Checking SysML Models against Safety and Security Properties

Pierre de Saqui-Sannes*
ISAE-SUPAERO, Université de Toulouse, France

Ludovic Apvrille†
LTCI, Telecom Paris, Institut Polytechnique de Paris, France

Rob Vingerhoeds‡
ISAE-SUPAERO, Université de Toulouse, France

Systems Engineering, or engineering in general, has long been relying on document-centric approaches. Switching to Model Based Systems Engineering, or MBSE for short, has extensively been discussed over the past three decades. Since about two decades, MBSE has been commonly associated with the modelling language SysML (Systems Modeling Language), that offers a standardized notation, not a methodology of using it. SysML needs therefore to be associated with a methodology supported by tools. In this paper, a methodology supported by the free and open-source software TTool is associated with SysML. This paper focuses discussion on methodological issues, leading the authors to share their experience in real-time systems modeling. Modeling with SysML is more than just drawing the different diagrams. Associated tools offer possibilities to analyze SysML models for specific properties. In this paper, verification addresses both safety and security properties. The TTool model checker inputs the SysML model enriched with safety properties to be verified and outputs a yes-no answer for each property. Security verification checks SysML models against confidentiality, integrity and authenticity properties. As an illustration of the proposed approach, an aircraft cockpit door control system is modeled in SysML and verified against safety and security properties.

*Professor, Department of Complex Systems Engineering, pdss@isae-supaero.fr.
†Professor, Department of Communication and Electronics, ludovic.apvrille@telecom-paris.fr
‡Professor, Head of Department of Complex Systems Engineering, AIAA Member, rob.vingerhoeds@isae-supaero.fr

Nomenclature

$AVATAR$	=	Automated Verification of reAl Time softwARe.
$CCSL$	=	Clock Constraint Specification Language
CSP	=	Constraint Satisfaction Programming
CTL	=	Computation Tree Logic.
$INCOSE$	=	International Council on Systems Engineering
$KAOS$	=	Knowledge Acquisition in Automated Specification
$MBSE$	=	Model-Based Systems Engineering
OMG	=	Object Management Group
$PRISM$	=	Probabilistic Symbolic Model Checker
$STPA$	=	System Theoretic Process Analysis
$SysML$	=	Systems Modeling Language
UML	=	Unified Modeling Language

I. Introduction

Systems Engineering, or engineering in general, has long been relying on document-centric approaches. Over the past three decades, researchers [1], industry practitioners [2] and contributors to standardization bodies [3] have discussed the benefits and potential of switching from document-centric systems engineering to Model-Based Systems Engineering. The acronym MBSE was coined at that time.

MBSE has been commonly associated with SysML, the Systems Modeling Language standardized [3] by OMG with the support of INCOSE. With SysML, OMG offers a standardized language, but not the way of using it. In other words, the SysML standard does not define a methodology, nor does it define associated tools. Consequently, the SysML language needs to be associated with a methodology that then needs to be supported by mature tools.

Contributions This paper discusses SysML modeling in terms of language, tools, and method.

- `Language.` SysML, which is a UML profile [4] and therefore a customization of the latter, can be customized in turn with a twofold objective: enhance its expression power and formalize its semantics. In this paper, the version of SysML named AVATAR [5] extends and formalizes the OMG-based version of SysML to address real-time systems.
- `Tools.` SysML editors are entry points to cater model simulators and formal verification tools that enable checking of SysML models against design errors. In this paper, the free and open-source software TTool [5] supports edition and formal verification of AVATAR models. Its native model checker computes properties

expressed inside the SysML model and returns the feedback in the same SysML model. Users of TTool are therefore not obliged to use external tools or to inspect the inner workings of the model checker. As far as security properties are concerned, TTool further enables checking of SysML models against confidentiality, integrity and authenticity properties.

- `Methods.` The method associated with AVATAR and TTool has been designed with real-time systems in mind. The method applies to a broad variety of these systems, including the pilot cabin door controller that serves as running example throughout this paper.

Structure The rest of this paper is organized as follows. Section II surveys related work. Section III details background information about SysML/AVATAR modeling, safety property verification relying on model checking, security property verification, and the method associated with SysML and TTool. Section IV introduces a case study: a Cockpit Door Control System. Sections V to X discuss application of SysML and formal verification to this control system. Section XI concludes paper and outlines future work.

II. Related Work

A. Formal Verification of Safety Properties in UML and SysML models

A survey of the literature indicates that checking UML and SysML diagrams against safety properties has been addressed with respect to activity diagrams [6, 7] and state machine diagrams [8–10], respectively. This section covers the two families of diagrams although the safety property verification approach discussed later on in this paper exclusively applies to state machine diagrams.

Formal verification of UML and SysML diagrams commonly relies on translating one UML or SysML model into a formal model [11]. Translation from UML/SysML to state/transition models has been formalized in the context of Petri nets [7, 8, 12, 13], automata for NuSMV model checker [14], timed automata [9] for UPPAAL model checker, hybrid automata [15], model checker NuSMV [16], probabilistic model checker PRISM [6, 15], and a theorem prover [17]. Translation from UML to process algebra has been investigated for RT-LOTOS [10] and CSP [18]. The family of correct by construction specification has been addressed with B [19].

Whatever the formal model (state/transition model, process algebra, or correct by construction model), the papers listed by previous paragraph link a UML or a SysML tool with an external verification tool, typically a model checker. This paper conversely presents a SysML tool (named TTool) that includes a native model checker [20]. Another important feature of TTool is backtracking of verification results to the initial SysML model. A feature that is missed by many tools, *e.g.*, [21].

B. Security models and verification

The closest approaches to our model-driven security verification framework are [22], [23] and [24]. UMLsec [22] is a modeling framework for defining security properties attached to software components that can be deployed over execution nodes and networks. It further supports many different stages of system development, ranging from requirements capture to tests. Logic-based formal verification supports the composition of software components.

In [23], Kordy et al. formally define Attack-Defense Trees. The trees can capture attacks, relations between attacks and countermeasures that can be related to the system under design. Our approach restricts the attacker's capabilities (Dolev-Yao attacker model [25]) but with a formal description of its capabilities. Moreover, we offer the possibility to model the system in terms of structure and behavior, which makes it possible to formally verify security properties.

More recently, [24] extended UML sequence diagrams for describing security protocols and verifying them. Like AVATAR designs addressed in this paper, sequence diagrams are translated into ProVerif [26] 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 AVATAR– allow modeling of precise behavioral properties more intuitively (such as conditional statements or loops). Further, our process includes verification of weak and strong authenticity.

C. Methods

The methods one may associate with SysML can be categorized into four groups:

1) Methods compliant with a standard which is a reference for systems engineering. Examples include ANSI EIA632 [27], IEEE 1220 [28], and IEC 15288 [29].
2) Methods compliant with a standard which is a reference for an application domain, such as ARP 4754A [30] for aeronautics [31].
3) Methods developed for specific tools whilst remaining applicable to a broad variety of systems. For instance, [1] and [32] associate a method with SysML and TTool [5] and discuss application to drones and communication architectures respectively.
4) Methods non initially developed for MBSE can be extended with MBSE features. [33] extends the STPA (*Systems Theoretic Process Analysis*) method with SysML and TTool, in particular to benefit from the model formal verification approaches supported by TTool. Another example of associating a SysML method with another method is discussed in [34] for the Formose project that associates SysML with KAOS.

III. Background

A. SysML/AVATAR

OMG (Object Management Group) and INCOSE (International Council on Systems Engineering) have jointly defined SysML, a System Modelling Language that is now an international standard at OMG [27] and one of the pillars of Model-Based Systems Engineering (MBSE). The SysML standard at OMG defines a notation, not the way of using it, leaving doors open for application to various domains, *e.g.,* real-time systems, and to the methods or processes practitioners of these application domains are familiar with.

Application of SysML to real-time systems has stimulated research work [35] on improving the expression power of the notation standardized by OMG [3]. For instance, the AVATAR modeling language [1] reuses SysML diagrams (Table1), but the package diagram. The use case diagram is the keystone of a functional analysis. The high level functions or services identified by the use cases can be documented verbatim or by other diagrams based on scenarios, flowcharts or architectural sketches. The design phase makes it possible to make technical choices to develop a system architecture and endow the blocks of this architecture with an internal behavior expressed in the form of state machines.

AVATAR extends the expression power of several SysML diagrams and introduces one diagram: the modeling assumptions diagram.

- Block definition diagrams (BDD) and internal block definition diagrams (IBD) are both supported. AVATAR allows one to merge one BDD and its associated IBD into one view that we call a 'block diagram' in the remainder of this paper. Block diagrams support synchronous and asynchronous communications with different flavors (lossy, non lossy, *etc.*).
- Finite state machine diagrams have been extended to cope with temporal indeterminism and work with temporal intervals instead of fixed delay values, making it possible to model the variability of transmission delays in computer networks. Further, timers are handled and transition blocks may contain random operators.
- Modeling assumptions diagrams have been introduced into AVATAR to encourage model designers to make modeling assumptions part of the SysML model and to document them properly. Assumptions identify simplification made at the time of creating the AVATAR model. For somebody receiving an AVATAR model, reading assumptions is essential for understanding modeling decisions.

B. Tools

Previous section has identified SysML diagrams, including the customization supported by TTool. One may distinguish between two groups of diagrams:

- Diagrams edited with TTool. These diagrams are exclusively processed by an editor. No other tool is available for checking the correctness of these diagrams, which means only interactions with experts may help beginners assessing the quality of their diagrams. This first group include requirement, modeling assumption, use case,

AVATAR Diagram	Usage
Requirement Diagram	Identifies and structures requirements to be met by the system.
Modeling Assumptions Diagram	Identifies the simplifications made by the author of the model.
Parametric Diagram	Describes mathematical equations by modeling elements.
Use Case Diagram	Sketches the system boundary and its main functions.
Sequence Diagram	Documents use cases in the form of a scenario.
Activity Diagram	Documents use cases in the form of a flow-chart.
Block Definition Diagram	Shows Blocks, their contents, and relationships.
Internal Block Diagram	Models the decomposition and interconnections of blocks.
State machine diagram	Models the behavior of one block in the architecture.

Table 1 AVATAR diagrams.

sequence, and activity diagrams.

- Diagrams edited with TTool and checked against design errors using the model simulator and formal verification modules of TTool. With this second group, which includes the block and state machine diagrams, beginners are provided with tool-assistance to debug the diagrams.

Table 2 categorizes SysML tools depending on the accessibility policy and functions offered by the tools.

Tool	Copyrighted by	Editor Source	Simulator	Safety Verification Model Checker	Security Verification
UML/PNO [8]	LAAS-CNRS	X	X	X	
TimeSquare [21]	INRIA	X	X		
Papyrus [36]	CEA	X			
Modelio [37]	Modeliosoft	X			
EA [38]	Sparx Systems	X	X		
Cameo [39]	Dassault Systems	X	X		
Rhapsody [40]	IBM	X	X	X	
TTool [5]	Telecom Paris	X	X	X	X

Table 2 SysML tools.

C. Model Checking for safety verification

Principle. In [41], Fisman and Pnueli define model checking as the method by which a desired behavioral property of a reactive system is verified over a given system (the model) through exhaustive enumeration (explicit or implicit) of all the states reachable by the system and the behaviors that traverse through them. Figure 1 pictorially identifies three main steps. The model checker is catered with a model of the system and a formal expression of the properties to be verified. The model checker processes the model and the properties, and outputs a "yes/no" answer stating whether the property is verified or not. The model checker also traces execution paths that lead to property violations. The tool must

indeed help the designer of the system interpreting verification results in the light of the model of the system the tool was catered with.

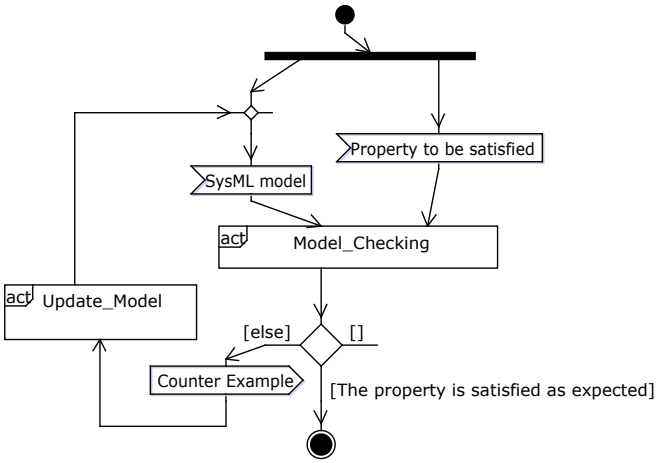


Fig. 1 Model checking SysML models.

The model checker implemented by TTool The model checker of TTool is catered with a SysML model and a set of properties.

In terms of model, the requirement diagrams, modeling assumptions diagrams, use case diagrams, sequence diagrams and activity diagrams are not processed by TTool's model checker. The later processes the block diagram depicting the architecture of the system under design and the set of state machines defining the inner workings of the blocks.

In terms of properties, TTool's model checker takes as input properties expressed using the logic whose operators are enumerated by Table 3. TTool supports CTL, which is a branching-time logic. CTL properties supported by TTool either start by A/E followed by "<>" or "[]" followed by a property p. p is a boolean expression on block states or attributes. TTool also supports a "leads-to" property $p \rightarrow q$ with p and q being boolean expression on block states and attributes.

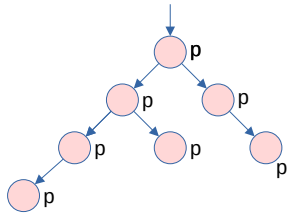
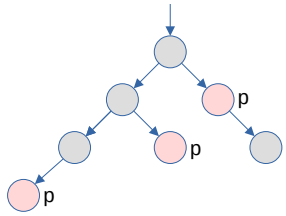
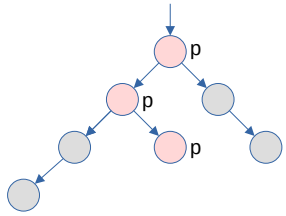
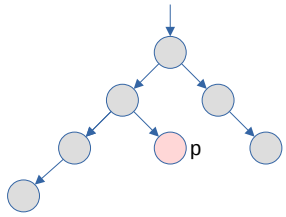
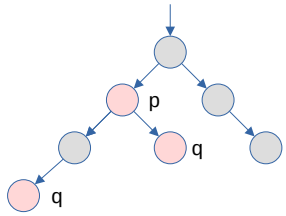
Operator	Meaning	Reachability Graph
A [] p	‘p’ satisfied in all states of all paths	
A <> p	‘p’ satisfied in at least one state of all paths	
E [] p	‘p’ satisfied in all states of at least one path	
E <> p	‘p’ satisfied in at least one state of at least one path	
p –> q	q is eventually satisfied in at least one state of all paths starting from the states in which p is satisfied	

Table 3 CTL logic formula supported by the model checker of TTool.

D. Security

Security aspects have been added to AVATAR in the scope of the SysML-Sec environment [42]. AVATAR has been enhanced in three ways:

1) **System model**. A «cryptoblock» defines security methods, such as `sencrypt` (for symmetric encrypt), `sdecrypt`

(for symmetric decrypt), and `aencrypt` (for asymmetric encrypt). These methods can be used to specify security mechanisms to setup secure exchanges. AVATAR also allows one to define public/private keys and to specify the initial knowledge of a block (*e.g.*, specifying the sharing of one symmetric key).

2) **Security properties**. Examples include the confidentiality of the attribute of a block, the integrity of a message exchange, the authenticity of a message exchange. These properties are expressed with specific pragmas.
3) **Formal verification of security properties** [43]. TTool transforms an AVATAR model into a ProVerif specification and back-traces results to the SysML model. Moreover, when a security property is not satisfied, TTool builds up a scenario (sequence diagram) representing the property violation.

E. Method

The INCOSE Systems Engineering Handbook [44] states development processes are inherently incremental, iterative, and recursive in nature. Usually the necessary insights for a system definition is obtained through exchanges between different development processes, in order to obtain a system definition matching the mission or business needs. Feedback loops in the development processes enable communication that accounts for ongoing learning and decisions, herewith facilitating incremental learning from analysis results with evolving technical solutions. Figure 2 depicts the incremental method associated with SysML and TTool. The method can be sketched as follows:

1) *Requirement capture* uses requirement diagrams to define stakeholder, user, and system requirements. Modeling assumptions diagrams list simplifications and other assumptions made at the time of creating the model.
2) *Analysis* is use case driven. Use cases identify the main functions and services to be offered by the system. Sequence and activity diagrams document the uses cases in the form of scenarios and flow-charts, respectively.
3) The *design* step architects the system in the form of a block diagram, and defines the inner workings of the blocks using state machine diagrams.
4) The last step checks the model against design errors by combining formal verification techniques addressing safety and security issues. When necessary, for example when one or more of the assumptions need to be updated or in case of a design error, the method recursively moves back up to the assumption selection or the design step.

IV. Case Study

The case study is inspired by the specification of Airbus cockpit doors [45].

The main concern for a door mechanism for commercial aircraft is safety and security. As frequently asserted, a closed and locked door is a safe and stable state for the cockpit, the pilots can operate the aircraft safely. The basic state is therefore that the door of a cockpit is closed and locked. Two ways can be used to enter into the cockpit.

First, a person can use a phone line located outside the cockpit to contact the flight crew inside the cockpit. Depending on the situation that can be assessed with a camera, the flight crew may decide to keep the door locked or to unlock it

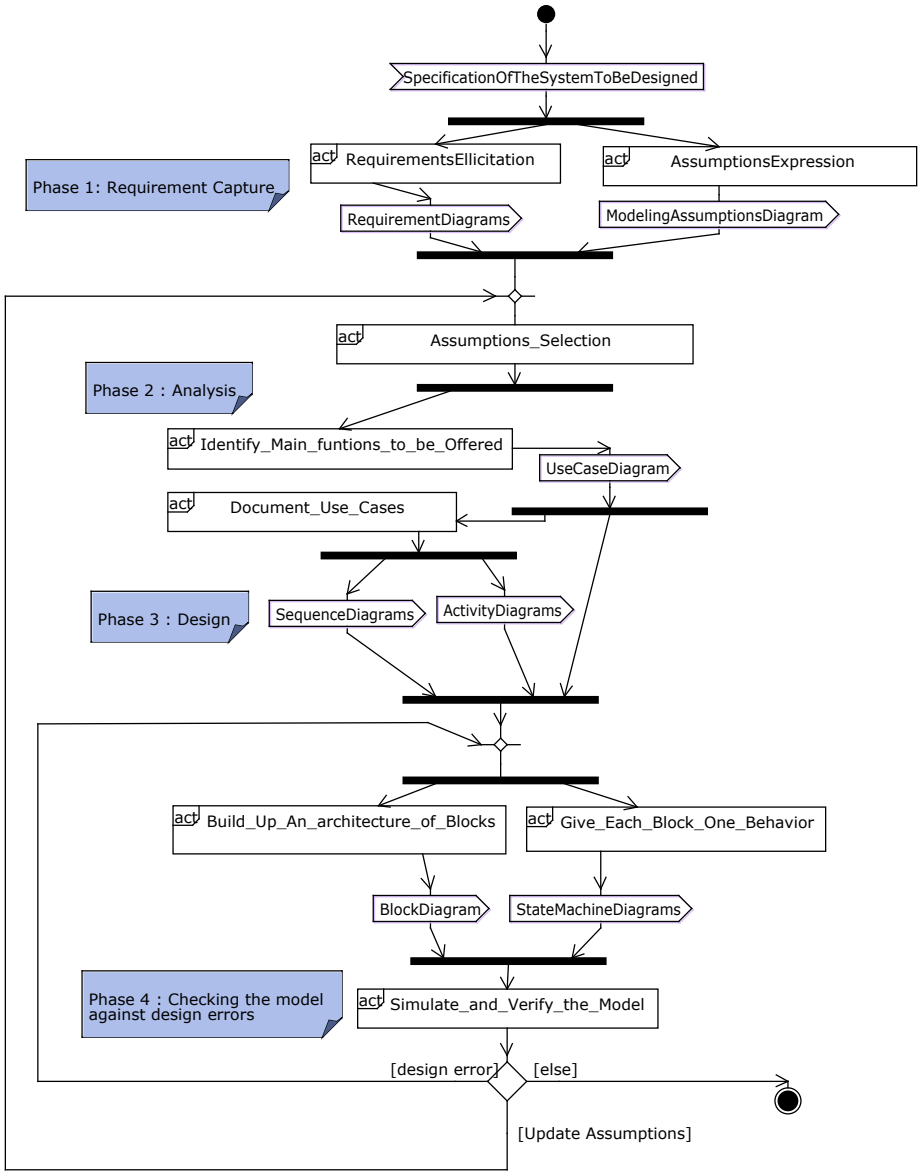


Fig. 2 Incremental modeling.

using a door button. This door button has three positions: norm (no action), lock (door remains locked, thus stopping any request to enter into the cockpit), unlock. Once released, the button comes back to the norm position automatically. Second, a person (*i.e.* a pilot or a crew member) may request the door to unlock using a secret code to be entered on a keypad located outside the cockpit. When the right code has been entered, a buzzer rings and an open signal is displayed just next to the door button to invite the pilot to unlock the door. However, one of the pilots can decide to set the button to lock: in that case, the entry is denied for 5 minutes before a new request can be performed.

Once a valid code has been entered, the cockpit door unlocks after 30s, except if the pilot decided to keep the door lock by moving the door lock/norm/unlock button to the lock position.

The cockpit door is thus controlled with a double-security mechanism. So, entering the code is not sufficient, since some person, *e.g.*, a steward, may be obliged to give the code to the criminal who unduly tries to enter the cockpit. Sniffing the network between the keypad and the cockpit equipment can be prevented by either physically isolating the concern communication elements (*e.g.,* buses) or by using security protocols and mechanisms.

V. Requirement Capture

Figure 3 depicts the requirement diagram derived from the specification given in previous section. Each box in the diagram contains one requirement together with one identifier and a `kind` attribute stating whether the requirement is functional or non-functional. The diagram links requirements with three different relations:

1) `is made up of (cross surrounded by a circle)`. It enables decomposition of complex requirements into more elementary ones.
2) «`refine`». It adds precision to another requirement.
3) «`deriveReq`». It enables deriving a technical requirement from a logical one.

In Figure 3, security requirements concern the confidentiality of code used to trigger an emergency call and the authenticity of the message sent from the keypad to the door management subsystem. Requirement with ID=10 specifies that the code shall remain confidential to attackers spying on the communication link between the keypad and the door. Requirement with ID=11 refers to an attacker replacing the keypad by a fake one, or replaying former messages on the communication link between the keypad and the door management subsystem.

Figure 4 shows how two blocks (see the architecture in figure 7) satisfy the confidentiality requirements expressed by the requirement diagram in figure 4.

VI. Modeling Assumptions

Figure 5 depicts the modeling assumptions diagram developed for the case study. The MAD consists of two parts. As many SysML models encountered in the literature, our SysML model ignores maintenance as well as the set up and shutdown phases of the system under design (Cockpit Door Control System in our case). The MAD enumerates additional simplifications made by the designers of the model.

VII. Analysis

In this section, 'Analysis' denotes the process of defining what the system has to do, by contrast to 'Design' that defines how the system will fulfill these tasks. Figure 6 depicts the use case diagram for the cockpit door controller. A rectangle sets the boundary of the system.

Here, what is inside the rectangle addresses the cockpit door controller, while external actors are depicted by stickmen outside the rectangle. The ovals define use cases that represent main functions or services that need to be offered by the

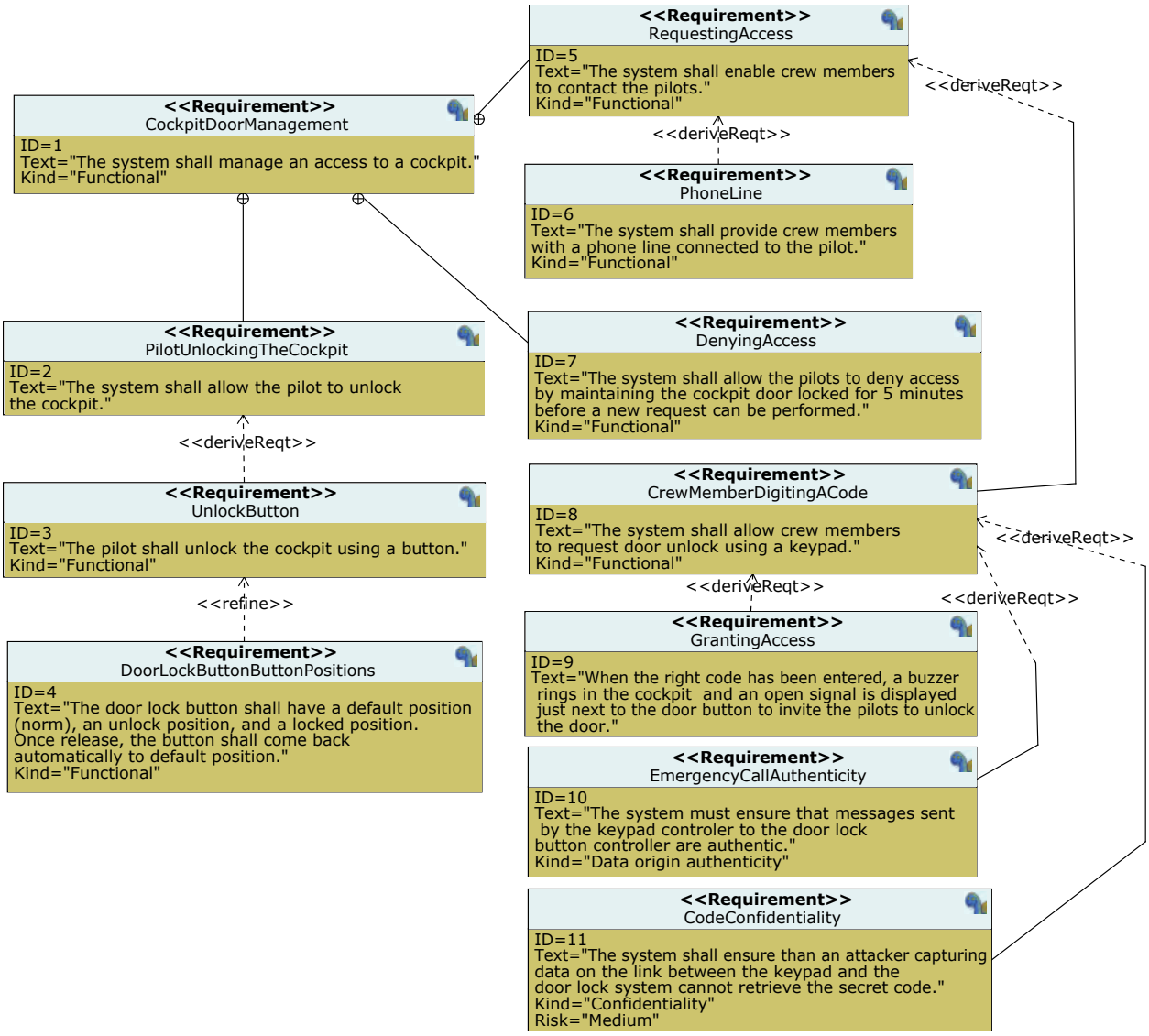


Fig. 3 Requirement diagram.

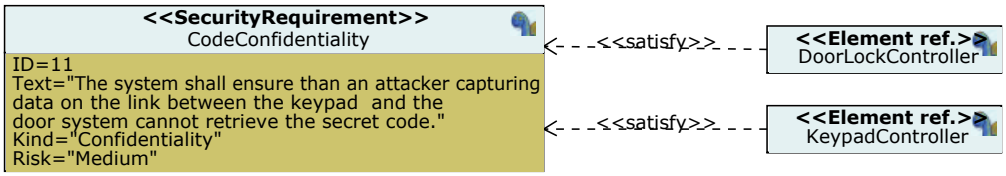


Fig. 4 Requirement traceability.

controller. One difficulty in defining use cases is to characterize functions at high level, not elementary actions. Logical inclusions between functions are modeled by «include» relations between pairs of use cases containing these functions. As mentioned, what is outside the rectangle represents the environment of the controller. That environment is characterized by a set of actors that interact with the use cases, as stated by the link relations between actors and use

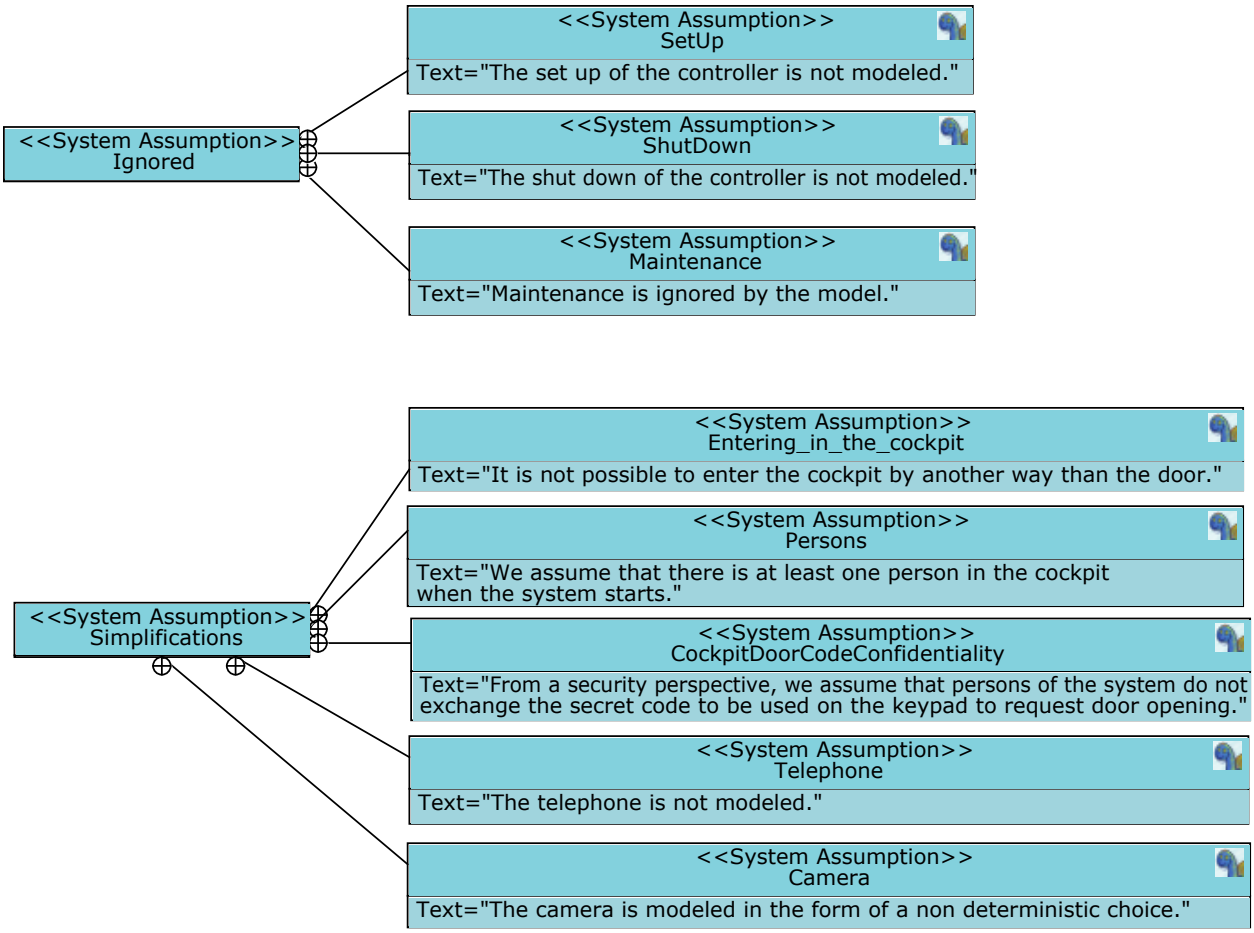


Fig. 5 Modeling assumptions diagram.

cases. One actor may be linked to one or several use cases. Similarly, one use case may be linked to one or several actors. It is further possible to draw inheritance relations between pairs of actors. For instance, on Figure 6, `Passenger`, `CabinCrew` and `Pilot` inherit from `Person` to say that passengers, the cabin crew and the pilot are specific types of persons. Note that all persons can manage the door lock, more obviously the pilot, but also a passenger if she/he manages to enter the cockpit.

The use case diagram in Figure 6 depicts a `ManageDoorLock` function that manages the overall control of the Door Locking System, including

- `HandleAccessRequests`. It handles the request coming from a keypad.
- `ReleaseLock`. It allows the pilots to unlock the door.
- `ForceLock`. It allows the pilots to keep the door locked, disabling the next request for the next 5 minutes.
- `InformCrew`. It keeps the crew informed on the current status of the door lock management.
- `InformPilots`. It keeps the pilots aware of the requests.

The environment is modeled by the following actors:

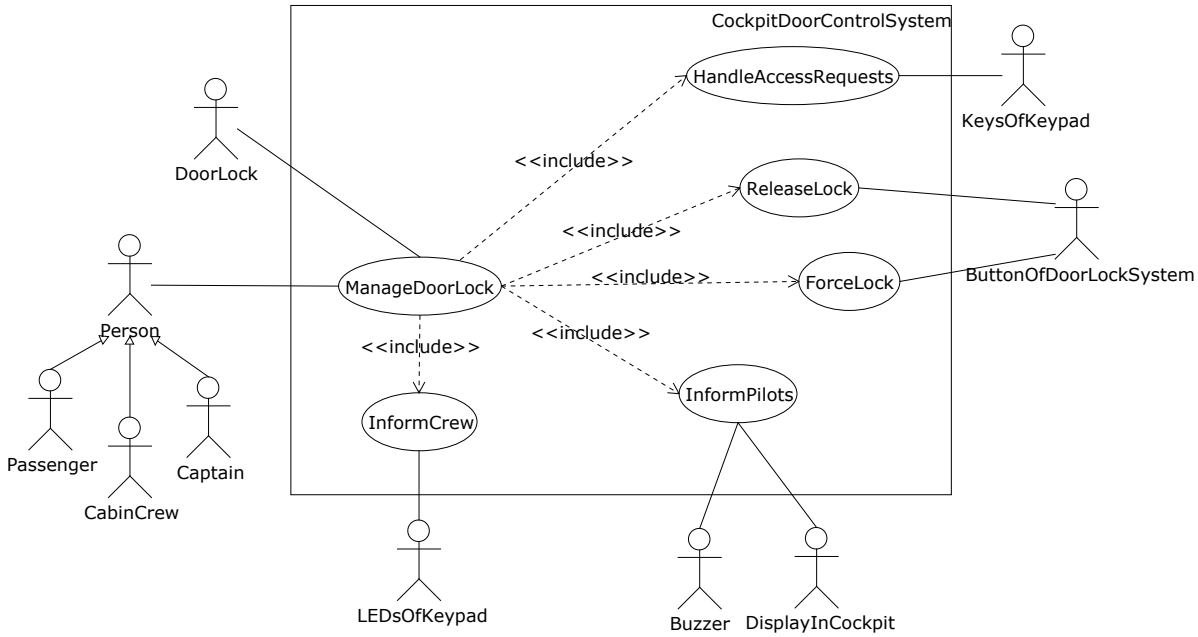


Fig. 6 Use case ciagram.

- `Person` - pilot, cabin crew or passenger.
- `Door` - the door itself.
- `Keypad` - the keypad the crew can use to request access to the cockpit using a code.
- `LockButton` - the button that allows the pilots to unlock or to reinforce the doorlock.
- `LEDsOffKeyPad` - a led indication that is to inform the crew on the status of the door lock and associated access request.
- `Buzzer` - a sound signal in the cockpit informing pilots on a request to unlock the door.
- `DisplayInCockpit` - an information display in the cockpit via which the pilots can obtain information on the access requests.

Writing good use cases to have a good basis for the design phase is not trivial. In [46] Rizzo-Aquino, de Saqui-Sannes and Vingerhoeds present UCCheck, a methodological assistant for use case diagrams creators. UCCheck detects errors such as confusion between high-level functions (appropriate for a use case diagrams) and elementary actions (appropriate for activity diagrams documenting the use cases).

VIII. Design

The first stage in a system's life cycle, the concept stage, focuses on the system concept. It involves understanding the implications of a system mission and core functionality. It addresses the requirements and assesses whether the required functionality with the required performance can be realised within the existing budget constraints. For this, trade-off indicators such as Figures of Merit (FoM) are used that allow comparing different architectures. The concept design

stage is one of the critical phases in systems development. It focuses on the concept: understanding the implications of a system mission and core functionality, a business case together with requirements, their interconnections and dependencies specified in *e.g.*, key performance indicators and trade-off indicators.

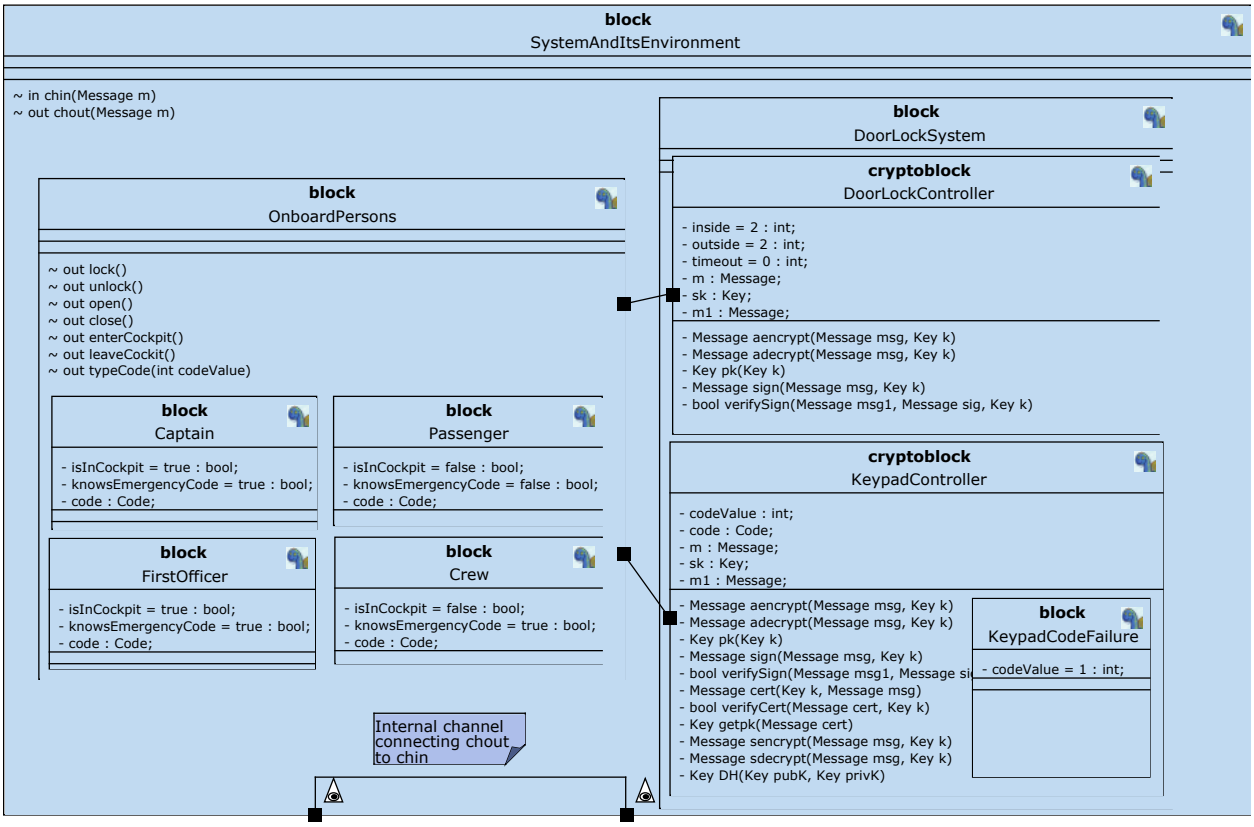


Fig. 7 Architecture of the door control system.

A logical architecture of the system design and its subsystems (the upper-level architecture) needs to be developed. This architecture is expected to meet the system requirements: a preliminary design of the product or service to develop [47]. The design step defines the architecture of the system and the behavior of the blocks the architecture is made of. System architecture is "the embodiment of concept, the allocation of physical/informational function to elements of form, and the definition of relationship among the elements and with the surrounding context" [48]. Architecting is a creative process in which the architect searches for innovative solutions to a specific problem.

A. Architectural Design

Figure 7 depicts the architecture of the door control system and the environment with which it interacts in the form of a block diagram view gathering both an block definition diagram and an internal block diagram. A block is defined by a set of attributes, methods that check and modify the values of these attributes, input signals, and output signals. For instance, `DoorLockController` has three attributes (two integer and one timer) and a list of input signals it can receive.

To receive or send a signal, a block needs a port connected to another port. In Figure 7, two black squares represent the ports of `OnBoardPersons` and `DoorLockController`. The two ports are connected to make communication feasible. Since `OnboardPersons` is considered part of the environment and `DoorLockController` part of the system, this interconnection between blocks specifies an interface between the system and its environment.

B. Behavioral Design

Each block of the architecture may contain attributes that are local variables not shared with other blocks, methods to test or modify the values of the attributes, and input/output signals conveyed by the ports owned by the block. Attributes may be integer or boolean values, records made up of several fields, and timers driven by set and reset commands.

An architecture of blocks is thus described in terms of interfaces. In a next step for each block an expected internal behavior needs to be defined. This might be achieved, for example, by describing the inner workings of blocks by pieces of software. Within SysML/AVATAR a slightly different approach was retained: each block has one and only one behavior that may be expressed in the form of a state machine (and C code). SysML state machines handle attributes, methods, signals and timers.

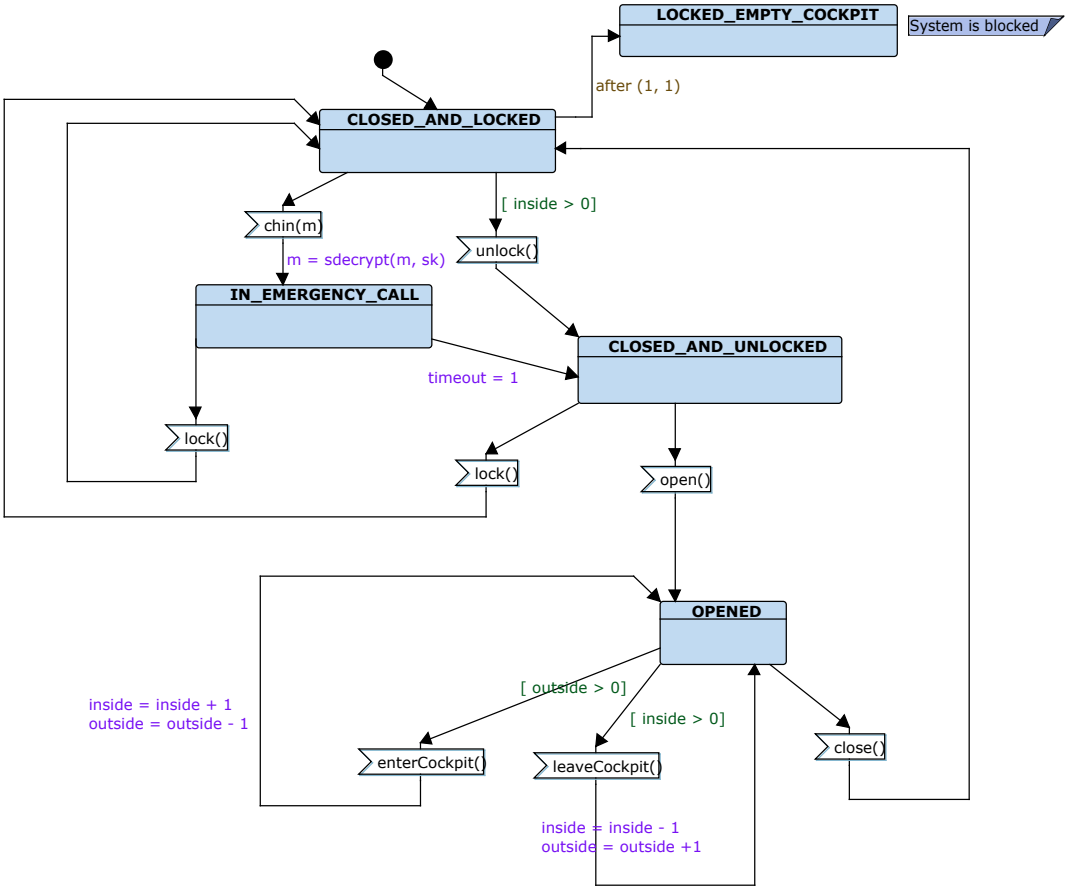


Fig. 8 State machine of the door lock controller.

Referring to the case study, Figure 8 depicts the state machine diagram of the door controller. The state machine in Figure 8 works as follows. The door controller starts in a state where the cockpit is both closed and locked. The controller is waiting for either the following signal: an emergency call or an unlock request. The latter may be accepted at the condition somebody is inside the cockpit. After entering the `CLOSE_AND_UNLOCKED` state, the controller may receive an `open` acceptation signal leading to a state where the door is open. Depending on the crew members and the pilots inside or outside the cockpit, the door controller accepts (or not) requests to enter or leave the cockpit.

The state machine in Figure 8 may return to its initial state. This will happen in all situations except when the transition linking `CLOSE_AND_UNLOCKED` state to `LOCKED_EMPTY_COCKPIT` state is fired. That transition is labelled by an `after(1,1)` clause, which means the controller should wait for one second before firing the transition. Given the semantics of time implemented by the simulator of TTool, transition to `LOCKED_EMPTY_COCKPIT` state may be fired if and only if no other transition (namely a transition without `after` clause) may be fired. Therefore, the value '1' assigned to the 'min' and 'max' attributes of the 'after' clause is arbitrary. Its role is merely to make transition to `LOCKED_EMPTY_COCKPIT` fireable if all other transitions are unable to be fired. Exhaustive analysis has shown that transition to `LOCKED_EMPTY_COCKPIT` is never fired, which lead us to conclude that when the cockpit is empty, somebody having the code can surely enter into the cockpit.

IX. Model Checking

A. Model Checking with CTL formulas

TTool includes a native model checker that implements the model checking approach introduced in section III.C. The editor of TTool enables editing the SysML model and the properties to be verified. The model checker of TTool processes the model and the properties. It outputs a yes/no answer for each property, and assists the designer of the model with counter-examples when the properties are not satisfied.

Users of TTool's native model checker are therefore not obliged to learn using an external model checker. Nor are they obliged to learn any internal language specific to TTool. As a mater of fact, the properties are expressed inside the SysML model and the yes/no answers are also reported inside the SysML model (see Figure 9). The properties are first expressed in so-called 'safety pragmas' (in other word a customized SysML comment) that is included in the block diagram. A second step consists in starting the model checker. The last step consists in coming back to the safety pragma to observe which property is verified (green V) or not verified (red X), or to open counter-examples.

Each line in a safety pragma expresses one property in the form of a CTL logic formula. Figure 9 lists four properties. This is how these properties can be phrased in English:

1) Is it possible to have one passenger in the cockpit, the latter's door being locked? The answer is true.

2) Is it possible for the cockpit to be locked and empty at the same time? The answer is false.

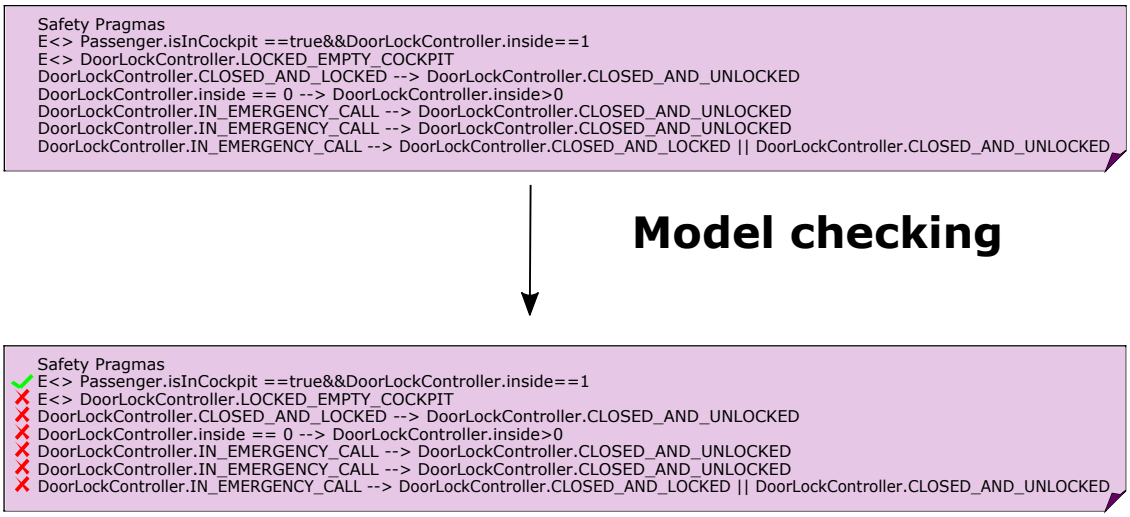


Fig. 9 Properties expressed and checked in the SysML model.

3) From a situation where the door is closed and the lock button is locked, will the system ever reach a situation where the door remains unlocked and the door button is unlocked? The answer is false.
4) From a state of emergency call, will the system ever reach a situation where the door is closed and the lock button is unlocked? The answer is negative.

B. Model Checking of reachability and liveness of states

An important aspect to verify is whether a particular state can be reached, allowing for example to check whether unwanted events can occur. Another aspect concerns liveness of a state, that is whether a particular state is included in which traces. To perform such model checking in TTool, there is no need to express properties in terms of CTL formulas. Properties verification boils down to checking whether one or several states in the state machines are reachable or not. For a given state in a state machine, one may perform either one of the following verification tasks:

- **Reachability of state S**. Is S reachable in at least *one* execution trace starting from the initial state of the block containing the state machine containing S?
- **Liveness of state S**. Is S present in *all* execution traces starting from the initial state of the block containing the state machine containing S ?

Before checking state S against reachability and liveness, one must select S and label it with a 'RL' option where R and L stand for 'Reachability' and 'Liveness', respectively. After model checking the SysML model, the R and L letters of the label are colored in green or red to indicate whether the property holds or not.

Figure 10 depicts an excerpt of the state machine associated with the cockpit door controller. The problem is to decide whether state `LOCKED_EMPTY_COCKPIT` is reachable or not. The model checker negatively answers, which means the SysML model makes it impossible to lock a cockpit that is empty.

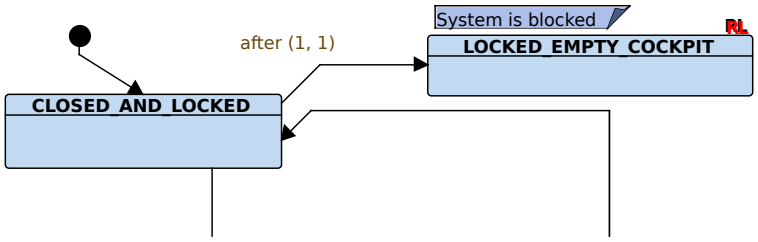


Fig. 10 States and actions checked against their reachability.

X. Security Verification

In previous section, model checking has been applied to verify the SysML model of the door controller against safety properties. This section reuses that SysML model to verify security properties. As usual when one applies two verification techniques to the same model, the two techniques do not necessarily extract the same data from the model. Let us illustrate what may be specific to security property verification. Security-exclusive information may include for example the possibility (or not) for an attacker to use communication links to perform attacks: a public link can thus be used to perform attacks, whereas a private one can not. A public link is identified with an eye within a triangle. For instance the internal communication channel `chout`-to-`chin` is public, as shown in Figure 7. Concerning the attacker, we assume a Dolev-Yao attacker model [25]. An attacker can listen to all messages on public links, forge new messages from the information he or she has learned while spying the link, and inject these messages in public links.

Three kinds of security properties can be expressed and verified with TTool / ProVerif.

1) The **Confidentiality** of a block attribute.
2) The **Integrity** of a message. Integrity is called "Weak authenticity" in TTool.
3) The **Authenticity** of a message.

Our case study specifies two security requirements addressing confidentiality of the secret code, and (weak and strong) authenticity of the message sent by the keypad to the door management subsystem. Let us consider Figure 11. The section labelled by `Security Property` shows how the properties are described in SysML block diagrams with the objective to be processed by TTool. The confidentiality property concerns the attribute `codeValue` of `Keypad`. Authenticity relates to the message m sent by `KeypadController` after entering state `sendingCode`, and received by `DoorLockController` just before entering the `IN_EMERGENCY` state.

Figure 11 also features security-exclusive characteristics. The "InitialSessionKnowledge" pragma expresses that each time the system is started, a new sk (for symmetric key) attribute value is shared between the `KeypadController` and the `DoorLockController` blocks. Since sk is of type `Key`, it means that sk is a symmetric key generated at system boot-up and shared between the two referred blocks.

Last but not least, state machines can also use security-exclusive methods, such as $sencrypt(Message, Key)$ for performing a symmetric encryption, $sdecrypt(Message, Key)$, $aencrypt(Message, key)$ for performing an

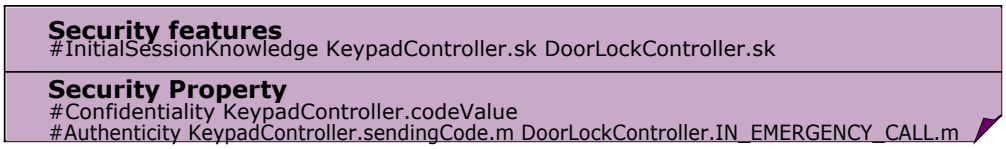
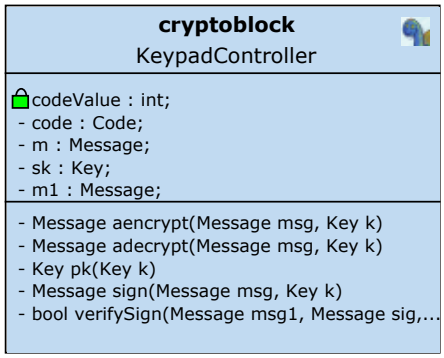


Fig. 11 Security properties included into the SysML model.

asymmetric encryption, and so on. These methods are ignored by the safety verification process. Actually, methods are used either for the security verification process or for executable code generation.

TTool relies on a model-to-proverif transformation* for checking security properties[43]. Results from ProVerif are then backtraced to models, either as a green/red/grey (satisfied, non satisfied, could not be proved) put next to the block attributes (for confidentiality) or next to security properties for (weak/strong) authenticity. In case a security property is not satisfied TTool can build a sequence diagram from the output of ProVerif: this sequence diagram shows a trace proving the security property violation.

Figure 12 shows how verification results are backtraced: the confidentiality property is satisfied, while the weak/strong authenticity property is not satisfied. Indeed, the symmetric key that is currently used to ensure the confidentiality cannot protect against *e.g.*, replay attacks. A possible countermeasure would be to use asymmetric cryptography and a unique identifier per message.



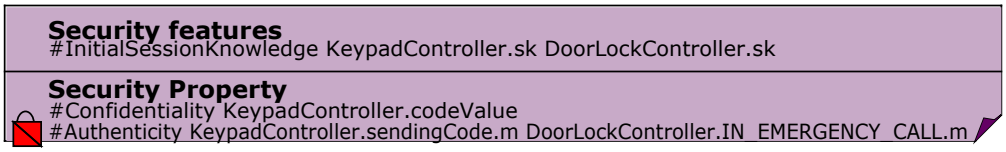


Fig. 12 Backtracing of security verification to block diagrams.

*This transformation has been mathematically proved for Confidentiality properties, and a proof has been sketched for authenticity properties [49]

XI. Conclusions

A. Contributions

The acronym MBSE was coined to denote system engineering approaches that take models as common references where previous document-centric approaches used to scatter information. Standardization of SysML at OMG has open promising avenues to give systems engineers a modeling language to share. SysML is a wide spectrum and graphic modeling language that may be customized to address one family of systems in particular.

AVATAR is an example of SysML customization that targets real-time systems. Modeling with SysML is more than just drawing the different diagrams, associated tools offer possibilities to analyze SysML models for specific properties. This paper shows how free and open source toolkit TTool enables formal verification of AVATAR models. Both safety and security properties can be verified, as shown on the Cockpit Door Control System that serves as running example throughout this paper.

B. Future Work

OMG is in the process of standardizing SysML v2. The new version of SysML will own both a textual notation and a graphic one. AVATAR and TTool will thus evolve to take the new textual syntax into account. SysML v2 extends SysML v1, for instance by adding port delegation to composition. Note: port delegation is already supported by TTool. We may expect SysML v2 to preserve the fundamentals of SysML v1. Consequently some difficulties encountered in teaching SysML v1 will remain at the time of teaching SysML v2. For instance, the authors of this paper have frequently noticed that use case diagrams are often misused by newcomers to SysML. In [46], Rizo Aquino, de Saqui-Sannes and Vingerhoeds have presented a prototype of methodological assistant that helps models designer to create and improve use case diagrams. This concept of methodological assistant deserves to be extended to other diagrams. Adding Artificial Intelligence engines to modeling assistant is a promising avenue to explore.

In terms of method, combination of STPA and SysML has been investigated in [33] to extend STPA with formal verification of safety properties. Next step will consist in taking security properties into account.

References

[1] Apvrille, L., de Saqui-Sannes, P., and Vingerhoeds, R. A., "An Educational Case Study of Using SysML and TTool for Unmanned Aerial Vehicles Design," *IEEE Journal on Miniaturization for Air and Space Systems*, Vol. 1, No. 2, 2020, pp. 117,129. doi:10.1109/JMASS.2020.3013325.

[2] Le Sergent, T., Dormoy, F.-X., and Le Guennec, A., "Benefits of Model Based System Engineering for Avionics Systems," *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016), Toulouse, France*, 2016, pp. 1, 11.

[3] OMG, *Systems Modeling Language, version 1.6, https://www.omg.org/spec/SysML/*, December 2019.

[4] OMG, *Unified Modeling Language, version 2.5.1., https://www.omg.org/spec/UML*, December 2017.

[5] *TTool, http://ttool.telecom-paris.fr/, last access on April 14,*, 2021.

[6] Ouchani, S., Ait Mohamed, O., and Debbabi, M., "A formal verification framework for SysML activity diagrams," *Expert Systems with Applications*, Vol. 41, No. 6, 2014. doi:10.1016/j.eswa.2013.10.064.

[7] Huang, E., McGinnis, L., and Mitchell, S., "Verifying SysML activity diagrams using formal transformation to Petri nets," *Systems Engineering*, Vol. 23, No. 1, 2019.

[8] Delatour, J., and Paludetto, M., "Uml/pno: A way to merge uml and Petri net objects for the analysis of real-time systems," *Oriented Technology: ECOOP'98 Workshop Reader*, 1998, p. 511–514. doi:10.1007/3-540-49255-0_169.

[9] Schafer, T., Knapp, A., and Merz, S., "Model checking uml state machines and collaborations," *Electronic Notes in Theoretical Computer Science*, Vol. 55, 2001, p. 357–369. doi:10.1016/S1571-0661(04)00262-2.

[10] Apvrille, L., Courtiat, J., Lohr, C., and Saqui-Sannes, P. d., "TURTLE: a real-time UML profile supported by a formal validation toolkit," *IEEE Transactions on Software Engineering*, Vol. 3, No. 7, 2004, pp. 473, 487. doi:10.1109/TSE.2004.34.

[11] Bruel, J.-M., "Integrating formal and informal specification techniques. why? how?" *Workshop on Industrial-Strength Formal Specification Techniques, Los Alamitos, CA, USA. IEEE Computer Society*, 1998, p. 50. doi:10.1109/WIFT.1998.766297.

[12] Szmuc, W., and Szmuc, T., "Towards Embedded Systems Formal Verification Translation from SysML into Petri Nets," *25th International Conference Mixed Design of Integrated Circuits and System (MIXDES)*, 2018, pp. 420–423. doi: 10.23919/MIXDES.2018.843687.

[13] Rahim, M., Boukala-Loualalen, M., and Hammad, A., "Hierarchical Colored Petri Nets for the Verification of SysML Designs-Activity-Based Slicing Approach," *4th Conference on Computing Systems and Applications (CSA 2020)*, Lecture Notes in Networks and Systems (LNNS), Vol. 199, Algiers, Algeria, 2020, pp. 131 – 142. URL `https://publiweb.femto-st.fr/tntnet/entries/17274/documents/author/data`.

[14] Wang, H., Zhong, D., Zhao, T., and Ren, F., "Integrating model checking with sysml in complex system safety analysis," *IEEE Access*, Vol. 7, 2019, p. 16561–16571. doi:10.1109/ACCESS.2019.2892745.

[15] Ali, S., "Formal verification of SysML diagram using case studies of real-time system," *Innovations in Systems and Software Engineering*, Vol. 14, No. 6, 2018, p. 245–262. doi:10.1007/s11334-018-0318-5.

[16] Mahani, M., Rizzo, D., Paredis, C., and Wang, Y., "Automatic Formal Verification of SysML State Machine Diagrams for Vehicular Control System," *SAE Technical Paper*, 2021. doi:doi.org/10.4271/2021-01-0260.

[17] Kausch1, M., Pfeiffer1, Raco1, D., and Rumpe, B., "Model-Based Design of Correct Safety-Critical Systems using Dataflow Languages on the Example of SysML Architectureand Behavior Diagrams," *AVIOSE'2021, Software Engineering 2021 Satellite Events, Bonn, Germany (virtual)*, 2021, pp. 1 – 22.

[18] Ando, T., Yatsu, H., Kong, W., Hisazumi, K., and Fukuda, A., "Formalization and model checking of sysml state machine diagrams by csp#," *Computational Science and Its Applications (ICCSA)*, 2013, p. 114–127. doi:10.1007/978-3-642-39646-5_9.

[19] Laleau, R., and Mammar, A., "An overview of a method and its support tool for generating B specifications from UML notations," *ASE2000. Fifteenth IEEE International Conference on Automated Software Engineering*, 2000, p. 269–272. doi:10.1109/ASE.2000.873675.

[20] Calvino, A. T., and Apvrille, L., "Direct Model-Checking of SysML Models," *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (Modelswards'2021), Vienna, Autrichia (online)*, 2021, p. 1..8.

[21] DeAntoni, J., and Mallet, F., "Timesquare: Treat your models with logical time," *International Conferenceon Modelling Techniques and Tools for Computer Per-formance Evaluation*, Springer, 2012, p. 34–41. doi:10.1007/978-3-642-30561-0_4.

[22] Jürjens, J., "Developing Secure Embedded Systems: Pitfalls and How to Avoid Them," *29th International Conference on Software Engineering (ICSE 2007)*, ACM, 2007, pp. 182–183. doi:10.1109/ICSECOMPANION.2007.30.

[23] Kordy, B., Mauw, S., Radomirović, S., and Schweitzer, P., "Foundations of Attack–Defense Trees," *Formal Aspects of Security and Trust*, edited by P. Degano, S. Etalle, and J. Guttman, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 80–95. doi:10.1007/978-3-642-19751-2_6.

[24] Shen, G., Li, X., Feng, R., Xu, G., Hu, J., and Feng, Z., "An Extended UML Method for the Verification of Security Protocols," *19th International Conference Engineering of Complex Computer Systems (ICECCS)*, 2014, pp. 19–28. doi:10.1109/ICECCS.2014.12.

[25] Dolev, D., and Yao, A., "On the security of public key protocols," *IEEE Transactions on Information Theory*, Vol. 29, No. 2, 1983, pp. 198–208. doi:10.1109/TIT.1983.1056650.

[26] Abadi, M., and Blanchet, B., "Analyzing Security Protocols with Secrecy Types and Logic Programs," *Journal of the ACM*, Vol. 52, 2005, pp. 102–146. doi:10.1145/1044731.1044735.

[27] ANSI, "EIA632 - Process for Engineering a System," *GEIA, Arlington, VA, USA*, 2003.

[28] 1220-2005, I., *IEEE Standard for Application and Management of the Systems Engineering Process*, 2005. doi:10.1109/IEEESTD.1999.88825.

[29] ISO, *IEC 15288 - https://www.iso.org/fr/standard/63711.html*, 2003.

[30] SAE, *ARP4754A: Guidelines for Development of Civil Aircraft and Systems*, 2010.

[31] Zhu, S., Tang, J., Gauthier, J.-M., and Faudou, R., "A Formal approach using SysML for capturing functional requirements in avionics domain," *Chinese Journal of Aeronautics*, 2018. doi:10.1016/j.cja.2019.03.037.

[32] de Saqui-Sannes, P., Vingerhoeds, R., and Apvrille, L., "Early Checking of SysML Models applied to protocols," *12th International Conference on Modeling, Optimisation and Simulation (Mosim 2018), Toulouse, France*, 2018, pp. 1,8.

[33] Rey de Suza, F., Melo Bezerra, J. d., Hirata, C., de Saqui-Sannes, P., and Apvrille, L., "Combining STPA with SysML Modeling," *4th annual IEEE International Systems Conference (SysCon 2020)*, 2020, pp. 1,8. doi:10.1109/SysCon47679.2020.9275867.

[34] Fotso, S. J. T., Laleau, R., Ruiz Barradas, H., Frappier, M., and Mammar, A., "A Formal Requirements Modeling Approach:Application to Rail Communication," *ICSOFT 2019: 14th International Conference on Software Technologies*, Scitepress, Prague, Czech Republic, 2019, pp. 170–177. doi:10.5220/0007809701700177, URL `https://hal.archives-ouvertes.fr/hal-02403931`.

[35] Wolny, S., Mazak, A., Carpella, C., Geist, V., and Wimmer, M., "Thirteen Years of SysML: A Systematic mapping Study," *Software and Systems Modeling*, Vol. 19, 2020, p. 111–169. doi:10.1007/s10270-019-00735-y.

[36] Gérard, S., Dumoulin, C., Tessier, P., and Selic, B., "Papyrus: A UML2 tool for domain-specific language modeling," *Model-Based Engineering of Embedded Real-Time Systems*, 2007, pp. 361, 368. doi:10.1007/978-3-642-16277-0_19.

[37] *Modelio, https://www.modeliosoft.com/en/modules/sysml-architect.html, last access April 14,*, 2021.

[38] *Entreprise Architect, https://sparxsystems.com/, last access April 14,*, 2021.

[39] Case, O., *SysML in Action with Cameo Systems Modeler*, 2017. doi:10.1016/C2016-0-00866-5.

[40] *Rhapsody, https://www.ibm.com/us-en/marketplace/systems-design-rhapsody, last access April 14,*, 2021.

[41] Fisman, D., and Pnueli, A., "Beyond regular model checking," *21st conference on Foundations of Software Technology and Theoretical Computer Science*, Vol. LNCS 2245, 2001, pp. 156, 170. doi:10.1007/3-540-45294-X_14.

[42] Apvrille, L., and Roudier, Y., "SysML-Sec: A SysML Environment for the Design and Development of Secure Embedded Systems," *INCOSE/APCOSEC 2013 Conference on system engineering, Yokohama, Japan, September 8-11*, 2013, pp. 655–664. doi:10.1007/3-540-49255-0_169.

[43] Ameur-Boulifa, R., Lugou, F., and Apvrille, L., "SysML Model Transformation for Safety and Security Analysis," *Security and Safety Interplay of Intelligent Software Systems*, edited by B. Hamid, B. Gallina, A. Shabtai, Y. Elovici, and J. Garcia-Alfaro, Springer International Publishing, Cham, 2019, pp. 35–49. doi:10.1007/978-3-030-16874-2_3.

[44] INCOSE, *INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities, 4th Edition*, 2015.

[45] Airbus, *https://airbusa320neoengine.blogspot.com/2020/03/airbus-a320-cockpit-door-locking-system.html, last access on April 14,*, 2021.

[46] Rizzo Aquino, E., de Saqui-Sannes, P., and Vingerhoeds, R., "A Methodological Assistant for Use Case Diagrams," *Modelsward 2020, Valetta, Malta.*, 2020. doi:10.5220/0008938002270236.

[47] Brazier, F., Langen, P. v., Lukosh, S., and Vingerhoeds, R., *Design, Engineering and Governance of Complex Systems*, NAP Foundation Press, 2015. doi:10.1016/j.ijproman.2013.10.007.

[48] Crawley, E., Cameron, B., and Selva, D., *System Architecture: Strategy and Product Development for Complex Systems*, Pearson Higher Education, Inc., 2015.

[49] Lugou, F., "Approaches for analyzing security properties of smart objects," Theses, Université Côte d'Azur, France, Feb. 2018. URL https://tel.archives-ouvertes.fr/tel-01791996.