AVATAR: A SysML Environment for the Formal Verification of Safety and Security Properties

Gabriel Pedroza, Ludovic Apvrille, Daniel Knorreck
System-on-Chip laboratory (LabSoC), Institut Telecom, Telecom ParisTech, LTCI CNRS
2229, route des Crêtes, B.P. 193, F-06904 Sophia-Antipolis Cedex
Email: {gabriel.pedroza, ludovic.apvrille, daniel.knorreck}@telecom-paristech.fr

***Abstract*—Critical embedded systems - e.g., automotive systems - are now commonly distributed, thus exposing their communication links to attackers. The design of those systems shall therefore handle new security threats whilst maintaining a high level of safety. To address that issue, the paper introduces a SysML-based environment named AVATAR. AVATAR can capture both safety and security related elements in the same SysML model. TTool [1], an open-source UML toolkit, provides AVATAR editing capabilities, and offers a press-button approach for property proof. Indeed, after having modeled an abstract representation of the system and given a description of the safety and security properties, the designer may formally and directly verify those properties with the well established UPPAAL and ProVerif toolkits, respectively. The applicability of our approach is highlighted with a realistic embedded automotive system taken from an ongoing joint project of academia and industry called EVITA [2].**

I. Introduction

The ever increasing prevalence of distributed systems has triggered the necessity to examine their resistance to security threats, in addition to safety issues. The extensive need for data exchange between system entities is satisfied by dedicated physical devices like field buses, or shared media like networks. Even if it is sometimes possible to physically protect dedicated links from malicious access, bets are off when wireless communication comes into play. Thus, the sophisticated communication patterns required by today's embedded systems open the door to new kind of attacks. Security flaws in systems may for instance compromise the confidentiality of messages or the privacy of users. Moreover, many of these embedded systems are safety critical, i.e., failures may not only have a severe economic impact but also endanger human beings. Examples are systems used in the field of transportation, energy generation and distribution, medical appliances, etc. In this paper, an automotive application serves as a representative of this class of systems. Indeed, while being critical, car systems are soon expected to offer advanced communications with other external systems, e.g., with road sign units, or with other cars [2].

A lot of work is concerned with modeling environments capturing safety properties during all development stages, including requirements engineering and design. To reduce verification complexity, formal proofs at design stage are usually applied to high-level models, or to subparts of more detailed designs. Unfortunately, current approaches are mostly dedicated to safety properties. And so, the developer has to come up with additional models relying on different formalisms amenable to security proofs. Additional effort has therefore to be spent on the creation and maintenance of two models - one for safety issues, and one for security ones - instead of one, and also on model consistency.

To overcome this limitation, we have introduced a SysML environment called AVATAR based on insights obtained from previous formally defined UML profiles [3] [4] [5]. AVATAR models are able to express both safety and security properties and provide means for their verification. The major contribution is that proofs can be conducted from the same model, and at the push of a button. Safety and security requirements are expressed in terms of SysML Requirement Diagrams, whereas the static and the behavioral aspects of the system are represented with Block and State Machine Diagrams respectively. Safety properties are further refined within Parametric Diagrams, and security properties are described within specific pragmas of Block Diagrams. Finally, safety and security proofs are accomplished by first transforming the SysML model to the respective domain specific language: *UPPAAL* for safety proofs, and *ProVerif* for security proofs. Modeling features and translators are implemented in TTool [1], an open-source UML toolkit supporting several profiles in addition to AVATAR.

Safety properties modeling and verification in AVATAR has been already introduced in [6]. This paper covers security issues only. The case study is taken from the European EVITA project [2], which focuses on security and safety matters of embedded automotive systems.

The paper is organized as follows. Section II reviews environments for modeling and proving safety or security properties. Section III introduces the AVATAR SysML environment, as published in [6]. The next section (section IV) explains how AVATAR can be efficiently extended to support security properties. The semantics of AVATAR for security is described in section V. A case study (section VI) illustrates the modeling and proof of security properties with AVATAR, based on an example extracted from the EVITA project. At last, section VII concludes the paper.

II. Related Work

A. Safety Properties

The following graphical environments offer a high level front-end for formal languages, and are mostly based on UML or SysML. In addition to the shortcoming of not addressing security features, the main differences to the AVATAR environment consist in the expressiveness of the respective profile, the underlying formal language, to what extent details of the latter are masked to the designer and the model transformation which may be more or less automated.

[7] relies on timed automata to model and analyze timeliness properties of embedded systems. The UPPAAL model checker is used to evaluate the automata which must be created manually. There is no automated translation routine from a high level language (UML,...) and thus the creation of the automata turns out to be error prone, and cannot easily be integrated into a wider-ranger methodology, including for example requirements capture and tracing.

[8] provides means for formal and simulation based evaluation of UML/SysML models for performance analysis of SoCs. UML Sequence diagrams constitute the starting point for the functional description. They are subsequently transformed into so called communication dependency graphs (CDGs) which capture the control flow, synchronization dependencies and timing information. CDGs are in turn amenable to static analysis in order to determine key performance parameters like BCRT and WCRT [1], and also I/O data rates. A drawback of this approach is that data flow independence has to be kept, thus preventing more sophisticated control flow structures to be modeled.

[9] advocates a nice graphical notation to formally capture requirements. System executions are expressed in the form of time line diagrams discriminating optional, mandatory, fail events and related constraints. As for other trace based approaches, conditional or varying system behavior cannot easily be expressed. Moreover, the approach does not address real-time or performance requirements.

The MARTE profile embraces VSL [10] which aims at specifying the values of constraints, properties and stereotype attributes particularly related to non-functional aspects. Even when used in combination with sequence diagrams, VSL makes it cumbersome if not impossible to specify complex sets of sequential behaviors.

The Rhapsody tool used by [11] similarly enables formal verification of SysML diagrams using UPPAAL. Unlike TTool, Rhapsody does not distinguish between requirements and properties. Nor it supports a property expression language - such as TEPE [6] - and computation operators in state machines. In terms of user-friendliness, TTool allows one to right-click on an action symbol and automatically verify the reachability of that action. In the same situation, the user of Rhapsody is obliged to manually enter a logic formula. The OMEGA2 environment [12] also has strong connections with Rhapsody for it implements the same semantics. OMEGA2 supports requirement diagrams as defined in SysML. Conversely ARTISAN [13] extends SysML to cope with continuous flows. ARTISAN models may contain probabilities and interruptible regions, two concepts not yet supported by AVATAR. The open-source environment Topcased also enables requirement modeling in a SysML fashion [14].

Electronic System Level (ESL), which is an emerging electronic design methodology, has stimulated research work on joint use of SysML and formal languages supported by simulation tools. Several papers discuss solutions where a model is designed in SysML and translated into VHDL-AMS [15] or Simulink [16]. Mechanical engineering is another area where SysML is combined with already existing domain specific languages, such as Modelica or bond graphs.

B. Security Properties

A wide range of methodologies and tools has been proposed for modeling and verifying security in embedded systems.

[17] proposes to verify cryptographic protocols with a probabilistic analysis approach. Protocols are represented as trees whose nodes capture knowledge whilst edges are assigned with transition probabilities. Although these trees could include malicious agents in order to model attacks and threats, security properties are nonetheless not explicitly represented. Moreover, for threat analysis, attacks should be explicitly expressed and manually solved. [18] defines a formal basic set of security services for accomplishing security goals. In this approach, security properties analysis strongly relies on designer's experience. Moreover, threat assessment is not easily feasible.

In more recent efforts, temporal logic languages are used for expressing security properties. For instance, [19] embeds a first order Linear Temporal Logic (LTL) in the theorem prover Isabelle/HOL, making it possible to model both a system and its security properties. Although this is a rigorous approach, security properties and goals can be built upon security concepts, unfortunately leading to non-easily reusable specific models. Another example of temporal logic based verification is presented in [20]. In that approach the designer has to establish assumptions, knowledge and communication axioms, and represent them in a temporal logic language, and so specialized skills are definitely necessary. Additionally, the proposal only targets authenticity and is protocol oriented.

Evaluating both security and performance is tackled in [21]. If the methodology aims to offer a good trade-off between Quality of Service (QoS) and Security, it nonetheless requires a qualitative evaluation of security leaks. Indeed, when security flaws are detected, the designer must decide to use a new security mechanism - and so to degrade performance - or to keep the leak, in case the designer assumes that it is of low importance in the system. As a consequence, this approach strongly relies on designer skills and experience.

Assessment of security in embedded systems mostly relies on formal approaches. However, [22] mixes formal and informal security properties. The authors argue that unified security

[1] Best Case Response Time, Worst Case Response Time

approaches won't provide enough flexibility to cope with highly heterogeneous requirements of distributed scenarios. Thus, the framework allows a user to define his own security properties and create dependencies between them, making the model probably difficult to reuse. Furthermore, the overall verification process is not completely automated, again requiring specific skills. Analogously, the Software Architecture Modeling (SAM) framework [23] aims to bridge the gap between informal security requirements and their formal representation and verification. Indeed, SAM uses formal and informal security techniques to accomplish defined goals and mitigate flaws. Thus, liveness and deadlock-freedom properties can be verified on LTL models relying on the Symbolic Model Verifier (SMV). Even if SAM relies on a well established toolkit - SMV - and considers a threat model, the "security properties to proof" process is not yet automated.

UMLsec [24] defines how to integrate security design elements, and security properties in a UML methodology. However, design elements and security properties are mixed on the same diagrams, as well as functional and non functional requirements. Also, UMLsec is not formally defined nor it covers threat analysis (e.g., attack trees).

As a conclusion, all these solutions usually make a trade-off between rigorousness and simplicity. On the one hand, security in embedded systems obviously requires rigorousness in the formal verification process. On the other hand, the complexity and diversity of systems and requirements, along with time-to-market and software-engineering criteria advocate for user-oriented tools with automated and simplified verification. AVATAR positively answers that need: it is based on a popular and friendly language (SysML), it is supported by an open-source toolkit (TTool) which relies on a recognized security verification toolkit (ProVerif). AVATAR further supports threat analyses and code generation, therefore supporting all secure system development phases: analysis, design, formal proof and code generation. On the contrary, other tools such as ST-Tool [25] and STA [26] are dedicated to one given system development phase, and based on non-reusable models.

III. The AVATAR SysML environment

A. Basics

The AVATAR environment reuses eight of the SysML diagrams (Package diagrams are not supported). It further structures Sequence Diagrams using an Interaction Overview Diagram (a diagram defined in UML2, not by SysML). The AVATAR profile is syntactically and semantically defined by a meta-model. Besides a syntax, a semantics and a tool support, a profile is also characterized by a methodology.

B. Methodology

The AVATAR methodology comprises the following stages:

1) **Requirement capture**. Requirements and properties are structured using AVATAR Requirement Diagrams. At this step, properties are just defined with a specific label as test cases.
2) **System analysis**. A system may be analyzed using usual UML diagrams, such as Use Case Diagrams, Interaction Overview Diagrams and Sequence Diagrams.
3) **System design**. The system is designed in terms of communicating SysML blocks described in an AVATAR Block Diagram, and in terms of behaviors described with AVATAR State Machines.
4) **Property modeling**. The formal semantics of properties is defined within TEPE [6] Parametric Diagrams (PDs). Since TEPE PDs involve elements defined during the system design phase (e.g, a given integer attribute of a block), TEPE PDs may be defined only after a first system design has been performed.
5) **Formal verification** of safety properties can finally be conducted over the system design, and for each test case.

Once all properties are proved to hold, requirements, system analysis and design, as well as properties may be further refined. Thereafter, and similarly to most UML profiles for embedded systems, the AVATAR methodological stages are reiterated. Having reached a certain level of detail, refined models may not be amenable to formal verification any more. Therefore the generation of prototyping code may become the only realistic option.

C. Block and State Machine Diagrams

Apart from their formal semantics, AVATAR Block and State Machine Diagrams only have a few characteristics which differ from the SysML ones.

An AVATAR block defines a list of attributes, methods and signals. Signals can be sent over synchronous or asynchronous channels. Channels are defined using connectors between ports. Those connectors contain a list of signal associations.

A block defining a data structure merely contains attributes. On the contrary, a block defined to model a sub-behavior of the system must define an AVATAR State Machine.

AVATAR State Machine Diagrams are built upon SysML State Machines, including hierarchical states. AVATAR State Machines further enhance the SysML ones with temporal operators:

- **Delay:** $after(t_{min}, t_{max})$. It models a variable delay during which the activity of the block is suspended, waiting for a delay between t_{min} and t_{max} to expire.
- **Complexity:** $computeFor(t_{min}, t_{max})$. It models a time during which the activity of the block actively executes instructions, before transiting to the next state: that computation may last from t_{min} to t_{max} units of time.

The combination of complexity operators ($computeFor()$), delay operators, as well as the support of hierarchical states - and the possibility to suspend an ongoing activity of a substate - endows AVATAR with main features for supporting real-time system schedulability analysis.

IV. Expressing Security in AVATAR

A. Current limitations of AVATAR for security

Let's consider the following Alice and Bob system, in which Alice wants to send a confidential data to Bob, using a pre-

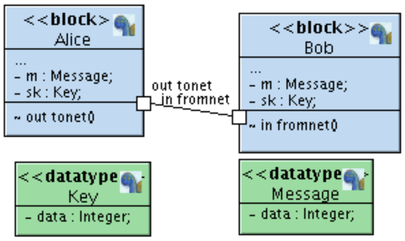


Fig. 1. Example 1: Block diagram of the Alice and Bob system

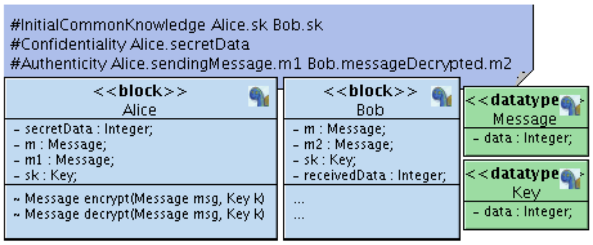

#InitialCommonKnowledge Alice.sk Bob.sk
#Confidentiality Alice.secretData
#Authenticity Alice.sendingMessage.m1 Bob.messageDecrypted.m2
<<block>>
Alice
- secretData : Integer;
- m : Message;
- m1 : Message;
- sk : Key;
~ Message encrypt(Message msg, Key k)
~ Message decrypt(Message msg, Key k)
<<block>>
Bob
- m : Message;
- m2 : Message;
- sk : Key;
- receivedData : Integer;
...
...
<<datatype>
Message
- data : Integer;
<<datatype>
Key
- data : Integer;

Fig. 2. Example 2: Block diagram of the Alice and Bob system

shared symmetric key, and relying on an authentic message containing the ciphered confidential data of Alice. Figure 1 presents the SysML internal block diagram of this Alice and Bob system.

Several limitations make it difficult to prove confidentiality and authenticity properties in this toy system:

- **Initial knowledge**: AVATAR provides no way to pre-share data, i.e., make data common to several blocks before the system starts.
- **Cryptographic functions**: Cryptographic systems commonly rely on a set of well-known functions (symmetric cipher and decipher, compute MAC, etc.) that are not defined in AVATAR, and have thus to be explicitly modeled by a designer.
- **Communication architecture**: AVATAR channels cannot be listened by an external Block, i.e., a network on which an attacker could listen packets has to be explicitly modeled.
- **Attacker model**: AVATAR does not include any attacker model. Having a default attacker model, e.g., based on Dolev-Yao [27], would avoid users of AVATAR to have to model by hand an attacker model.
- **Security properties**: Security properties cannot be defined in AVATAR Requirement Diagrams, nor in TEPE, nor at Block Diagram level.

B. Extending AVATAR for security purpose

We now show how we have addressed all aforementioned limitations in AVATAR.

- **Initial knowledge**

SysML offers several ways to share data between classes, using for example *block attributes*, or using a dedicated block storing that shared knowledge. Unfortunately, those two solutions suffer two drawbacks:

1) The sharing is not really explicit, i.e., it is not clear which block intends to use a given block attribute, or given data of a dedicated block.
2) The sharing is defined for the entire system execution: in security, we are interested by the **pre** sharing of information, not by the sharing of this information during the entire system execution.

To overcome those two limitations, we propose to use specific directives - or pragmas - in notes of Block Diagrams. The pragma is as follows:

```
♯ InitialCommonKnowledge BlockID.attribute
[BlockID.attribute]*
```

Figure 2 contains the internal block diagram of the Alice and Bob system enhanced with AVATAR extensions for security. Similarly to the first design, the new design has two blocks (*Alice* and *Bob*), and declares two data structures (*Key* and *Message*). A note now declares that sk is a pre-shared data (a key) between Alice and Bob:

```
♯ InitialCommonKnowledge Alice.sk Bob.sk
```

- **Cryptographic functions**

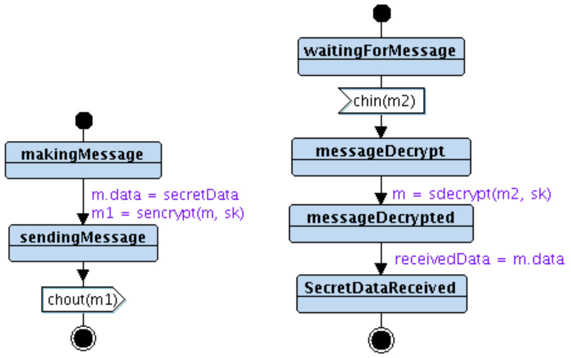
AVATAR includes the definition of a specific block called *cryptographic block*. That block defines a set of cryptographic functions that can be used as usual methods by the state machine of these blocks (some methods are visible in Figure 2). For example, Alice and Bob declare a set of cryptographic methods. A few examples of cryptographic functions we have defined:

- $encrypt(Message\ msg, Key\ k)$ and $decrypt(Message\ msg, Key\ k)$, for encrypting and decrypting messages with asymmetric keys, respectively.
- $sencrypt(Message\ msg, Key\ k)$ and $sdecrypt(Message\ msg, Key\ k)$, for encrypting and decrypting messages with symmetric keys, respectively.
- $MAC(Message\ msg, Key\ k)$ and $verifyMAC(Message\ msg, Key\ k, Message\ macm)$, for computing the MAC of a message, and verifying the MAC of a message, respectively.

For instance, the behavior of *Alice* and *Bob* is provided within two respective State Machine Diagrams (see Figures 3-a and 3-b, respectively). Alice first puts its $secretData$ into a message $m.data = secretData$, then encrypts this message $m1 = sencrypt(m, sk)$ with the symmetric encryption function, and finally sends the resulting message on the broadcast channel $chout(m1)$. *Bob* waits for a message on the broadcast channel $chin(m2)$. Then, *Bob* tries to decrypt the received message $m = sdecrypt(m2, sk)$ and then extracts from the message the $secretData$ sent by Alice: $receivedData = m.data$.

- **Communication architecture**

AVATAR communications were firstly based upon unidirectional one-to-one synchronous or asynchronous communications. We now assume that each block has two special signals: $chout$ and $chin$, which can be respectively used for sending and receiving purpose respectively. No channels need to be declared for using those signals, which are considered to



a – Alice b – Bob

Fig. 3. Example 2: State Machine Diagrams of the Alice and Bob system

be sent on / received from a public broadcast asynchronous channel common to all blocks. This public broadcast channel is particularly interesting for modeling messages that can be probed by an attacker.

- **Attacker model**

In AVATAR, the attacker model is implicit, i.e., there is no need to model an attacker either at Block Diagram level, or at State Machine Diagram level. More precisely, AVATAR relies on the attacker model of ProVerif, see section V-B.

- **Security properties**

Security properties are modeled in two different stages of the AVATAR methodology (see section III): at requirement capture stage, and at property modeling stage.

1) Security-oriented requirements: We have already published an extension of SysML Requirement Diagrams supporting security requirements [28]. Basically, our contribution relies on the identification of attacks represented within attack trees in a SysML Parametric Diagram, on the use of a security-domain identifier (e.g., *confidentiality*, *authenticity*, *privacy*, etc.) added to SysML requirements, and on a precise methodology applied to identify these security requirements, based on use cases, attack trees, and a very first design of system functions.

2) Security properties: TEPE [6] has already been proposed for modeling safety-related properties in AVATAR. However, safety properties are commonly complex to express. Indeed, they generally involve several attributes and signals of blocks, as explained in [6] on an elevator system. On the contrary, security properties can be usually defined with a type (e.g., confidentiality), and with elements related to that kind (e.g., the confidentiality of the attribute of a block). This simplicity advocates for a simple modeling solution, not based on a different diagram. Finally, our solution relies on *pragmas* provided in notes of Block Diagrams: *confidentiality* and *authenticity* can be directly expressed at this level.

Confidentiality

Confidentiality in AVATAR can be modeled as a simple pragma provided in the note of a Block Diagram. The confidentiality must be specified as the confidentiality of an attribute of a block:

```
♯ Confidentiality block.attribute
```

Coming back to the example provided in Figure 2, the following statement models that the attribute $secretData$ of *Alice* shall remain confidential i.e., never accessible to an attacker:

```
♯ Confidentiality Alice.secretData
```

Authenticity

Authenticity in AVATAR is also modeled as a pragma. An authenticity pragma states that a message $m2$ received by a block $block2$ was necessarily sent before in a message $m1$ by a block $block1$. The authenticity pragma specifies two states: one of the sender block, i.e. one state $s1$ of $block1$, and one state $s2$ of $block2$. Also, in the state machine diagram of $block1$, $s1$ corresponds to the state right **before** the sending of $m1$. Analogously, $s2$ corresponds to the state right **after** message $m2$ has been received and accepted as authentic. Finally, the authenticity pragma is as follows:

```
♯ Authenticity block1.s1.m1 block2.s2.m2
```

For example, in Figure 2, the authenticity pragma states that all messages $m1$ sent by *Alice* after state $sendingMessage$ shall be authentic for *Bob* receiving it into a message named $m2$ before its state $messageDecrypted$.

```
♯ Authenticity Alice.sendingMessage.m1
Bob.messageDecrypted.m2
```

V. Semantics of AVATAR

A. General approach

Previous section has emphasized several particularities of AVATAR which obviously have to be taken into account by the underlying proof mechanism. Just to mention a few of these particularities: AVATAR assumes public broadcast channels between blocks that can be listened up by an attacker. AVATAR also assumes a Dolev-Yao attacker model [27].

Several environments target the proof of security properties, e.g., SHVT [29], AVISPA [30], and ProVerif [31]. Proofs within the SHVT environment cannot be automatically conducted: so, it was excluded. In AVISPA, the tool is system dependent, and really focused on cryptographic protocols, which is not the case of AVATAR which targets embedded systems in general. ProVerif is based on process algebra, and is therefore well suited for modeling communicating entities as found in embedded systems. Also, to our experience, ProVerif nicely solves the trade-off between expressiveness, complexity and automation of formal approaches.

B. ProVerif

ProVerif [31] is a toolkit that relies on Horn clauses resolution for the automated analysis of security properties over cryptographic protocols. ProVerif takes as input a set of Horn Clauses, or a specification in pi-calculus (a process algebra) and a set of queries. ProVerif outputs whether each query is satisfied or not. In the latter case, ProVerif tries to identify a trace explaining how it came to the conclusion that a query is not satisfied.

In ProVerif, a specification takes the form of a system represented as *spi-calculus processes* and properties represented as *queries*.

- **Processes** are themselves composed of a *declaration part*, of *a definition of a set of sub-processes*, and the definition of a *main process*.
  The declaration part can be used to declare global terms, including channels. Functions and functions reduction can also be declared.
  Main process constructs are summarized in Table I, the semantics of which is explained informally and in pi-calculus. [32] offers a complete specification of the ProVerif grammar.

TABLE I
MAIN PROVERIF PROCESS CONSTRUCTS

Construct	Semantics
`out(c,M); P`	Sending of M in channel c, and execution of process P, i.e., $\bar{c}\langle M\rangle.P$.
`in(c, M);P`	Receiving of Message M from channel c, and execution of process P, i.e., $c(M).P$.
`new a; P`	Definition of a new term a, and subsequent execution of process P, i.e., $(\nu a)P$.
`begin(M); P`	Execution of an event, and subsequent execution of process P, i.e., *begin*$(M).P$.
`!P`	Replication of process P, i.e., an infinite number of instances of P are executed.
`P|Q`	Parallel composition between processes P and Q.
...	... other pi-calculus constructs, such as *if then* constructs, etc.

- **Properties** are represented with ProVerif *queries*. Queries are formal clauses in which the left hand side of the implication is a set of facts that should be accomplished whilst the right hand side includes the hypotheses to be verified. Queries can be used to express confidentiality [33] and authenticity requirements [31].
  Confidentiality queries directly express which data shall not be accessible to the attacker, e.g., that a private key shall not be accessible to an attacker:

```
query attacker:myKey.
```

Authenticity of messages relies on ProVerif events. Whenever a message `m` sent by a process `A` to a process `B` shall be authenticated, one event shall be included in each process: one shall be included in `A` before the sending of `m` (e.g., `eventSendM`), and one after the receiving of `m` (e.g., `eventReceiveM`). Since the attacker is not allowed to execute events, it suffices to prove that to each receiving event of `m` corresponds exactly one sending of `m`. Finally, an injective query is used to model authenticity:

```
query evinj:eventReceiveM(x) ==>
      evinj:eventSendM(x).
```

Important note: ProVerif also makes it possible to study the reachability of events, based on queries. And so, ProVerif may also be used for proving safety properties. Reachability is always studied on the system augmented with the attacker.

ProVerif integrates its own **attacker model**, which is itself a process implementing a Dolev-Yao approach [27]. This process acts like an adversary relying upon a set of known names, variables and terms which is referred to as *knowledge*. Attacker knowledge increase relies on public channel probing and execution of functions non prohibited to the attacker. Finally, ProVerif is not intended to perform computational attacks nor proves, but CryptoVerif [34] could be used for that purpose.

To verify a query, ProVerif implements a resolution algorithm [31], [35] [36] that first translates the complete pi-process specification to Horn Clauses [31]. To verify a query, the resolution algorithm determines, based upon a set of inference rules, if the attacker reasoning is able to derive a trace that contradicts the query, thus proving that the query is false. Otherwise, if the attacker is unable to find such a trace, then the property is satisfied. Additionally, if facts on which the query is based upon are not reachable, the algorithm informs that the query can not be proved.

C. Translation

The translation process takes as parameter an AVATAR design, including its sets of pragmas (as defined in section IV), and outputs a ProVerif specification.

Briefly, the translation process is as follows:

Definition 1: AVATAR translation process

Let $\mathcal{T}$ the translation process that takes as input a Block Diagram BD, and a set of pragmas P, and Pr the resulting ProVerif specification:

$Pr = \mathcal{T}(BD, P)$.

- A BD is composed by three graphical entities named $<< block >>$, $<< datatype >>$ and $<< pragmas >>$. A block contains a set of attributes, a set of functions, a set of signals and a reference to a State Machine Diagram (SMD). $<< datatype >>$ can be ignored for the translation process since they can easily be removed.
- An SMD is a set of interconnected logical operators: *start states*, *stop states*, *transitions* - with attribute settings and function calls -, *choices*, *states*, *sending in a channel*, *receiving from a channel*.
- The type of a pragma in P is either $InitialCommonKnowledge$, $Confidentiality$, or $Authenticity$.

$\mathcal{T}$ applies the following set of rules:

1) For each block $b \in BD$, a "first" process `fp` is generated. Then, for each state s of the State Machine Diagram *smd* of b, another process `ps` is generated.
2) `fp` instantiates all attributes that are not listed in $InitialCommonKnowledge$ or $Confidentiality$ pragmas: '`new attr;`'. Then, `fp` makes a call to the `ps` process corresponding to the start state of *smd*.
3) Each `ps` is created as follows. An event is first called for tracing the reachability of states '`event entering_state_nameofs();`'. Then, each branch of logical operators linked from s is considered until another state is reached on that branch:

- Sending on a channel c of a message m is translated as an 'out(c, m);'.
- Receiving on a channel c of a message m is translated as an 'in(c, m);'.
- The assignment of a variable is translated using a 'let' operator, e.g.:
  'let m1.data = (m2.data, m3.data);'.
- The call of a cryptographic function is translated with a ProVerif cryptographic function and a 'let' operator:
  'let mac = MAC(msg1.data, Key.data);'.
- The call of a non-cryptographic function is translated with a simple call to an event having the name of the corresponding function, and with the same parameters, e.g., 'event function(par0, par1);'.
- The various branches starting from state s are selected using the 'if...else' ProVerif statement.

4) The main process mp of the ProVerif specification instantiates all attributes listed in $InitialCommonKnowledge$ pragmas. Then, it instantiates in parallel, and for an infinite number of sessions, all fp processes, e.g., '(!fp$_1$)|(!fp$_2$)|...|(!fp$_n$)'.

5) $Confidentiality$ pragmas referencing a block b and an attribute $attr$ of b are translated as a declaration of $attr$ as follows: 'private free attr.' and with a query of the following form:
'query attacker:attr.'

6) $Authenticity$ pragmas of the form $b1.state1.attr1 b2.state2.attr2$ are translated using statements of the following form: 'query evinj:b2_state2(attr2) ==> evinj:b1_state1(attr1).'.
Additionally, in the process ps where $s = s1$, a call to 'event b1_state1(attr1);' is added at the beginning of the process. Similarly, a call to 'event b2_state2(attr2);' is added at the beginning of the process ps where $s = s2$.

A few of these translations rules are presented in tables II and III. Even if AVATAR supports algebraic integer operations (+, −, /) in SMD transitions, those operations have no meaning at ProVerif level and so, they are translated as a new variable instantiation using the 'new <variable_name>;' statement. Similarly, only basic boolean expressions of guards can be directly converted into ProVerif (e.g., comparison between two variables).

TABLE II
A FEW AVATAR BLOCK DIAGRAM → PROVERIF TRANSLATION RULES

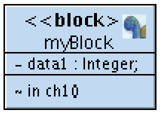
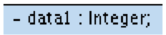
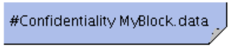
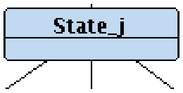
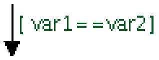
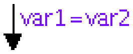
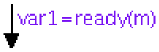
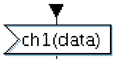
AVATAR	ProVerif	Semantics
Block declaration 	let *myBlock*$_0$=	Block declaration with name *myBlock*. The initial process *myBlock*$_0$ is declared
Block data types 	new *data1*;	Integer attribute named *data1*
Block input signal	free *ch1*.	Declaration of input channel *ch1*. Quite similar rule for *output* or *private* channel
Common knowledge *val* in blocks $\{B_i\}$	new *val*;	The static variable *val* preshared by $\{Block_i\}$, *i=1...n*, has the same initial value
Confidentiality pragma 	query attacker: *data*.	*data* is defined as confidential

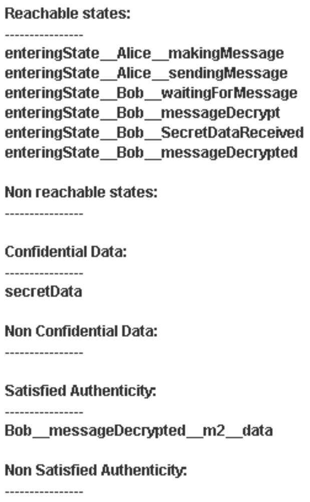
TABLE III
A FEW AVATAR STATE MACHINE DIAGRAM → PROVERIF TRANSLATION RULES

AVATAR	ProVerif	Semantics
State declaration 	let $myBlock_i$= event *myBlock_State* $_j$(); ($myBlock_{i+1}$ |...| $myBlock_{i+n}$).	The state $State_j$ with n outgoing transitions. Each subprocess $myBlock_{i+k}$ corresponds to a single transition
Guard declaration 	if var1=var2 then $myBlock_i$.	Transition with condition *var1* equals to *var2*
Simple assignation 	let *var1=var2* in $myBlock_i$.	Transition with assignation of *var2* to *var1*
Assign predefined function	let *var1* = *encrypt*(*m*,*k*) in $myBlock_i$.	Transition with assignment of *var1* to result of pre-defined AVATAR method *encrypt*
Assign non predefined function 	event ready(m); new var1;	Transition with *ready* being a non predefined AVATAR function
Input signal 	in(*ch1*,*Data*);	Reads a signal from channel *ch1*, and stores its content in *Data*
Stop state	0.	The end of a process branch

D. Toolkit support

TTool [1] fully supports AVATAR, including the security extensions presented in this paper: TTool therefore implements a press-button approach for verifying confidentiality and authenticity security properties from AVATAR models: TTool executes $\mathcal{T}$, then it makes a call to ProVerif, and outputs the following results:

- For each $Confidential$ pragma, TTool indicates whether data provided in the pragma was proved to be secret, or not.



```
Reachable states:
----------------
enteringState__Alice__makingMessage
enteringState__Alice__sendingMessage
enteringState__Bob__waitingForMessage
enteringState__Bob__messageDecrypt
enteringState__Bob__SecretDataReceived
enteringState__Bob__messageDecrypted

Non reachable states:
----------------

Confidential Data:
----------------
secretData

Non Confidential Data:
----------------

Satisfied Authenticity:
----------------
Bob__messageDecrypted__m2__data

Non Satisfied Authenticity:
----------------
```

Fig. 4. Example of verification results as displayed by TTool

- For each $Authenticity$ pragma, TTool indicates whether the authenticity holds, or not.
- For each state s of each smd, TTool indicates whether s is reachable, or not. Reachability is studied for both the system and the attacker model running together, which means that a given state of the system may be reachable because of the attacker.
- Each query that could not be proved is also listed. An option makes it possible to obtain the trace leading to the query violation.

Figure 4 shows the results provided by TTool when applying the verification on the Alice-Bob example.

VI. Case study

A. Keying Protocol Description

This protocol was verified in the scope of the EVITA project [37]. Its specification is provided in [38].

The Keying Protocol with Key Master aims to securely distribute a randomly generated key among the members of a group of in-car Electronic Control Units (ECUs). The key to be distributed is referred to as Session Key (*SesK*). In our description, the ECU that creates the *SesK* is referred as generator (ECU1). The generator creates the *SesK* and sends it to the Key Master ECU (ECU-KM) for group distribution (see Figure 5). Since the Key Master owns the Pre-shared Symmetric Key (PSK) of every ECU in the group, the generator encrypts the *SesK* with its PSK. Further, this message includes a time stamp and is protected with a MAC (Message 1). After reception, the Key Master verifies Message 1 (See Figure 5) and in case of a valid request, the *SesK* is imported into an internal module named Hardware Security Module (HSM) [39]. From this point, the Key Master is responsible for *SesK* distribution. Consequently, the *SesK* is protected with the PSK of the respective target ECU (ECUN). Again, this message is time stamped and MAC protected (Message 2). After reception of Message 2, the target ECU first verifies its validity and then imports the new *SesK* into its local HSM. Finally, a message including an acknowledgement flag (ACK) is sent by the target ECU to the Key Master thus informing *SesK* acceptance (Message 3). Message 3 also includes a time stamp and is MAC protected. The Key Master receives the acknowledgement and a security check is performed. The Key Master has to repeat Messages 2 and 3 for every ECU in the group (different from the generator). After *SesK* distribution, the Key Master informs the generator about the result: partial or total accomplishment. The message includes the respective ACK code, the time stamp and is MAC protected (Message 4). Finally, the generator verifies Message 4.

Once distributed, the *SesK* key allows for confidential and authentic unidirectional communications between the generator and the rest of the group.

B. Protocol Modeling in AVATAR

This quite complex cryptographic protocol is modeled in AVATAR as follows (see Figure 6):

- One AVATAR block is used for ECU1, one for the Key Master, and for each ECU of the group (*ECUN*).
- Pre-shared keys are modeled using *InitialCommonKnowledge* pragmas.
- Symmetric and MAC cryptographic functions are used by the states machines of each block. The State Machine Diagram of the Key Master contains around 15 states.
- Only one public common channel is used for the communication between ECUs. We thus assume that an attacker can listen to all communications between ECUs. We also assume that communications between internal ECU components (e.g. CPU and HSM [39]) cannot be listened up by an attacker, as stated in [38].

Two confidentiality properties and one authenticity property have been modeled for the sake of this case study: the confidentiality of *SesK* states that an attacker shall never be able to obtain this key. The attacker shall also never be able to retrieve a confidential data that ECU1 sends after the protocol run, and using *SesK*. An authenticity property states that an attacker shall never send an authentic message *Message1* to the key Master ECU-KM, i.e., the first message of the protocol sent to the Key Master cannot be forged by an attacker. Other authenticity properties of the protocol can obviously be proved the same way.

This is an excerpt of the verification results as displayed by TTool, and computed by ProVerif in a transparent way.

```
Non reachable states:
enteringState__ECU1__NotAnACK

Confidential Data:
confData
SesK__data

Satisfied Authenticity:
KM__decipherOK__msgauth__data
```

The three properties are satisfied. Moreover, all states of the system are reachable, apart from one: that state corresponds to the acceptance by ECU1 of a message with a non valid MAC.

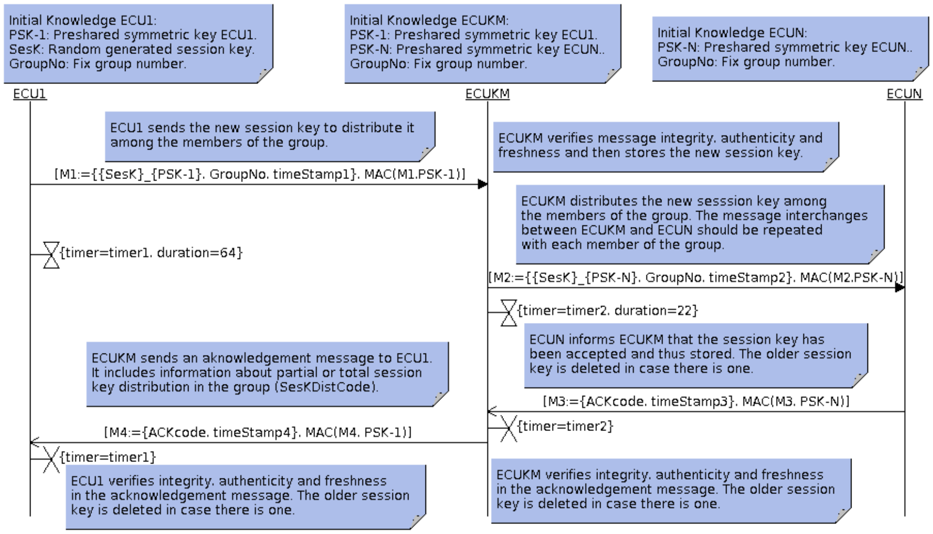

Initial Knowledge ECU1:
PSK-1: Preshared symmetric key ECU1.
SesK: Random generated session key.
GroupNo: Fix group number.
Initial Knowledge ECUKM:
PSK-1: Preshared symmetric key ECU1.
PSK-N: Preshared symmetric key ECUN..
GroupNo: Fix group number.
Initial Knowledge ECUN:
PSK-N: Preshared symmetric key ECUN..
GroupNo: Fix group number.
ECU1
ECUKM
ECUN
ECU1 sends the new session key to distribute it among the members of the group.
ECUKM verifies message integrity, authenticity and freshness and then stores the new session key.
[M1:={{SesK}_{PSK-1}, GroupNo, timeStamp1}, MAC(M1,PSK-1)]
ECUKM distributes the new sesssion key among the members of the group. The message interchanges between ECUKM and ECUN should be repeated with each member of the group.
{timer=timer1, duration=64}
[M2:={{SesK}_{PSK-N}, GroupNo, timeStamp2}, MAC(M2,PSK-N)]
{timer=timer2, duration=22}
ECUN informs ECUKM that the session key has been accepted and thus stored. The older session key is deleted in case there is one.
ECUKM sends an aknowledgement message to ECU1. It includes information about partial or total session key distribution in the group (SesKDistCode).
[M3:={ACKcode, timeStamp3}, MAC(M3, PSK-N)]
{timer=timer2}
[M4:={ACKcode, timeStamp4}, MAC(M4, PSK-1)]
{timer=timer1}
ECU1 verifies integrity, authenticity and freshness in the acknowledgement message. The older session key is deleted in case there is one.
ECUKM verifies integrity, authenticity and freshness in the acknowledgement message. The older session key is deleted in case there is one.

Fig. 5. Sequence Diagram of the Keying Protocol

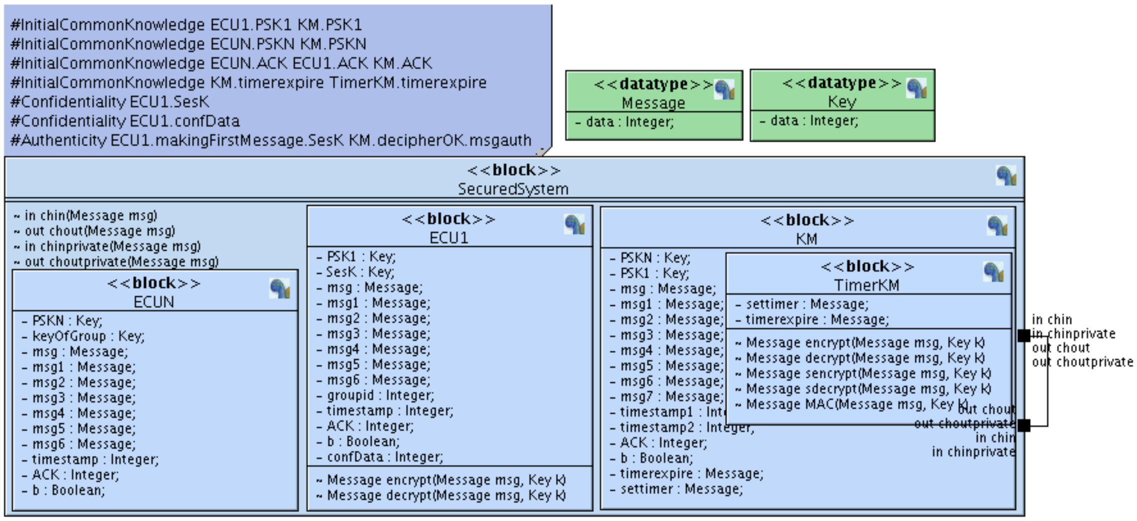


Fig. 6. AVATAR Block Diagram for the Key Master protocol

VII. Conclusions and future work

AVATAR is a SysML based environment targeting the modeling and formal verification of distributed, real-time and embedded systems. While safety properties have already been discussed in a previous contribution [6], security properties can now be modeled and proved from an AVATAR model. For security proofs, AVATAR relies on ProVerif which provides a formal framework for security proofs, based on Horn clauses resolution. TTool fully supports AVATAR, including its security extensions. No or little knowledge of ProVerif is necessary to perform security proofs.

Even if ProVerif is a strong formal approach, it only targets verification of confidentiality and authenticity properties thus limiting AVATAR security proof capabilities. To our knowledge, richer notion of attackers are indeed required to address other security properties, e.g. message freshness. Additionally, ProVerif attacker and semantics are not yet suitable for temporal analyses. Thus, we are currently working in two different but complementary directions. First, we would like to extend security properties that ProVerif can handle: integrity and freshness properties. The second direction is to enhance AVATAR with the new security properties that ProVerif could handle. For example, when ProVerif will support freshness properties, it will probably become necessary to enhance the AVATAR-to-Proverif translation process, so as to handle differently the temporal operators of AVATAR.

Following techniques proposed in [40], code generation preserving security properties is also one of our next research topics. As shown and explained in [41], maintainability of the automatically generated code shall subsequently be addressed.

References

[1] LabSoc, “TTool,” in *http://labsoc.comelec.enst.fr/turtle/ttool.html*.

[2] “The EVITA european project,” http://www.evita-project.org/.

[3] L. Apvrille, J.-P. Courtiat, C. Lohr, and P. de Saqui-Sannes, “TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit,” in *IEEE transactions on Software Engineering*, vol. 30, no. 7, Jul 2004, pp. 473–487.

[4] L. Apvrille, W. Muhammad, R. Ameur-Boulifa, S. Coudert, and R. Pacalet, “A UML-based environment for system design space exploration,” in *Electronics, Circuits and Systems, 2006. ICECS’06. 13th IEEE International Conference on*, Nice, France, Dec. 2006.

[5] S. Ahumada *et al.*, “Specifying Fractal and GCM components with UML,” in *XXVI International Conference of the Chilean Computer Science Society (SCCC’07)*, Iquique, Chile, nov 2007.

[6] D. Knorreck, L. Apvrille, and P. D. Saqui-Sannes, “TEPE: A SysML language for timed-constrained property modeling and formal verification,” in *Proceedings of the UML&Formal Methods Workshop (UML&FM)*, Shanghai, China, November 2010.

[7] M. Hendriks and M. Verhoef, “Timed automata based analysis of embedded system architectures,” in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006, p. 8.

[8] A. Viehl, T. Schonwald, O. Bringmann, and W. Rosenstiel, “Formal performance analysis and simulation of UML/SysML models for ESL design,” *Design, Automation and Test in Europe, 2006. DATE’06. Proceedings*, vol. 1, pp. 1–6, March 2006.

[9] M. H. Smith, “Events and constraints: a graphical editor for capturing logic properties of programs,” in *Proceedings of the 5th International Symposium on Requirements Engineering*, 2001, pp. 14–22.

[10] OMG, “A UML profile for MARTE, beta 2, www.omg.org,” 2008.

[11] E. C. da Silva and E. Villani, “Integrando SysML e model checking para v&v de software crítico espacial,” in *Brasilian Symposium on Aeropspace Engineering and Applications, São José dos Campos, SP, Brasil*, September 2009.

[12] I. Ober and I. Dragomir, “OMEGA2: A new version of the profile and the tools (regular paper),” in *UML&AADL’2009 - 14th IEEE International Conference on Engineering of Complex Computer Systems*. Potsdam: IEEE, June 2009, pp. 373–378.

[13] M. Hause and J. Holt, “Testing solutions with UML/SysML,” in *http://www.artist-embedded.org/docs/Events/2010/UML_AADL /slides/Session1_Matthew_Hause.pdf*, 2010.

[14] M. Audrain and B. Marconato, “TOPCASED 3.4 tutorial - requirement management,” in *http://www.topcased.org/index.php? documentsSynthesis=y&Itemid=59*, 2010.

[15] “SysML companion,” in *http://www.realtimeatwork.com/?page_id=683*.

[16] Y. Vanderperren and W. Dehaene, “From UML/SysML to Matlab/Simulink: current state and future perspectives,” in *DATE’06: Proceedings of the conference on Design, automation and test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 93–93.

[17] M. J. Toussaint, “A new method for analyzing the security of cryptographic protocols,” in *Journal on Selected Areas in Communications*, vol. 11, No. 5. IEEE, June 1993.

[18] D. Trcek and B. J. Blazic, “Formal language for security services base modelling and analysis,” in *Elsevier Science Journal, Computer Communications*, vol. Vol. 18, No. 12. Elsevier Science, 1995.

[19] M. Drouineaud, M. Bortin, P. Torrini, and K. Sohr, “A first step towards formal verification of security policy properties for RBAC,” in *Proceedings of the Fourth International Conference on Quality Software (QSIC’04)*, vol. 0-7695-2207-6/04. IEEE, 2004.

[20] C. Dixon, M.-C. F. Gago, M. Fisher, and W. van der Hoek, “Using temporal logics of knowledge in the formal verification of security protocols,” in *Proceedings of the 11th International Symposium on Temporal Representation and Reasoning (TIME’04)*, vol. IEEE 1530-1311/04, 2004.

[21] A. Aldini and M. Bernardo, “A formal approach to the integrated analysis of security and QoS,” in *Reliability Engineering and System Safety*, vol. Vol. 92. Elsevier, 2007, pp. 1503–1520.

[22] A. M. na and G. Pujol, “Towards formal specification of abstract security properties,” in *The Third International Conference on Availability, Reliability and Security*, vol. 0-7695-3102-4/08. IEEE, 2008.

[23] Y. Ali, S. El-Kassas, and M. Mahmoud, “A rigorous methodology for security architecture modeling and verification,” in *Proceedings of the 42nd Hawaii International Conference on System Sciences*, vol. 978-0-7695-3450-3/09. IEEE, 2009.

[24] J. Jürjens, “UMLsec: Extending UML for secure systems development,” in *Proceedings of the 5th International Conference on The Unified Modeling Language*, ser. UML’02. London, UK, UK: Springer-Verlag, 2002, pp. 412–425.

[25] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone, “St-tool: a case tool for security requirements engineering,” in *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, aug.-sept. 2005, pp. 451 – 452.

[26] M. Boreale and M. G. Buscemi, “Experimenting with STA, a tool for automatic analysis of security protocols,” in *Proceedings of the 2002 ACM symposium on Applied computing*, ser. SAC’02. New York, NY, USA: ACM, 2002, pp. 281–285.

[27] D. Dolev and A. Yao, “On the security of public key protocols,” *IEEE trans. on Information Theory*, vol. 29, pp. 198–208, 1983.

[28] M. Idrees, Y. Roudier, and L. Apvrille, “A framework towards the efficient identification and modelling of security requirements,” in *5ème Conf. sur la Sécurité des Architectures Réseaux et Systèmes d’Information (SAR-SSI 2010)*, Menton, France, May 2010.

[29] P. Ochsenschläger, J. Repp, and R. Rieke, “The sh-verification tool,” in *Proceedings of the Thirteenth International Florida Artificial Intelligence Research Society Conference*. AAAI Press, 2000, pp. 18–22.

[30] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. H. Drielsma, P. H. O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron, “The AVISPA tool for the automated validation of internet security protocols and applications,” in *CAV 2005*, ser. LNCS, vol. 3576. Springer Verlag, 2005, pp. 281–285.

[31] B. Blanchet, “Automatic verification of correspondences for security protocols,” *Journal of Computer Security*, vol. 17, no. 4, pp. 363–434, Jul. 2009.

[32] ——, “Proverif automatic cryptographic protocol verifier user manual,” CNRS, Département d’Informatique École Normale Supérieure, Paris, Tech. Rep., July 2010.

[33] ——, “From Secrecy to Authenticity in Security Protocols,” in *9th International Static Analysis Symposium (SAS’02)*, ser. Lecture Notes on Computer Science, M. Hermenegildo and G. Puebla, Eds., vol. 2477. Madrid, Spain: Springer Verlag, Sep. 2002, pp. 342–359.

[34] B. Blanchet and D. Pointcheval, “The computational and decisional Diffie-Hellman assumptions in CryptoVerif,” in *Workshop on Formal and Computational Cryptography (FCC 2010)*, Edimburgh, United Kingdom, Jul. 2010.

[35] M. Abadi and B. Blanchet, “Analyzing Security Protocols with Secrecy Types and Logic Programs,” in *29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL 2002)*. Portland, Oregon: ACM Press, Jan. 2002, pp. 33–44.

[36] B. Blanchet and A. Podelski, “Verification of cryptographic protocols: Tagging enforces termination,” *Theoretical Computer Science*, vol. 333, no. 1-2, pp. 67–90, Mar. 2005, special issue FoSSaCS’03.

[37] A. Fuchs, S. Gürgens, L. Apvrille, and G. Pedroza, “On-Board Architecture and Protocols Verification,” EVITA Project, Tech. Rep. Deliverable D3.4.3, 2010.

[38] H. Schweppe, M. S. Idrees, Y. Roudier, B. Weyl, R. E. Khayari, O. Henniger, D. Scheuermann, G. Pedroza, L. Apvrille, H. Seudié, H. Platzdasch, and M. Sall, “Secure on-board protocols specification,” EVITA Project, Tech. Rep. Deliverable D3.3, 2010.

[39] H. Seudié, J. Shokrollahi, B. Weyl, A. Keil, M. Wolf, F. Zweers, T. Gendrullis, M. S. Idrees, Y. Roudier, H. Schweppe, H. Platzdasch, R. E. Khayari, O. Henniger, D. Scheuermann, L. Apvrille, and G. Pedroza, “Secure on-board architecture specification,” EVITA Project, Tech. Rep. Deliverable D3.2, 2010.

[40] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel, “Weaving executability into object-oriented meta-languages,” in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, vol. 3713, pp. 264–278.

[41] O. Chebaro, L. Broto, J.-P. Bahsoun, and D. Hagimont, “Self-TUNeing of a J2EE clustered application,” in *Engineering of Autonomic and Autonomous Systems, 2009. EASe 2009. Sixth IEEE Conference and Workshops on*, 2009, pp. 23 –31.