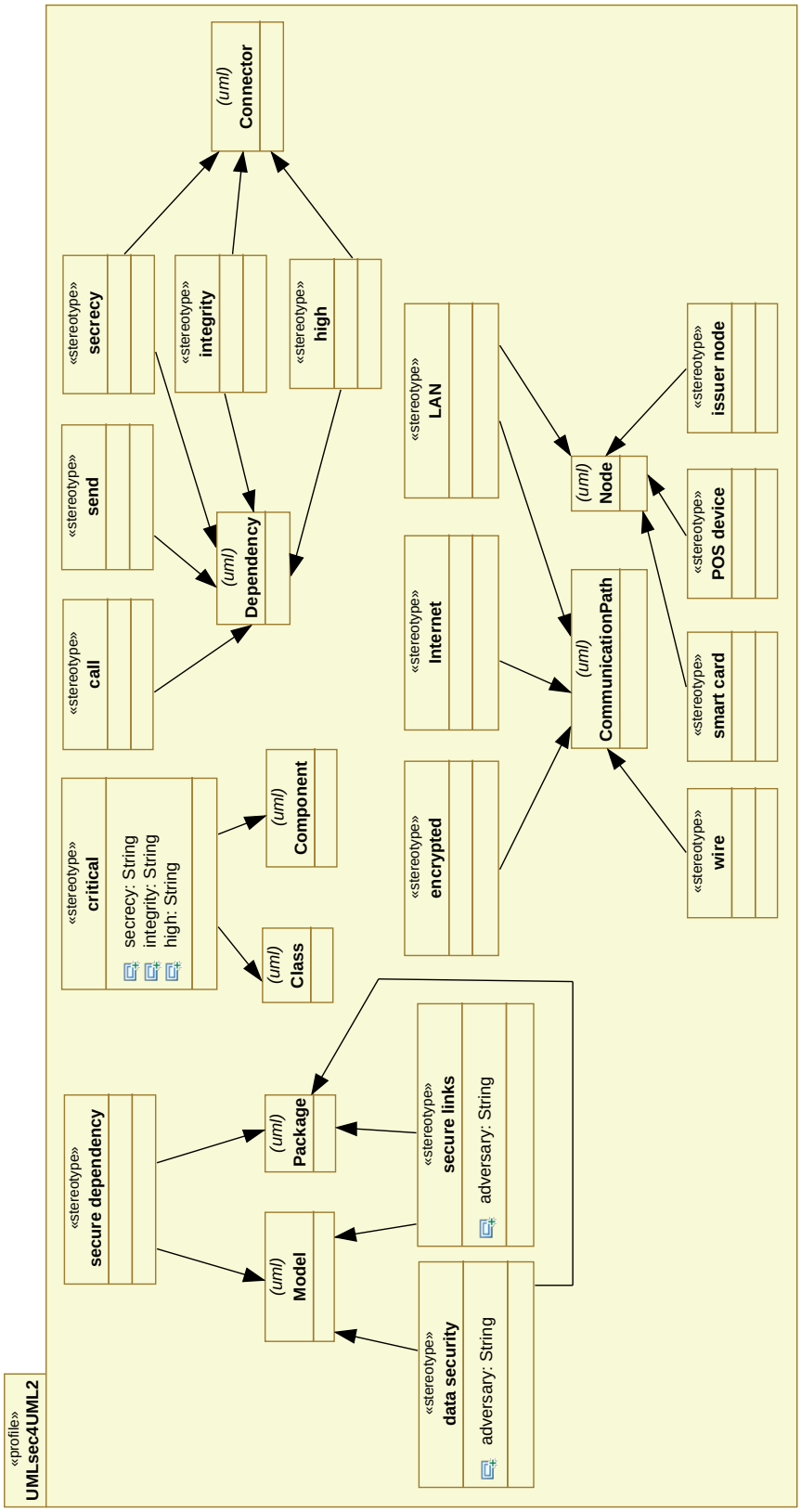

Figure 3: Profile Diagram of UMLsec4UML2-Profile

1. Developer uses a compatible UML editing tool to create a UML2.3 model.
2. Automatic verification of the OCL constraints directly within the UML editing tool; developer inspects text reports and possibly corrects model.
3. Usage of further UMLsec-Tools (as Eclipse-Plugins).
4. Developer inspects text reports and possibly corrects model.

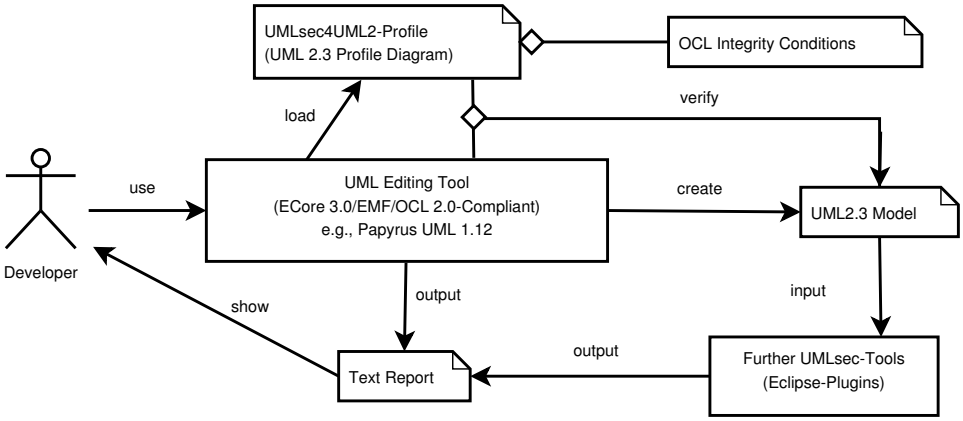

Figure 4: Workflow for using the UMLsec4UML2-Profile

In the following, we specify as examples the stereotypes `<<secure dependency>>` for class diagrams in Section 4.2.1 and `<<secure links>>` for deployment diagrams in Section 4.2.2.

4.2.1 Stereotype `<<secure dependency>>`

The `<<secure dependency>>` stereotype (Jürjens, 2005, pp. 59 ff.) is developed to label subsystems containing static structure diagrams (especially class diagrams). UML2.3 does not contain a metaclass subsystem. Instead, subsystems can be modeled as instances of the metaclass component stereotyped with `<<subsystem>>`. In the UMLsec4UML2-profile, the `<<secure dependency>>` extends the metaclasses Model and Package as depicted in Figure 3. This way, one can label arbitrary models with this stereotype, and one can make use of the advanced modeling possibilities for packages, e.g., package imports, merge, nesting, and so on.

Structure diagrams contained in a package might contain `<<call>>` and/or `<<send>>` dependencies between structural elements such as classes. Hence, the `<<call>>` and `<<send>>` stereotypes are modeled in Figure 3 as extensions of the metaclass Dependency. The source and target elements of a dependency can be be marked with the stereotype `<<critical>>` to express that some data that might be transmitted between source and target should fulfill certain security properties. Note that is possible to make use of `<<interface>>` classes, so that dependencies are connected to the interfaces. Then, classes that realize these interfaces are marked with the stereotype `<<critical>>`.

The stereotype `<<critical>>` has the tags {`secrecy`}, {`integrity`}, and {`high`}, which allow to specify security properties in more detail. Consequently, we introduce a stereotype `<<critical>>` in Figure 3 as an extension to the metaclasses class and component with the properties secrecy, integrity, and high of type String. The fact that this stereotype is an extension of the metaclass component allows one to apply it to component and composite structure diagrams, too.

In addition to the tag definitions, corresponding dependencies must be marked with the stereotypes `<<secrecy>>`, `<<integrity>>`, and/or `<<high>>`. So, these stereotypes are introduced in Figure 3 as extensions of the metaclass Dependency.

Note that the constraints associated with the `<<secure dependency>>` stereotype are explained in detail in Section 4.3.

An example of a class diagram contained in a package stereotyped `<<secure dependency>>` is shown in Figure 5. In this example, the class Class_1 is equipped with an explicit interface class Interface_0. It is also possible to omit explicit interface classes, and connect classes directly via dependencies.

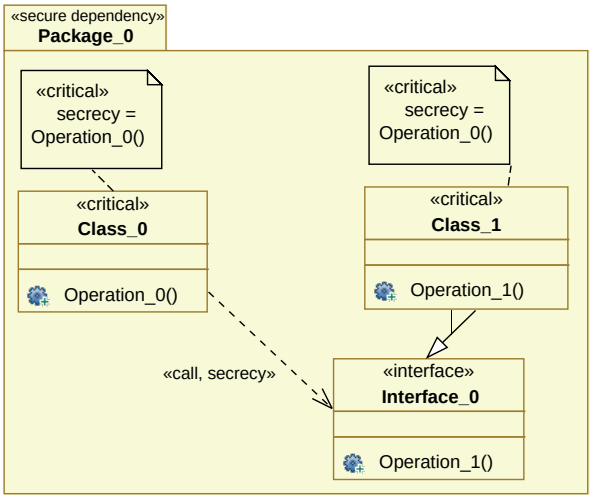

Figure 5: Class Diagram with Explicit Interface Class Contained in a Package Stereotyped `<<secure dependency>>`

4.2.2 Stereotype `<<secure links>>`

The `<<secure links>>` stereotype (Jürjens, 2005, p. 59) is developed to label subsystems containing deployment diagrams. In the UMLsec4UML2-profile, `<<secure links>>` extends the metaclasses Model and Package as depicted in Figure 3. Moreover, it is equipped with a tag adversary of type String, which can have the values default (Jürjens, 2005, p. 57) and insider (Jürjens, 2005, p. 58).

A deployment diagram contained in a package stereotyped `<<secure links>>` might contain communication paths, which can be marked `<<Internet>>`, `<<encrypted>>`, `<<LAN>>`, or `<<wire>>` (see Figure 1 for details). These stereotypes represent types of communication links, and they are associated with possible threats (Jürjens, 2005, pp. 57 ff.). Consequently, we introduce these stereotypes in Figure 3 as an extension to the metaclass Communication Path. Communication paths connect nodes, which can be marked `<<LAN>>`, `<<smart card>>`, `<<POS device>>`, or `<<issuer node>>`. These stereotypes represent node types, and similar to the stereotypes for communication paths, they are associated with possible threats (Jürjens, 2005, pp. 57 ff.). We introduce these stereotypes in Figure 3 as an extension to the metaclass Node. Based on an adversary tag and on stereotyped communication paths and nodes an attacker setting is defined in detail.

According to UML2.3, nodes can contain any element that can be contained in a package (not only components, as it is the case for UML1.5). Hence, dependencies (marked according to the `<<secure dependency>>` stereotype) between nodes or between parts, connectors, or classes contained in nodes are used to model security-critical call or send relationships.

Note that the constraints associated with the `<<secure links>>` stereotype are explained in detail in Section 4.3.

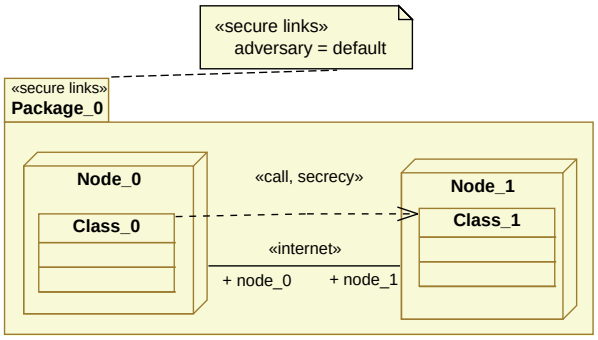

Figure 6: Deployment Diagram Contained in a Package Stereotyped `<<secure links>>`

An example of a deployment diagram contained in a package stereotyped `<<secure links>>` is shown in Figure 6. There, the dependency stereotyped `<<secrecy>>` connects classes contained in the nodes that communicate via a connection stereotyped `<<Internet>>`. The value of the adversary tag is default.

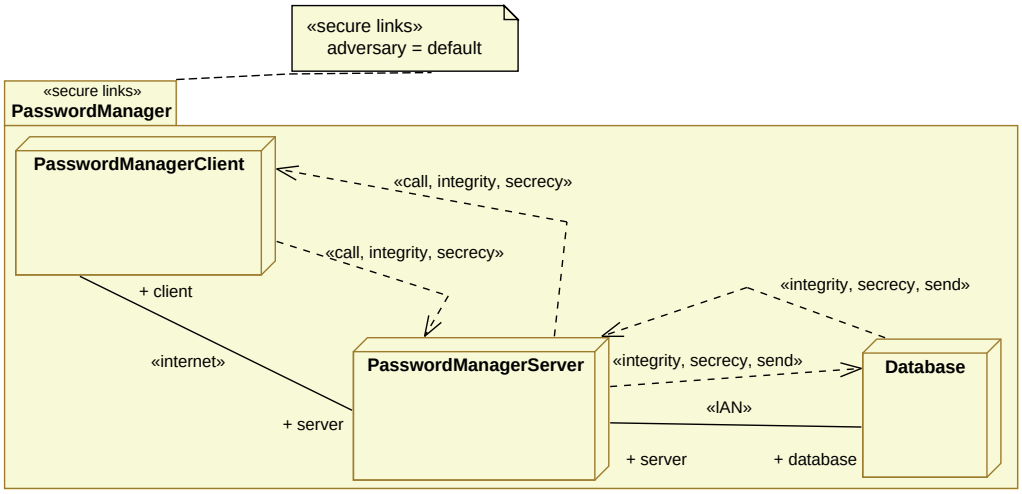

Figure 7: Deployment Diagram for Password Manager Example Contained in a Package Stereotyped `<<secure links>>`

A deployment diagram for the running example of a password manager as described in Section 3 is shown in Figure 7. The diagram is contained in a package stereotyped `<<secure links>>`, and it consists of the three nodes PasswordManagerClient, PasswordManagerServer, and Database. The communication path between the first two nodes is of type `<<Internet>>`, and the one between the last two nodes is of type `<<LAN>>`. Since the password manager should preserve the confidentiality and integrity of the data, i.e., passwords, usernames, etc., transmitted between the nodes, corresponding dependencies labeled `<<secrecy>>` and `<<integrity>>` are included.

4.3 OCL Integrity Conditions for UMLsec4UML2

The UMLsec4UML2-profile presented in Section 4.2 is enriched with integrity conditions denoted in *Object Constraint Language* (OCL) (UML Revision Task Force, 2006; Warmer & Kleppe, 2003). OCL is part of UML (UML Revision Task Force, 2010), and it is a notation to describe *constraints* on object-oriented modeling artifacts. A constraint is a restriction on one or more elements of an object-oriented model. In fact, the OCL constraints realize the functionality of a part of the tool support developed for the UMLsec-profile developed for UML1.5. More specifically, the static checks available in this tool are covered by the OCL constraints in the UMLsec4UML2-profile.

As an example, we present OCL integrity conditions for the `<<secure dependency>>` constraints. These integrity conditions are part of the UMLsec4UML2-profile. In the following, we explain the constraints informally, and we present the corresponding OCL integrity conditions:

Listing 1 For each `<<call>>` or `<<send>>` dependency connecting two classes contained in a package that is equipped with the stereotype `<<secure dependency>>` the source and target classes must be equipped with the stereotype `<<critical>>` if the dependency has the stereotype `<<secrecy>>`, `<<integrity>>`, or `<<high>>`.

Listing 2 If the dependency has the stereotype `<<secrecy>>`, then the source and target classes must have tags {`secrecy`} with equal tagged values.[1]

A UMLsec4UML2 model fulfills the `<<secure dependency>>` constraints as defined for UMLsec for UML1.5 if it fulfills the conjunction of the previously presented OCL integrity conditions.

[1]Note that similar constraints apply for the tags {`integrity`} and {`high`}.

```
(Package.allInstances()->select(p |
   (p.oclIsTypeOf(Package)) and (p.oclAsType(Package).getAppliedStereotypes().name->includes('
      secure dependency')))).allOwnedElements()->select(d |
         (d.oclIsTypeOf(Dependency)) and
         ((d.oclAsType(Dependency).getAppliedStereotypes().name->includes('call'))
            or (d.oclAsType(Dependency).getAppliedStereotypes().name->includes('send'))
         )->forAll(d |
            (d.oclAsType(Dependency).source->forAll(oclIsTypeOf(Class))) and
            (d.oclAsType(Dependency).target->forAll(oclIsTypeOf(Class))) and
            (d.oclAsType(Dependency).getAppliedStereotypes().name->includes('secrecy') and
             d.oclAsType(Dependency).source.getAppliedStereotypes().name->includes('critical') and
             d.oclAsType(Dependency).target.getAppliedStereotypes().name->includes('critical')) or
            (d.oclAsType(Dependency).getAppliedStereotypes().name->includes('integrity') and
             d.oclAsType(Dependency).source.getAppliedStereotypes().name->includes('critical') and
             d.oclAsType(Dependency).target.getAppliedStereotypes().name->includes('critical')) or
            (d.oclAsType(Dependency).getAppliedStereotypes().name->includes('high') and
             d.oclAsType(Dependency).source.getAppliedStereotypes().name->includes('critical') and
             d.oclAsType(Dependency).target.getAppliedStereotypes().name->includes('critical'))
            )
```

Listing 1: OCL Constraint: secureDependency

```
(Dependency.allInstances()->select(
   d | d.oclAsType(Dependency).getAppliedStereotypes().name->includes('secrecy'))->forAll(
      d_1 | (
         d_1.oclAsType(Dependency).source.getAppliedStereotypes()->any(
            s | s.oclAsType(Stereotype).name='critical').getValue(
               d_1.oclAsType(Dependency).source.getAppliedStereotypes()->any(
                  s | s.oclAsType(Stereotype).name='critical'),'secrecy')
            )
      = (
         d_1.oclAsType(Dependency).target.getAppliedStereotypes()->any(
            s | s.oclAsType(Stereotype).name='critical').getValue(
               d_1.oclAsType(Dependency).target.getAppliedStereotypes()->any(
                  s | s.oclAsType(Stereotype).name='critical'),'secrecy')
         )
      )
)
```

Listing 2: OCL Constraint: secrecyTaggedValuesAreEqual

The current version of this part of the UMLsec4UML2-profile has the following limitation: scenarios with `<<interface>>` classes are not yet covered, i.e., dependencies must connect `<<critical>>` classes directly.

In the following, we present OCL integrity conditions for the `<<secure links>>` constraints. These integrity conditions are part of the UMLsec4UML2-profile. In the following, we explain the constraints informally, and we present the corresponding OCL integrity conditions:

Listing 3 If there exists a package stereotyped `<<secure links>>` and the adversary tag has the value default, and if there exists a communication path stereotyped `<<Internet>>` between two nodes, then the existence of a dependency stereotyped `<<call>>` or `<<send>>` and `<<secrecy>>`, `<<integrity>>`, or `<<high>>` between the same two nodes is not allowed.

Listing 4 Communication paths are stereotyped `<<Internet>>`, `<<encrypted>>`, `<<LAN>>`, or `<<wire>>`, or they are marked with none of these stereotypes.

A UMLsec4UML2 model fulfills the `<<secure links>>` constraints as defined for UMLsec for UML1.5 if it fulfills the conjunction of the previously presented OCL integrity conditions.

In summary, the UMLsec4UML2-profile offers a convenient way to develop UMLsec models as described in Figure 4, and to automatically verify that these models fulfill the OCL integrity conditions included in the profile.

```
(Package.allInstances()->select(p |
   (p.oclIsTypeOf(Package)) and
   (p.oclAsType(Package).getAppliedStereotypes().name->includes('secure links')) and
   (p.oclAsType(Package).getValue(p.oclAsType(Package).getAppliedStereotype('UMLsec4UML2::secure
       links'),'adversary').oclAsType(String)='default')
      )
).allOwnedElements()->select(c |
   (c.oclIsTypeOf(CommunicationPath)) and
      (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('Internet') and)
   ).allOwnedElements()->select(p |
      (p.oclIsTypeOf(Property)) and
      ((p.oclAsType(Property).type.oclAsType(Node).clientDependency.oclAsType(Dependency).
          getAppliedStereotypes().name->includes('call')) or
       (p.oclAsType(Property).type.oclAsType(Node).clientDependency.oclAsType(Dependency).
          getAppliedStereotypes().name->includes('send'))) and
      ((p.oclAsType(Property).type.oclAsType(Node).clientDependency.oclAsType(Dependency).
          getAppliedStereotypes().name->includes('secrecy')) or
       (p.oclAsType(Property).type.oclAsType(Node).clientDependency.oclAsType(Dependency).
          getAppliedStereotypes().name->includes('integrity')) or
       (p.oclAsType(Property).type.oclAsType(Node).clientDependency.oclAsType(Dependency).
          getAppliedStereotypes().name->includes('high')))
      )->isEmpty()
```

Listing 3: OCL Constraint: secureLinksWithDefaultAttackerAndInternet

```
(Package.allInstances()->select(p |
   (p.oclIsTypeOf(Package)) and
   (p.oclAsType(Package).getAppliedStereotypes().name->includes('secure links'))
      )
).allOwnedElements()->forAll(c |
   (c.oclIsTypeOf(CommunicationPath)) and
   (((c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('Internet')) and
      not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('encrypted')) and
      not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('LAN')) and
      not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('wire')))
    or
    ((c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('encrypted')) and
      not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('Internet')) and
      not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('LAN')) and
      not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('wire')))
    or
    ((c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('LAN')) and
      not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('Internet')) and
      not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('encrypted')) and
      not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('wire')))
    or
    ((c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('wire')) and
      not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('Internet')) and
      not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('encrypted')) and
      not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('LAN')))
    or
    (not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('wire')) and
     not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('Internet')) and
     not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('encrypted')) and
     not (c.oclAsType(CommunicationPath).getAppliedStereotypes().name->includes('LAN')))
    )
)
```

Listing 4: OCL Constraint: communicationPathsStereotypedOnlyOnce

5 Specification of Architectural Security Patterns Using UMLsec4UML2

We presented a pattern- and component-based approach named SEPP (as introduced in Section 1) to construct secure software systems in (Schmidt, 2010) that especially deals with the early software development phases. SEPP makes use of *security problem frames* (SPF) and *concretized security problem frames* (CSPF), which constitute patterns for security requirements engineering. SPFs are patterns for structuring, characterizing, and analyzing problems that occur frequently in security engineering. CSPFs involve first solution approaches for the problems described by SPFs.

Then, the security requirements previously analyzed and specified are realized by *generic security architectures* (GSA), which constitute architectural patterns. They are related to CSPFs, and they

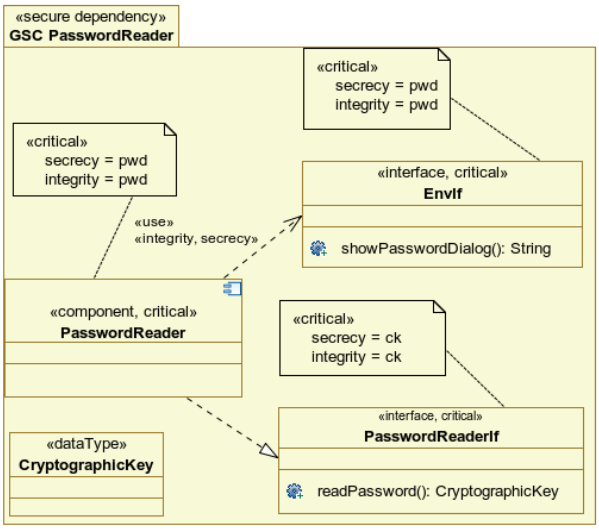

Figure 8: Class Diagram Stereotyped `<<secure dependency>>` to Define the Port Type PasswordReader

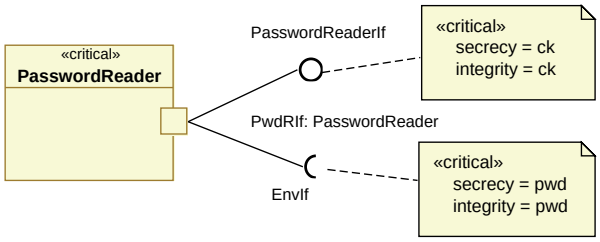

Figure 9: Composite Structure Diagram for the GSC PasswordReader Using the Port Type Defined in Figure 8

consist of *generic security components* (GSC) and *generic non-security components* (GNC). After a GSA is instantiated for each CSPF instance of a given software development problem, the different GSAs must be composed to obtain a global GSA. Finally, this global architecture is refined, and the result is a secure software product built from existing and/or tailor-made security components.

In (Schmidt, 2010, Part III), the GSAs are expressed using plain UML. In this paper, we present an approach to express GSAs using our UMLsec4UML2-profile, and to prepare the ground for tool support to check whether composed GSAs still fulfill the security requirements provided by the individual GSAs.

While UML1.5 has several disadvantages for the specification of software architectures compared to its successor UML2.3 as explained by Ivers et al. (2004); Pérez-Martínez and Sierra-Alonso (2004), UML2.3 can be used to model software architectures (Avgeriou, Guelfi, & Medvidovic, 2004; Björkander & Kobryn, 2003; Medvidovic, Rosenblum, Redmiles, & Robbins, 2002). For example, UML2.3 supports the concepts of *parts*, i.e., black-box components, and *connectors*.

As presented in (Schmidt, 2010, Part III), GSCs (similar to GSAs) are represented by composite structure diagrams and sequence diagrams. An example of a composite structure diagram for the GSC PasswordReader is shown in Figure 9. The port PwdRIf is typed PasswordReader as defined in Figure 8.

Both diagrams are stereotyped `<<secure dependency>>`[2] and the values of the {`secrecy`} and {`integrity`} tags refer to the password retrieved from the environment using interface EnvIf, and to the cryptographic key processed by the PasswordReader component that provides this cryptographic key to other components.

While the `<<secure dependency>>` constraint of the `<<use>>` dependency between the PasswordReader component and the EnvIf interface is fulfilled (see Figure 8), a similar statement about the {`secrecy`} and {`integrity`} tags of the `<<critical>>` stereotype of the PasswordReaderIf interface cannot be made

[2]Note that when using Papyrus UML, composite structure diagrams cannot be contained in packages. However, the model itself can be stereotyped as specified in the UMLsec4UML2-profile depicted in Figure 3.

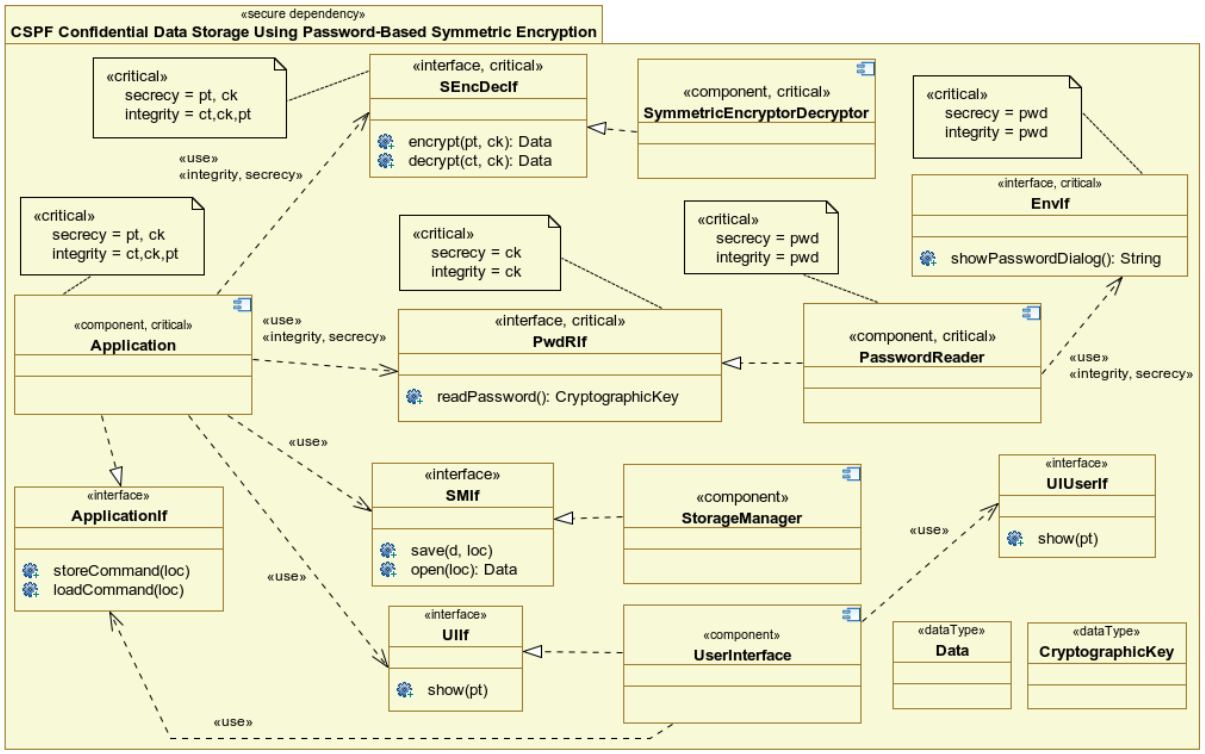

Figure 10: Port Type Definitions for GSA of CSPF Confidential Data Storage Using Password-Based Symmetric Encryption

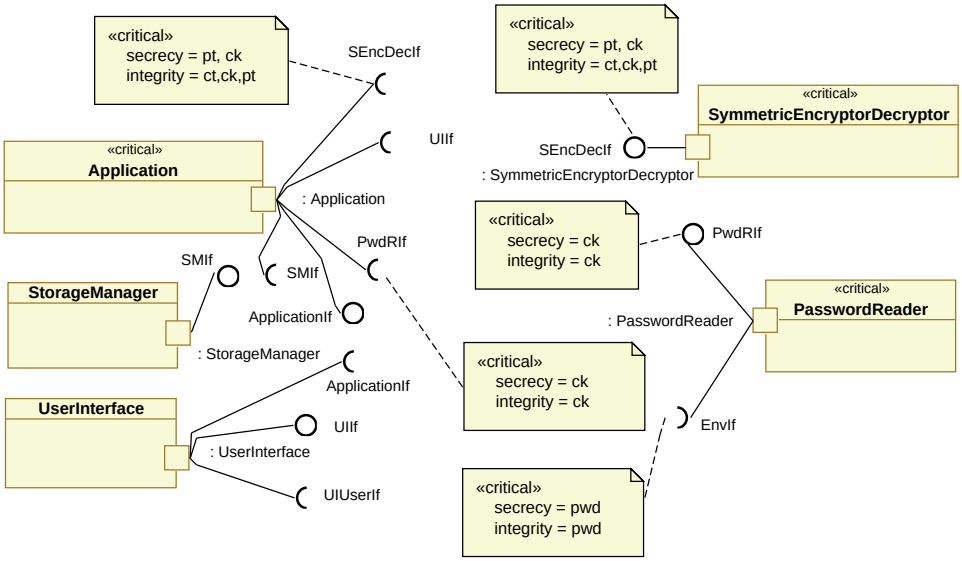

Figure 11: Structural View with Lollipop Notation of GSA for CSPF Confidential Data Storage Using Password-Based Symmetric Encryption

(see Figure 8). To check the `<<secure dependency>>` constraint, the component by which the PasswordReaderIf interface is used must be known and analyzed.

Figures 10 and 11 show the structural view of a GSA presented in (Schmidt, 2010, Part III) related to the problem class of confidential data storage using password-based symmetric encryption. This GSA makes use of the previously presented GSC PasswordReader.

The GSA is contained in a package stereotyped `<<secure dependency>>`. According to the original

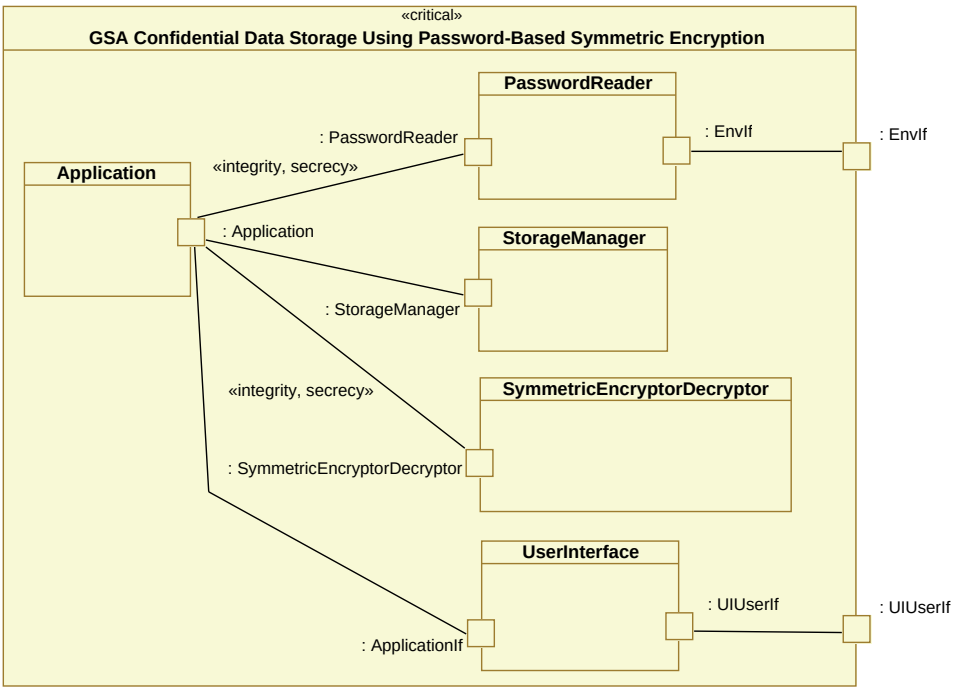

Figure 12: Structural View with Connectors of GSA for CSPF Confidential Data Storage Using Password-Based Symmetric Encryption

UMLsec for UML1.5, this stereotype refers to dependencies stereotyped `<<call>>` or `<<send>>`. Composite structure diagrams, which we use to model the structural views of GSAs, contain dependencies stereotyped `<<use>>`[3] to specify that components make use of interfaces of other components contained in an architecture. For example, the component Application in Figure 10 depends on the components SymmetricEncryptorDecryptor, PasswordReader, and StorageManager. It makes use of the interfaces realized by these components. Hence, the UMLsec4UML2-profile is extended to additionally cover `<<use>>` dependencies. The constraint expressed by the `<<secure dependency>>` stereotype is basically unchanged: the values of the {`secrecy`}, {`integrity`}, and/or {`high`} tags of the `<<critical>>` stereotypes of two components that are connected via a `<<use>>` dependency stereotyped `<<secrecy>>`, stereointegrity, and/or `<<high>>` should be equal. In our example in Figure 10 all `<<use>>` dependencies are stereotyped `<<secrecy>>` and `<<integrity>>`, and the tagged values of the `<<critical>>` stereotypes of the involved components are equal. Hence, the `<<secure dependency>>` constraint is fulfilled.

Figures 13 and 14 show the architecture of the node PasswordManagerClient in the deployment diagram in Figure 7. In this software architecture, several GSAs are composed.

6 Conclusion

We presented in this paper the new UMLsec4UML2-profile, which integrates modeling and verification activities. UMLsec4UML2 makes the whole expressiveness of UML2.3 available for architectural security modeling. We validated and discussed the approach presented in this paper using the sample development of a password management software.

[3] In contrast to the stereotypes `<<call>>` and `<<send>>`, the stereotype `<<use>>` is predefined in the UML2.3 metamodel.

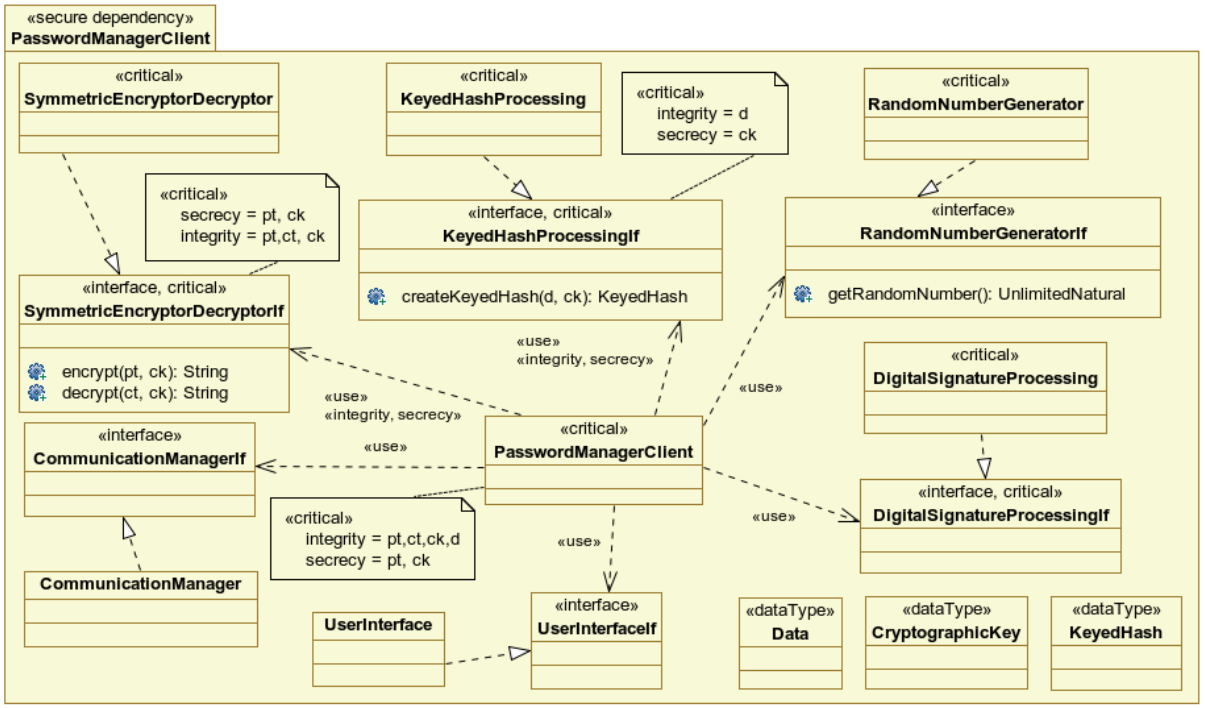

Figure 13: Class Diagram for Password Manager Example Stereotyped `<<secure dependency>>` to Define Port Types

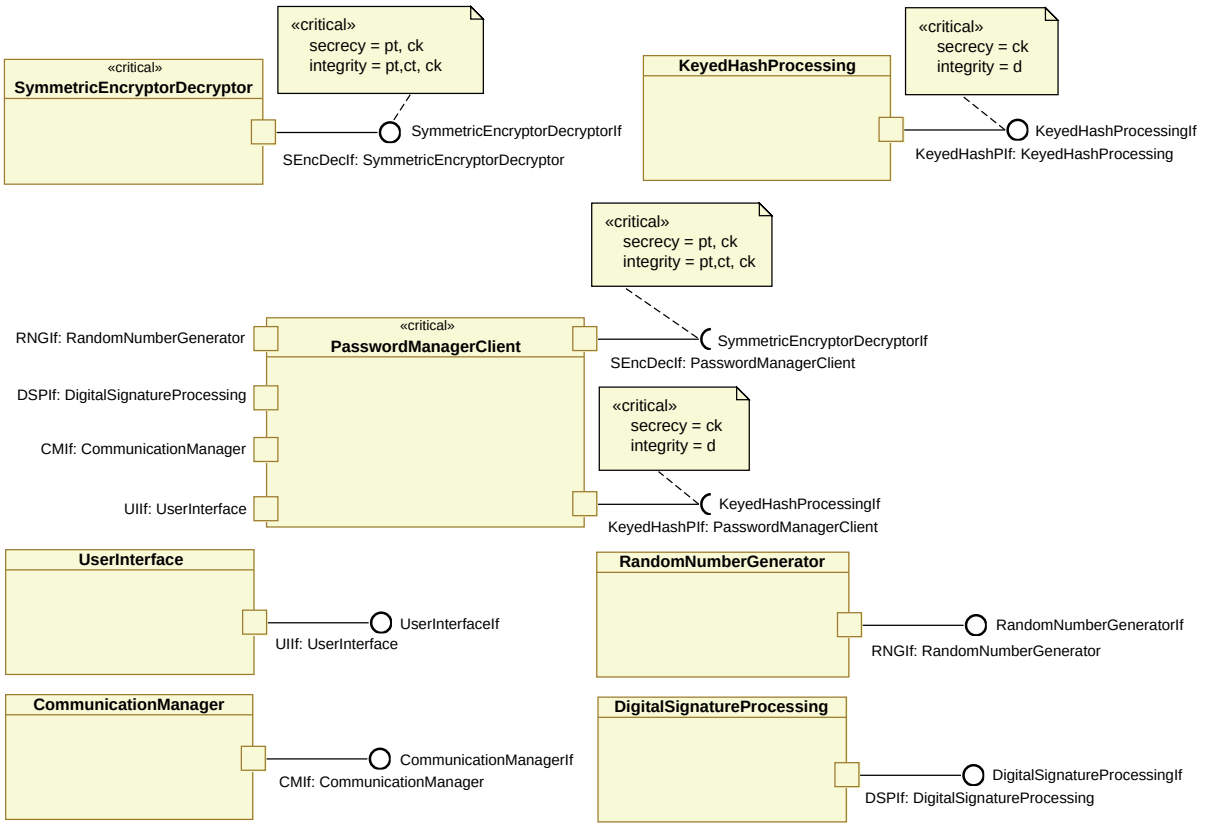

Figure 14: Composite Structure Diagram for Password Manager Example Stereotyped `<<secure dependency>>` Using the Port Types Defined in Figure 13

References

Avgeriou, P., Guelfi, N., & Medvidovic, N. (2004). Software architecture description and UML. In *UML satellite activities* (pp. 23–32). Springer.

Björkander, M., & Kobryn, C. (2003). Architecting systems with UML 2.0. *IEEE Software*, *20*(4), 57–61.

Deshmukh, M., Kawana, P., & Erkulla, S. (2010). *Development of a web-based password manager using SEPP.* (Report of student project at University Duisburg-Essen, Germany)

Eclipse - An Open Development Platform. (2010, June). (`http://www.eclipse.org/`)

Ferraiolo, D. F., & Kuhn, D. R. (1992). Role-based access control. In *Proceedings of the national computer security conference* (pp. 554–563).

Ivers, J., Clements, P., Garlan, D., Nord, R., Schmerl, B., & Silva, J. R. O. (2004). *Documenting component and connector views with UML 2.0* (Tech. Rep. No. CMU/SEI-2004-TR-008). Carnegie Mellon Software Engineering Institute.

Jürjens, J. (2005). *Secure systems development with UML.* Springer.

MagicDraw UML 16.8. (2010, June). (`http://www.magicdraw.com`)

Medvidovic, N., Rosenblum, D. S., Redmiles, D. F., & Robbins, J. E. (2002). Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineering and Methodology*, *11*(1), 2–57.

Model Development Tools Project (MDT). (2010, June). (`http://www.eclipse.org/modeling/mdt/`)

MOSKitt – MOdeling Software Kitt. (2010, June). (`http://www.moskitt.org`)

Object Management Group (OMG). (2009, August). OMG business process model and notation (bpmn) [Computer software manual]. (`http://www.omg.org/cgi-bin/doc?dtc/09-08-14.pdf`)

Papyrus UML 1.12. (2010, June). (`http://www.papyrusuml.org`)

Pérez-Martínez, J. E., & Sierra-Alonso, A. (2004). UML 1.4 versus UML 2.0 as languages to describe software architectures. In *Proceedings of the european workshop on software architectures (EWSA)* (pp. 88–102). Springer.

Schmidt, H. (2010). *A pattern- and component-based method to develop secure software.* Deutscher Wissenschafts-Verlag (DWV) Baden-Baden. (Online version: `http://www.mathomhouse.de/phdthesis.html`)

SPASS Theorem Prover 3.5. (2010, June). (`http://www.spass-prover.org/`)

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2009). *EMF: Eclipse modeling framework* (Vol. Second Edition). Addison-Wesley.

SUN Java 6 Standard Edition. (2010, June). (`http://java.sun.com/javase/6/docs/api/`)

Topcased 3.4.1. (2010, June). (`http://www.topcased.org`)

UML Revision Task Force. (2006, May). Object constraint language specification [Computer software manual]. (`http://www.omg.org/docs/formal/06-05-01.pdf`)

UML Revision Task Force. (2010, May). OMG unified modeling language: Superstructure [Computer software manual]. (`http://www.omg.org/spec/UML/2.3/Superstructure/PDF/`)

Warmer, J., & Kleppe, A. (2003). *The object constraint language 2.0: Getting your models ready for MDA* (2nd ed.). Pearson Education.

XML - Extensible Markup Language. (2010, June). (`http://www.w3.org/XML/`)