Security patterns modeling and formalization for pattern-based development of secure software systems

B. Hamid, S. Gürgens and A. Fuchs

Abstract Pattern-based development of software systems has gained more attention recently by addressing new challenges such as security and dependability. However, there are still gaps in existing modeling languages and/or formalisms dedicated to modeling design patterns and the way how to reuse them in the automation of software development. The solution envisaged here is based on combining metamodeling techniques and formal methods to represent security patterns at two levels of abstraction to fostering reuse. The goal of the paper is to advance the state of the art in model and pattern-based security for software and systems engineering in three relevant areas: (1) develop a modeling language to support the definition of security patterns using metamodeling techniques; (2) provide a formal representation and its associated validation mechanisms for the verification of security properties; and (3) derive a set of guidelines for the modeling of security patterns within the integration of these two kinds of representations.

Keywords Secure software systems, Trust, Security, Pattern, Meta-model, Model Driven Engineering, Formal modeling.

B. Hamid
IRIT, University of Toulosue
118 Route de Narbonne, 31062 Toulouse Cedex 9, France
Tel: +33 (0)5 6150 2386
Fax: +33 (0)5 6150 4173
E-mail: brahim.hamid@irit.fr

S. Gürgens
Fraunhofer Institute for Secure Information Technology SIT
75, 64295 Darmstadt, Germany
E-mail: : sigrid.guergens@sit.fraunhofer.de

A. Fuchs
Fraunhofer Institute for Secure Information Technology SIT
75, 64295 Darmstadt, Germany
E-mail: andreas.fuchs@sit.fraunhofer.de

1 Introduction

1.1 Motivation and background

During the last decades, the systems have grown in terms of complexity and connectivity. In the past security was not such a critical concern of system development teams, since it was possible to rely on the fact that a system could be easily controlled due to its limited connectivity and, in most of the cases, its dedicated focus. However, nowadays, systems are growing in terms of complexity, functionality and connectivity. The aforementioned challenges in modern system development push the Information and Communication Technologies (ICT) community to search for innovative methods and tools for serving these new needs and objectives. Regarding system security, in the cases of modern systems, the "walled-garden" paradigm is unsuitable and the traditional security concepts are ineffective, since they are based on the fact that it is possible to build a wall between the system and the outer world.

Application developers usually do not have expertise in security. Hence capturing and providing this expertise by way of security patterns [1,2] has become an area of research in the last years. Security patterns shall enable the development of secure and dependable applications while at the same time liberating the developer from having to deal with the technical details. On the other hand, Model Driven Engineering (MDE) [3,4] provides a very useful contribution for the design of secure and trusted systems: It allows to reduce time/cost of understanding and analyzing system artefacts description due to the abstraction mechanisms, and it reduces the cost of the development process; thanks to the generation mechanisms. Hence security pattern integration has to be considered at some point in the MDE process.

When security requirements are determined, architecture and design activities are conducted using modeling-techniques and tools for higher quality and seamless development. Often, formal modeling is used as a complementary approach when verification is needed. On the other hand, the integration of security features using this approach requires high expertise in developing the architecture and the design and demands both application domain-specific knowledge and security expertise. Hence capturing and providing this expertise by means of security patterns can enhance systems development by integrating them in the different development life-cycle stages [5,6].

Security solutions can be described as security patterns, and the use of these patterns results in products that are already established in the respective domains, usually in the form of COTS components. Patterns define best practices and their idea is to be reused and thus to support help designers reuse them in new designs. Security requirements are specified independently from patterns and technological products at a very early stage of system design and then refined using the system model until they can be matched with existing security patterns. The main problem for the developer is to select and connect the security patterns to be used with the rest of the application. Stating more precisely some aspects of the pattern description would make these aspects more convenient. Even if a pattern requires tailoring, starting from a precise description facilitates its selection and application.

Security patterns are still not widely adopted in system and software security engineering due to the weakness/lack of appropriate methods and tools to assist the designer/developer to use them during the system engineering life cycle. The application and composition of patterns are still achieved as an ad-hoc process (eg., search, select, apply, validate). A security pattern targets some particular properties that characterize it, and the integration, composition and application of a security pattern should maintain these properties. A rigorous treatment of security properties needs to be based on clear formal semantics that enable system developers to precisely specify security requirements. Thus, the precise but flexible specification and description of security patterns are pre-requisites to their successful integration and composition for their application.

In this paper, we propose a new design framework for the specification and the validation of security patterns intended for software development automation in the context of systems with stringent security requirements. In our vision, a security pattern is a subsystem exposing pattern functionalities through interfaces and targeting security properties. The associated pattern design framework provides two additional and complementary types of representations: a semi-formal representation through metamodeling techniques and a rigorous formal representation through a prefix-closed formal language. We explain in particular how the formalization and the validation help the specification of security patterns and the specification of guidelines for their correct usage. In addition, we discuss the integration of pattern definition and application development processes towards a correct-by-construction Pattern-Based system and software Security Engineering (PBSE) approach. The proposed integrated modeling and formal framework is illustrated through an example of a secure communication pattern.

To provide a concrete example, we will investigate a case study from Intelligent Transport Systems (ITS). In particular, electric vehicles provide new interesting use cases related to charging and reverse charging. Indeed, due to the connection with smart grids, the full infrastructure has to be adapted to arising new needs.

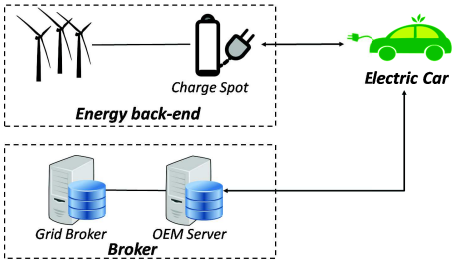

Fig. 1 Interconnection between an Electric Vehicle and a Smart Grid

The case study chosen in this paper and presented in Fig. 1 illustrates the different connections within a smart grid related to cars. The grid can sell energy to a car (i.e., charging mode) but can also buy energy from the car (i.e., reverse charging mode). To this end, data such as prices, parking time slot or banking information must be exchanged between the car and the grid through brokers. Hence the communication channels between the assets must be trustworthy, i.e. satisfy strong requirements in terms of authenticity and confidentiality. In the Intelligent

Transport Systems (ITS) domain, the ISO/IEC 15118 highly recommends to use TLS [7] for ensuring security properties.

1.2 Intended contribution

The basis formulation of the approach presented in this article has been previously published in a research paper at the International ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS'11) [8]. This work extends ideas described in this earlier paper and presents a holistic approach for the design, formalization and validation and integration of the modeling and formal approach for the definition of security patterns. Specifically, we provide a more comprehensive and complete description of our approach. The work presented in this paper has the following aspects:

- Modeling: We propose a semi-formal modeling framework using MDE techniques to specify patterns for security properties and constraints independently from end-development applications and execution platforms.
- Verification: For security pattern verification, we apply the techniques for formally proving security properties of systems provided by the Security Modeling Framework SeMF developed by Fraunhofer SIT, following the abstraction levels of the application development process.
- Integration: We derive a set of guidelines for the correct usage of security patterns using the semi-formal representation of the pattern and its associated proof in tandem.

1.3 Organization of the contribution

The rest of this paper is organized as follows. In Section 2, we review related works addressing pattern specification and validation. An overview our modeling approach including a set of definitions is presented in Section 3. Then, Section 4 presents in detail the pattern modeling language and illustrates the pattern modeling process in practice. Section 5 presents the formal validation process, and Section 6 shows its application to a specific domain-independent and a domain-dependent pattern, respectively. Then, Section 7 presents the integrated framework and the process for security patterns definition. In Section 8, we discuss our contribution and explain in particular how the formalization helps the specification of security patterns and their usage as building blocks for secure system and software engineering. Finally, Section 9 concludes this paper with a short discussion about future works. We investigate some open issues, mainly the issues of generalization and implementation including the usability of the proposed modeling framework.

2 Related work

Design patterns constitute a solution model to generic design problems, applicable in specific contexts. Several tentatives exist in the security design pattern literature [1,9–13]. They allow to solve very general problems that appear frequently

as sub-tasks in the design of systems with security and dependability requirements. These elementary tasks include secure communication, fault tolerance, etc. Particularly, [1] presented a collection of patterns to be used when dealing with application security. [11] describes a hybrid set of patterns to be used in the development of fault-tolerant software applications. An extension to the framework presented in [11] for the development of dependable software systems based on a pattern approach is proposed in [12]. The pattern specification consists of a service-based architectural design and deployment restrictions in form of UML deployment diagrams for the different architectural services.

2.1 Pattern modeling approach -abstraction

To give a flavor of the improvement achievable by using specific languages, we look at the pattern formalization problem. *UMLAUT* [14] is an approach that aims to formally model design patterns by proposing extensions to the UML meta model 1.3. They used OCL language to describe constraints (structural and behavioral) in the form of meta collaboration diagrams. In the same way, *Role-Based Meta modeling Language (RBML)* [15] is able to capture various design perspectives of patterns such as static structure, interactions, and state-based behavior. The framework *LePUS* [16] offers a formal and visual language for specifying design patterns. It defines a pattern in an accurate and complete form of formula with a graphical representation. A diagram in LePUS is a graph whose nodes correspond to variables and whose arcs are labeled with binary relations.

While many security patterns have been designed, still few works propose general development techniques for security patterns. A survey of approaches of security patterns is proposed in [10]. For the first approach of this kind [17], design patterns are usually represented by diagrams with notations such as UML object, annotated with textual descriptions and examples of code. There are some well-proven approaches [18] based on Gamma et al. However, this kind of techniques does not allow to reach the high degree of pattern structure flexibility which is required to achieve our target. The framework promoted by LePUS [16] is interesting but the degree of expressiveness proposed to design a pattern is too restrictive.

2.2 Pattern composition and application

With regard to the usage of security patterns in secure software systems development, the *Design Pattern Modeling Language (DPML)* [19] allows the incorporation of patterns in UML class models. In [20], the authors explained how pattern integration can be achieved by using a library of precisely described and formally verified solutions. Another attempt has been made in [21] which creates a meta-model for both the problem and the design pattern. Then, a mapping between the two models enables to create an integrated model using model transformations. The work of [9] reports an empirical experience regarding the adoption and elicitation of security patterns in the Air Traffic Management (ATM) domain, and shows the power of using patterns as a guidance to structure the analysis of operational aspects when they are used at the design stage. In [22], the authors

introduced an approach for the composition of web services using so-called secure service orchestration (SESO) patterns. These patterns express primitive (e.g., sequential, parallel) service orchestrations, which are proven to have certain global security properties if the individual services participating in them have themselves other security properties. Hence SESO patterns determine the criteria (security, interface and functional) that should be satisfied by the services that could instantiate them. These criteria are used to drive a discovery process through which the pattern can be instantiated.

2.3 Model-driven security engineering

With regard to the modeling of security and dependability in model-driven development, UMLsec [23], SecureUML [24] and [25], to name a few, and our proposal are not in competition but they complement each other by providing different views to a secure information system. The importance of models and MDE in security engineering has been highlighted in [26–28] and confirmed recently in [29]. The modeling of basic concepts related to security and software architecture requirements are established and well-know. There are several security risk analysis methods that exist such as EBIOS [30], CORAS [31] and security-HAZOP [32]. In these methods, a threat and a risk analysis is executed using methods like the threat modeling with attack trees as described in [33]. The output of these methods is a set of recommendations and guidelines to detect possible risks, evaluate them and then mitigate them.

2.4 Formal methods for security engineering

In system and software engineering formal methods are used for the precise specification of the modeling artefacts across the development life cycle for validation purposes [34], particularly in the development of security critical systems [35, 36]. A number of works have considered the composition of security policies and developed solutions for particular areas like access control or Web services [37–39]. Formal methods rely on mathematically rigorous procedures to search through the possible execution paths of a model for test cases and counterexamples. Regarding the verification of security properties, early work discusses the verification of cryptographic protocols and is based on an abstract (term-based) representation of cryptographic primitives that can be automatically verified using model checking and theorem proving tools. One research line in this category is authentication logics, the first of these logics being the BAN Logic [40]. The Inductive Approach by Paulson [41] started another research line. Early work in the area of model checking can be traced back to [42], see [43] for a survey. One of the more recent approaches is AVISPA [44] which provides the High Level Protocol Specification Language (HLPSL [45]) and four different analysis tools. Another approach [46] has been used to find security flaws in a number of key exchange, authentication and non-repudiation protocols and has more recently been applied to analyze certain scenarios based on Trusted Computing [47]. To the best of our knowledge, none of the above-described approaches is able to integrate the security solution validation into the MDE refinement process of the application. In contrast, in [48],

a first approach of our validation method was conducted, using it to prove certain security properties being provided by the architecture for automotive on-board networks of the project EVITA.

2.5 Positioning

To summarize, in software engineering, design patterns are considered effective tools for the reuse of specific knowledge. However, a gap between the development of systems using patterns and the pattern information still exists. This becomes even more visible when dealing with specific concerns namely security and dependability for several application sectors. A modeling framework that brings semi-formal and rigorous representation concepts all together in a unified manner, to best of our knowledge, does not exist yet.

In concept, our modeling framework is similar to the one proposed in [20]. Nevertheless they used a rigid structure (a pattern is defined as a quadruplet) and consequently their approach is not usable to capture specific characteristics of patterns for several domains. Although we found similarities between the approach in [21] and ours, we want to go further than the transformation by defining a full process for a proven integration, and be able, within this defined process, to leave the user free to alter the automatic result, while always checking the correctness at the end.

In contrast to other formal security engineering methods, the used formal security framework, referred to as SeMF [49], is not following the attack nor the risk based approaches. Its basis are a set of desired security properties and associated assumptions. With SeMF it is possible to validate if properties like trust, authenticity or confidentiality hold under given assumptions. The side benefit is case a stated assumption does not hold is that possible consequences in regard to security properties can be estimated. The proof itself is conducted mostly with pencil and paper and the resulted proof artifacts will be utilized by the designer as input to the pattern-based development process. See Section 5.1 for a short introduction to SeMF.

From a different point of view, we agree with the argumentations given in [50] to justify why the precise specification and formalization of a pattern by definition restricts its "degree of freedom for the design", and hence there are no success stories of works dealing with pattern development. This is not only related to security patterns. Note however, that these works do not address the validation activity which is an important issue in any design activity and more particularly in security engineering. We claim that security is subject to rigorous and precise specification and the proposed literature (to the best of our knowledge) fails to meet these two objectives. To remedy these contradictory needs, we support the specifications of security patterns at two levels of abstractions, domain independent and domain specific, in both a semi-formal and formal representation through metamodeling techniques and a rigorous formal representation through a formal approach. This allows to support some variability of the pattern.

3 Conceptual model

In this section, we present our conceptual model. The goal of this model is to illustrate the addressed problems and to have a common understanding of all the concepts used in this paper.

In software engineering, separation of concerns promotes the separation of general-purpose services from implementations. In our context, we target the separation of the general purpose of a pattern from the mechanisms used to implement it. This is an important issue to understand the use of patterns in the scope of security engineering. The layer in which patterns and their related mechanisms are integrated depends on the assurance a client has in the services of other concerned layers. We begin with a motivating example.

3.1 Informal description of the motivating example

Since messages passing across any public network can be intercepted and manipulated, the problem we address in our example is how to ensure that the data is secure in transit, in particular how to guarantee data authenticity. We show the feasibility of our approach through the example of Secure Communication Pattern (SCP). On domain-independent level, this pattern uses abstract send and receive actions and abstract communication channels that are assumed to provide authenticity. However, on domain-specific level, SCPs are slightly different with regard to the application domain. A system domain may have its own mechanisms and means, protocols that can be used to implement this pattern range from SSL, TLS, Kerberos, IPSec, SSH, to WS-Security. In summary, they are similar in the goal, but different in the implementation issues. This is the motivation to handle the modeling of security patterns by following abstraction. As an example, on the domain-specific level we use the TLS mechanism [7] as a concrete implementation of the SCP.

The TLS mechanism is composed of two phases: The *TLS Handshake* that establishes a secure channel, and the *TLS Record* in which this channel can be used to exchange data securely. The client initiates the TLS handshake by providing the server with a random number and information about the cryptographic algorithms it can handle. The server replies by choosing the actual algorithm to use, optionally requiring the client to authenticate itself, and by sending a random number of its own and its certificate issued by some Certification Authority trusted by both the server and the client.

For authenticating itself, in the final handshake message the client includes its own certificate, a signature on all three handshake messages generated with the client's private key, and a third random number encrypted using the server's public key contained in the server's certificate. After having verified the certificates and signature, both client and server use the exchanged random numbers to generate session keys for generating and verifying message authentication codes (MACs) and for encrypting and decrypting messages during the TLS record phase.

Since the key used by the client for generating a MAC/encrypting a message is used by the server only for MAC verification/decryption and vice versa, and since these keys are based on one random number confidential for the client and the

server, the keys establish a channel that provides authenticity and confidentiality for both client and server.

3.2 Definitions and concepts

Security patterns are not only defined from a platform independent viewpoint (i.e., they are independent from the implementation), they are also expressed in a way consistent with domain-specific models. Particularly a security pattern at domain-independent level exhibits an abstract solution without specific knowledge on how the solution is implemented with regard to the application domain. The objective is to reuse the generic model artifacts for several different industrial application domain sectors and also to enable their customization regarding domain-specific knowledge and requirements to produce the respective domain-specific artifacts. Thus, the question of how to support these concepts should be captured in the specification languages.

Definition 1 (Domain) A domain is a field or a scope of knowledge or activity that is characterized by the concerns, methods, mechanisms, etc., employed in the development of a system. The actual clustering into domains depends on the given group/community implementing the target methodology.

In our context, a domain may include knowledge about protocols, processes, methods, techniques, practices, OS, HW systems, measurement and certification related to the specific domain. For example, in the group of Safety Standards, IEC 61508 is a domain-independent standard and EN 50126, EN 81 and ISO 26262 are a railway domain-specific standard, an elevator domain standard and an automotive domain standard, respectively.

To specify security patterns, we build on a metamodel for representing these patterns in the form of a subsystem providing appropriate interfaces and targeting security properties to enforce the security system requirements. The so-called **external interfaces** will be used to make the pattern's functionality available to the application, while the **technical interface** supports interactions with security primitives and protocols of the application domain, including HW platforms. We capture the security capabilities of the pattern through a novel concept called **Property**.

The proposition presented in this paper is based on a Model-Driven Engineering (MDE) approach and on three levels of abstraction: (i) Pattern Specification Metamodel (SEPM), (ii) Domain-Independent Pattern Model (DIPM) and (iii) Domain-Specific Pattern Model (DSPM). This decomposition aims at allowing the design of multi-concerns applications in the context of safety by avoiding the great complexity normally introduced when combining extra-functional concerns and domain-specific artifacts, and targets overcoming the lack of formalism of the classical textual pattern form.

4 Pattern specification metamodel (SEPM)

The System and Software Engineering Pattern Metamodel (SEPM), as depicted in Fig. 2, defines a new formalism for describing patterns and constitutes the base of

our pattern modeling language. Such a formalism describes all the concepts (and their relations) required to capture all the facets of patterns. These patterns are specified by means of a domain-independent generic representation and a domain-specific representation.

Our representation of both domain-independent and domain-specific patterns is based on GoF template style [17] (informal representation), but extends this approach to fit with the security and dependability needs. We keep the template elements in the form of attributes and deeply refine them by defining new concepts in order to fit with the aforementioned needs.

The principle classes of the metamodel and their links with the property models are described with a UML class diagram in Fig. 2. For the semi-formal representation one may use the property and constraint modeling languages presented in [51, 8], or equivalent [52,53]. These approaches use security property models as model libraries to define the security properties and constraints of the pattern. In this paper we focus on a formal representation of security properties based on a rigorous semantics in order to enable precise specification and validation processes. In the following, we depict in more details the meaning of principle concepts used to rewrite a pattern by means of the SEPM language.

- SepmPattern. An SepmPattern is a subsystem describing a *solution* for a particular recurring security design *problem* that arises in a specific design *context*. As we shall see, we keep these descriptions as attributes of the subsystem in the form of textual data. Additionally we derived a set of concepts to support automatic pattern specification and to ease pattern classification, identification and validation.
- placeHolderAppEntity. It represents the application element to be used during the pattern integration-application in the designs. In other words, such a pattern element will be replaced with one element from the application design, or created if it exists in the pattern but not in the application. Moreover, it will be used to reason about the pattern properties and its provided design solution.
- SepmDIPattern. This is an SepmPattern denoting some abstract representation of a security pattern at domain-independent level. This is the key entry artifact to model pattern at domain *independent* level (DIPM). The behavior of an SepmDIPattern is defined in terms of provided and required interfaces. An SepmDIPattern serves as a type whose conformance is defined by these *interfaces*. Larger pieces of a system's functionality may be assembled by integrating patterns into an encompassing pattern or an assembly of patterns. An SepmDIPattern may be manifested by one or more artifacts which in turn may be deployed in their execution environment.
- externalInterface. A subsystem (pattern) provides/requires appropriate interfaces to exhibit pattern functionality in order to manage its application. An SepmDIPattern interacts with its environment (application) with externalInterfaces.
- SepmProperty. A SepmProperty is a particular characteristic of a pattern related to the concern the pattern is dealing with and dedicated to capture its intent in a certain way. ecurity and dependability properties for example are SepmProperties. The concept is used to describe the security aspects of the subsystem to enforce the security system requirements. An SepmProperty is

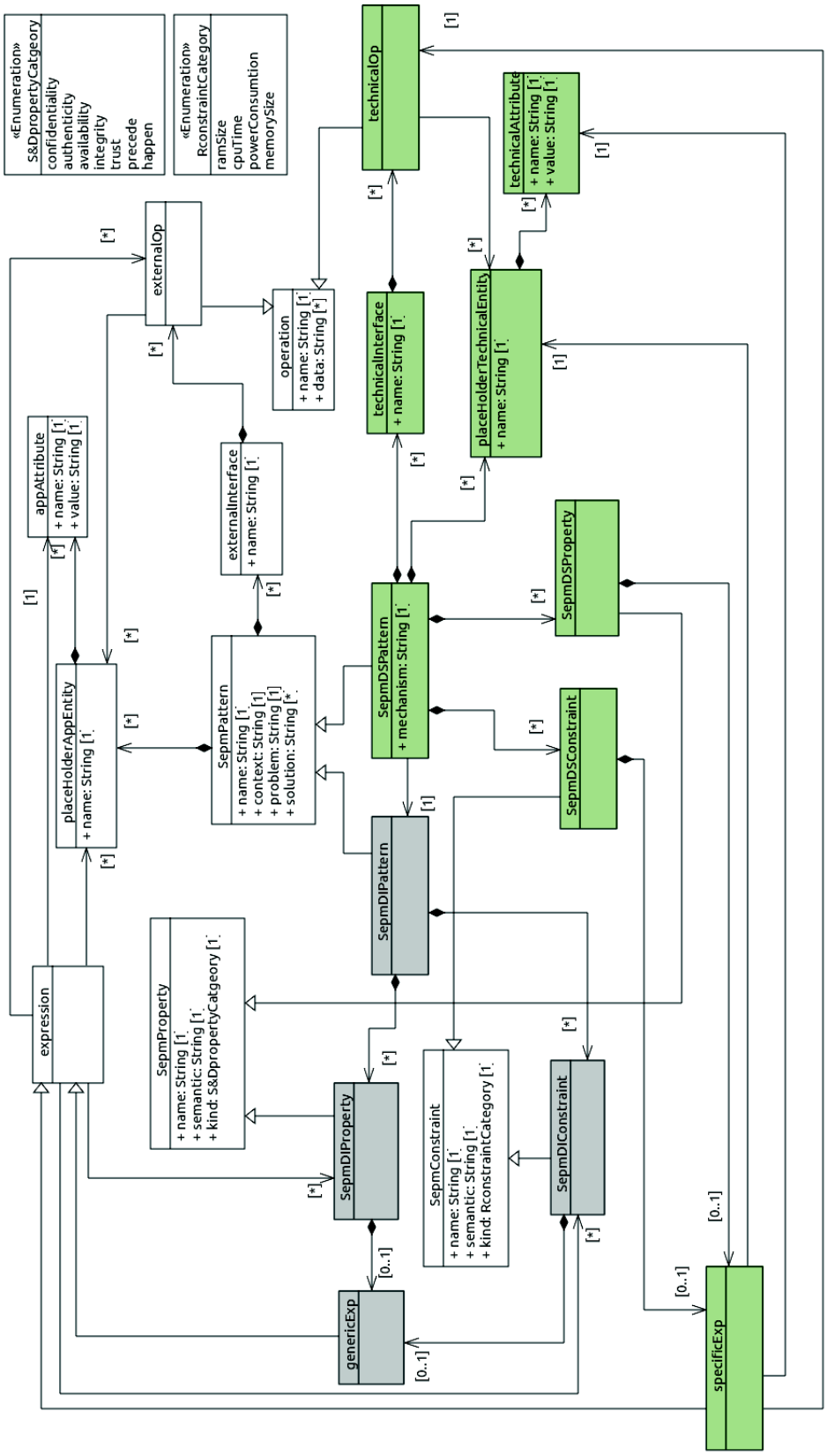

Fig. 2 The SEPM metamodel- Overview

defined through a *name*, a *semantic*, a *kind* and an *expression*. The set of possible kinds is defined in the enumeration S&DpropertyCategory. As we shall see in Section 5, each property of a pattern will be validated at the time of the pattern validating process, and hence the expression is formalized and refined to match the expected semantic.

- SepmConstraint. A constraint is a condition that holds or must hold during the application of a pattern. It is based on the notion of pre and post condition specification as commonly used in many formal methods. In our context, the assumptions derived during the formalization and validation processes of the pattern will be compiled as a set of constraints. For instance, resource constraints which will have to be satisfied by the domain application before the pattern application can be performed and after the pattern is applied. A SepmConstraint is also defined through a *name*, a *semantic*, a *kind* and an *expression*. The set of possible kinds is defined in the enumeration RconstraintCategory.
- SepmDSPattern. It is an SepmPattern and a refinement of an SepmDIPattern. This is the key entry artifact to model patterns at domain *specific* level (DSPM). Since most known techniques that deal with security and dependability are cryptography-based and redundancy-based models, respectively, we introduce the SepmDSPattern with a *mechanism* attribute to make abstraction of such notions in the SepmDIPattern model. In the example introduced in Section 3.1, TLS is one technique to achieve secure communication, and there are alternative ways to achieve the same goal. In addition to the interfaces of an SepmDIPattern, a SepmDSPattern has TechnicalInterfaces to interact with the platform representing the application domain.
- technicalInterface. These interfaces allow implementing interaction with the platform. In addition, they support interactions with security and dependability primitives and protocols. For instance, at a domain-specific level, it is possible to define links with software or hardware modules for the cryptographic key management. Please note that an SepmDIPattern does not have a TechnicalInterface.
- placeHolderTechnicalEntity. It represents the platform element to be used during the pattern integration-application in the designs. In other words, such a pattern element will be replaced with one element from the application domain, or created if it exists in the pattern but not in the application. A set of attributes (technicalAttribute) may be defined to describe its characteristics.

4.1 Generic property metamodel (GPRM)

The Generic PRoperty Metamodel (GPRM) [51], which is depicted with Ecore notations in Fig. 3, is a metamodel defining a new formalism (i.e. a language) for describing property libraries including units, types and property categories. This metamodel allows to make the typing of properties and constraints of the patterns (i.e. SepmProperty, SepmConstraint) and other artifacts extensible instead of being limited by predefined type enumerations. For instance, security and dependability attributes [54] such as authenticity, confidentiality and availability are defined as categories. These categories require a set of measure types (degree,

metrics, . . .) and units (boolean, float,. . .). To this end, we instantiate the appropriate type library and its corresponding unit library. These models are used as external model libraries to type the properties of the patterns. Especially during the editing of the pattern we define the properties and the constraints using these libraries.

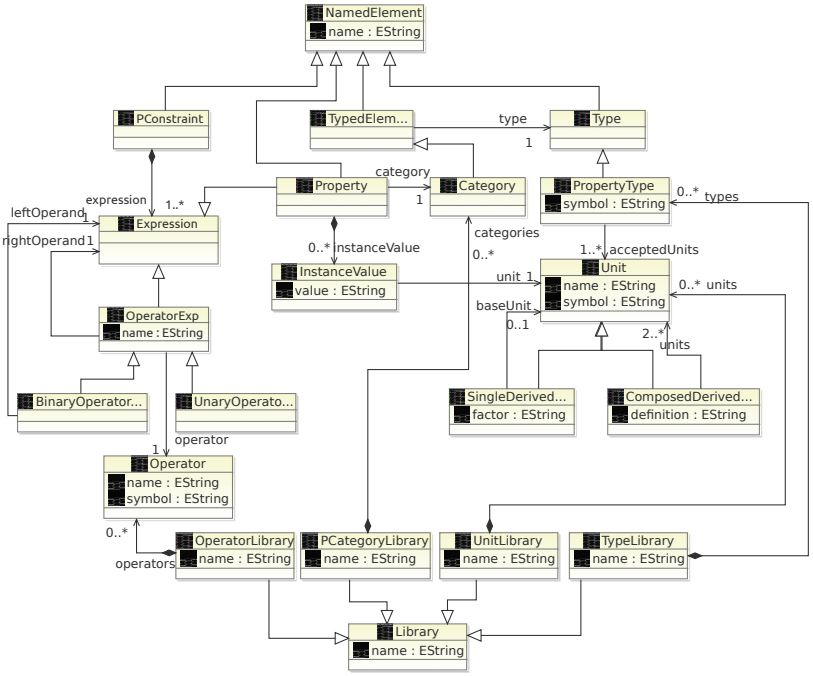

Fig. 3 The (simplified) GPRM Metamodel

4.2 Pattern model specification process: overview

The first step in our specification process is the understanding of the pattern's informal representation. The target representation is the SEPM metamodel, simplified in Figure 2 for the purpose of our study (semi-formal and formal representation of security pattern). For instance, the informal description given in the previous Section reflects our understanding of the representation of secure communication patterns given in the literature [55]. At the DIPM level, this description gives rise to specifying the following elements: interfaces of type SepmExternalInterface and security properties of type SepmProperties. At the DSPM level, the description involves the following elements: interfaces of type SepmExternalInterface and SepmTechnicalInterface and security properties of type SepmProperties. The description with varying levels of abstraction is managed by inheritance. Once

there is a good understanding of the pattern informal representation structure, the pattern can be specified using the SEPM metamodel.

The first step (A1) is to create a basic pattern subsystem P as an instance of the SepmPattern, as shown in Fig. 4. The instance is given a name and set of attributes representing the pattern. In our example, an instance of SepmPattern is created and named 'SecureCommPattern'.

Once the basic pattern subsystem has been specified, the designer starts the pattern artifact development process (the structured activity A2). It consists of the following procedures: (1) the specification of the pattern P at domain-independent level, denoted by P_{DI}, by inheritance, (2) the refinement of P_{DI} to specify one of the pattern P representations at domain-specific level, denoted by P_{DS}, by inheritance and a refinement. For each of these two procedures, interfaces are added to expose some of the pattern's functionalities. For each such interface, an instance of SepmExternalInterface is added to the pattern's interfaces collection.

The next step after creating interfaces is (3) the creation of property instances: For every security property to be provided by the pattern, an instance is created in the pattern's properties collection. A property is given a name and an expression in terms of the external interfaces in a property language (GPRM). During this activity the pattern artifacts are built conforming to the SEPM specification language.

The next activity to be performed (A3) concerns to syntactically check the design conformity of the pattern to its metamodel (SEPM). Then, activity (A4) deals with the pattern validation. It reflects the formal validation of a pattern using the process introduced in Section 5, resulting in a set of validation artifacts. At this point, the pattern designer may generate documentation (A6). If the pattern has been correctly defined, i.e. conforms to the pattern modeling language, and formally validated, the pattern is ready for publication into the model-based repository (A7). Otherwise, the issues causing the validation to fail are identified and the pattern is re-built (A5) by correcting and/or completing its relevant constructs, taking the found issues into account.

The DIPM and DSPM level constructions concerning the secure communication pattern example are described below. For the sake of simplicity, we only specify those elements of activity A2 *Develop pattern artifacts* that we need in order to explain our proof (see Section 5).

4.3 Modeling pattern at DIPM

At DIPM level, the security pattern subsystem and its related elements are created by inheritance. The first activity is to set the pattern key words (secure channel, communication) to ease the future search of the pattern. The next activity specifies the external interfaces of the pattern exposing its functionalities through function calls. In our example, we identified two external interfaces, one for the client and one for the server, providing the following functions:

- $establCh(P, ch(P, Q)) : P$ establishes a channel $ch(P, Q)$ with Q,
- $send(P, ch(P, Q), m) : P$ sends message m to Q on the channel $ch(P, Q)$,
- $recv(P, ch(P, Q), m) : P$ receives and accepts message m from Q on the channel $ch(P, Q)$,
- $closeCh(P, ch(P, Q)) : P$ closes the channel $ch(P, Q)$ shared with Q.

with $P \in \{C_1, \ldots, C_n\}$ and $Q = S$ or $Q \in \{C_1, \ldots, C_n\}$ and $P = S$, $ch(C, S) = ch(S, C)$ denotes the communication channel of a client $C \in \{C_1, \ldots, C_n\}$ and the server S, and m a message.

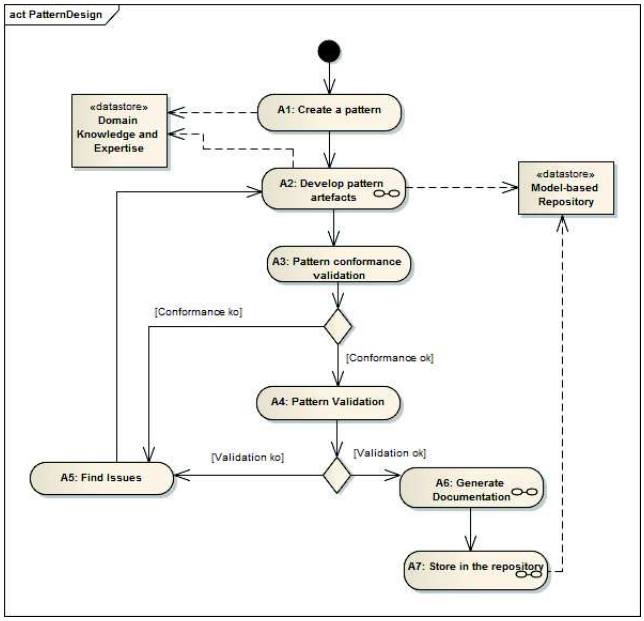

Fig. 4 Pattern development process

The next concern of the process is the definition of the pattern properties and constraints. The supporting activities require the availability of a set of property libraries. For a security property, after the instantiation of the appropriate libraries, one instance is created for each library. This instance remains active for the complete duration of the process. The imported model libraries will be used during the definition of the properties to type the category. For the example of Secure communication pattern, we specify the security property:"authenticity of sending and receiving". To type the category of this property we use a category from the ones defined in the security category library "Authenticity".

4.4 Modeling pattern at DSPM

At DSPM level, the security pattern and some of its related elements are also created by inheritance. Once a SepmDSPattern is created, every pattern external interface is identified and modeled as a refinement of the SepmExternalInterface in the pattern's interfaces collection. Then, following the pattern's description of the particular solution the pattern shall represent, each of the pattern's technical interfaces is identified and modeled by an instance of SepmTechnicalInterface in the pattern's interfaces collection. The next step is the specification of properties. Each property is represented by an instance of *SepmProperty* in the pattern's properties collection. A property is given a name and an expression in terms of the pattern's external and technical interfaces in a property language (GPRM).

The external interfaces are defined as a refinement of the DIPM external interfaces. In addition to the refinement of the concepts used at DIPM, the process in-

volves the definition of technical interfaces. These two activities are complementary and can be mutually reinforcing when undertaken simultaneously. For instance, the establishment of a channel using the TLS mechanism is a refinement of the DIPM action $establCh(P, ch(P, Q))$: We add the random numbers generated by P and Q, respectively, using the corresponding technical interfaces ($genRnd(P, Q)$), which results into $establCh(P, ch(P, rnd_P, preMS_P, Q, rnd_Q))$.

A subset of the functions provided by the external interfaces are:

- $establCh(P, ch(P, rnd_P, preMS_P, Q, rnd_Q))$: P establishes a channel $ch(\ldots)$ with Q,
- $send(P, ch(P, rnd_P, preMS_P, Q, rnd_Q), m, mac(preMS_P, \ldots, m))$: P sends m and the corresponding MAC to Q on the channel $ch(\ldots)$,
- $recv(P, ch(P, rnd_P, preMS_P, Q, rnd_Q), m, mac(preMS_P, \ldots, m))$: P receives m and the corresponding MAC from Q on the channel $ch(\ldots)$,
- $closeCh(P, ch(P, rnd_P, preMS_P, Q, rnd_Q))$: P closes the channel $ch(\ldots)$ shared with Q.

with P, Q as defined above, rnd_P and $preMS_P$ denoting a random number and the premaster secret, respectively, generated by P, m denoting a message and $mac(preMS_P, \ldots, m)$ the message authentication code generated using the premaster secret.

The technical interfaces are defined as a set of functions related to the use of TLS to refine the secure communication pattern. A subset of the functions provided by the internal interfaces are:

- $genRnd(P, rnd_P)$: P generates a random number rnd_P,
- $getCert(P, cert)$: P has access to its certificate,
- $getKey(P, PuK_{CA})$: P has access to the CA's public key,
- $verifyCert(P, PuK_{CA}, cert)$: P verifies the certificate $cert$,
- $genMac(P, ch(P, rnd_P, preMS_P, Q, rnd_Q), m, mac(preMS_P, \ldots, m))$: P generates the message authentication code (MAC) for a message using its own TLS shared secret for MAC generation

The interfaces (external and technical) with the required libraries of properties are then used in the next activity for specifying the pattern properties and constraints. The pattern interfaces specified above are likely those a developer will start with. However, the proof presented in Section 6 will show that these interfaces need to be refined.

5 The formalization and validation process

In this section, we introduce our approach for pattern validation based on our Security Modeling Framework SeMF. The next section gives a brief overview of the underlying formal semantics, Section 5.2 introduces the validation artifacts that will be applied in Section 6 where we prove the secure communication example both on DIPM and DSPM level.

5.1 The security modeling framework SeMF

In SeMF, the specification of any kind of cooperating system is based on a prefix-closed formal language whose alphabet is composed of the actions that can happen

in the system. More specifically, we use a set of agents (where the term "agent" denotes any entity acting in the system such as a human being, a device, a part of a device like an embedded system, etc.), and a set Σ of actions performed by the agents (the letters of the alphabet). The system's behavior can then be formally described by a prefix closed formal language $B \subseteq \Sigma^*$, i.e. by the set of its possible sequences of actions. The actions of the DIPM to be introduced in Section 4.3 can for example be derived from its external interface: $EstablCh(\ldots)$,$Send(\ldots)$, $Recv(\ldots)$, and $CloseCh(\ldots)$. We denote the set of letters in a word $\omega \in \Sigma^*$ by $alph(\omega)$ and the number of occurrences of any action of a set $\Gamma \subseteq \Sigma$ in a word ω by $card(\Gamma, \omega)$. If Γ consists of only one action a, we simply say $card(a, \omega)$.

Different formal models of the same application/system are partially ordered with respect to different levels of abstraction. Formally, abstractions are described by so-called alphabetic language homomorphisms that map action sequences of a more concrete abstraction level to action sequences of a more abstract level while respecting concatenation of actions.

We further extend the system specification by two components: *agents' initial knowledge* about the global system behavior and *agents' local views*. The initial knowledge $W_P \subseteq \Sigma^*$ of agent P about the system consists of all traces P initially considers possible, i.e. all traces that do not violate any of P's assumptions about the system. Every trace that is not explicitly excluded is assumed to possibly happen in the system. An agent P may assume for example that a message that was received must have been sent before leading to all sequences containing receive actions without prior send actions to be excluded.

In a running system P can learn from actions that have occurred. Satisfaction of security properties obviously also depends on what agents are able to learn. After a sequence of actions $\omega \in B$ has happened, every agent P can use its *local view* λ_P of ω to determine the sequences of actions it considers to have possibly happened. Examples of an agent's local view are that an agent can see only its own actions, or that an agent P can see the message that was sent over a network bus but cannot see who sent it, in which case e.g. $\lambda_P(send(sender, message)) = send(message)$.

Security properties can now be defined in terms of a system specification, i.e. in terms of actions, agents, the agents' initial knowledge and local views. Note that system specification does not require a particular level of abstraction. Further, it does not contain the specification of malicious behavior. In contrast to e.g. model checking approaches that explicitly specify an adversary's capabilities and try to prove that still the system cannot reach an undesired state, our approach works from the opposite: Based on the formal semantics of our framework, we specify the basic properties of the system that can be assumed to hold and use these and adequate SeMF Building Blocks (SeBBs, see below) to prove that the system satisfies specific security requirements. In the following section we will introduce our validation artifacts.

5.2 Validation artifacts

Security properties. One important set of artifacts is of course the security properties a system shall provide. In the following we will explain the basic ideas of those properties that are relevant for this paper without going into the formal details.

For more information about our formal framework and the formal definitions of security properties we refer the reader to [49], [56], and [57].

We call a particular action $a \in \Sigma$ authentic for an agent P (after a sequence of actions ω has happened) if in all sequences that P considers to have possibly happened a must have happened.

In many cases we require a particular instantiation of this property to hold:

auth$(\mathbf{a}, \mathbf{b}, \mathbf{P})$ denotes that whenever a particular action b has happened, it must be authentic for agent P that action a has happened as well. Whenever an agent has for example verified a digital signature using a specific public key (action b), in any of the sequences the agent considers possible, the owner of the private signature key must have generated the signature (action a).

precede$(\mathbf{a}, \mathbf{b})$ holds if all sequences of actions in B that contain an action b also contain an action a. Since this holds in particular for those that end on b and B is prefix closed, a happens before b. The usage of a random number for session key generation for example (action b) must always be preceded by the generation of the random number (action a).

not-precedes$(\mathbf{a}, \mathbf{b})$ denotes the contrary: it holds if all sequences of actions in B that end with b do not contain a. For instance, the verification of a digital signature using a specific agent's public key (action b) is never preceded by signature generation performed by an agent other than the owner of the respective private signature key (action a).

not-happens$(\mathbf{a})$ specifies the obvious: action a does not happen in the system.

In order to capture that some action happens at a particular point in time we use the concept of a phase class Φ. This is essentially a part of the system behavior that is closed with respect to concatenation. Maximal words $v \in \Phi$ in a phase class are those for which every continuation leads out of the phase class, i.e. for which holds $va \notin \Phi$ for all $a \in \Sigma$. Note that in general the word of a phase class need not be a word in B. In our proof, we will use a particular phase class $\Phi(St, T)$ that is defined by its starting actions St and terminating actions T: any action in St starts the phase class, and the first occurrence of any action in T terminates it. An authentication protocol for example can be described by a phase class: it may start with the sending of a random number, and possible termination actions can be signature verification and timeout.

Based on the concept of phase classes, we can express a specific action b happening within, e.g. an authentication phase is (authentically for some agent) preceded by another action a:

precede-wi-phase$(\mathbf{a}, \mathbf{b}, \mathbf{\Phi})$ holds if for all sequences of actions in the phase class Φ, if action b is contained in the sequence, then a is contained as well.

auth-wi-phase$(\mathbf{a}, \mathbf{b}, \mathbf{P}, \mathbf{\Phi(B{\uparrow}W_P)})$ holds if for all sequences of actions ω of the system that contain b, any sequence x of actions that agent P cannot distinguish from ω and considers possible satisfies the following property: If x contains a word v of the phase class $\Phi(B{\uparrow}W_P)$ (denoting Φ extended to P's initial knowledge W_P) (i.e. $x = uvz$) which in turn contains b, then this word v also contains a. If for instance an agent verifies a message authentication code within a TLS session, it is authentic for this agent that it has been generated within this session by the agent with which the session was established.

Our concept of parameter confidentiality expresses that an agent R shall not know the actual value of a particular parameter. This means that if the agent monitors any sequence of actions ω in which this parameter occurs, it cannot dis-

tinguish the actual value from any other possible value of this parameter. Whether or not R can recognize the parameter value depends to a great extend on R's initial knowledge and its local view. Let $\mathcal{A}(par)$ denote the set of actions (the context) in which *par* shall be confidential, and let *who* denote a set of agents. Then $\mathbf{conf}(\mathcal{A}(\mathbf{par}), \mathbf{par}, \mathbf{who})$ denotes that for all agents not in *who*, the parameter *par* is parameter confidential with respect to $\mathcal{A}(par)$, i.e. agents not in *who* consider all parameter values possible whenever an action of $\mathcal{A}(par)$ occurs. Our concept of parameter confidentiality is actually more complex than the above explanations disclose; however its full expressiveness is not needed in this paper, we refer the interested reader to [58].

Parameter confidentiality is obviously violated if an agent that is not allowed to know a specific parameter value excludes possible parameter values a priori or can deduce its value from other actions that need not even contain the parameter. If for example an agent knows that a specific system's random number generator produces random numbers of a restricted range, parameter confidentiality is violated (brute force attacks might reveal the random number). Hence $\mathbf{all\text{-}values\text{-}possible}(\mathbf{who}, \mathbf{par}, \mathbf{M})$ holds if all agents $R \notin who$ do neither exclude any value of parameter $par \in M$ in their initial knowledge W_R nor are able to deduce the value from other actions. The two formal properties captured by this predicate can be used to prove one of our main SeBBs (SeBB.2, see below), we refer the interested reader to [57].

We use two more concepts derived from the above notion of parameter confidentiality:

$\mathbf{restricted\text{-}conf}(\mathcal{A}(\mathbf{par}), \mathbf{par}, \mathbf{who}, \mathbf{\Gamma})$ requires parameter confidentiality only to hold for all words in the set $\Gamma \subseteq \Sigma^*$.

$\mathbf{conf\text{-}during\text{-}phase}(\mathcal{A}(\mathbf{par}), \mathbf{par}, \mathbf{who}, \mathbf{\Phi}(\mathbf{B}))$ requires parameter confidentiality to hold for all words $\omega \in B$ which extend into the phase class $\Phi(B)$. Note that parameter confidentiality may not hold for a proper prefix of such a word, hence this predicate can be used to express that some data may be known until a specific point in time (the starting action of the phase class) and then forgotten.

trust: Finally we introduce our notion of trust which allows to capture basic trust assumptions (like trust in a public key infrastructure) and to reason about these. Trust is a relation between an agent and a property: $\mathbf{trust}(\mathbf{P}, \mathbf{prop})$ denotes that agent P trusts a property *prop* to hold in the system if the property holds in the agent's conception of the system, i.e. in the agent's initial knowledge W_P. P's conception of the system may defer from the actual system. P may for example not have all information about the system behavior and believe more sequences of actions to be possible than B actually contains. Note that trust of an agent in a property is again a property.

Assumptions. Any validation of a security property holding in a system must make use of some assumptions. In order to prove for example that the TLS Handshake results in authentic shared secrets for both the client and the server, we need to assume that the public key of the certification authority is authentic for both of them. In SeMF, these assumptions are again specified as security properties.

SeMF building blocks. A SeMF Building Block (SeBB) is essentially a visualization of a proof, concerning either a formal implication between security properties or

a security mechanism. Hence, a SeMF Building Block consists of three different parts:

- The *internal properties* (*assumptions*) that are assumed to be satisfied by the system the SeBB shall be applied to.
- The mechanism or instrument that makes use of the internal properties. There are two different types: F-SeBBs which constitute a formal proof within SeMF, based on the formal definitions of the internal properties, and M-SeBBs whose proof is performed by means of our framework but is additionally based on assumptions external to SeMF, capturing expert knowledge about security mechanisms like protocols and cryptographic primitives. Such knowledge may for example concern the fact that a digital signature must have been generated using a specific signature key, i.e. cannot be guessed or generated by other means.
- The *external properties* are those that are proven to hold for the overall system, given that the internal properties hold.

An example for an F-SeBB is the transitivity of precedence: $precede(a, b)$ and $precede(b, c)$ imply $precede(a, c)$. An example for an M-SeBB captures the RSA signature mechanism: If the private key is confidential for its owner (internal property), then a signature verification action using the respective public key is always authentically for the verifier preceded by the respective signature generation action by the owner of the private key (external property).

SeBBs can be used to prove that a particular pattern provides a particular property. The assumptions that need to be satisfied in order for the pattern to provide the desired security property represent the internal properties of one or more SeBBs. By consecutively applying appropriate SeBBs we then search for a proof path that ends with the property provided by the pattern as external property. We identify all assumptions that are included in the security pattern to be verified. For a pattern describing TLS, for example, we may have the assumption that the server's private key be confidential for the server. For each of the security requirements the pattern is supposed to provide, we then search for a way to repeatedly apply appropriate SeBBs leading from a (sub)set of the assumptions identified by the pattern to the respective security requirement. The (sub)set of assumptions serves as internal properties for the first round of SeBB application which produces a set of external properties. These are taken as internal properties for the next application of SeBBs, etc., until the last step in which the desired security requirement is (one of) the external propertie(s) of the applied SeBB.

For pattern development, we use SeBBs in the reverse way, starting with the property provided by the pattern as external SeBB property and deriving the pattern assumptions as internal SeBB properties by consecutively applying adequate SeBBs. Formally, this constitutes a proof that given the assumptions hold, the pattern provides the desired property. The application of SeBBs will be explained in more detail in Section 6.

F-SeBBs can be applied on all abstraction levels and are thus domain independent, while M-SeBBs are concerned with particular security mechanisms, hence can be considered domain specific.

In the following we present the most important F-SeBBs that we will use in our proof. For simpler F-SeBBs (such as the one regarding transitivity of precedence) and the proofs we refer the reader to [57].

Deriving authenticity within a phase class (SeBB.1)
External Property:
$$auth\text{-}wi\text{-}phase(a, c, P, \Phi(B{\uparrow}W_P))$$
Internal Property:
$$trust(P, precede\text{-}wi\text{-}phase(a, b, \Phi(B)))$$ $$\wedge auth\text{-}wi\text{-}phase(b, c, P, \Phi(B{\uparrow}W_P))$$

This SeBB is analogous to SeBB.3.1.13 handling authenticity with respect to a phase class (see [57]) and can be proven analogously. We will use it in the first step of proving that the example DIPM pattern provides a specific authenticity property related to a phase class.

Extending restricted parameter confidentiality (SeBB.2)
External Property:
$$restricted\text{-}conf(\mathcal{A}(par), par, who, \{\omega a\})$$
Internal Property:
$$restricted\text{-}conf(\mathcal{A}(par), par, who, \{\omega\})$$ $$\wedge all\text{-}values\text{-}possible(who, par, M)$$ $$\wedge \omega a \in B$$ $$\wedge \forall R \notin who : \lambda_R \text{ does not reveal } par \text{ in } a$$

This SeBB essentially states that restricted parameter confidentiality is not violated by actions that do neither reveal the parameter nor allow to draw conclusions about its value.

Our idea of security pattern validation is now the following: In order to prove that a particular DIPM model provides a specific property, only F-SeBBs will be applied. As a result we identify a set of assumptions that need to hold in order for the security property to hold within the DIPM model.

For each of the assumptions in this set, we have two options: either we can argue that the DIPM model satisfies the respective assumption, or we need a pattern on DSPM level that provides the respective security property. We then use appropriate F-SeBBs and further those M-SeBBs that capture the particular security mechanisms that are used in the DSPM pattern (e.g. generation and verification of message authentication codes) to prove that the DSPM pattern provides the desired security property. Again, our proof results in a set of basic assumptions which the DSPM model needs to satisfy in order for the pattern to

provide the desired security property. Hence we need to find arguments why the model does satisfy these assumptions.

We then have a (set of) DSPM pattern that provides a particular security property *prop* given a set of assumptions holds, and a DIPM pattern that provides the desired security property we started with, given that an adequate representation of *prop* holds. What remains to be proven is that the two respective systems are related. This can be done by using a homomorphism that preserves *prop* and will be explained in the next paragraph.

Security property preserving homomorphisms. As explained in Section 5.1, a homomorphism is an abstraction that maps a concrete system to an abstract one by preserving concatenation of actions. For defining a particular homomorphism, we specify which of the concrete actions are mapped onto which of the abstract actions and onto the empty word, respectively. Under certain conditions a homomorphism can preserve specific properties: if the conditions hold, and if the property holds in the abstract system, the respective property also holds in the concrete system. For the formal proof of sufficient conditions for preserving authenticity and confidentiality, we refer the reader to [59] and [60], respectively, conditions concerning the preservation of phase classes are presented in [61].

This idea can now be applied to relate DIPM and DSPM models. For each of the assumptions that we need to hold in order for the DIPM pattern to work correctly, we do the following: We specify an adequate homomorphism that maps the DSPM model onto the DIPM model. We then prove that this homomorphism preserves the security property represented by the assumption which implies that the homomorphism "transports" this security property into the DSPM model. We finally prove that the respective property in terms of the DSPM model is identical to the property we have proven for this model to hold.

6 Validating secure communication patterns

In this section, we will show how the validation artifacts introduced in the previous section can be used for pattern validation. Exemplarily we will apply them to the example domain DIPM and DSPM patterns for secure communication introduced in Sections 4.3 and 4.4, respectively. We will explain how a proof can be conducted that each time the server side of the communication channel receives a message on the channel, for the server it authentically originates from the client side of the channel.

6.1 Formalizing the DIPM secure communication pattern

The agents of the formal model that corresponds to the DIPM interfaces introduced in Section 4.2 are a set of clients $\{C_1, \ldots, C_n\}$ and server S, the actions Σ^*_{DI} correspond to the DIPM pattern functions presented in Section 4.3.

We assume that each agent can only see its own actions. The security property provided by the pattern can be expressed as follows: Each time the server receives a message m on a channel it has established with client C, it is authentic for the server that this client has sent this message after it has established the channel

(i.e. within the word of a phase class that starts with the client establishing the channel and ends with the server closing it). Formally:

$$\begin{aligned} &\textit{auth-wi-phase}(\textit{send}(C, \textit{ch}(C,S), m), \textit{Recv}(S, \textit{ch}(S,C), m), S, \\ &\Phi(\{\textit{EstablCh}(C, \textit{ch}(C,S))\}, \{\textit{CloseCh}(S, \textit{ch}(C,S))\})(B{\uparrow}W_S)) \end{aligned} \tag{P-DI}$$

6.2 Proving the DIPM solution

For the first step of our proof, we need a SeBB whose external property is authenticity within a phase class. We use SeBB.1 introduced in Section 5, instantiating action a of the SeBB with the send action by the client, b and c with the receive action by the server, and the phase class $\Phi(B)$ with the one represented by establishing and closing, respectively, the abstract communication channel.

Thus we conclude that property P-DI holds if the following two assumptions hold:

$$\begin{aligned} &\textit{trust}(S, \textit{precede-wi-phase}(\textit{Send}(C, \textit{ch}(C,S), m), \textit{Recv}(S, \textit{ch}(S,C), m), \\ &\Phi(\{\textit{EstablCh}(C, \textit{ch}(C,S))\}, \{\textit{CloseCh}(S, \textit{ch}(C,S))\})(B))) \end{aligned} \tag{A-DI1}$$

$$\begin{aligned} &\textit{auth-wi-phase}(\textit{Recv}(S, \textit{ch}(S,C), m), \textit{Recv}(S, \textit{ch}(S,C), m), \\ &S, \Phi(\{\textit{EstablCh}(C, \textit{ch}(C,S))\}, \{\textit{CloseCh}(S, \textit{ch}(C,S))\})(B{\uparrow}W_S)) \end{aligned} \tag{A-DI2}$$

We first prove property A-DI2. For this we note that the server sees its own actions, thus in particular the action of receiving a message is authentic for the server itself. Further, the server will never accept a message on a channel it has already closed. When trying to formalize this assumption, we encounter the problem that in our formalization of the abstract communication channel used by the client and the server (see Section 4.3) we use a constant: $\textit{ch}(C,S)$ never changes for client C. So we cannot formalize directly that every receive action for a particular message m occurs within an active channel, i.e. within a word of the phase class, as there may be two actions $\textit{Recv}(S, \textit{ch}(C,S), m)$ in a word, one occurring in the phase class, the other one occurring in a word belonging to a newly established channel, and our formalization does not allow to distinguish between the two.

This gives rise to a change in our interface specification that allows to distinguish between different channels which enables us to formalize that channels are not established twice. This change furthermore corresponds to the idea of channels representing a specific period of time within the system behavior. The new interface specification and resulting set of actions is thus as follows:

$\textit{EstablCh}(P,\ ch_k(P,Q))$	P establishes a channel ch_k with Q.
$\textit{Send}(P, ch_k(P,Q), m)$	P sends message m on the channel ch_k shared with Q.
$\textit{Recv}(P, Q, ch_k(P,Q), m)$	P receives and accepts message m on the channel ch_k shared with Q.
$\textit{CloseCh}(P, ch_k(P,Q))$	P closes the channel ch_k shared with Q.

with $P \in \{C_1, \ldots, c_n\}$ and $Q = S$ or vice versa, $ch_k(C,S) = ch_k(S,C)$ and $k \in \mathbb{N}$. We can now formalize that a server receive action never occurs without an active channel as follows:

No message acceptance without active channel: (Ass.1)

$$Recv(S, ch_k(S,C), m) \in alph(\omega) \Rightarrow$$
$$\exists v \in \Phi(\{EstablCh(C, ch_k(C,S))\}, \{CloseCh(S, ch_k(C,S))\})(B)), u, z \in \Sigma_{DI}^* :$$
$$\omega = xvz \wedge Recv(S, ch_k(S,C), m) \in alph(v)$$

Thus, receipt and acceptance of a message by the server occur authentically for the server within the phase class that corresponds to the active channel being established and closed, respectively. Hence we do not need a further mechanism to enforce this authenticity property.

Regarding property A-DI1, we note that there is no reason that would allow us to just assume that the server trusts into the precedence of its receive action by a client send action within the phase class corresponding to the abstract communication channel. So the next step of a proof is to find another F-SeBB with this property as external property. In a more complex model with for example more actions in between the send and receive action we would certainly be able to apply other F-SeBBs (e.g. the one that captures the transitivity of precede). However, in this simple example model, no other F-SeBB can be applied. Hence this concludes our proof with respect to the abstract model.

We now have to consider a concrete solution, i.e. we have to find and validate a domain-specific secure communication pattern that provides an equivalent property. There exist various different possibilities for such patterns that refine the abstract communication channel and thus the abstract pattern. One possibility is e.g. to execute a Diffie Hellman based key exchange algorithm. Another possibility that we will focus on in the next section is the establishment of a TLS channel.

6.3 Formalizing the DSPM secure communication pattern

We refine the DIPM model by using TLS [7]. The formal model corresponding to the TLS pattern discussed in Section 4.2 contains the same set of agents, namely clients $C_1, \ldots, C_n$ ($n \in \mathbb{N}$) and server S. Its action set Σ_{DS} constitutes a refinement of Σ_{DI}, containing the DIPM actions with additional parameters and new actions modeling additional TLS operations. These actions together correspond to the external and technical interface function calls presented in Section 4.4, taking into account the findings discussed in the previous section. They comprise both the TLS Handshake and Record phase. However, for brevity we disregard some of the actions as they are not relevant for our proof.

The following description of the TLS Handshake protocol is an abstraction of the protocol as specified in [7] that focuses on the security relevant messages. For example, all server hello messages sent by the server are subsumed into one send action, other messages (e.g. *changeCipherSpec*) are omitted. We further assume that the premaster secret is directly used for session key generation, abstracting

from the fact that TLS uses a master key derived from the premaster secret for this. We finally abstract from implementation issues such as the actual agents that are acting in a TLS protocol (e.g. client browser, TLS library, and Networking Stack). However, we use a TCP channel to represent the intended receiver and assumed sender, respectively, of a message.

The table below lists those actions that are important for our proof. One possible and desired sequence of actions of a client C and server S that results, as we will show in Section 6.4, in establishing a secure TLS channel, will be presented in the appendix.

$genRnd(P, rnd_P)$	Agent P generates a random number. We do not model implementation dependent details such as how client and server link random numbers to server and client to establish a channel with and simply assume this link to be provided.
$send\text{-}hs(P, tcp_j, m$	Agent P sends a message (e.g. its random number and some information about the possible algorithm and key lengths etc.), using a particular TCP channel tcp_j established for (not yet secure) communication.
$recv\text{-}hs(P, tcp_j, m)$	Agent P receives the message on the TCP channel tcp_j.
$verifyCert(C, PuK_{CA}$, $cert(PrK_{CA}, PuK_S, ID_S))$	The client verifies the server's certificate with the CA's public key and extracts the server's public key.
$encrypt(C, PuK_S, preMS_C$, $cipher(PuK_S, preMS_C))$	The client encrypts the pre-master secret using the server's public key.
$establCh(P, SKmac(rnd_C, preMS_C, rnd_S^k))$	The client/server uses the three exchanged random numbers to generate its session key for MAC generation/verification, thereby establishing a secure channel.
$finished(P, tcp_j, HMAC(preMS_C, label(C), hash(m1,$ $m2, m3)))$	Agent P signals having finished the TLS Handshake. The message is composed of an HMAC including the premaster secret and the hash of all Handshake messages excluding this finished message (denoted by $m_1, \ldots, m_3$).

As a symmetric algorithm, both MAC generation and verification use the same symmetric key, so both the client and the server generate the same key

$SKmac(rnd_C, preMS_C, rnd_S^k)$. Since we focus on MAC generation and verification, respectively, we disregard the fact here that other keys, e.g. for encryption and decryption of messages are generated as well. The establishment of a channel is a refinement of the respective abstract action: We first add the random numbers generated by C and S, respectively, which results into $establCh(P, ch_k(C, rnd_C, preMS_C, S, rnd_S^k))$. We can then simplify this action by disregarding the index k of the parameter ch_k (since it is contained in rnd_S^k), by removing the channel parameters C and S (since these are represented by the respective random numbers), and by removing rnd_C, as it is not relevant for our proof. We finally exchange ch for $SKmac$ in order to emphasize that on the domain-specific level the abstract channel is represented by the TLS session key. Hence in the following we denote the client MAC generation and server MAC verification key by $SKmac(preMS_C, rnd_S^k)$.

After having established the TLS channel, the TLS Record protocol can be used to exchange messages via this channel. The following describes one possible sequence of actions:

$genMac(C, SKmac(preMS_C, rnd_S^k), m, mac(SKmac(preMS_C, rnd_S^k), m))$
: The client generates the message authentication code (MAC) for message m using its own TLS shared secret for MAC generation.

$send(C, tcp_j, SKmac(preMS_C, rnd_S^k), m, mac(SKmac(preMS_C, rnd_S^k), m))$
: The client sends m and the corresponding MAC to the server using the TLS channel $SKmac(preMS_C, rnd_S^k)$.

$recv(S, tcp_j, SKmac(preMS_C, rnd_S^k), m, mac(SKmac(preMS_C, rnd_S^k), m))$
: The server receives m and corresponding MAC on the TLS channel.

$verifyMac(S, SKmac(preMS_C, rnd_S^k), m, mac(SKmac(preMS_C, rnd_S^k), m))$
: The server verifies, using its shared secred for verification, that the message authentication code for m is correct and originates from the client.

$closeCh(S, SKmac(preMS_C, rnd_S^k))$
: This action corresponds to the *close_notify* action as specified in [7] with which the server closes the TLS session related to $preMS_C$ and rnd_S^k.

We now need to specify the security property that the TLS pattern provides. Obviously, the authenticity property provided by the HMAC mechanism refers to the generation and verification of an HMAC, so instead of using the send and receive actions as parameters of the property (in equivalence to the actions used to specify the abstract authenticity property P-DI), we use the actions of MAC generation and verification. Hence the security property that corresponds to the trust property assumed to hold for the abstract communication model (this correspondence will be proven in Section 6.5) is that the server trusts into the precedence of its own verification action by the MAC generation action of the client within the phase class that starts with the client establishing the TLS channel (i.e. generating the MAC key) and ends with the closure of the TLS session:

$$
\begin{aligned}
&trust(S, \textit{precede-wi-phase}(\\
&genMac(C, SKmac(preMS_C, rnd_S^k), m,\\
&mac(SKmac(preMS_C, rnd_S^k), m)),\\
&verifyMac(S, SKmac(preMS_C, rnd_S^k), m,\\
&mac(SKmac(preMS_C, rnd_S^k), m)),\\
&\Phi(\{establCh(C, SKmac(preMS_C, rnd_S^k))\},\\
&\{closeCh(S, SKmac(preMS_C, rnd_S^k))\})))
\end{aligned}
\tag{P-DS}
$$

6.4 Proving the TLS mechanism

In order to prove that the TLS mechanism provides this property, we first introduce some TLS specific SeBBs and assumptions and discuss why they are appropriate.

6.4.1 Assumptions and M-SeBBs

We assume the server's private RSA key for decryption to be confidential for the server:

Confidentiality of the server's private key: (Ass.2)

$$conf(\mathcal{A}(PrK_S), PrK_S, \{S\})$$

Considerations regarding random numbers and premaster secret

The first two random numbers that are generated are sent in plain-text, so they are considered known by all agents. Regarding knowledge of the premaster secret $preMS_C$, we assume that client and server do not make it available other than through the Handshake messages. That is to say, the session keys are generated on the basis of the premaster secret exchanged; thus their confidentiality is violated if the premaster secret gets known. Hence we assume that the premaster secret is not known by any agents prior to generation and only to the client until directly after its generation. One mechanism to satisfy this property is to use a random number as premaster secret. Hence we assume that the client indeed generates a random number as premaster secret (the client could very well use an already existing one). Random numbers are generated "virtually unique" which is formalized by the following assumption about their "absolute uniqueness":

Random number generation only once: (Ass.3)

$$\forall \omega \in B \forall rnd \in \mathbb{N} : card(\{genRnd(P, rnd) \mid P \in \mathbb{P}\}, \omega) \leq 1$$

We assume a random number to be confidential for the agent that generates it from system start until its generation. Note that this assumption only holds if the agent uses an appropriate random number generator that provides both confidentiality and non-predictability of random numbers (see Assumption Ass.3). This leads to

Confidentiality of random number until generation: (Ass.4)

$$\begin{aligned} &\mathit{restricted\text{-}conf}(\mathcal{A}(rnd), rnd, \emptyset, \{\omega \in B \mid genRnd(P, rnd) \notin alph(\omega)\} \\ \wedge &\mathit{restricted\text{-}conf}(\mathcal{A}(rnd), rnd, \{P\}, \{pre(\omega) \in B \mid suf_1(\omega) = genRnd(P, rnd)\} \end{aligned}$$

with $pre(\omega)$ denoting the set of prefixes of ω and $suf_1(\omega)$ denoting its last action.

Session key generation

We assume that a session key cannot be guessed, so before using the session key for MAC generation, it has to be established:

Session key generation before usage: (Ass.5)

$$\begin{aligned} &precede(establCh(P, SKmac(preMS_C, rnd_S^k)), \\ &genMAC(Q, SKmac(preMS_C, rnd_S^k), m, mac(SKmac(preMS_C, rnd_S^k), m))) \end{aligned}$$

In other words, if agent R knows the session key, it has either generated it, or received it from the server or the client. This, however, would render the whole protocol useless. So we assume that neither the session keys nor the premaster secret are leaked or made available deliberately.

We assume that the session key generation algorithm in [7] is secure in the sense that it requires knowledge of the premaster secret. Since we further assume that session keys are not made available other than (potentially) after ending the TLS session, we have the following assumption:

Confidentiality of session keys: (Ass.6)

$$\begin{aligned} &\mathit{conf\text{-}during\text{-}phase}(\mathcal{A}(preMS_C), preMS_C, who, \Phi(St, T)) \\ &\Rightarrow \\ &\mathit{conf\text{-}during\text{-}phase}(\mathcal{A}(SKmac(preMS_C, rnd_S^k)), SKmac(preMS_C, rnd_S^k), who, \\ &\Phi(St, T)) \end{aligned}$$

A session key involving a specific random number, in particular the premaster secret, cannot be generated before the random number has been generated.

Session key generation: (Ass.7)

$$precede(genRnd(P, preMS_C), establCh(Q, SKmac(preMS_C, rnd_R^k)))$$

A MAC is generated and verified, respectively, with the same session key that must be known to both the client and the server. Assuming that the server acts correctly, it will never deliberately use the client's MAC generation key to generate a MAC and will only use this key to verify a MAC. Hence the only reason that a server's MAC verification action is preceded by an action in which the server itself generates this MAC is that the server's MAC generation key is identical to the one of the client. According to [7], the key generation algorithm is based on all random numbers exchanged in the Handshake: the client's one sent in the first message, the server's one sent in the second message, and the premaster secret sent again by the client in the third message. So even if the client violates the protocol specification and uses old numbers, the fact that the server's random number is used guarantees that each of the session keys are different to session keys generated in any other TLS session before. While the client can establish the channel (i.e. session key) based on the server's random number arbitrarily often, each time producing the same key and thus the same channel, for simplicity we assume that it happens only once:

Session key generation only once: (Ass.8)

$$\forall \omega \in B \forall k \in \mathbb{N} : card(establCh(C, SKmac(preMS_C, rnd_S^k)), \omega) \leq 1$$

Further, the key generation algorithm guarantees that all session keys are different from each other. Hence we can assume that the server never accidentally uses the client's MAC generation key to generate a MAC, even more, it does not use it for anything but verifying a client's MAC. Formally:

MAC generation key not used by server: (Ass.9)

$$not\text{-}happens(genMac(S, SKmac(preMS_C, rnd_S^k), m, \\ mac(SKmac(preMS_C, rnd_S^k), m)))$$

Rather than using this assumption, a more low level analysis would formulate low level assumptions and SeBBs in order to formally prove Assumption Ass.9. However, this would constitute a verification of the internal TLS implementation. Though possible this is out of the scope of the work described here.

Closing of sessions

The TLS session can be terminated by either the client or the server by sending a close_notify message. The recipient must respond with a close_notify of its own. Of course, if an adversary intercepts a close_notify message for example sent by the client, the server will never receive it. So with no additional measures taken, the protocol would allow a situation in which the server considers the TLS session still valid while the adversary retrieves the session keys or premaster secret (by

some cryptanalysis algorithm). This would allow the adversary to impersonate the client. Hence an appropriately implemented protocol will include a timeout action by the server (and the client, respectively) that terminates any TLS session before the respective premaster secret and session keys can get known. We abstract from this by assuming that a close action will always happen in due time, i.e. before the session keys and premaster secret can get known. Since we want to investigate the TLS protocol from the server's point of view, for simplicity we further disregard the fact that the client can close the session and assume that it is always the server who terminates it. We then assume that after the server has closed a session, the respective premaster secret and session keys are potentially known to all agents. Accordingly, we assume that the server never accepts a message sent on a channel it has already closed:

Message acceptance within active channels: (Ass.10)

$$\begin{aligned} &verifyMac(S, SKmac(preMS_C, rnd_S^k), m, mac(SKmac(preMS_C, rnd_S^k), m)) \\ &\qquad \in alph(\omega) \Rightarrow \\ &\qquad \exists v \in \\ &\Phi(\{establCh(C, SKmac(preMS_C, rnd_S^k))\}, \{closeCh(SKmac(preMS_C, rnd_S^k))\}), \\ &\qquad u, z \in \Sigma_{DS}^* : \omega = uvz \wedge verifyMac(S, SKmac(preMS_C, rnd_S^k), m, \\ &\qquad\qquad mac(SKmac(preMS_C, rnd_S^k), m)) \in alph(v) \end{aligned}$$

Note that since in this document we address authenticity rather than confidentiality of messages, we do not discuss the question when exactly a session key can get known, i.e. at which point in time after the closure of a session the respective session key may be known by an adversary and confidentiality of messages sent in this session is violated.

For technical reasons within the formal proof, we further assume that the server does not close a session that has not yet started:

No close of not existing session: (Ass.11)

$$not\text{-}precedes(closeCh(SKmac(preMS_C, rnd_S^k)), establCh(C, SKmac(preMS_C, rnd_S^k)))$$

Agents' local views

We further assume that spying on the client's or server's platform is not possible, and assume typing of messages, i.e. we assume that each of the TLS messages contains enough information to recognize its particular role in the protocol (the protocol step and session it is meant to be used for). Thus we can disregard problems that could occur if some TLS message can be misinterpreted and used in a different step or session of TLS.

The set $\mathcal{A}(preMS_C)$ of actions in the context of which the premaster secret shall be confidential contains all actions that have a parameter $preMS_C$ (see the

appendix for the full set of actions):

$$\mathcal{A}(preMS_C) := \{genRnd(C,\ldots), establCh(C,\ldots), encrypt(C,\ldots), sign(C,\ldots), \\ send\text{-}hc(C,\ldots), finished(C,\ldots), recv\text{-}hc(S,\ldots), verifySig(S,\ldots), \\ decrypt(S,\ldots), establCh(S,\ldots), finished(S,\ldots), closeCh(S,\ldots)\}$$

However, since we assume that spying on the server's or client's platform is not possible, all actions processed completely within a platform cannot be monitored by an adversary, i.e. an adversary's local view of these actions is the empty word. For the send and receive actions performed by the client and the server, respectively, we note that the premaster secret is encrypted. Consequently, no agent, when looking at these actions, can actually see the premaster secret. The equivalent holds for the *finished* messages that include the premaster secret only in the HMAC (we assume here that an HMAC is a one-way function), and for the *closeCh* message that contains $preMS_C$ as a parameter of the channel. This implies that the local views λ_R of agents R other than the client and server do not reveal the premaster secret in these actions. Trivially, λ_R does not reveal any parameter of an action a if $\lambda_R(a) = \varepsilon$. We define

$$\mathcal{A}_\sharp(preMS_C) := \mathcal{A}(preMS_C) \setminus \{genRnd(C, preMS_C), \\ encrypt(C, PuK, preMS_C, cipher(PuK, preMS_C))\}$$

Local views do not reveal $preMS_C$: (Ass.12)

$$\forall R \in \mathbb{P} \setminus \{C, S\} : a \in \mathcal{A}_\sharp(preMS_C) \Rightarrow \lambda_R \text{ does not reveal } preMS_C \text{ in } a$$

Agent's initial knowledge regarding possible values of premaster secret

The actions of a TLS session that contain the premaster secret can be used in different orders (e.g. the client can generate the premaster secret before or after actually sending the first message to the server). There may even be different actions used in different types of sessions. The actions could for example contain the operating system of the device they are performed on. If there was a known weakness in random number generation of one of the possible operating systems, the particular type of a TLS session would indeed have an influence on its security. However, since we assume a properly working random number generator, confidentiality of the premaster secret does not depend on the particular order of actions or type of session that is being used. Consequently, we need not differentiate between different types of TLS sessions, hence our model does not contain any such differentiation. This does not hold in general as there may very well be weaknesses of a protocol, e.g. the generation of a not sufficiently long random number that occurs only in specific types of sessions.

Additionally, when using a properly working random number generator, i.e. a pseudo random number generator with a secure seed, even if an agent knows one random number (e.g. a malicious server that had some time in the past a TLS session established with the client), it will not be able to deduce previously generated random numbers or to predict future ones. This is formalized by assuming that there is no relation between the values of premaster secrets in different sessions. Hence in our model an agent has no way to conclude the value of one premaster secret by knowing the value of another one. This is again not true in general: if a protocol for example uses a sequence number, even if it is confidential for agent R, R knows that the sequence numbers are simply incremented which constitutes

a relation between different sessions. In fact, it even violates confidentiality in the case R itself is involved in one session. TLS that is based on a random number generator that does not provide unpredictability suffers from the same security flaw. Hence the quality of the random number generator is of paramount importance for the security of TLS. Abstracting from the relation between different TLS sessions implies that an agent R observing the system must assume that in any of the sessions, any parameter value in the range of the premaster secret can occur.

In view of the above considerations and considering all activities that constitute the TLS protocol, we note that there is no action, containing or not containing the premaster secret, that narrows the range of its values that agent R considers possible. Since we further assume that a priori all agents $R \neq C, S$ assume all values of $preMS_C$ in the given range to be possible, we conclude the following assumption with $M(preMS_C)$ denoting the range of random number values of the premaster secret:

Agents' initial knowledge does not exclude parameter values: (Ass.13)

$$\textit{all-values-possible}(\{C, S\}, preMS_C, M(preMS_C))$$

6.4.2 Domain specific mechanisms

If client or server make the premaster secret available deliberately, this is equivalent to making all the session keys themselves available. As said above, this in turn would render the whole protocol useless. So we assume that neither the session keys nor the premaster secret are made available deliberately. Hence the only actions that operate on the premaster secret are the ones specified in the TLS Handshake. In particular, the first action that makes the premaster secret available to the outside world is the one in which it is sent encrypted.

The effect of encrypting $preMS_C$ is modeled by the following RSA encryption SeBB. Only an agent that owns the particular private key corresponding to the public key used for encryption can decrypt the cipher-text and thus may know the plain-text. For technical reasons we consider $preMS_C$ potentially known by all agents owning the decryption key already in the moment when it is encrypted. Note that this SeBB implicitly assumes that no agent $P \notin who$ can actually see the parameter at the moment of encryption.

RSA encryption (SeBB.3)
External Property: $$\mathit{restricted\text{-}conf}(\mathcal{A}(par), par, who \cup \{Q\}, \{\omega a\})$$
Internal Property: $$\begin{aligned} &\mathit{restricted\text{-}conf}(\mathcal{A}(par), par, who, \{\omega\}) \\ \wedge a = &\mathit{encrypt}(P, PuK_Q, par, \mathit{cipher}(PuK_Q, par)) \\ \wedge &\omega a \in B \\ \wedge &\mathit{conf}(\mathcal{A}(PrK_Q), PrK_Q, \{Q\}) \end{aligned}$$

For the proof of this and the other M-SeBB presented in this section, we refer the reader to the appendix.

With the same arguments as above we assume that the client encrypts each premaster secret only with one public key, i.e. does not encrypt it using different public keys, because this again would constitute a deliberate violation of confidentiality. Note that this assumption can only be safely made if the server's public key is only used within TLS protocol runs. Since the client wants to establish a TLS session with server S, we can further assume that it only uses PuK_S for encrypting the premaster secret.

No double encryption of premaster secret: (Ass.14)

$$\begin{aligned} &\mathit{encrypt}(C, PuK, preMS_C, \mathit{cipher}(PuK, preMS_C)) \in \mathit{alph}(\omega) \\ &\Rightarrow PuK = PuK_S \end{aligned}$$

A premaster secret that has not yet been generated cannot be encrypted:

No operation on not existing premaster secret: (Ass.15)

$$\begin{aligned} &\forall \omega \in B_{DS} : \\ &\mathit{genRnd}(P, preMS_C) \notin \mathit{alph}(\omega) \\ \Rightarrow \omega\ &\mathit{encrypt}(P, PuK, preMS_C, \mathit{cipher}(PuK, preMS_C)) \notin B_{DS} \end{aligned}$$

Finally, the following SeBB captures the nature of a MAC mechanism which guarantees that a MAC verification action is always preceded by a MAC generation action. If further the key used to generate the MAC is generated after some particular action starting a phase class, then MAC verification happening within this phase class implies that the MAC generation must also have happened within this phase class.

MAC generation and verification (SeBB.4)
External Property:
$$\begin{aligned} &\textit{precede-wi-phase}(genMac(P, key, m, mac(key, m)),\\ &verifyMac(Q, key, m, mac(key, m)), \Phi(\{s\}, T)(B)) \end{aligned}$$
Internal Property:
$$\begin{aligned} &\textit{conf-during-phase}(\mathcal{A}(key), key, \{P, Q\}, \Phi(\{s\}, T)(B))\\ &\wedge \textit{not-precedes}(genMac(Q, key, m, mac(key, m)),\\ &verifyMac(Q, key, m, mac(key, m)))\\ &\wedge \textit{precede}(s, genMac(P, key, m, mac(key, m)))\\ &\wedge verifyMac(Q, key, m, mac(key, m))) \in alph(\omega) \Rightarrow\\ &\exists v \in \Phi(\{s\}, T), u, z \in \Sigma^* :\\ &\omega = uvz \wedge verifyMac(Q, key, m, mac(key, m))) \in alph(v)\\ &\wedge \forall \omega \in B : card(s, \omega) \leq 1 \end{aligned}$$

6.4.3 The proof

The security property that we want to prove for the TLS pattern and that corresponds to the trust property assumed to hold for the DIPM model (this correspondence will be addressed in the next section) is that the server trusts into the precedence of its own verification action by the MAC generation action of the client within the TLS session starting with session key generation by the client. Formally, this phase class is specified as $\Phi(\{establCh(C, SKmac(preMS_C, rnd_S^k))\}, \{closeCh(S, SKmac(preMS_C, rnd_S^k))\}$. Hence the property we want to prove for the TLS mechanism can be formally specified as

$$\begin{aligned} &trust(S, \textit{precede-wi-phase}(\\ &genMac(C, SKmac(preMS_C, rnd_S^k), m,\\ &mac(SKmac(preMS_C, rnd_S^k), m)),\\ &verifyMac(S, SKmac(preMS_C, rnd_S^k), m,\\ &mac(SKmac(preMS_C, rnd_S^k), m)),\\ &\Phi(\{establCh(C, SKmac(preMS_C, rnd_S^k))\},\\ &\{closeCh(S, SKmac(preMS_C, rnd_S^k))\}))) \end{aligned} \tag{P-DS}$$

As explained in Section 5.2, trust of S into the above precede property means that this property holds in the server's conception of the system. More precisely, if S trusts this property to hold, precedence of MAC verification by MAC generation holds within the phase class being extended into the respective one within what

S believes to be the system behavior (formally denoted by W_S, see Definition 15 of [62] for more information). In order to facilitate the proof, we, therefore, first prove that the precede property holds in B_{DS}, using the assumptions introduced in Section 6.4.1. We then argue that S trusts all these assumptions to hold, hence the respective assumptions hold in W_S which allows us to perform an equivalent proof within this system.

So we need to prove the following:

$$\begin{aligned} &\textit{precede-wi-phase}(genMac(C, SKmac(preMS_C, rnd_S^k), m, \\ &\qquad mac(SKmac(preMS_C, rnd_S^k), m)), \\ &\qquad verifyMac(S, SKmac(preMS_C, rnd_S^k), m, \\ &\qquad mac(SKmac(preMS_C, rnd_S^k), m)), \\ &\qquad \Phi(\{establCh(C, SKmac(preMS_C, rnd_S^k))\}, \\ &\qquad \{closeCh(S, SKmac(preMS_C, rnd_S^k))\})) \end{aligned} \tag{P-DS1}$$

We do this by applying SeBB.4.The last two properties required by SeBB.4 are assumed to hold (assumptions Ass.5 and Ass.8, and Ass.10, respectively). Hence it remains to prove the following two internal properties:

$$\begin{aligned} &\textit{conf-during-phase}(\mathcal{A}(SKmac(preMS_C, rnd_S^k)), SKmac(preMS_C, rnd_S^k), \\ &\qquad \{C, S\}, \Phi(\{establCh(C, SKmac(preMS_C, rnd_S^k))\}, \\ &\qquad \{closeCh(S, SKmac(preMS_C, rnd_S^k)\})) \end{aligned} \tag{P1}$$

$$\begin{aligned} &\textit{not-precedes}(genMac(S, SKmac(preMS_C, rnd_S^k), m, \\ &\qquad mac(SKmac(preMS_C, rnd_S^k), m)), \\ &\qquad verifyMac(S, SKmac(preMS_C, rnd_S^k), m, \\ &\qquad mac(SKmac(preMS_C, rnd_S^k), m))) \end{aligned} \tag{P2}$$

Proving property P2 is trivial: since by Ass.9 the server never generates a MAC using $SKmac(preMS_C, rnd_S^k)$, this action does not precede any other action of the system, in particular it does not precede the server's MAC verification action. So it remains to prove the confidentiality of $SKmac(preMS_C, rnd_S^k)$.

Proving confidentiality of $\mathbf{SKmac(preMS_C, rnd_S^k)}$

We start proving confidentiality of the session key by first proving that the premaster secret that it is based on is confidential during the TLS session. Confidentiality of the session key then follows with Ass.6. Hence we need to show

$$\begin{aligned} &\textit{conf-during-phase}(\mathcal{A}(preMS_C), preMS_C, \{C, S\}, \\ &\qquad \Phi(\{establCh(C, SKmac(preMS_C, rnd_S^k))\}, \\ &\qquad \{closeCh(S, SKmac(preMS_C, rnd_S^k))\})) \end{aligned} \tag{P5}$$

In order to prove this, we first prove confidentiality of the premaster secret during the phase class starting with its generation and then show the above property.

Let $preMS_C^*$ and rnd_S^* be fixed but arbitrary premaster secrets and random numbers, respectively, and let $C \in \{C_1, \ldots, C_n\}$. For ease of reading the following proof, we use the abbreviation $\Phi(preMS_C^*, rnd_S^*) := \Phi(\{genRnd(C, preMS_C^*)\}, \{closeCh(S, SKmac(preMS_C^*, rnd_S^*))\})$. We start with focusing on words that end with generating this particular premaster secret. Let $\Omega(preMS_C^*)$ denote all words $\omega \in B_{DS}$ that continue with client C generating the premaster secret, i.e. $\Omega(preMS_C^*) := \{\omega \in B_{DS} \mid suf_1(\omega^{-1}(B_{DS})) = \{genRand(C, preMS_C^*)\}$ where ω^{-1} denotes the set of all continuations of ω in B_{DS}. Since we assume the client to generate a random number as premaster secret, Ass.4 implies

$$\begin{aligned} &\textit{restricted-conf}(\mathcal{A}(preMS_C^*), preMS_C^*, \{C\}, \\ &pre(\{\omega v \in B_{DS} \mid \omega \in \Omega(preMS_C^*)), v = genRnd(C, preMS_C^*)\}) \end{aligned} \tag{P6}$$

This trivially implies

$$\textit{restricted-conf}(\mathcal{A}(preMS_C^*), preMS_C^*, \{C, S\}, \Gamma) \tag{P7}$$

with $\Gamma = pre(\{\omega v \in B_{DS} \mid \omega \in \Omega(preMS_C^*)), v = genRnd(C, preMS_C^*)\}$. Note that the words for which confidentiality of the premaster secret holds are the prefixes of those that extend into the phase class $\Phi(preMS_C^*, rnd_S^*)$ by just the starting action. From this we then deduce restricted confidentiality for the prefixes of all arbitrary words that extend into this phase class. We do this by induction over the length of such words.

1. Induction basis: by property P7.
2. Induction hypothesis: let property P7 hold for $\Gamma_0 = \{pre(\omega v_0) \in B_{DS} \mid \omega \in \Omega(preMS_C^*), v_0 \in \omega^{-1}(B_{DS}) \cap \Phi(preMS_C^*, rnd_S^*)\}$.
3. Induction step: let $\omega \in \Omega(preMS_C^*), a \in \Sigma_{DS}$ with $v_0 a \in \omega^{-1}(B_{DS}) \cap \Phi(preMS_C^*, rnd_S^*)$.
   (a) $a \notin \mathcal{A}(preMS_C^*)$. Then trivially, λ_R does not reveal $preMS_C^*$ for any agent $R \neq C, S$. Hence Assumption Ass.13 together with SeBB.2 implies

$$\begin{aligned} &\textit{restricted-conf}(\mathcal{A}(preMS_C^*), preMS_C^*, \{C, S\}, \\ &\{pre(\omega v_0 a) \in B_{DS} \mid \omega \in \Omega(preMS_C^*), v_0 a \in \omega^{-1}(B_{DS}) \cap \\ &\qquad \Phi(preMS_C^*, rnd_S^*)\}) \end{aligned} \tag{P9}$$

   (b) $a \in \mathcal{A}_\sharp(preMS_C^*)$. By Assumption Ass.12, the local views of agents other than C and S do not reveal the premaster secret, hence again, Assumption Ass.13 together with SeBB.2 implies property P9 to hold. If further $a = closeCh(S, SKmac(preMS_C, rnd_S^k))$, then $v_0 a$ is maximal in $\Phi(preMS_C^*, rnd_S^*)$ and confidentiality of $preMS_C^*$ within the phase class is proven.
   (c) $a = encrypt(P, PuK, preMS_C^*, cipher(PuK, preMS_C^*))$. Since $genRnd(C, pre{-}MS_i) \in alph(\omega v_0 a)$, Ass.3 implies $genRnd(P, preMS_C^*) \notin alph(\omega v_0 a)$ for all $P \neq C$, and with assumption Ass.15 it follows $\omega v_0 encrypt(P, PuK, preMS_C^*, cipher(PuK, preMS_C^*)) \notin B_{DS}$ for all $P \neq C$. Ass.14 implies $a = encrypt(C,$

$PuK_S, preMS^*_C, cipher(PuK_S, preMS^*_C)$). Then SeBB.3 together with assumption Ass.2 implies that the owner of the private key corresponding to PuK_S, i.e. the server S, can be added to the set of agents allowed to know $preMS^*_C$ (which we formally have done already for property P7), i.e. we can again conclude that property P9 holds.

(d) $a = genRnd(P, rnd)$ As explained above, by Ass.3 an agent other than the client cannot have generated $preMS^*_C$, hence no other agent's local view reveals $preMS^*_C$. Further, generation of a different random number does not reveal knowledge of $preMS^*_C$, formalized by Ass.13, thus property P9 holds.

Since property P9 holds for the prefixes of all arbitrary words that extend into the phase class $\Phi(preMS^*_C, rnd^*_S) = \Phi(\{genRnd(C, preMS^*_C)\}, \{closeCh(S, SKmac(preMS^*_C, rnd^*_S)\}))(B_{DS})$, it holds in particular for the prefixes of all maximal words, hence confidentiality during this phase class holds:

$$\begin{aligned} &conf\text{-}during\text{-}phase(\mathcal{A}(preMS^*_C), preMS^*_C, \{C, S\}, \\ &\Phi(\{genRnd(C, preMS^*_C)\}, \{closeCh(S, SKmac(preMS^*_C, rnd^*_S))\})(B_{DS}) \end{aligned} \tag{P10}$$

By Ass.7, the premaster secret must be generated before channel establishment. Then using Ass.11, it can easily be proven that property P5 holds. With Ass.6 it follows that property P1 holds.

Concluding the proof

Having shown that properties P1 and P2 hold, SeBB.4 implies that property P-DS1 holds.

As explained at the beginning of our proof, we now assume that S trusts all assumptions to hold. This includes in particular that the server has to trust that the client uses an appropriate random number generator. This is a strong assumption, considering that deficiencies of operating systems (e.g. Debian) have been detected that used a random number generator with insufficiently large output. The problem is further increased since the server usually has no means to authentically identify the client's operating system. Nevertheless, if the server does not trust the quality of the random numbers, it can simply refuse to establish a TLS channel with the client. If now the server trusts all assumptions to hold, this implies that all assumptions hold in what the server believes to be the system i.e. with respect to its initial knowledge W_S. Since this is in itself a system, and assuming all assumptions to hold in this system, the analogous proof can be performed for W_S. This in turn implies that property P-DS1 holds in W_S, which by our notion of trust as introduced in Section 5.2 implies that property P-DS holds in B_{DS}.

The approach is described and applied in more detail in [48]. For more information on the used notion of trust please refer to [56,63].

6.4.4 Other properties of TLS

Note that TLS provides more properties than the ones discussed here. What we have proven so far is that after MAC verification, the server may trust that the MAC was generated by the client with whom it has established the TLS channel, given the server trusts in the assumptions about the client behavior. This does not

say anything about this client's identity yet. However, if the client authenticates itself, TLS does not only provide authenticity of the MAC generation by *some* client to the server but also allows the server to know *which* client generated the MAC. In order to prove this we would need more assumptions, e.g. concerning the server's trust in the CA's private key being confidential for the CA. These assumptions would allow us to reason about the client's certificate and signature related to the Handshake messages and also to base the assessment of trust in the client behavior onto the reputation it has with the CA for certificate issuing. This is subject of future work.

6.5 Correspondence between DIPM and DSPM

We have now achieved proofs that (i) assuming that Ass.1 and property A-DI1 hold, the DIPM model provides property P-DI, and (ii) assuming that assumptions Ass.2 to Ass.15 hold, the DSPM model provides property P-DS. In the final proof step, we have to show that the DIPM model is an abstraction of the DSPM model that preserves property A-DI1, and that this property, transfered to the DSPM model, is identical to property P-DS.

In order to show this, we use an appropriate homomorphism that maps the actions of the TLS model onto the actions of the abstract communication model, and then show that this homomorphism preserves trust in precede within the phase class corresponding to establishing and closing, respectively, the abstract communication channel.

Since the aim of this homomorphism is to relate the trust properties P-DI and P-DS, an obvious mapping is the one that on the one hand relates the actions relevant for the abstract and concrete *precede* properties, and on the other hand relates the abstract and concrete phase classes. Hence we define the following homomorphism:

$$
\begin{aligned}
&h(establCh(P, SKmac(preMS_C, rnd_S^k))) && = EstablCh(P, ch_k(S,C))\\
&h(genMac(C, SKmac(preMS_C, rnd_S^k), m,\\
&\quad mac(SKmac(preMS_C, rnd_S^k), m))) && = Send(C, ch_k(C,S), m)\\
&h(verifyMac(S, SKmac(preMS_C, rnd_S^k), m,\\
&\quad mac(SKmac(preMS_C, rnd_S^k), m))) && = Recv(S, C, ch_k(C,S), m)\\
&h(closeCh(S, SKmac(preMS_C, rnd_S^k))) && = CloseCh(S, ch_k(C,S))
\end{aligned}
$$

$(P \in \{S, C\})$ and map all other actions onto the empty word ε.

In order to preserve trust in precedence, the homomorphism needs to satisfy $h(W_S) \subseteq W'_S$, i.e. the image of a sequence of actions that S considers initially possible in the DSPM model must again be considered possible by S in the DIPM model (see [62] for a proof). This can easily be proven by induction, making use of the fact that $h(\omega) \notin W'_S$ implies $h(\omega)$ to contain an action $Recv(S, ch_k(C,S))$ happening outside the channel $ch_k(C,S)$ which in turn contradicts assumption Ass.1. Hence h preserves trust into precedence.

In [62] we have shown that with $\Phi(\{EstablCh(C, ch_k(C,S))\}, \{closeCh(S, ch_k(C, S))\})$ a phase class in the domain independent pattern system, $\Phi(\{h^{-1}(EstablCh(C, ch_k(C,S)))\}, \{h^{-1}(CloseCh(S, ch_k(C,S)))\}) = \Phi(\{EstablCh(C, SKmac(\ preMS_C, rnd_S^k))\}, \{CloseCh(S, SKmac(preMS_C, rnd_S^k))\})$ is a phase class in the domain-specific pattern system. This in turn is the phase class the precedence property

transfered by the homomorphism refers to. This implies that property A-DI1 transfered to the TLS model is identical to property P-DS which concludes our proof.

7 Integration of modeling techniques and formal validation methods

In our vision, a security pattern is a subsystem exposing pattern functionalities through interfaces and targeting security properties. It is an ideal development context for integrating metamodeling techniques and formal approaches.

7.1 Towards a unified modeling and formal design framework for security pattern definition

At high level of abstraction, security properties are derived from security requirements with clear semantics, albeit with informal description. At the semi-formal description level, the link of these requirements to the pattern is supported by the pattern properties documentation and categories. Security categories for instance may be provided as external model libraries targeting specific domains. The documentation of a specific security property belonging to a specific category includes a semantic description to support the decision of whether a specific pattern actually fulfills this particular security property. At the formal description level, the actions of the formal models representing DIPM and DSPM, respectively, are derived from the patterns' external and technical interfaces.

Hence, the proposed accompanying formalization and validation framework that supports precise specification of patterns will be used to prove a particular property and a particular parameter of a system. The resulting validation artifacts may mainly (1) complete the definitions of, and (2) provide formal semantics for the pattern interfaces and properties in the context of security. This way, validation artifacts are linked with the patterns and can be used for pattern integration and validation.

The domain refinement is applied during the formal validation process for the specification and validation of patterns. As it follows the MDE paradigm for system's design, using patterns on different levels of abstraction, it allows for integration into the system's design process, hence supports this process. To this end, the proposed representation takes into account the simplification and the enhancement of activities such as patterns selection/search, based on the classified properties, and pattern integration, based on a high level description of interfaces.

When developing a pattern by rewriting it using the SEPM modeling language and its accompanying formal SeMF model, integrated modeling and formal techniques will be applied to achieve an elegant correct-by-construction process. We now present an overview of our development process. Along this description, we will give the main keys to understand why our process is based on a general and constructive approach. The procedure described in the following is executed until the pattern subsystem is successfully validated against its provided security properties. This may include several iterations. Once the procedure is finished, the resulted SeMF model, including the pattern formal specification and the proof artifacts, can be transformed to an SEPM model, with some of the artifacts just

being appended to the model in the form of guidelines as documentation for the pattern usage. The procedure consists of the following steps:

- Step 1. Specify a pattern using the SEPM representation, as described in Section 4.2.
- Step 2. From SEPM metamodel to SeMF. The SEPM metamodel represents concepts of a security pattern at both domain-independent and domain-specific level. SeMF can express structural and behavioral computing systems in the form of agents, actions and security and dependability properties. These concepts can be used to represent similar concepts of the SEPM metamodel such as using actions to express functions calls of the pattern's interfaces, properties and assumptions to express pattern's properties etc. The transformation/mapping from an SEPM model to an SeMF model is generic and allows to transform each interface of the metamodel to a set of actions, a function call to an action with the corresponding signature, and the properties to the corresponding SeMF properties (auth, precede, conf etc, see Section 5.2). The level of details of the SeMF models follows the two levels of abstractions of the SEPM metamodel (DIPM and DSPM). For instance, the function call $send(C, ch(C, rnd_C, preMS_C, S, rnd_S), m, mac(preMS_C, \ldots, m))$ from the SEPM model is transformed to action $send\text{-}hs(C, tcp_j, SKmac(preMS_C, rnd_S^k), m, mac(preMS_C, rnd_S^k), m))$ as part of the SeMF model.
- Step 3. Validation-Proof process. Using the generated SeMF model (as a result of Step 2 or Step 4, see below) as input, we start the validation-proof process in the SeMF framework to produce a first assessment on the validity of the pattern specification against its security and dependability properties.
  (i). If there exists such a proof, and the formal specification is an extended version of the one generated from the semi-formal one (Step 2 or Step 4), the proof process stops, and the procedure is continued with Step 5.
  (ii). Otherwise, the procedure is continued with Step 4.
- Step 4. The proof and specification process continues (Finding issues). If the proof fails and reasons for this can be found, the description of interfaces and the properties have to be completed/modified. In Section 6.2, when trying to prove the first version of the DIPM pattern, we have already shown a proof that failed and gave rise to a change in the abstract interface specification. Another possible proof failure could be caused by modeling the server's public key by PuK instead of PuK_S, with the argument that there is only one server involved in our model. However, in the induction step of the proof, part (c), we need to deduce that the action of encrypting the premaster secret reveals this secret only to the server. This implies that the client does not use any other agent's public key for encryption which is expressed by Ass.14. However, not being able to distinguish the server's public key from any other public key that might exist in the system would not allow to express this assumption, hence the need to change the interface specification accordingly.
  - Check if the description of actions' signatures and their use in the specification of the basic SeMF properties are appropriately representing the formal definition. For instance, a function call from SEPM model may need an additional parameter.
  - Check if the formalized security properties are an appropriate refinement of the semantics of the formal ones.

After having regenerated the SeMF model, the procedure resumes with Step 3.

– Step 5. From SeMF to SEPM metamodel. To perform this transformation we require the user to analyze the proof and extract information about all the necessary interface's functions calls and security properties from actions' signatures and assumptions from the SeMF model. This information will be used to instantiate SEPM concepts to improve the corresponding SEPM model. Note, however, that this transformation is quite restricted and does not support full SeMF notations and expressiveness.

In the following, we will deal with examples on how to derive guidelines to cover the non-captured information, lost during the SeMF to SEPM transformation, to assist the developer in correctly using the validated patterns.

When using the pattern, an application developer will be concerned with the security requirements expressed in terms of the external interface, i.e. in terms of the function calls used by the application. So any formal proof needs to refer to these function calls. On the other hand, the solution described in a respective DSPM pattern must be modeled in terms of refined function calls. Stored in a repository, validated security patterns are then made available to be integrated into an MDE process to develop secure applications for various different domains. Beyond this, we store in the repository associated validation artifacts i.e. properties and assumptions. At this point, we propose a set of guidelines for how to correctly use the validated patterns.

Intuitively, we propose to handle the assumptions that the proof is based on as a set of *constraints* that need to be satisfied in order for the pattern and the solution it specifies to provide the proven pattern properties in terms of its interfaces. In other words, we execute the different steps of the proof and the related assumptions as requirements on the external and technical interfaces.

7.2 Revisiting the motivating example

In order to prove for example that the TLS handshake results in authentic shared secrets for both the client and the server, we need to assume that both own the authentic public key of the certification authority. We now exemplarily introduce some constraints derived from the assumptions used in the proof of the TLS Handshake that the application developer needs to verify in order to ensure that the pattern used indeed provides the required authenticity and trust properties:

– Implementation of sender and receiver (i.e. client and server) application entities. The sender application entity must be implemented adhering to assumption Ass.14 which states that the client does not encrypt the premaster secret with two different public keys (it might encrypt it twice using the same key). The receiver application entity must be implemented in compliance with assumption Ass.1, i.e. must not accept a message after having closed the respective channel.
– Key Handling. The HMAC algorithm works with shared secrets. These shared secrets must be deployed in such a way that they are only known to the sender and the receiver application entities (see Ass.12, Ass.6). Further, it must be assured that the receiver will not use the shared secret of the sender to compute

HMACs as expressed through Ass.9. This also means that the same shared secret must not be used for bi-directional transmissions.
– Random number generation. The sender application entity must ensure that it uses a random number as basis for the premaster secret and that the random number generator produces unpredictable random numbers (see e.g. Ass.4).

8 Synthesis and discussion

In this paper, we have proposed a new framework that enables the specification and validation of domain-independent security and dependability and their customization towards specific domains. In contrast to the informal representation of security patterns introduced in [17], we propose two additional and complementary types of representations: a semi-formal representation through metamodeling techniques and a rigorous formal representation through the formal Security Modeling Framework SeMF. However, we keep the template elements in the form of attributes and we deeply refine them by the definition of new concepts in order to fit with security engineering needs.

8.1 Recapitulation and perspectives

The DIPM pattern exhibits an abstract solution without specific knowledge on how the solution is implemented with regard to the application domain. Following an MDE process, the domain-independent model of a pattern is then refined towards a domain-specific level, taking into account domain artifacts, concrete elements such as mechanisms to use, devices that are available, etc. Consequently, a security pattern at domain-specific level contains the respective information. These two levels of abstractions are captured using new concepts related to the different kind of knowledge described by the pattern, rather than using existing software constructs.

In our work, we used the MDE philosophy. We do not use the software concepts (object or component constructs) recommended by Model-Driven Architecture (MDA), for example. However, the SEPM language is capable to target specific software modeling languages such as those recommended by MDA, using model transformation techniques. Regarding the well known MDA levels (PIM and PSM), there is an overlap between these levels and our two abstraction levels. For example, in the Intelligent Transport Systems (ITS) domain, the ISO/IEC 15118 highly recommends to use TLS for ensuring security properties, and we can find different platforms supporting the respective DSPM pattern.

The documentation and implementation prototype developed in the course of our work includes an EMF editor that can be used to specify security patterns and allows the user to publish the patterns in a repository and to retrieve them again. Further the editor provides a number of guidelines that will facilitate (1) the populating of the repository with further security patterns, and (2) the transformation of the security patterns into platform dependent specifications.

Mainly, the results of our work contribute to the following:

1. Security pattern modeling framework to get a common representation of patterns for several domains (1) to capture the essence of the pattern, (2) to provide enough detail to enable the usability of the pattern by a non-specialist, (3) to provide sufficient information to be validated, (4) to provide sufficient explanation to enable the usability of the pattern in domains other than the one in which the pattern was defined.
2. Repository of integrated models (security patterns, resource models, trust models, engineering process models,...).
3. Repository access tool allowing application designers to capitalize on the MDE even if they are not experts in modeling.
4. Patterns are formally validated when they are saved in the repository. This formal validation supports the application developer when integrating the pattern and is thus an important means to guarantee the correctness of this step.
5. When a pattern has been formally validated, implementations with automatically derived guidelines for platform dependent implementation of the patterns will be available.
6. In order to integrate a pattern in a system (application), some significant additional information about the pattern is required, for instance, the interfaces and their requirements. The goal is to capture how the pattern interacts with the system (that may include other patterns), especially when dealing with software and hardware components.

On the other hand, the proposed pattern and property specification languages supports the specification of security patterns and their related properties, mainly security and resource properties. In addition to that, the languages may be used to specify other kind of patterns. For instance, memory, concurrency and distributed patterns [64].

In a wider scope, new requirements and their related artifacts can be addressed. For instance integration, composition, analysis, simulation and instantiation are important needs that we can consider in our framework to serve for systematic construction of large complex systems with multiple concerns.

8.2 Revisiting the use case

Finally, the SCP pattern studied in this paper is very relevant for electric vehicle connected to smart grids. Indeed, due to normative restrictions, TLS based communication has to be implemented. Since this application is not dedicated to power control systems, car makers do not use usual operating systems such as OSEK. For telematic and infotainment applications, Linux is mainly chosen. Finally, this kind of application is written in Java or C. In this case study, Linux and C language is used. For the security functions, the OpenSSL library is chosen.

The SCP pattern defines internal and external interfaces. The internal interfaces are provided by the OpenSSL library. The external interface is related to ISO/IEC 15118 and defines *send* and *receive* functions. In both methods, the prototype is composed of client and server addresses, a MAC and a message. From the point of view of the ISO/IEC 15118 standard, the charging spot and the electric vehicle are respectively the server and the client.

The ISO/IEC 15118 standard only defines different messages written in XML. In other words, the *send* and *receive* functions of the pattern are not specified at

the standard level. The developer can directly use the external interface of the pattern and focus on the message definition. Therefore, the development cost and the necessary expert knowledge is reduced due to the reusability of the pattern and thus the expert knowledge. Moreover, security properties are ensured by formal validation in the both cases.

8.3 Integrating pattern and application development

The ultimate goal of our work is to promote a correct-by-construction pattern-based system and software security engineering approach. Depending on the level of details the system developer starts with, he/she may use either DI or DS patterns. Here we discuss briefly the interaction with and the impact on the application development process regarding these two possibilities.

In the overall process there are three sets of the independent parallel processes: (i) an overall functional architecture and platform development process responsible for delivering new system functionality and platforms, (ii) pattern development processes which deliver a set of patterns that provide security and dependability functions, and (iii) the product development process which is basically an integration process.

In the very early requirement and design phases, one important activity is to analyze the possibility of realizing the solutions that will meet system requirements. In a pattern-based approach this implies that it is necessary to analyze whether these requirements can be fulfilled by available patterns. This means that the system developers must be aware of DI and DS patterns to use. Some extra activities are required when system developers fail to identify appropriate DI and/or DS patterns.

A system developer having the necessary expertise can just use the pattern design process introduced in this paper to develop their own solutions. This also holds in case there is no DS pattern available that specifies the security solution to be used.

If an appropriate DI pattern is not available, one possibility that is compliant with our pattern-based approach is to negotiate the requirements and modify them to be able to use the existing DS patterns.

If an appropriate DI pattern as well as DS patterns are available, the system developer needs to identify those DS patterns that indeed represent a refinement of the DI pattern. There are again various possibilities. The system developer herself can verify that there is a homomorphism preserving the DI pattern's security property and mapping the DS model onto the DI model. This may reduce to checking some conditions to hold for the DI and DS model, respectively. To find such conditions depends on the security property in question and is subject to future work. Another possibility is to adapt an approach introduced by the project SERENITY (see [65]) and include in the DS pattern another artifact that specifies a DI pattern's interfaces and their refinement towards those of the DS pattern.

9 Conclusion

Application developers usually do not have expertise in security and dependability. Hence capturing and providing this expertise by way of security patterns has become an area of research in the last years. Security patterns shall enable the development of secure and dependable applications while at the same time liberating the developer from having to deal with the technical details. Model-driven engineering (MDE) provides a very useful contribution for the design of secure and trusted systems, since it bridges the gap between design issues and implementation concerns. Hence Security pattern integration has to be considered at some point in the MDE process.

In this paper, we have proposed an approach for security pattern modeling and validation that follows the MDE paradigm. Our approach is based on metamodeling techniques that allow to specify and validate security patterns at different levels of abstraction. A security pattern at domain-independent level allows the application developer to identify security requirements and select a respective abstract solution without specific knowledge on how the solution is designed and implemented. Thus a DIPM pattern can easily be integrated into the overall abstract system specification. Following the MDE process, the system model is then refined towards a domain-specific level, taking into account domain artifacts, concrete elements such as mechanisms to use, devices that are available, etc. Consequently, a security pattern at domain-specific level contains the respective information.

Pattern validation follows these two abstraction levels, i.e. we validate a DIPM pattern and possible DSPM instantiations independently. However, the additional final validation step proves that the latter is indeed a refinement of the former which in turn proves that the overall application system indeed satisfies the security requirements initially specified by the application developer. This process may significantly reduce the cost of engineering the system, since it enables to address security issues early in the system development process while at the same time relieving the developer from the technical details.

The proposed integrated modeling and formal framework is illustrated through an example of a secure communication pattern. By this, we point out the feasibility and effectiveness of the proposed specification and design frameworks. The SEPM modeling language provides an environment where a designer can model patterns at DIPM and DSPM and the security modeling framework SeMF provides an environment for the validation of a pattern at DIPM and DSPM levels against its security and dependability properties.

Furthermore, we walk through a prototype as a set of Eclipse plug-ins[1], as a proof of concept. All used metamodels are specified using Eclipse Modeling Framework (EMF). The approach presented here has been evaluated in the context of the TERESA project[2] resulting in the development of a repository with more than 30 security patterns. The conformance validation tool embedded in the tool suite helps designers to rectify the problems in models and correct them with adequate measures if such problems occur, while the manually performed formal validation gives rise to feedback in the form of guidelines for the pattern application.

[1] http://www.semcomdt.org

[2] http://www.teresa-project.org/

The next step of this work consists in defining a correct-by-construction pattern-based security engineering process. It aims at providing the correct-by-construction integration of a design pattern into an application while offering a certain degree of liberty to the designer using it. In order to be able to validate the integration, we must have a formal specification of the pattern, i.e., its properties, constraints and related validation artifacts, as input to the pattern-based development process. Yet an important task remains to be performed when integrating a security pattern into an application: It has to be ensured that the assumptions used for proving the correctness of the DSPM pattern are indeed satisfied by the particular environment of the application. While so far these assumptions are specified in terms of formal security properties, future steps consist in transforming them into environment constraints through external model libraries (i.e., security and resource properties).

Another objective for the near future is to provide automated tool support for pattern-based development, preferably based on a widely known and accepted model-based approach in industry such as UML [66]. For that, we plan to investigate the possibility to transform our design artifacts into UML and their corresponding validation artifacts into OCL [53]. More precisely, we will on the one hand specify sufficient conditions of language homomorphisms for preserving further security properties (e.g. trust). On the other hand, we will investigate the transformation of these conditions into other modeling domains. For example, we will refine, where possible, security properties representing assumptions for patterns into very basic properties and formulate these as OCL constraints referring to a UML model of specific application domains.

References

1. J. Yoder, J. Barcalow, Architectural Patterns for Enabling Application Security, in: Conference on Pattern Languages of Programs (PLoP 1997), 1998.
2. M. Schumacher, Security Engineering with Patterns - Origins, Theoretical Models, and New Applications, Vol. 2754 of Lecture Notes in Computer Science, Springer, 2003.
3. B. Selic, The Pragmatics of Model-Driven Development, IEEE Software 20 (5) (2003) 19–25.
4. C. Atkinson, T. Kühne, Model-Driven Development: A Metamodeling Foundation, IEEE Softw. 20 (5) (2003) 36–41.
5. A. V. Uzunov, E.B.Fernandez, K. Falkner, Securing distributed systems using patterns: A survey, Journal of Computers & Security 31 (5) (2012) 681–703.
6. B. Hamid, J. Geisel, A. Ziani, J. Bruel, J. Perez, Model-Driven Engineering for Trusted Embedded Systems Based on Security and Dependability Patterns, in: SDL Forum, 2013, pp. 72–90.
7. The TLS Protocol Version 1.2, rfc5246 (2008).
8. B. Hamid, S.Gurgens, C. Jouvray, N. Desnos, Enforcing S&D Pattern Design in RCES with Modeling and Formal Approaches, in: J. Whittle (Ed.), ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS), Vol. 6981, Springer, 2011, pp. 319–333.
9. V. D. Giacomo, M. Felici, V. Meduri, D. Presenza, C. Riccucci, A. Tedeschi, Using Security and Dependability Patterns for Reaction Processes, in: Proceedings of the 2008 19th International Conference on Database and Expert Systems Application, IEEE Computer Society, Washington, DC, USA, 2008, pp. 315–319.
10. N. Yoshioka, H. Washizaki, K. Maruyama, A survey of security patterns, Progress in Informatics (5) (2008) 35–47.
11. F. Daniels, The Reliable Hybrid Pattern: A Generalized Software Fault Tolerant Design Pattern, in: Proc. of the Pattern Language of Programs (PLoP'97), 1997.

12. M. Tichy, , D. Schilling, H. Giese, Design of self-managing dependable systems with UML and fault tolerance patterns, in: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, WOSS '04, ACM, New York, NY, USA, 2004, pp. 105–109.
13. A. Maña, E. Fernandez, J. Ruiz, C. Rudolph, Towards Computer-oriented Security Patterns, in: The 20th International Conference On Pattern Languages Of Programs PLoP13, 2013.
14. A. L. Guennec, G. Sunyé, J.-M. Jézéquel, Precise modeling of design patterns, in: In Proceedings of UML'00, Springer-Verlag, 2000.
15. D.-K. Kim, R. France, S. Ghosh, E. Song, A UML-based meta-modeling language to specify design patterns, in: Patterns, Proc. Workshop Software Model Eng. (WiSME) with Unified Modeling Languages, 2004.
16. E. Gasparis, J. Nicholson, A. H. Eden, LePUS3: An Object-Oriented Design Description Language, in: In: Gem Stapleton et al. (eds.) DIAGRAMS, LNAI 5223, 2008, pp. 364–367.
17. E. Gamma, R. Helm, R. E. Johnson, J.Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
18. B. P. Douglass, Real-time UML: Developing Efficient Objects for Embedded Systems, Addison-Wesley, 1998.
19. D. Mapelsden, J. Hosking, J. Grundy, Design pattern modelling and instantiation using DPML, in: CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific, Australian Computer Society, Inc., 2002, pp. 3–11.
20. D. Serrano, A. Mana, A.-D. Sotirious, Towards Precise and Certified Security Patterns, in: Proceedings of 2nd International Workshop on Secure systems methodologies using patterns (Spattern 2008), IEEE Computer Society, 2008, pp. 287–291.
21. G. E. boussaidi, H. Mili, Representing and applying design patterns: what is the problem?, in: Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MODELS), Springer, 2005, pp. 186–200.
22. A. Maña, E. Damiani, S. Gürgens, G. Spanoudakis, Extensions to Pattern Formats for Cyber Physical Systems, in: Proceedings of the 31st Conference on Pattern Languages of Programs (PLoP 14), 2014.
23. J. Jürjens, UMLsec: Extending UML for Secure Systems Development, in: Proceedings of the 5th International Conference on The Unified Modeling Language, UML '02, Springer-Verlag, London, UK, 2002, pp. 412–425.
24. T. Lodderstedt, D. Basin, J. Doser, SecureUML: A UML-Based Modeling Language for Model-Driven Security, in: Proceedings of the 5th International Conference on The Unified Modeling Language, UML '02, Springer-Verlag, London, UK, 2002, pp. 426–441.
25. B. Hamid, A. Radermacher, A. Lanusse, C. Jouvray, S. Gérard, F. Terrier, Designing fault-tolerant component based applications with a model driven approach, in: IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2008), Vol. 5287 of Lecture Notes in Computer Science, Springer, 2008, pp. 9–20.
26. D. Basin, J. Doser, T. Lodderstedt, Model driven security: From UML models to access control infrastructures, ACM Transactions on Software Engineering and Methodology (TOSEM) 5 (1) (2006) 39–91.
27. D. Basin, M. Clavel, J. Doser, M. Egea, Automated Analysis of Security-Design Models, Information and Software Technology 51 (2009) 815–831.
28. J. Jensen, M. G. Jaatun, Security in Model Driven Development: A Survey, in: Proceedings of the 2011 Sixth International Conference on Availability, Reliability and Security. ARES '11, IEEE Computer Society,, 2011, pp. 704–709.
29. L. Lucio, Q. Zhang, P. H. Nguyen, M. Amrani, J. Klein, H. Vangheluwe, Y. L. Traon, Advances in Model-Driven Security, Advances in Computers 93 (2014) 103–152.
30. J. McDonald, N. Oualha, A. Puccetti, A. Hecker, F. Planchon, Application of EBIOS for the risk assessment of ICT use in electrical distribution sub-stations, in: In PowerTech (POW- ERTECH), IEEE, 2013, pp. 1–6.
31. F. Braber, I. Hogganvik, M. Lund, K. Stølen, F. Vraalsen, Model-based security analysis in seven steps - a guided tour to the CORAS method, BT Technology Journal 25 (1) (2007) 101–117.
32. T. Srivatanakul, J. A. Clark, F. Polack, Effective security requirements analysis: HAZOP and use cases, Information Security, Lecture Notes in Computer Science (3225) (2004) 416–427.
33. B. Schneier, Attack Trees, Modeling Security Threats, Dr. Dobb's Journal.
34. M. Rodano, K. Giammarc, A Formal Method for Evaluation of a Modeled System Architecture, Procedia Computer Science 20 (2013) 210–215.

35. C. E. Landwehr, Formal Models for Computer Security, ACM Computing Surveys 13 (1981) 247–278.
36. P. Devanbu, S. Stubblebine, S. S. Premkumar, T. Devanbu, Software engineering for security - a roadmap, in: Proceedings of the Conference on The Future of Software Engineering, ICSE '00, ACM, 2000, pp. 227–239.
37. A. J. Lee, J. P. Boyer, L. E. Olson, C. A. Gunter, Defeasible security policy composition for web services, in: Proceedings of the fourth ACM workshop on Formal methods in security, ACM, 2006, pp. 45–54.
38. G. Bruns, D. S. Dantas, M. Huth, A simple and expressive semantic framework for policy composition in access control, in: Proceedings of the 2007 ACM workshop on Formal methods in security engineering, ACM, 2007, pp. 12–21.
39. G. Bruns, M. Huth, Access control via belnap logic: Intuitive, expressive, and analyzable policy composition, ACM Transactions on Information and System Security (TISSEC) 14 (1) (2011) 1–27.
40. M. Burrows, M. Abadi, R. Needham, A Logic of Authentication, ACM Transactions on Computer Systems 8.
41. L. Paulson, Proving Properties of Security Protocols by Induction, Tech. Rep. 409, Computer Laboratory, University of Cambridg (1996).
42. G. Lowe, An Attack on the Needham-Schroeder Public-Key Protocol, Information Processing Letters 56 (3) (1995) 131–133.
43. B. Roscoe, P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, The modelling and Analysis of Security Protocols, Addison Wesley, 2000.
44. AVISPA, The HLPSL tutorial, a beginner's guide to modelling and analysing internet security protocols, http://www.avispa-project.org.
45. Y. Chevalier, L. Compagna, J. Cuellar, D. P. Hankes, J. Mantovani, S. Mdersheim, L. Vigneron, A high level protocol specification language for industrial security-sensitive protocols, in: Workshop on Specification and Automated Processing of Security Requirements (SAPS 2004), 2004.
46. S. Gürgens, C. Rudolph, Security analysis of efficient (un-)fair non-repudiation protocols, Formal Asp. Comput. 17 (3) (2005) 260–276.
47. S. Gürgens, C. Rudolph, D. Scheuermann, M. Atts, R. Plaga, Security evaluation of scenarios based on the TCG's TPM specification, in: J. Biskup, J. Lopez (Eds.), Computer Security - ESORICS 2007, Vol. 4734, 2007.
48. A. Fuchs, and S. Gürgens and L. Apvrille and G. Pedroza, D3.4.3 - On-Board Architecture and Protocols Verification, Tech. rep., EVITA-Project (2010).
49. S. Gürgens, P. Ochsenschläger, C. Rudolph, On a formal framework for security properties, International Computer Standards & Interface Journal (CSI), Special issue on formal methods, techniques and tools for secure and reliable applications 27 (5) (2005) 457–466.
50. U. Zdun, P. Avgeriou, Modeling Architectural Patterns Using Architectural Primitives, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, ACM, New York, NY, USA, 2005, pp. 133–146.
51. A. Ziani, B. Hamid, S. Trujillo, Towards a Unified Meta-model for Resources-Constrained Embedded Systems, in: 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), IEEE, 2011, pp. 485–492.
52. OMG, A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems,Beta 2, http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf (June 2008).
53. OMG, OCL 2.2 Specification, http://www.omg.org/spec/OCL/2.2 (2010).
54. A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions on Dependable and Secure Computing 1 (2004) 11–33.
55. M. Schumacher, E. Fernandez, D. Hybertson, F. Buschmann, Security Patterns: Integrating Security and Systems Engineering, John Wiley & Sons, 2005.
56. A. Fuchs, S. Gürgens, C. Rudolph, A Formal Notion of Trust – Enabling Reasoning about Security Properties, in: Preceedings of Fourth IFIP WG 11.1 International Conference on Trust Management, Vol. 321, Springer, 2010, pp. 200–215.
57. A. Fuchs, S. Gürgens, N. Lincke, D. Weber, D5.2 & d5.4 - Application of formal validation to relevant examples & Guidelines for platform dependent implementation v1, Tech. rep., TERESA-Project (2012).

58. S. Gürgens, P. Ochsenschläger, C. Rudolph, Parameter confidentiality, in: Informatik 2003 - Teiltagung Sicherheit, Gesellschaft für Informatik, 2003.
59. S. Gürgens, P. Ochsenschläger, C. Rudolph, Authenticity and provability - a formal framework, in: Infrastructure Security Conference InfraSec 2002, Vol. 2437, 2002, pp. 227–245.
60. S. Gürgens, P. Ochsenschläger, C. Rudolph, Abstractions preserving parameter confidentiality, in: European Symposium On Research in Computer Security (ESORICS 2005), 2005, pp. 418–437.
61. A. Fuchs, S. Gürgens, D05.1 Formal Models and Model Composition, Tech. rep., ASSERT4SOA Project (2011).
62. A. Fuchs, S. Gürgens, D5.1v2.5 - Formal Validation Approach, Tech. rep., TERESA-Project (2012).
63. A. Fuchs, S. Gürgens, C. Rudolph, Formal Notions of Trust and Confidentiality - Enabling Reasoning about System Security, Journal of Information Processing 19 (2011) 274–291.
64. D. B. Powel, Real-time design patterns : robust scalable architecture for real-time systems, The Addison-Wesley object technology series, Addison-Wesley, Boston, San Francisco, Paris, 2003.
65. L. Compagna and P. El Khoury and R. Harjani and C. Kloukinas and K. Li and A. Maña and A. Muñoz and G. Pujol and J.F. Ruiz and A. Saidane and D. Serrano and S.K. Sinha and G. Spanoudakis, A5.D2.5 - Patterns and Integration Schemes Languages, Tech. rep., SERENITY-Project (2008).
66. OMG, OMG Unified Modeling Language (OMG UML), Superstructure, http://www.omg.org/spec/UML/2.2/Superstructure (February 2009).

Appendix

Example of establishing a secure TLS channel. In the following, we present one possible and desired sequence of actions of a client C and server S that result, as we have shown in Section 6.4, in establishing a secure TLS channel. Note that the actions need not occur in exactly this order, other orders are possible to achieve the same result. Further, some of the actions may not be needed at all or may be implemented differently within particular implementations. Keys to be used for encryption for example may be passed from the calling application or may be accessible directly. Since our proof focuses on authenticity of the client we disregard all actions that are concerned with server authentication.

$genRnd(C, rnd_C)$	The client generates a random number. We do not model implementation dependent details such as how the client (and vice versa the server) links random numbers to the server (client) to establish a channel with and simply assume this link to be provided.
$send\text{-}hs(C, tcp_j, rnd_C, info(C))$	The client sends its random number and some information about the possible algorithm and key lengths etc. to the server, using a particular TCP channel tcp_j established for (not yet secure) communication with the server.
$recv\text{-}hs(S, tcp_j, rnd_C, info(C))$	The server receives the client's message on the TCP channel tcp_j.
$genRnd(S, rnd_S^k)$	The server generates a random number to be used subsequently to establish a session with client C.
$getCert(S, cert(PrK_{CA}, PuK_S, ID_S))$	The server retrieves its certificate for its public key.
$send\text{-}hs(S, tcp_j, rnd_S^k, cert(PrK_{CA}, PuK_S, ID_S), info(S))$	The server sends its certificate and information about the possible algorithms and a request for client authentication to the client on TCP channel tcp_j.
$recv\text{-}hs(C, tcp_j, rnd_S^k, cert(PrK_{CA}, PuK_S, ID_S), info(S))$	The client receives the server's message on TCP channel tcp_j.
$getKey(C, PuK_{CA})$	The client retrieves the CA's public key.
$verifyCert(C, PuK_{CA}, cert(PrK_{CA}, PuK_S, ID_S))$	The client verifies the server's certificate with the CA's public key and extracts the server's public key.

$genRnd(C, preMS_C)$	The client generates the pre-master secret.
$encrypt(C, PuK_S, preMS_C, cipher(PuK_S, preMS_C))$	The client encrypts the pre-master secret using the server's public key.
$establCh(C, SKmac(rnd_C, preMS_C, rnd_S^k))$	The client uses the three exchanged random numbers to generate its session key for MAC generation.
$send\text{-}hs(C, tcp_j, cipher(PuK_S, preMS_C))$	The client sends the encrypted premaster secret to the server.
$finished(C, tcp_j, HMAC(preMS_C, label(C), hash(m1, m2, m3)))$	The client signals having finished the TLS Handshake to the server. The message is composed of an HMAC including the premaster secret and the hash of all Handshake messages excluding this finished message (denoted by $m_1, \ldots, m_3$).
$recv\text{-}hs(S, tcp_j, cipher(PuK_S, preMS_C))$	The server receives the encrypted premaster secret.
$decrypt(S, PrK_S, cipher(PuK_S, preMS_C))$	The server decrypts the premaster secret.
$establCh(S, SKmac(rnd_C, preMS_C, rnd_S^k))$	The server uses the three exchanged random numbers to generate its session key for MAC verification.
$finished(S, tcp_j, HMAC(preMS_C, label(S), hash(m1, m2, m3)))$	The server signals having finished the TLS Handshake to the client. The message is composed of an HMAC including the premaster secret and the hash of all Handshake messages excluding this $finished$ message.

Proof of SeBB.3

Proof Trivially, if for a specific sequence ω of actions, all agents in $\mathbb{P} \setminus who$ consider all values of par possible, then all agents in the smaller set $\mathbb{P} \setminus (who \cup \{Q\})$ still consider all values of par possible, i.e. $restricted\text{-}conf(\mathcal{A}(par), par, who, \{\omega\})$ implies $restricted\text{-}conf(\mathcal{A}(par), par, who \cup \{Q\}, \{\omega\})$. If ω is continued by encrypting par with an RSA public key, then the security property provided by RSA encryption ensures that nobody but the owner of the public key can gain knowledge about the value of par, hence $restricted\text{-}conf(\mathcal{A}(par), par, who \cup \{Q\}, \{\omega a\})$ holds.

Proof of SeBB.4

Proof Let $\omega \in B$ and $verifyMac(Q, key, m, mac(key, m)) \in alph(\omega)$. Then it follows that exists $v \in \Phi(\{s\}, T), x, z \in \Sigma^*$ with $\omega = uvx$ and $verifyMac(Q, key, m, mac(key,$

$m)) \in alph(v)$. Further, by the nature of the MAC algorithm, a MAC verification action is always preceded by a MAC generation action, hence $genMac(R, key, m, mac(key, m)) \in alph(\omega)$ for some $R \in \mathbb{P}$. Since the MAC generation action in turn is preceded by the start action s of the phase class, it follows $genMac(R, key, m, mac(key, m)) \in alph(v)$. Further, since s is the only phase class start action in ω, there is no other word v' contained in ω with $verifyMac(Q, key, m, mac(key, m)) \in alph(v')$. Finally, the key key is confidential for P and Q during this phase class, thus $R \in \{P, Q\}$, and since Q never itself generates a MAC before having verified it, it follows that P has done so and, therefore, the assertion.