Security Asset Elicitation for Collaborative Models

Maria Vasilevskaya[1]
maria.vasilevskaya@liu.se

Linda Ariani Gunawan[2]
gunawan@item.ntnu.no

Simin Nadjm-Tehrani[1]
simin.nadjm-tehrani@liu.se

Peter Herrmann[2]
herrmann@item.ntnu.no

[1]Department of Computer and Information Science, Linköping University, Linköping, Sweden
[2]Department of Telematics, Norwegian University of Science and Technology, Trondheim, Norway

ABSTRACT

Building secure systems is a difficult job for most engineers since it requires in-depth understanding of security aspects. This task, however, can be assisted by capturing security knowledge in a particular domain and reusing the knowledge when designing applications. We use this strategy and employ an information security ontology to represent the security knowledge. The ontology is associated with system designs which are modelled in collaborative building blocks specifying the behaviour of several entities. In this paper, we identify rules to be applied to the elements of collaborations in order to identify security assets present in the design. Further, required protection mechanisms are determined by applying a reasoner to the ontology and the obtained assets. We exemplify our approach with a case study from the smart metering domain.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.10 [**Software Engineering**]: Design—*Methodologies*

General Terms

Design, Security

Keywords

Model, domain, assets, security, ontology, smart grid, tools

1. INTRODUCTION

Model-driven engineering (MDE) for security deals with development of security-enhanced systems at the early phase. Principles of MDE enable us to describe functionality and security enforcement mechanisms as separate concerns. This in turn allows the system and security engineers to work independently. However, the decision to select an appropriate enforcement mechanism and to apply it, still requires additional knowledge about the security domain from a system engineer.

The SecFutur project [4] addresses this problem by developing a process that leverages domain-specific modelling. The process includes two main phases: (I) capturing security knowledge (by a security engineer) and (II) applying it to a functional system specification in a certain domain (by a system engineer).

In this paper we refine the phase of applying the captured security knowledge. This phase consists of the following steps: (1) The system model is analysed to elicit present assets using a set of rules. (2) A system engineer enumerates a subset among all elicited assets to be protected. (3) For each selected asset from step 2, the captured (domain-specific) security knowledge is consulted to obtain available security properties. (4) A system engineer chooses a subset among all security properties, identified in step 3, according to system security requirements derived, e.g., from threat analysis. (5) Available enforcement mechanisms satisfying the chosen security properties are inferred from the captured (domain-specific) security knowledge.

The outcome of these steps is a list of enforcement mechanisms, which are subsequently integrated into a system model. In this paper we elaborate on steps 1, 3, and 5. In particular, we use the model-based engineering technique SPACE [10] and its tool-set Arctis [13] to provide step 1. Furthermore, we support the above steps by extending existing tools with new ones.

In the following, we describe SPACE and Arctis by means of a case study. Sect. 3 outlines the phase of capturing security knowledge in a particular domain. As a solution for step 1, we present an asset elicitation technique in Sect. 4. Thereafter, the implementation of steps 3 and 5 is explained in Sect. 5. Sect. 6 sketches the developed tool support. We end this paper with a summary of some related works followed by concluding remarks.

2. COLLABORATIVE MODELS

To design both functional and security aspects, we employ collaboration-oriented models that are based on UML activities [16]. Distributed system specifications are composed from *collaborative building blocks* [10] describing local behaviour within components as well as their interaction. This specification style is suitable for specifying security protocols because such mechanisms are inherently collaborative.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MDSec'12 , September 30 2012, Innsbruck, Austria
Copyright 2012 ACM 978-1-4503-1806-8/12/09 ...$15.00.

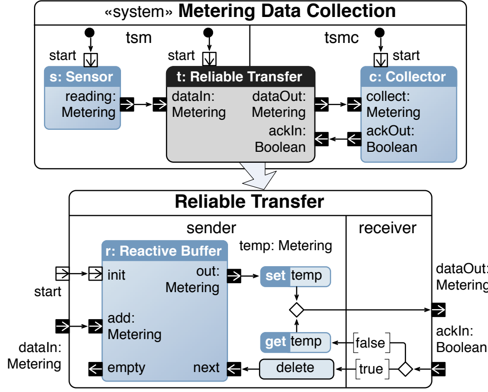


Figure 1: Metering data collection scenario

Moreover, since a large part of an application specification (on average about 70% [11]) is taken from reusable building blocks, the specification style also facilitates rapid system development. The models have a formal semantics [12] which enables verification of relevant properties, e.g., the correct integration of building blocks into activities, by the model checker included in the tool-support Arctis [13]. Furthermore, as will be shown later in Sect. 4 the semantics enables the formulation of rules for security asset elicitation.

The model of our case study from the smart metering domain is depicted in the upper part of Fig. 1. It is an application for collecting metering data stored in a Trusted Sensor Module (TSM) by a TSM Collector (TSMC). These two physical components are modelled by the partitions *tsm* and *tsmc* of the activity diagram. The delivery of the metering data is specified by the collaborative block *t: Reliable Transfer*. In contrast to all other activity node types which are mapped to exactly one partition, these collaborative blocks span across several ones. The rest of the system behaviour is modelled by two local building blocks, which are located on a single partition each. Block *s: Sensor* models the electricity meter device which periodically reports electricity consumption. Block *c: Collector* encapsulates the behaviour of storing the metering data. The blocks contain pins at their frames which enable to link them by activity edges with other nodes. Each block refers to an activity diagram that describes a detailed internal behaviour as shown in the lower part of Fig. 1 for block *t*.

Due to its Petri net-like semantics, an activity models a behaviour as control and object flows of tokens along its nodes and edges. A system start is marked by a token flowing from each of the initial nodes (•). In the upper part of Fig. 1, it means blocks *s*, *t*, and *c* are started. When metering data is available on the *tsm* entity, its value, which is encapsulated as an object of type *Metering*, is carried by a token emitted from the sensor block via pin *reading* and forwarded to the transfer block through pin *dataIn*. As shown in the lower part of the figure, block *Reliable Transfer* contains block *r: Reactive Buffer* used to store metering data temporarily, when there is other data being sent but not yet acknowledged. If data is received (via pin *add*) when the buffer is empty, it is emitted (via pin *out*) immediately; otherwise it is buffered. Subsequent data, if any, is retrieved when a token flowing into pin *next*. A token carrying metering data flows from pin *out* of block *r* and passes operation

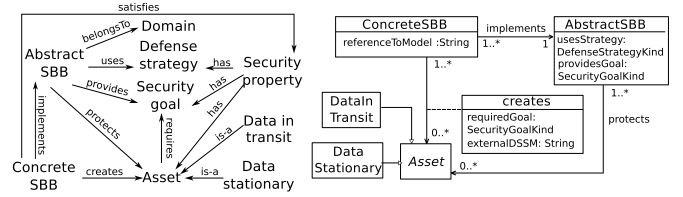


Figure 2: Security ontology (left) and its UML representation (right)

set temp which stores a copy of the data to variable *temp* as well as a merge node (◇). Thereafter, the token is sent to the other entity as depicted by the edge crossing the partition border. As shown in the upper part of Fig. 1, block *c* receives the data through pin *collect* and emits a token via *ackOut* containing an acknowledgment, which is, thereafter, forwarded to block *t*. The lower part of the figure depicts that the token reaches a decision node (◇) with two labelled outgoing edges. The edge labelled with *false* is followed when the measurement is not approved by the *tsmc*, i.e., it has not passed a validation test. Here, the previous metering value is retrieved from variable *temp* and resent. The *true* edge means a successful transfer of metering data. In this case, operation *delete*, which removes the value of variable *temp*, is called and consecutive data is obtained from the buffer.

3. CAPTURING SECURITY KNOWLEDGE

As already mentioned, our approach aims to support engineers who are not security experts in developing protected systems. To achieve this goal, security-related knowledge in a particular application domain, which is typically possessed by security experts, needs to be captured, stored, and presented to enable its application and reuse. To represent security concepts and their relationships, we employ ontology technologies. Since most developers are familiar with the UML [16], we use a class diagram to express a security ontology. Hence, security knowledge in a particular domain is essentially an instance of a class diagram, i.e., an object diagram. We call it *Domain-Specific Security Model* (DSSM) and store it in a library for reuse. A DSSM, essentially, extends the concepts in our ontology with rules and assertions. Thus, we can reason on a DSSM and obtain from it security relevant information (see Sects. 5 and 6).

Herzog et al. [8] defined an extensive information security ontology, which is suitable for our approach since it uses assets as its core concept. We adapted this ontology to accommodate protection mechanisms that are modelled as *Security Building Blocks* (SBBs). Our ontology has three basic concepts in common with Herzog et al. [8], namely, asset, security goal, and defense strategy (see the left part of Fig. 2). Within a system model, *assets* are objects of value to be protected. There are two types of assets, i.e., *stationary assets* resting on a physical component, and *assets in transit*, which refer to objects being communicated among several components. A *security goal* like confidentiality is achieved when an asset is protected with a countermeasure. *Defense strategy* describes how a countermeasure is applied, e.g., prevention or detection.

We introduce *Abstract SBB* to model general countermeasures (e.g., cipher) and *concrete SBB* for their implementa-

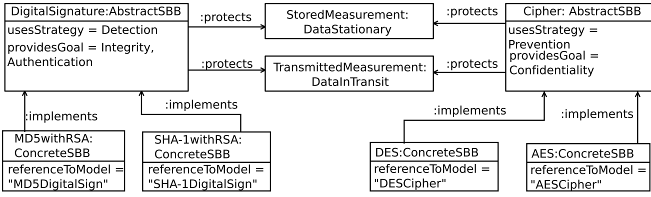


Figure 3: A fragment of the metering DSSM

tions (e.g., AES). An abstract SBB *protects* an asset by *using* a defense strategy, and therefore, *provides* a particular security goal. A concrete SBB *implements* an abstract SBB and may *create* other assets, e.g., an AES key when AES cipher is used. In addition to SBBs, we add two new concepts, namely *domain* and *security property*. The *domain* concept represents application domains. Each abstract SBB *belongs* to a domain. *Security property* is the core concept in our ontology aggregating the three notions of asset, security goal, and defense strategy. A concrete SBB *satisfies* a security property. Note that we do not focus on the concept of threats since enforcing a security property inherently works against any threat that would violate that security property.

The representation of the ontology in an UML class diagram is depicted in the right part of Fig. 2. The four elements asset, abstract SBB, concrete SBB, data stationary, and data in transit are directly mapped to corresponding elements in the UML representation. So do relations *implements* and *protects*. The *is-a* relation is modelled as the generalisation link. Instances of security goal and defense strategy of the ontology are represented as enumerations (not shown in Fig. 2). The *uses* and *provides* relations and their related concepts are mapped as the *usesStrategy* and *providesGoal* attributes of the class *AbstractSBB*. The *creates* relation is represented as an association class, whose attribute *requiredGoal* implements the relation *requires* from the ontology. The second attribute of the class (i.e., *externalDSSM*) refers to another domain if a created asset needs protection provided by a concrete SBB from another domain. The name of an object diagram (i.e., DSSM) expresses our domain concept, since we can say that all abstract SBBs defined within a certain DSSM *belong to* this domain. The security property concept of the ontology is essentially represented by the triple of asset, security goal, and defense strategy. Therefore, we can directly extract this information from DSSM without introducing any dedicated UML representation for it. Finally, a new attribute *referenceToModel* of the class *ConcreteSBB* refers to the Arctis collaborative building block that implements the corresponding protection mechanisms.

A fragment of the metering DSSM associated with our collection scenario (Fig. 1) is shown in Fig. 3. This association is done based on matching of system and security knowledge domains. This DSSM contains two assets: *StoredMeasurement* (i.e., data stationary) and *TransmittedMeasurement* (i.e., data in transit). The assets are protected by two abstract SBBs, namely, *Cipher* and *DigitalSignature*. *Cipher* provides confidentiality as a security goal, while *DigitalSignature* maintains integrity and data authenticity. *DES* and *AES* are concrete SBBs that implement *Cipher*. The other two objects are concrete SBBs for *DigitalSignature*.

4. ASSET ELICITATION

We define asset elicitation as identification of assets within a system model and their matching assets described in an associated DSSM. Rules in Sect. 4.1 are applied (as explained in Sect. 4.2) to a collaborative-based system model to identify available assets in the specification and to classify the assets according to the ontology described in the previous section. The application of these rules is automated and presented as *asset analyser* in Sect. 6.

4.1 Rules for Assets Identification

According to the SPACE semantics [12], an activity is a directed graph with a set of activity nodes V and connecting edges E. Fig. 4 presents identification rules **R1**-**R7**. In the rules, we use the following functions:

- Two functions mapping an activity node and edge to their particular types, i.e., $kind_V : V \rightarrow K_V$ and $kind_E : E \rightarrow K_E$, where $K_V = \{operation, local, collaboration, merge, join, fork, decision, other\}$ and $K_E = \{object, control\}$
- Two functions mapping a given node to the set of its incoming and outgoing edges, i.e., $in_E : V \rightarrow 2^E$ and $out_E : V \rightarrow 2^E$.
- Two functions that return an object flowing to (reps. from) a given node through an edge, i.e., $in_O : E \times V \rightarrow ON$ and $out_O : V \times E \rightarrow ON$, where ON is the set of all object nodes of a given activity.
- A function mapping a given asset to a class from our ontology, i.e., $class : A \rightarrow K_A$, where $K_A = \{transit, stationary\}$ and A is the set of assets constructed from elements of the set ON.
- A function mapping a node to the partition, which it belongs to, i.e., $part : V \rightarrow 2^P$, where $1 \leq |2^P| \leq 2$ and P is a set of all partitions of a given activity diagram. The case where $|2^P| = 2$ corresponds to a collaboration node (e.g., see the t node in Fig. 1).
- Two functions that return the source and target nodes of a given edge, i.e., $s : E \rightarrow V$ and $t : E \rightarrow V$ respectively.
- A function mapping a merge node and the set of its incoming object edges to its outgoing object, i.e., $fMerge : V \times 2^E \rightarrow ON$. Likewise, we define function $fJoin$ for a join node. According to the SPACE semantics, only one outgoing edge is allowed for merge and join nodes (see the rule **OUT1** in [12]).
- A function mapping a fork node, its incoming object edge, and one of its outgoing edges to an object flowing through this outgoing edge, i.e., $fFork : V \times 2^E \times E \rightarrow ON$. Likewise, we define function $fDecision$ for a decision node. For the sake of generality we allow that the second argument of $fFork$ and $fDecision$ is a set of edges. However, according to the SPACE semantics [12], fork and decision nodes can have only a single incoming edge (see the rule **IN3** in [12]).

The rules **R1** and **R2** express that for an operation, local, or collaboration node q the stationary data asset (i.e., ast)

R1:
$$\frac{\begin{array}{l} q \in V, e \in in_E(q), kind_E(e) = object, \\ kind_V(q) \in \{operation, local, collaboration\} \end{array}}{\exists ast \in A : ast = in_O(e, q), class(ast) = stationary}$$

R2:
$$\frac{\begin{array}{l} q \in V, e \in out_E(q), kind_E(e) = object, \\ kind_V(q) \in \{operation, local, collaboration\} \end{array}}{\exists ast \in A : ast = out_O(q, e), class(ast) = stationary}$$

R3:
$$\frac{\begin{array}{l} e \in E, kind_E(e) = object, |part(s(e))| = |part(t(e))| = 1, \\ part(s(e)) \neq part(t(e)), q \in V, \\ kind_V(q) \in \{operation, local\}, e \in out_E(q) \end{array}}{\exists ast \in A : ast = out_O(q, e), class(ast) = transit}$$

R4:
$$\frac{\begin{array}{l} e \in E, kind_E(e) = object, |part(s(e))| = |part(t(e))| = 1, \\ part(s(e)) \neq part(t(e)), m \in V, kind_V(m) = merge, \\ e \in out_E(m), q \in V, in_E(m) \cap out_E(q) \neq \emptyset, \\ kind_V(q) \in K_V \setminus \{other\} \end{array}}{\begin{array}{l} \exists ast \in A : ast = fMerge(m, in_E(m)), \\ class(ast) = transit, \end{array}}$$

R5:
$$\frac{\begin{array}{l} e \in E, kind_E(e) = object, |part(s(e))| = |part(t(e))| = 1, \\ part(s(e)) \neq part(t(e)), d \in V, kind_V(d) = decision, \\ e \in out_E(d), q \in V, in_E(d) \cap out_E(q) \neq \emptyset, \\ kind_V(q) \in K_V \setminus \{other\} \end{array}}{\begin{array}{l} \exists ast \in A : ast = fDecision(d, in_E(d), e), \\ class(ast) = transit \end{array}}$$

R6:
$$\frac{\begin{array}{l} e \in E, kind_E(e) = object, |part(s(e))| = |part(t(e))| = 1, \\ part(s(e)) \neq part(t(e)), j \in V, kind_V(j) = join, \\ e \in out_E(j), q \in V, in_E(j) \cap out_E(q) \neq \emptyset, \\ kind_V(q) \in K_V \setminus \{other\} \end{array}}{\exists ast \in A : ast = fJoin(j, in_E(j)), class(ast) = transit}$$

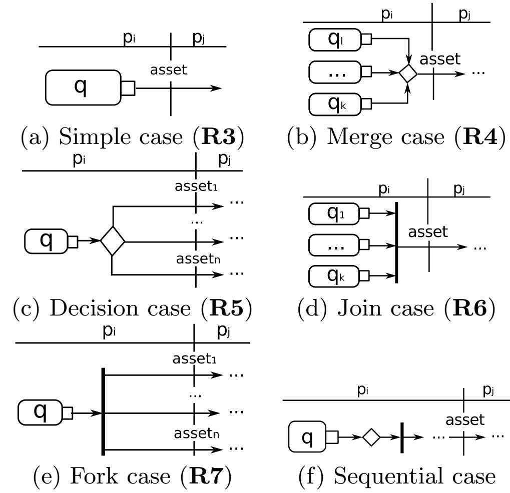
R7:
$$\frac{\begin{array}{l} e \in E, kind_E(e) = object, |part(s(e))| = |part(t(e))| = 1, \\ part(s(e)) \neq part(t(e)), f \in V, kind_V(f) = fork, \\ e \in out_E(f), q \in V, in_E(f) \cap out_E(q) \neq \emptyset, \\ kind_V(q) \in K_V \setminus \{other\} \end{array}}{\begin{array}{l} \exists ast \in A : ast = fFork(f, in_E(f), e), \\ class(ast) = transit \end{array}}$$

Figure 4: Rules for asset identification

is observed if this node has an incoming (**R1**) resp. outgoing (**R2**) edge e of the kind *object*. The rules **R3** to **R7** are applied to an object flow crossing a border of two partitions, which corresponds to the data in transit concept. **R3** describes the case that an object leaves an operation node and goes directly to another partition. By the rules **R4** to **R7**, we cover the cases that a flow passes a merge, join, decision, or fork node before crossing a partition border. Figs. 5.(a) to 5.(e) illustrate the cases of **R3** to **R7** respectively. Fig. 5.(f) depicts an example of a sequential case, which is implicitly covered by **R1** to **R7**, i.e., when more than one control node precede an edge crossing a border.

4.2 Application of Rules

The functions `traverseBlocks` and `traverseEdges` depicted in Fig. 6 outline the application of **R1** to **R7** to a system model, where each activity A is represented by a tuple of nodes and edges (V, E). The function `traverseBlocks` goes through the nodes V of a certain activity A and applies rules **R1** and **R2**. Thereafter, in the case that a considered



(a) Simple case (**R3**) (b) Merge case (**R4**)
(c) Decision case (**R5**) (d) Join case (**R6**)
(e) Fork case (**R7**) (f) Sequential case

Figure 5: Illustration of rules

```
function traverseBlocks (Activity A)
    ∀v ∈ V :
        R1 − R2
        if kind_V(v) = local then
            traverseBlocks(v)
        endif
        if kind_V(v) = collaboration then
            traverseEdges(v),
            traverseBlocks(v)
        endif
function traverseEdges (Activity A)
    ∀e ∈ E :
        R3 − R7
```

Figure 6: Functions to traverse a system model

node is a local block, `traverseBlocks` is recursively applied to its internal behaviour. Likewise, a collaborative block requires application of both the `traverseBlocks` and `traverseEdges` functions. For example, this is the case for the analysis of block t in our scenario in Fig. 1. Here, the function `traverseEdges` applies rule **R4** to the merge before the crossing edge from partition *sender* to *receiver* in the activity on the bottom of Fig. 1. For the decision node before the crossing edges in the opposite direction, rule **R5** is used. As a result, four data in transit assets are elicited: two assets *ackIn* incoming to the *get* and *delete* nodes; two assets *temp* outgoing from the *get* and *set* nodes.

5. SECURITY PROPERTY AND SBB

Each DSSM is used to enrich our ontology with a set of rules and assertions for the corresponding concepts described in Sect. 3. We refer to such an ontology as the *enriched ontology*. Therefore, the search for suitable security properties (step 3 introduced in Sect. 1) and concrete SBBs (step 5) for a specific asset and domain can be realised as ontology inference. To achieve this, we formulate (1) a query to find available security properties for a given asset and (2) a query to retrieve concrete SBBs that satisfy certain security properties (step 4). Thereafter, these queries are used as inputs to an ontology reasoner that returns a list of security properties and concrete SBBs respectively.

We employ the HermiT reasoner [1]. To formulate the queries, we use the Manchester syntax [3] where *and* and

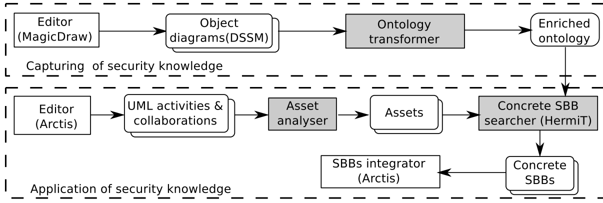


Figure 7: Tool support

only are the syntax keywords denoting *sets intersection* and *universal quantifier* respectively. The rest of words in the queries are names of corresponding concepts and relations (see Fig. 2).

Security properties, which are available for a given asset (i.e., the query (1) above), are accessed in the enriched ontology executing the following query[1]:

```
SecurityProperty and has only [Asset]
```

For example, one of the identified assets in Fig. 1 is *temp*. It corresponds to the *TransmittedMeasurement* asset from the metering DSSM. We execute the query for the *TransmittedMeasurement* (TM) asset. It returns the following triples:

```
SP1: [TM, Confidentiality, Prevention]
SP2: [TM, Integrity, Detection]
SP3: [TM, Authentication, Detection]
```

A system engineer chooses a subset out of all available security properties (i.e., step 4 in Sect. 1) and proceeds with a search for concrete SBBs.

Concrete SBBs that satisfy certain security properties in a given domain (i.e., the query (2) above) are accessed in the enriched ontology executing the following query:

```
ConcreteSBB and satisfies only [SecurityProperty]
and belongsTo only [Domain]
```

For our data collection scenario, we execute this query for the selected property SP_1 and the *metering* domain. This returns the concrete SBBs *AES* and *DES*, defined by the abstract SBB *Cipher*, and their related Arctis building blocks, i.e., *AESCipher* resp. *DESCipher*. Thereafter, a system engineer selects one alternative based on, e.g., resource constraints. The formalisation of the basis and assistance for such decisions is not in the scope of this paper.

As mentioned in Sect. 3, integrating a concrete SBB may create new assets as expressed by the *creates* and *requires* relations in our ontology. Hence, a further search is needed to fulfil also the security goals required for these new assets. For this purpose, we apply the following strategy: First, we search for security properties available for created assets using query (1). Thereafter, those security properties that have the goals stated in the attribute *requiredGoal* are selected (see Fig. 2). Afterwards, we search for concrete SBBs that satisfy the security properties within a domain specified by the *externalDSSM* attribute using query (2). This may lead to recursive execution of the queries (1) and (2) until all security goals of all created assets are fulfilled.

[1]Values in square brackets denote parameters of queries.

6. TOOL SUPPORT

Fig. 7 depicts a set of tools to support capturing and application of security knowledge. The grey boxes denote the components developed within the presented work.

Capturing. A security engineer creates DSSMs as object diagrams using any UML editor supporting the XMI format [16]. In our work, we use MagicDraw [2]. Thereafter, the information captured by these diagrams (i.e., DSSMs) enriches the ontology. Omitting details, the *ontology transformer* converts elements of a DSSM and their relations to corresponding axioms on classes, relations, and individuals.

Application. A system engineer uses a modelling tool (in our case the Arctis tool [13]) to create a functional model of a given system, i.e., UML activities and collaborations. Afterwards, the model is analysed by our *asset analyser* that implements the rules presented in Section 4. The outcome of the step is a set of elicited assets. This information goes to our *concrete SBB searcher* that is based on the queries presented in Sect. 5, where a system engineer makes a choice on considered assets, security properties, and domain. The outcome is a set of all concrete SBBs captured by DSSMs that protect the assets. Thereafter, an engineer may select a subset of these SBBs and proceed to integrate these concrete SBBs using the (Arctis) modelling tool capabilities.

7. RELATED WORKS

Several model-based approaches have been proposed to integrate security aspects into a system design, e.g., UMLsec [9] and SecureUML [14]. UMLsec [9] is a UML profile, which helps to incorporate the security-related functionality into a system model and to verify satisfaction of certain security requirements. SecureUML [14] deals with design and verification of role-based access control systems. Security patterns [17] is another well-known methodology, which is a general approach to capture security solutions for reoccurring security problems. According to Nhlabatsi et al. [15], SecureUML and UMLsec are examples of languages to express and apply security patterns. In this paper we exploit the SPACE method [12] to describe and integrate security building blocks into a system design. Earlier, this method has been already applied for security aspects [7].

An important step preceding actual integration of security building blocks is the selection of such. We have observed that the model-based approaches mentioned above do not address this task in detail. However, approaches to select a security mechanism are elaborated in the context of security patterns [17]. A fundamental principle to assist in selection of security patterns is their sufficient classification. Such classification can be built of three classes (Fernandez et al. [6]) or be represented as a multi-dimensional matrix of classes (VanHilst et al. [18]) including such categories as an architectural layer (e.g., network), lifecycle stage (e.g., design), domain (e.g., enterprise systems), etc. In our work the basis for selection of concrete SBBs are domain, security goal, defense strategy, and asset. Thus, there are some overlapping concepts in our work compared to the dimensions elaborated for security patterns.

Washizaki et al. [19] develop the *dimensional graph (DG)* concept to formalise the multi-dimension classification of security patterns. Each DG uses a UML object diagram to show relations of a pattern to a set of dimensions of interest. We formalise our categories as the ontology, which

allows utilising advantages of its reasoning capabilities for selection of SBBs.

Asset identification is an essential stage for many risk analysis methods that precedes selection of security mechanisms , e.g., as in the CORAS method [5]. Identification of assets is normally done by a security expert through informal analysis of a system model or other (often informal) representation. We have proposed to automate this task in presence of a formal system model. Thus, we have defined the rules for traversing a system model expressed as SPACE collaborative activities to identify assets and classify them in accordance with knowledge captured within the ontology.

8. CONCLUDING REMARKS

In this paper, we have refined our approach for model-based security engineering. In particular, we present a technique for asset elicitation applied to a system model. The core of this technique is a set of rules to traverse collaborative models. Elicited assets are used to guide the security building blocks' selection process. The task of SBBs selection was formulated as ontology inference. The proposals are supported with a set of tools, which complement other modelling tools used in our approach, i.e., the Arctis [13] and MagicDraw modelling tools [2]. We used a fragment of the smart metering system, which is currently under industrial development, to exemplify the employed modelling language and to illustrate the key aspects of our proposal.

We will continue applying and validating our technique and tools for different parts of the case study mentioned above. In addition to the smart metering system, we will consider other complex and diverse industrial case studies presented in the SecFutur project [4]. Moreover, our further work includes enhancing the set of rules with other semantics- and context-dependent information to provide a more comprehensive list of elicited assets.

9. REFERENCES

[1] HermiT Reasoner. `http://hermit-reasoner.com`.

[2] MagicDraw. `http://www.magicdraw.com`.

[3] Ontology Language Manchester Syntax. `http://www.w3.org/TR/owl2-manchester-syntax/`.

[4] The SecFutur project: Design of Secure and Energy-efficient Embedded Systems for Future Internet Application. `http://www.secfutur.eu`.

[5] F. Braber, I. Hogganvik, M. S. Lund, K. Stølen, and F. Vraalsen. Model-Based Security Analysis in Seven Steps – a Guided Tour to the CORAS Method. *B*T Technology Journal, 2007.

[6] E. B. Fernandez, H. Washizaki, N. Yoshioka, A. Kubo, and Y. Fukazawa. Classifying security patterns. In *1*0th Asia-Pacific web conference on Progress in WWW research and development, 2008.

[7] L. A. Gunawan, F. A. Kraemer, and P. Herrmann. A Tool-Supported Method for the Design and Implementation of Secure Distributed Applications. In *I*nternational Symposium on Engineering Secure Software and Systems. Springer, 2011.

[8] A. Herzog, N. Shahmehri, and C. Duma. An Ontology of Information Security. *J*ournal of Techniques and Applications for Advanced Information Privacy and Security, IGI Global, 2007.

[9] J. Jürjens. *Secure System Development with UML*. Springer-Verlag, 2005.

[10] F. A. Kraemer. *Engineering Reactive Systems: A Compositional and Model-Driven Method Based on Collaborative Building Blocks*. PhD thesis, Norwegian University of Science and Technology, August 2008.

[11] F. A. Kraemer and P. Herrmann. Automated Encapsulation of UML Activities for Incremental Development and Verification. In *I*nternational Conference on Model Driven Engineering, Languages and Systems. Springer, 2009.

[12] F. A. Kraemer and P. Herrmann. Reactive Semantics for Distributed UML Activities. In *F*ormal Techniques for Distributed Systems. Springer, 2010.

[13] F. A. Kraemer, V. Slåtten, and P. Herrmann. Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. *J*ournal of Systems and Software, 2009.

[14] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *I*nternational Conference on The Unified Modeling Language, UML. Springer-Verlag, 2002.

[15] A. Nhlabatsi, A. Bandara, S. Hayashi, C. Haley, J. Jurjens, H. Kaiya, A. Kubo, R. Laney, H. Mouratidis, B. Nuseibeh, T. Tun, H. Washizaki, N. Yoshioka, and Y. Yu. Security patterns: Comparing modelling approaches. *S*oftware Engineering for Secure Systems: Industrial and Research Perspectives, IGI Global, 2010.

[16] Object Management Group. *Unified Modeling Language: Superstructure, version 2.4.1*.

[17] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley and Sons, 2005.

[18] M. VanHilst, E. B. Fernández, and F. A. Braz. A multi-dimensional classification for users of security patterns. In *I*nternational Workshop on Security in Information Systems, 2008.

[19] H. Washizaki, E. B. Fernandez, K. Maruyama, A. Kubo, and N. Yoshioka. Improving the classification of security patterns. In *I*nternational Workshop on Database and Expert Systems Application, 2009.