SysML Model Transformation for Safety and Security Analysis

Rabéa Ameur-Boulifa[1](✉), Florian Lugou[2], and Ludovic Apvrille[1]

[1] LTCI, Télécom ParisTech, Université Paris-Saclay, Paris, France
Rabea.Ameur-Boulifa@telecom-paristech.fr
[2] Prove & Run, Paris, France

Abstract. While the awareness toward the security and safety of embedded systems has recently improved due to various significant attacks, the issue of building a practical but accurate methodology for designing such safe and secure systems still remains unsolved. Where test coverage is dissatisfying, formal analysis grants much higher potential to discover security vulnerabilities during the design phase of a system. Yet, formal verification methods often require a strong technical background that limits their usage. In this paper, we formally describe a verification process that enables us to prove security-oriented properties such as confidentiality on block and state machine diagrams of SysML. The mathematical description of the translation of these formally defined diagrams into a ProVerif specification enables us to prove the correctness of the verification method.

Keywords: Model-Driven Engineering · Verification · Safety · Security · Embedded systems

1 Introduction

In our increasingly connected world, security is a growing concern for embedded systems. This remark firstly applies to critical systems such as connected vehicles or industrial systems. There are already many approaches (i.e. methods, models and tools) to evaluate critical aspects of these systems independently from their security: real-time schedulability, formal verification techniques based e.g. on model-checking or correct-by-construction techniques. Model-Driven Engineering often considers safety aspects with coherence checks between diagrams or with model-to-formal-specification algorithms in order to evaluate safety properties from e.g. UML diagrams. Concerning security aspects, a usual practice is to rely on dedicated models and tools that are focused on the security aspect e.g. ProVerif [1] and Avispa [6], and are thus not compatible with safety-related models and tools. As a result, security is often seen as *the right way to use the right tools*, if not totally ignored. This however leads to more subtle bugs when out-of-the-box cryptographic solutions are not suitable, and in particular when the importance of an asset or communication is misunderstood. Such a security

B. Hamid et al. (Eds.): ISSA 2018/CSITS 2018, LNCS 11552, pp. 35–49, 2019.
https://doi.org/10.1007/978-3-030-16874-2_3

issue can be minor when the number of devices affected is small and when the vulnerability can be fixed easily, e.g. with a software patch. However, this is typically not the case for embedded systems where design flaws can be impossible to fix and can affect a whole range of products. Even when a security vulnerability is discovered before the product is released, the amount of work needed to rethink the whole architecture may be prohibitive.

To facilitate the design of critical systems with security requirements, we suggest enhancing safety-related models with security mechanisms, and to offer, from the same model, safety-to-formal-specification and in addition, security-to-formal-specification transformations. In the paper, we present the SysML-Sec environment that supports both safety and security. Then, we elaborate on the SysML-Sec-model-to-security-formal-specification that was first sketched in [16]. This transformation algorithm is valuable as it enables us to perform security verification on general-purpose design models and thus avoids error-prone duplication of models. However, the transformation algorithm had not been formally described yet. This paper gives a formal description of the transformation algorithm in order to prove the correctness of the method. Throughout the paper, we will illustrate our explanation of the different phases of modeling and verification on a pedagogical example. Although the example has purposely been kept to its bare minimum so that the reader can easily refer to it, it could still be used as a sub-part of a greater real-life design. In the presented scenario, two participants (called Alice and Bob) communicate through an unsafe (public) channel. Alice repeatedly sends sensitive data to Bob. The messages are encrypted by Alice before being transmitted over the public channel. The two participants have beforehand shared a cryptographic key and we assume the way the sharing was performed does not need to be modeled. In practice, the key could have been physically shared, built from asymmetric key material (through a Diffie-Hellman protocol for instance) or it could have been provided to Alice and Bob by a trusted third party. The key used by Alice to encrypt her communications periodically changes, and thus a new key is created. So each time Alice sends a new message, she attaches the newly created key so that Bob is able to decrypt the next message. We typically want to verify that the data sent by Alice can not be retrieved by a potential attacker eavesdropping and manipulating messages on the public channel. Other more complex security protocols and systems have been modeled and verified using the method described in this paper.

The verification method enables us to prove confidentiality and authenticity properties on these models within an acceptable time (less than 5 min on a general-purpose computer). We will not detail these case studies in the current paper but refer the interested reader to the SysML-Sec website[1] where the corresponding models are freely accessible.

The paper is organised as follows: in Sect. 2, we present the methodology chosen here and give a formal description of the modeling language (a SysML profile). Section 3 presents the basic model ProVerif language and we give a translation of SysML model to ProVerif model. Section 4 acts as a validation of

[1] http://sysml-sec.telecom-paristech.fr/.

our approach that can be used to assert the validity of our translation. Section 5 surveys related work before concluding in Sect. 6.

2 SysML-Sec Language

SysML-Sec [5] is a modeling language following a model-driven approach to design embedded systems with safety, security and performance constraints. This modeling language was chosen as it enables the user to analyze behaviors that will be implemented by the system and specifically targets embedded systems. Moreover, it is supported by a free and open-source tool to which the presented algorithm was added.

Designing an Application: Basically, SysML-Sec supports two main modeling phases:

1. The **system-level HW/SW partitioning** phase includes capturing functional elements of the target application, modeling candidate architectures and finally mapping functional elements—including communications between functions—to candidate architectures. Then a verification sub-phase follows in which safety, security and performance constraints are evaluated in order to select the "best" HW/SW partition.
2. A **software design** phase follows a successful partitioning phase. Software components are first built from high-level functions mapped onto processor nodes at the previous phase. Then, they are progressively refined. Refinement typically concerns the accurate description of algorithms and protocols, including security protocols.

Design elements of the two phases are built from (safety and security) requirements. Verification is supported in all modeling stages in order to assess the security and safety requirements. Attack trees also help capture potential attacks that are feasible in the considered mapping models.

TTool is a free and open-source tool that supports the different phases and models of SysML-Sec. It offers a press-button approach for safety, security and performance verification, and can backtrace verification results to modeling views.

Software Design Verification: As formalized below, a software design is built upon communicating blocks whose behaviors are described with state machine diagrams. Software design verification intends to evaluate the fulfillment of safety and security properties. Safety verification checks a large set of properties including safety (e.g. deadlock-free) and liveness (e.g. reachability) properties. Properties can be modeled either with a subset of temporal logic language e.g. CTL, or with the use of observers in the model that are expressed with state machine diagrams. TTool relies on UPPAAL model checking tool for verification.

2.1 Syntax

In the software design phase, the SysML-Sec diagrams intend to describe a software *design*. This section provides a formal definition of software designs.

Definition 1 Design. *A design is defined by a network of blocks interconnected by links and a set of pragmas:*
$\mathcal{D} = \langle \mathcal{B}, \mathcal{C}, \mathcal{P} \rangle$ *where* $\mathcal{B}$ *is a set of blocks,* $\mathcal{C}$ *is a set of channels, and* $\mathcal{P}$ *is a set of pragmas.*

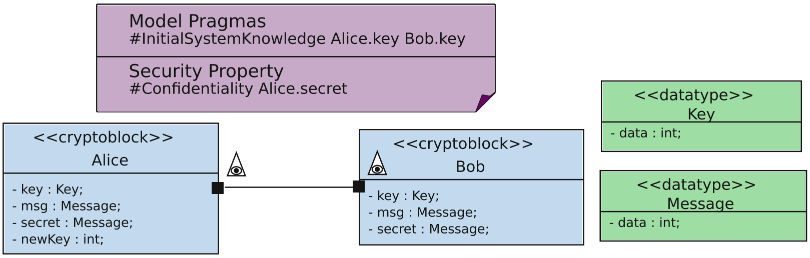


Fig. 1. A graphical representation of SysML-Sec design example

Figure 1 displays two blocks *Alice* and *Bob* as well as a public link—as denoted by the illuminati symbol—between the two. In this paper, we don't mention data types as they only act as syntactic sugar as far as security analysis is concerned.

SysML blocks consist of a set of methods and attributes. Communication ports can be attached to a block, and to each port are attached interfaces and signals [12]. For simplicity, we directly attach signals to SysML blocks.

Definition 2 Block. *A block is a tuple:*
$\mathsf{block} = \langle ident, \mathcal{A}, \mathcal{M}, \mathcal{S}, \mathsf{behav} \rangle$ *where*

- *ident is a block name.*
- $\mathcal{A}$ *is a set of attributes.*
- $\mathcal{M}$ *is a set of methods.*
- $\mathcal{S}$ *is a set of directed signals. For each* $s \in \mathcal{S}$, $\mathsf{type}(s) \in \{in, out\}$.
- behav *is a state machine diagram.*

We define a function *block* that, for a given design $\mathcal{D}$, returns the set of its blocks; and functions *sig* and *att* that *b* return the set of signals and attributes for a given block respectively.

Definition 3 Channel. *A channel connects signals between blocks:* $\mathsf{channel} = \langle \mathsf{type}, \mathcal{R} \rangle$ *where* type *is a physical property which can be either* $\mathsf{private}$ *or* public, *and* $\mathcal{R}$ *is one-to-one correspondence between two sets of signals,* $\mathcal{R} \subseteq sig(b_1) \times sig(b_2)$ *where* $b_1, b_2 \in block(\mathcal{D})$ *such that* $\forall (s_1, s_2) \in \mathcal{R}$, $\mathsf{type}(s_1) \neq \mathsf{type}(s_2)$.

SysML design supports the notion of pragma. Pragmas enable us to describe properties of the system in the initial state, and to query a property of the design that will be checked during verification. To simplify this description, we will consider only two types of pragmas which: - express that two attributes have the same value at the beginning of the execution ($\mathcal{P}_{init}$); - query the confidentiality of an attribute ($\mathcal{P}_{secret}$).

Definition 4 Pragma. *Let $\mathcal{D}$ be a design. We define a pragma as a pair: $\mathcal{P} = (\mathcal{P}_{init}, \mathcal{P}_{secret})$ where*
$\mathcal{P}_{init} \subseteq \left(\bigcup_{\mathsf{b} \in block(\mathcal{D})} att(\mathsf{b})\right)^2$ *and* $\mathcal{P}_{secret} \subseteq \bigcup_{\mathsf{b} \in block(\mathcal{D})} att(\mathsf{b})$

A state machine diagram is a labelled transition system with variables named attributes; a state machine diagram can have guards and assignments of attributes on transitions. Attributes can be manipulated, defined, or accessed. Let f range over function names, x_i range over variable names, and c are channel names. The set $\mathcal{A}ctions$ of action terms in state machine diagrams is defined as follows:

$$\begin{array}{rll} a \in \mathcal{A}ctions ::= & f(x_1, \ldots, x_n) & \text{function call} \\ & \mid x := exp & \text{assignment expression} \\ & \mid c\langle x \rangle & \text{input action} \\ & \mid \bar{c}\langle x \rangle & \text{output action} \\ & \mid \nu.x & \text{random action} \\ & \mid \varepsilon & \text{empty action} \end{array}$$

Expressions (exp) in SysML-Sec can be variables and function calls (x and $f(x_1, \ldots, x_n)$). The set $\mathcal{G}uards$ is the set of boolean expressions.

Definition 5 State Machine Diagram. *A state machine diagram is a rooted directed graph:* $\mathsf{behav} = \langle \mathcal{Q}, q_0, q_\perp, \mathcal{E} \rangle$ *where*

- *$\mathcal{Q}$ is a set of nodes.*
- *$q_0 \in \mathcal{Q}$ is an initial state node.*
- *$q_\perp \in \mathcal{Q}$ is a (possibly empty) final state node.*
- *$\mathcal{E} \subseteq \mathcal{Q} \times \mathcal{G}uards \times \mathcal{A}ctions \times \mathcal{Q}$.*

A name is given by the designer to each state. We define a labelling function $\mathbb{L}$ that returns the name of a given state. Given an edge $e = (q, g, a, q')$, we define functions $source(e) = q$, $guard(e) = g$, $action(e) = a$, and $target(e) = q'$. A trace $\sigma \in \mathcal{A}ctions^*$ is a sequence of actions $a_0\ a_1, \ldots a_n$ such that there is a sequence of states $q_0\ q_1, \ldots q_n$ and $(q_{i-1}, g, a_i, q_i) \in \mathcal{E}$ for all $i = 1, \ldots, n$.

Syntactic Constraints on Activity Diagram. TTool enforces some basic properties on the state machine diagrams, namely:

1. The initial state node may only occur in the source of an edge.
2. The final state node may only occur in the target of an edge.
3. For any state node, there is a path from the initial state node to this node.

4. Any state node different from the final state node has at least one outgoing transition.

We introduce the notion of *basic block* that we will use in our translation. A basic block can be seen as a sub-design that offers a single point of entry and that can be triggered by several points. Precisely, it is a connected sub-graph for which all the states have exactly one incoming edge, except for one state that we name *root*. We will use $\mathcal{O}ut$ function that returns the set of transitions outgoing from a given state. We also define a predicate *UniqueOut* and *UniqueIn* that take a state q and return true only if no two different transitions have q as a source and target state respectively.

$$UniqueOut(q) \Leftrightarrow \begin{pmatrix} \forall (q_1, g_1, a_1, q_1'), (q_2, g_2, a_2, q_2') \in \mathcal{E}. \\ q_1 = q \wedge q_2 = q \Rightarrow g_1 = g_2 \wedge a_1 = a_2 \wedge q_1' = q_2' \end{pmatrix}$$

$$UniqueIn(q) \Leftrightarrow \begin{pmatrix} \forall (q_1, g_1, a_1, q_1'), (q_2, g_2, a_2, q_2') \in \mathcal{E}. \\ q_1' = q \wedge q_2' = q \Rightarrow q_1 = q_2 \wedge g_1 = g_2 \wedge a_1 = a_2 \end{pmatrix}$$

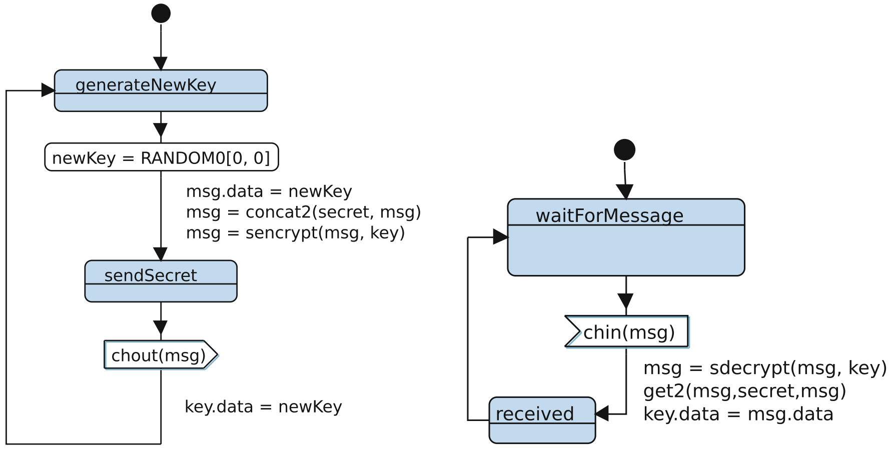
Figures 2a and b show the graphical representation of the two state machine diagrams of *Alice* and *Bob* respectively. Note that empty actions and "true" guards are not shown in the diagrams. States are depicted by colored boxes (except for the initial state which is a circle), transitions by arrows, and actions are either represented by textual expressions next to arrows (for function calls and assignment expression) or by white boxes with various forms (for the other types of actions). For instance, the state machine of Alice is composed of an initial state linked to a state named *generateNewKey* by an empty transition. This state is linked to another state *sendSecret* by a transition bearing 4 actions: a random action and 3 assignment expressions. Another transition links *sendSecret* to *generateNewKey* and bears an output action. Note that in the diagrams, multiple actions appear on each transition. This is semantically equivalent to multiple chained transitions, each of which bearing a single action and a `true` guard.

3 From SysML-Sec to Proverif

Our goal is to provide an environment to design safe and secure systems with the SysML language. Our plan is to give a formal definition of the behavioural semantics of SysML, and get a standard code to do the security analysis. This section describes the behavioural semantics of SysML design allowing security analysis.

3.1 ProVerif Language

ProVerif [7] is a cryptographic protocol verification tool operating on a symbolic model. ProVerif specifications are described in a custom language following a well-defined structure [8]. It consists of a sequence of declarations and a process. Our translation use a core of ProVerif language, excluding only some declarations. In detail, it covers the following features, which form a complete language for generating well-formed code for security analysis:



(a) Alice

(b) Bob

Fig. 2. State machine diagrams in the SysML-Sec methodology

- **Function** declaration (referred to by *fun* and *reduc* keywords). They are typically used to describe cryptographic primitives such as *hash*, *symmetric encryption*, etc. and they don't depend on the particular design we are translating.
- **Variable** declaration (denoted by *channel* and *free* keywords). They declare channels and other variables that are shared by every participant and can be either public or private.
- **Queries** (referred to by *query* keyword) express the security properties that a user wishes to prove on the design
- **Sub-processes** declaration (referred to by *let* keyword). Each sub-process declaration contains a behavioral description of part of the state machine diagrams of the design. They may be referenced by other sub-processes or by the main process. If they are not referenced by anyone, they are simply ignored.
- The **main process** (referred to by *process* keyword), which is the entry point of the design. It can reference any sub-process.

Global structure of an example of ProVerif code is presented in Listing 1.1.

In particular, we see a constructor declaration (`sencrypt`), a destructor declaration (`sdecrypt`), two shared variables declarations (`token___Bob___0` and `token___Alice___0`), a confidentiality query, the declaration of a sub-process (`Bob___0`) and the main process which creates a new private name (`Alice___key___data`).

```
(* Functions *)
fun sencrypt (bitstring, bitstring): bitstring.
reduc forall x: bitstring, k: bitstring;
        sdecrypt (sencrypt (x, k), k) = x.
...
```

```
(* Variables *)
free token___Bob___0: bitstring [private].
free token___Alice___0: bitstring [private].
...

(* Queries *)
query attacker(new Alice___secret__data).
...

(* Sub-processes *)
let Bob___0 =
    new strong___Bob___02: bitstring;
    out (chControl, strong___Bob___02);
    ...

(* Main process *)
process
    new Alice___key__data: bitstring;
    ...
```

Listing 1.1. Global structure of a ProVerif file

3.2 Translation of SysML-Sec Design to ProVerif

We now give the semantics of a SysML-Sec design, expressed as a translation from SysML-Sec designs into ProVerif specifications. For each SysML-Sec design $\mathcal{D}$, the interpretation function is expressed under the form:

$$[\![\mathcal{D}]\!]_{\mathcal{E}} = F_{\mathcal{E}}(\mathcal{D}) \oplus V_{\mathcal{E}}(\mathcal{D}) \oplus Q_{\mathcal{E}}(\mathcal{D}) \oplus P_{\mathcal{E}}(\mathcal{D}) \oplus \text{"process"} \oplus Main_{\mathcal{E}}(\mathcal{D})$$

It relies on several auxiliary functions for expressing the semantics of specific parts of the designs. The core entities of this semantics include five functions: $F_{\mathcal{E}}(\mathcal{D})$ for generating the set of functions, $V_{\mathcal{E}}(\mathcal{D})$ for generating the set of variables, $Q_{\mathcal{E}}(\mathcal{D})$ for generating the set of queries from pragmas, $P_{\mathcal{E}}(\mathcal{D})$ for generating the set of processes, and $Main_{\mathcal{E}}(\mathcal{D})$ that generates the main process that manages global instantiation of other processes. The construction of these functions relies on the notion of *environment* denoted $\mathcal{E} = (\mathcal{E}_q, \mathcal{E}_v)$ that keeps track of the states that have to be visited ($\mathcal{E}_q$) and those that have already been visited ($\mathcal{E}_v$) during state machine traversal.

Before defining the interpretation function, it is helpful to introduce some notations. We use the quote (") character to indicate the beginning and ending of a string (corresponding to ProVerif instruction). Quoted strings placed next to each other are concatenated (by $\oplus$ operator) to produce a whole string (complete source code). $\overrightarrow{a}^{a \in \mathcal{S}}$ denotes a list of parameters over the set $\mathcal{S}$.

(1) Declarations part

Functions. They include a list of common cryptographic primitives that can be used in all SysML-Sec designs. They also include additional functions tok and untok (used to protect variables), and a pair of encryption and decryption functions that are added to each private channel.

Variables. They consist of three types channels used for public communication, channels controlling messages (referred to by chctrl) and variable for each basic block (referred to by `token_...`). Note that the `token_...` variables can only be generated once the sub-processes are generated.

Queries. In this paper, we focus on the confidentiality property. For each variable `v` for which the designer would like to check the confidentiality, we generate a query of the form `"query attacker(new v)"`.

(2) Processes generation

Sub-processes. They are generated by walking through the state machine diagram of every basic block of the SysML-Sec design. To do this, the interpretation function relies on a queue of states to be visited $\mathcal{E}_q$ that is initialized to contain the *root* state of each basic block, and a list $\mathcal{E}_v$ that contains all the states that have already been visited (which is empty at the beginning). While there are unexplored states, one state s is picked from the $\mathcal{E}_q$ set, it is added to the explored set $\mathcal{E}_v$ set, a sub-process is created by using the first function $[\![s]\!]^p_{\mathcal{E}}$ (see Table 1). The idea is that the translation function goes through the whole *basic block* starting from the root and generates a Proverif instruction for each constructor encountered by calling the appropriate interpretation function. All interpretation functions are defined in Table 1. They use the terminology *fresh* variable which means that the variable is a new one and it has no occurrence anywhere in the code except in the instruction that creates it. Informally, the interpretation functions, as described in Table 1, translate states to a corresponding ProVerif `event` used for reachability queries; and transitions by translating their guards into `if` conditions ($[\![., .]\!]^t_{\mathcal{E}}$) and their actions into ProVerif instructions ($[\![., .]\!]^a_{\mathcal{E}}$). The continuation of the translation of following states is completed by $[\![.]\!]^c_{\mathcal{E}}$ function. Two interpretation functions require special attention: multiple outgoing transitions and transitions linking states of two different basic blocks. For the former, the resulting ProVerif process generates a token for each possible transitions and makes them available to the attacker ($[\![.]\!]^m_{\mathcal{E}}$). Then, it triggers the path by asking the attacker to accept one token. For the latter, the process also generates a token ($[\![.]\!]^b_{\mathcal{E}}$). This token must contain the current state of the block (as described by its attributes) and the identifier of the basic block to be called (the `token_` variables). In order to prevent the attacker from replaying previous tokens, the token includes a nonce that is issued by the callee. This token is protected from modification and spying by the attacker by encapsulating it into a private function `tok`.

Main Process. The main process is then appended to the end of the ProVerif specification. Its purpose is first to create one unique `tok(...)` message for each state machine so that the attacker can *call*[2] the process corresponding to each basic block whose root is the initial state of a state machine. To create each token for a block, the main process needs to instantiate the attributes of the block, wait for a nonce and send the token. Then, it runs all the created processes in parallel (as denoted by the | operator) infinitely (as denoted by the ! operator).

[2] The term *call* here is abusive. Indeed, the attacker has no control over the execution flow of each process. It is however able to pass a token to a particular process which is blocked waiting for it.

Table 1. Interpretation function of state machine diagrams

$$\llbracket q\rrbracket^{p}_{\mathcal{E}} = \begin{cases} \text{"let proclabel\_}\mathbb{L}(q) = \\ \text{new nonce;} \\ \text{out (chctrl, nonce);} \\ \text{in(chctrl, token);let (=token\_}\mathbb{L}(q), \text{=nonce, } args) = \text{untok(token)"} \oplus \llbracket q\rrbracket^{s}_{\mathcal{E}} \\ \\ \texttt{with } args = \overrightarrow{a}^{a\in att(b)} \end{cases}$$

$$\llbracket q\rrbracket^{s}_{\mathcal{E}} = \begin{cases} \texttt{"."} & \texttt{if } q = q_{\perp} \\ \text{"event enteringState\_}\mathbb{L}(q)\text{();"} \oplus \llbracket q,e\rrbracket^{t}_{\mathcal{E}} & \texttt{if } UniqueOut(q) \\ \text{"event enteringState\_}\mathbb{L}(q)\text{();"} \oplus \llbracket q\rrbracket^{m}_{\mathcal{E}} & \texttt{otherwise} \end{cases}$$

$$\llbracket q,e\rrbracket^{t}_{\mathcal{E}} = \begin{cases} \text{"if } guard(e) \text{ then"} \oplus \llbracket q,e\rrbracket^{a}_{\mathcal{E}} & \texttt{if } guard(e) \neq \text{true} \\ \llbracket q,e\rrbracket^{a}_{\mathcal{E}} & \texttt{otherwise} \end{cases}$$

$$\llbracket q\rrbracket^{m}_{\mathcal{E}} = \begin{cases} \bigoplus\limits_{e\in \mathcal{O}ut(q)} \text{"new } x_e\text{;out (chctrl, } x_e\text{);"} \oplus \text{"in (chctrl, } c\text{);"} \bigoplus\limits_{e\in \mathcal{O}ut(q)} \left(\text{"if } c = x_e \text{ then"} \oplus \llbracket q,e\rrbracket^{t}_{\mathcal{E}}\right) \\ \texttt{where } c \texttt{ and } x_e \texttt{ are fresh variables} \end{cases}$$

$$\llbracket q,e\rrbracket^{a}_{\mathcal{E}} = \begin{cases} \text{"let } x = exp \text{ in"} \oplus \llbracket target(e)\rrbracket^{c}_{\mathcal{E}} & \texttt{if } action(e) = x := exp \\ \text{"new } x\text{;"} \oplus \llbracket target(e)\rrbracket^{c}_{\mathcal{E}} & \texttt{if } action(e) = \nu.x \\ \text{"out } (c,x)\text{;"} \oplus \llbracket target(e)\rrbracket^{c}_{\mathcal{E}} & \texttt{if } action(e) = \bar{c}\langle x\rangle \\ \text{"in } (c,x)\text{;"} \oplus \llbracket target(e)\rrbracket^{c}_{\mathcal{E}} & \texttt{if } action(e) = c\langle x\rangle \\ \llbracket target(e)\rrbracket^{c}_{\mathcal{E}} & \texttt{if } action(e) = f(x_1,\ldots,x_2) \mid \varepsilon \end{cases}$$

$$\llbracket q\rrbracket^{c}_{\mathcal{E}} = \begin{cases} \llbracket q\rrbracket^{s}_{\mathcal{E}} & \texttt{if } UniqueIn(q) \\ \llbracket q\rrbracket^{b}_{\mathcal{E}} & \texttt{otherwise} \end{cases}$$

$$\llbracket q\rrbracket^{b}_{\mathcal{E}} = \begin{cases} \text{"in(chctrl, nonce);out(chctrl, tok(token\_}\mathbb{L}(q)), \text{nonce, } args))\text{."} & \texttt{if } q \in \mathcal{E}_v \text{ or } q \in \mathcal{E}_q \\ \text{"in(chctrl, nonce);out(chctrl, tok(token\_}\mathbb{L}(q)), \text{nonce, } args))\text{."} & \texttt{otherwise} \\ \quad \mathcal{E}_q = \mathcal{E}_q \cup \{q\} & \\ \\ \texttt{with } args = \overrightarrow{a}^{a\in att(b)} & \end{cases}$$

$$Main_{\mathcal{E}}(\mathcal{D}) = \left(\bigoplus_{b\in block(\mathcal{D})} \left(\bigoplus_{a\in att(b)} \text{"new } a\text{;"} \oplus \text{"in(chctrl, nonce);} \right.\right.$$
$$\left.\left. \text{out(chctrl, tok(token\_}\mathbb{L}(q_0), \text{nonce, } args))\text{"}\right)\right)$$
$$\text{"}\Big|\text{"}_{q\in\mathcal{E}_v} \left(\text{" ! proclabel\_}\mathbb{L}(q)\text{"}\right)$$

with $args = \overrightarrow{a}^{a\in att(b)}$

4 Validation

The purpose of this section is to provide arguments validating the semantics given in this paper. The first part shows formally that we didn't introduce any new information in our translation process; the second part focuses on an example to show how our translation works in practice.

4.1 Correctness Theorem

We first proved that our translation algorithm is sound: if there is a possible disclosure of a secret in the software design, then there is a disclosure in the ProVerif specification. Soundness of translation algorithm states that each ProVerif code generated by $Main_{\mathcal{E}}(\mathcal{D})$, is compliant with the software design $\mathcal{D}$, according to the property of confidentiality.

Proposition 1. *If a term M is a secret in the SysML-Sec model, then M is a secret in the generated ProVerif specification.*

The proof is done by induction on the length of all possible execution traces of SysML-Sec model (proof detailed in [15]).

For checking properties like confidentiality, ProVerif tries to prove it by finding all possible *execution traces* that would lead to a violation of this property in an *approximated model*. This approximated model—which is needed since proving secrecy properties in the Dolev-Yao model has been proved to be undecidable in the general case [4,10]—is constructed so that each possible trace on the *real* model produces a possible trace in the approximated model. As such, ProVerif can issue three types of results (given for secrecy here):

- Property is **true**. ProVerif did not find any trace leading to a violation of the property in the approximated model. Since the approximation is sound, this means that the property is true also on the real model.
- Property is **false**. ProVerif has found a trace on the approximated design and has managed to construct a corresponding trace on the real model. The trace found is provided with the result by ProVerif.
- Property **cannot be proved**. ProVerif has found a trace on the approximated design but this trace did not match a valid trace on the real model. In this case, ProVerif is not able to conclude but the trace on the approximated model is returned so that the designer can decide whether this matches a valid trace or not.

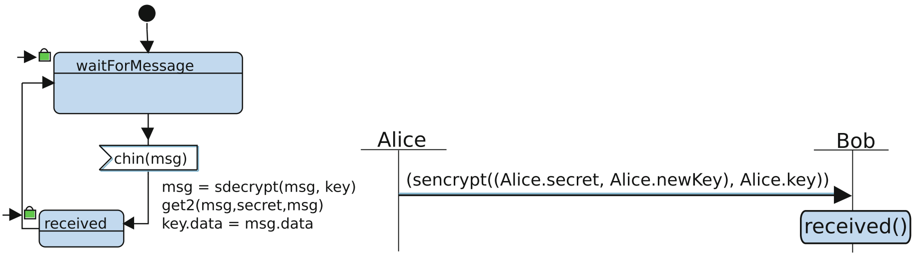
We keep these three possible results and make them available to the designer through the TTool interface.

4.2 Verification Results in TTool

In order to enable the designer to simultaneously see the results of the previous verification and accordingly continue modeling, verification results are displayed on the diagrams that are built by the designer. Results for the reachability, confidentiality and authenticity properties are displayed on the block and state machine diagrams in the form of green (when property is true) or red (when property is false) locks. For instance, we can see in Fig. 3a that the *waitForMessage* and *received* states are reachable. Also, in order to ease debugging and when it is available, the designer is provided with a trace that shows why the property is true (for instance how a state is reachable) or false (how a secret can be disclosed). This trace is automatically constructed based on the trace issued by ProVerif and displayed as a sequence diagram. As such, the trace presents the messages exchanged by the participants (all blocks and the attacker) and the states that each block goes through. As shown in Fig. 3b, we see how the *received* state inside Bob's state machine can be reached by receiving the message sent by Alice to Bob containing the data: (sencrypt((Alice.secret, Alice.newKey), Alice.Key).

5 Related Work

Assessing security properties when designing software components mostly relies on formal approaches. For example, [20] proposes verifying cryptographic protocols with a probabilistic analysis approach. Protocols are represented as trees whose nodes capture knowledge while edges are assigned transition probabilities. Although these trees could include malicious agents in order to model attacks and threats, nevertheless security properties are not explicitly represented. Moreover, for threat analysis, attacks should be explicitly expressed and manually solved. [21] defines a formal basic set of security services for accomplishing security goals. In this approach, security property analysis strongly relies on the designer's experience. Moreover, threat assessment is not easily feasible. There are numerous approaches for formal verification of security properties. Most of them are not automated and cannot be used as an engineering tool e.g. [9,17] and [2]. Among the research dedicated to engineering-oriented security verification that we are aware of, the closest are [13,14] and [19]. UMLsec [13] is a modeling framework aimed at defining security properties of software components and their composition within a UML framework. It also features a rather complete framework addressing various stages of model-driven secure software engineering from the specification of security requirements to tests, including



(a) State machine of Bob

(b) Seq. diag. of an attack trace

Fig. 3. Trace expressing ProVerif results (Color figure online)

logic-based formal verification regarding the composition of software components. In [14], Kordy et al. exposed a formal description of attack-defense trees. In these diagrams, interactions between the attacker and the system (defender) are modeled as attacks and countermeasures. In this sense, our approach is different as it relies on attacker capabilities and on a description of the system behaviour, meaning that the verification algorithm presented in this paper is able to prove that a design is secure against a certain class of attacker, without prior knowledge of the form of the attack. On the other hand, verification algorithms on attack-defense trees can solely prove that a countermeasure is efficient against a specific attack. More recently, [19] developed an expanded UML model extending the sequence diagrams of UML for security protocol verification. Their approach includes translating models into ProVerif for verification of confidentiality and correspondence. While sequence diagrams are particularly well suited to evaluating observational equivalence properties as they show the messages exchanged between participants, state machine diagrams –as used in this paper– allow modeling of precise behavioural properties more intuitively (such as conditional statements or loops). Furthermore, our process includes verification of weak and strong authenticity.

This paper expands on previous publications on SysML-Sec, proposing how to better model certain situations (e.g., loops) and their models-to-ProVerif transformation, taking into account the capabilities and limitations of ProVerif. We thus manage to limit cases where the proof of security properties would fail, without impacting the verification capabilities of SysML-Sec diagrams.

6 Conclusion

The paper describes a formal and novel Model-Driven Approach for (safety) and security modeling and verification of embedded systems. The paper itself focuses on the formal SysML-to-ProVerif transformation, and sketches a proof of the soundness of our approach. Last but not least, this new transformation is already available in TTool, and it includes backtracing capabilities. The overall approach is exemplified with a toy example. However, it has already been successfully applied to a large range of systems, including an authenticated and non-authenticated versions of the TLS protocol, an implementation of the X3DH protocol used by messaging applications such as Signal/Telegram or a key exchange protocol targeting Intel SGX architecture, and the design of the embedded architecture of an autonomous vehicle. Our formal description set the frameworks for a future proof of equivalence or soundness. Proof limitations of ProVerif could also be addressed using other proving techniques, e.g. relying on Prolog.

References

1. Abadi, M., Blanchet, B.: Analyzing security protocols with secrecy types and logic programs. J. ACM **52**, 102–146 (2005)
2. Ali, Y., El-Kassas, S., Mahmoud, M.: A rigorous methodology for security architecture modeling and verification. In: Proceedings of the 42nd Hawaii International Conference on System Sciences (2009)
3. Allamigeon, X., Blanchet, B.: Reconstruction of attacks against cryptographic protocols. In: 18th IEEE Workshop on Computer Security Foundations, CSFW-18 2005 (2005)
4. Amadio, R.M., Lugiez, D., Vanackère, V.: On the symbolic reduction of processes with cryptographic functions. Theor. Comput. Sci. **290**, 695–740 (2003)
5. Apvrille, L., Roudier, Y.: Designing safe and secure embedded and cyber-physical systems with SysML-Sec. In: Desfray, P., et al. (eds.) Model-Driven Engineering and Software Development, vol. 580, pp. 293–308. Springer, Switzerland (2016). https://doi.org/10.1007/978-3-319-27869-8_17
6. Armando, A., et al.: The AVISPA tool for the automated validation of internet security protocols and applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_27
7. Blanchet, B., et al.: An efficient cryptographic protocol verifier based on prolog rules. In: CSFW, vol. 1, pp. 82–96 (2001)
8. Blanchet, B., Smyth, B., Cheval, V.: Automatic cryptographic protocol verifier. User Manual and Tutorial, Technical report (2015)
9. Drouineaud, M., Bortin, M., Torrini, P., Sohr, K.: A first step towards formal verification of security policy properties for RBAC. In: QSIC 2004 (2004)
10. Durgin, N., Lincoln, P., Mitchell, J., Scedrov, A.: Undecidability of bounded security protocols. In: Workshop on Formal Methods and Security Protocols (1999)
11. Eames, D.P., Moffett, J.D.: The integration of safety and security requirements. In: Felici, M., Kanoun, K. (eds.) SAFECOMP 1999. LNCS, vol. 1698, pp. 468–480. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48249-0_40
12. OM Group: System modeling language specification (SysML), version 1.5. Technical report
13. Jürjens, J.: Developing secure embedded systems: pitfalls and how to avoid them. In: 29th International Conference on Software Engineering (2007)
14. Kordy, B., Mauw, S., Radomirović, S., Schweitzer, P.: Foundations of attack–defense trees. In: Degano, P., Etalle, S., Guttman, J. (eds.) FAST 2010. LNCS, vol. 6561, pp. 80–95. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19751-2_6
15. Lugou, F.: Environments for analyzing the security of smart objects. Ph.D. thesis, Télécom ParisTech, France (2018)
16. Lugou, F., Li, L.W., Apvrille, L., Ameur-Boulifa, R.: SysML models and model transformation for security. In: 4th International Conference on Model-Driven Engineering and Software Development (2016)
17. Maña, A., Pujol, G.: Towards formal specification of abstract security properties. In: The Third International Conference on Availability, Reliability and Security. IEEE (2008)
18. Pedroza, G., Knorreck, D., Apvrille, L.: AVATAR: a SysML environment for the formal verification of safety and security properties. In: The 11th IEEE Conference on Distributed Systems and New Technologies, NOTERE 2011 (2011)

19. Shen, G., Li, X., Feng, R., Xu, G., Hu, J., Feng, Z.: An extended UML method for the verification of security protocols. In: 19th International Conference on Engineering of Complex Computer Systems (ICECCS) (2014)
20. Toussaint, M.J.: A new method for analyzing the security of cryptographic protocols. IEEE J. Sel. Areas Commun. **11**, 702–714 (1993)
21. Trcek, D., Blazic, B.J.: Formal language for security services base modelling and analysis. Elsevier Sci. J. Comput. Commun. **18**, 921–928 (1995)