

Security Analysis: From model to system analysis

Drouot Bastien, Valery Monthe, Sylvain Guérin, Joël Champeau

▶ **To cite this version:**

Drouot Bastien, Valery Monthe, Sylvain Guérin, Joël Champeau. Security Analysis: From model to system analysis. CRiSiS 2022 : International Conference on Risks and Security of Internet and Systems, Dec 2022, Sousse, Tunisia. hal-03866297

HAL Id: hal-03866297
https://hal.science/hal-03866297

Submitted on 24 Nov 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Security Analysis: From model to system analysis

Bastien Drouot, Valery Monthe, Sylvain Guérin, and Joel Champeau

Lab STICC, ENSTA Bretagne, Brest ,France.
{fistname.lastname}@ensta-bretagne.fr

Abstract. There is a wide range of security solutions on cyber-physical systems, most aimed at preventing an adversary from gaining access to the system. However, to make a cyber-physical system more resilient and discover possible attack scenarios, it is necessary to analyze systems by taking into account their interactions with their environment. Standard formal analysis approaches are based on a model of the system. From a quantitative and qualitative point of view, the results of these analyzes depends on the model abstraction relative to the system. Usually, property verification is performed with formulas expressed in specific logics such as LTL or CTL. One of the problems is the semantic gap between textual requirements and these formalisms. In a security context, attacker interests are also necessary to take into account in the properties expression, in addition to system requirements.
In this article we propose an approach allowing to analyze a real cyber-physical system while taking into account the interests of an attacker and while reducing the semantic gap between the textual requirements and logic formulas. The proposed methodology relies on the property specification patterns and the specification of an interface related to the state of the deployed embedded software. The motivating example used in this article comes from an industrial partner included in a collaborative project.

Keywords: Cyber-Security · Modeling · Formal Methods · Model-Checking · Property Specification · Case Study.

1 Introduction

Security of Cyber Physical Systems is a real challenge especially for communicating ones. To improve the security of these systems at design time, formal methods can be used to provide analysis support at behavioral level. This technique is based on the use of formal models which provide system abstraction to mainly focus on the communicating system behavior [3]. To apply efficiently the formal methods, the analyses lead to verify properties regarding the system models. One of the challenges of the approach is to formalize properties from requirements expressions which are mainly textual with a dedicated text structure in the best case.

In a security context, formal properties are not only derived from requirements but must also integrate the potential interests of attackers. These interests provide objectives to the analysis on the system, and include the behavior of attackers interacting with the system. The security property expression is mainly based on temporal logics to take into account propositions which integrate time. In many cases, the expression of these

security properties remains an issue for domain experts without experience on formal methods.

So one of our goal is to bridge the gap between the textual security requirements and the formal properties to help the domain expert to conduct an analysis on their system.

Another main drawback of formal methods is the use of models for the system under study. Models are powerful to take into account a dedicated viewpoint of the system and the abstraction relative to this viewpoint provides an efficient focus for the issue to address. In a previous step, we have demonstrated that the formal methods can be applied successfully on our reference system with a full MBSE approach [11]. This previous experiment also allowed us to analyze our results and identify research opportunities. Indeed for embedded systems, application behavior is often dependent on the deployment of the software component on the embedded target. Therefore, models allow analysis but they are not always sufficient to completely guarantee the behavior of the deployed software, especially to know if the system is resilient to attacks.

To avoid the system model drawback, we suggest a methodology based on the analysis of the embedded software code. However the models remain relevant to model the environment. The methodology we propose here is a heterogeneous approach with models for environment specification and the embedded software itself. The base of our approach is the OBP model checker which provides the capacity to take into account heterogeneous formalisms [10],[2].

The rest of the paper is organized as follows. In Section 2, we present the works on which we base our approach, including the property specification patterns and OBP model checker. In Section 3, we introduce the Car Reservation System (CRS), the case study used throughout this paper to illustrate our approach. After that we describe the shared API between system and environment and used to catch the state of the deployed embedded software in Section 4. In Section 5 we present our implementation of the property specification patterns taking into account the interests of the attacker in security properties. In Section 6 we analyse our results and share the lessons learned on our methodology. And we conclude in Section 8.

2 Background

2.1 Previous MBSE approach

In an initial phase, we applied a full MBSE approach [11] on our motivating example based on UPPAAL formal models to perform a methodology on qualitative risk analysis. The goal of our methodology is to identify the risks which are not elicited during the functional analysis and test phases. Based on this experiment, we have identified several key aspects on our methodology: 1) Environment models provide an efficient way to specify the behavior of external entities constraining system behavior. These models define an execution context for the system related to use case scenarios identified by the system designer. 2) Based on models of the system and the environment, we can define security properties to identify risks on system components or behaviors, i.e an action sequence. The scenarios violating the security properties are possible attacks on the system. 3) The gain obtained with LTL properties is to be put in perspective relatively to the difficulty to express these properties. In fact, these LTL expressions are far

from textual requirements. 4) A full MBSE approach is powerful at specification or design time but models remain an issue to take into account the embedded system code. In practice, deployed software is rarely fully derived (generated) from the analyzed model, and contains a significant amount of manual code.

Lessons learned at the end of these first experiments reveal two major issues : 1) how to bridge the gap between the requirements or attacker interests which are informally described and a formal formalism, and 2) how to take into account the real embedded system software in a formal verification approach for security analysis.

2.2 Property Specification Patterns

The purpose of verification is to make sure that a system meets its requirements. A requirement can be simply defined as an expectation or constraint that a service, product or system must satisfy. The property is obtained by describing this expectation or constraint in a formal language, i.e. in the form of a logical formula to be verified. There are several temporal logics that can be used to specify properties: LTL, CTL, TCTL, TLA, TLA+, etc.

Let's take an example: Consider the following requirement: "*the end of task A leads to the end of task B*". To express this requirement in temporal logic, we must first define atomic propositions, which are predicates whose evaluation result is a boolean. The corresponding logic formula can be written in the chosen temporal logic, such as:

$$P : \textit{task A is ended}.\ \ S : \textit{task B is ended}.;\quad CTL : AG\ (\ P \rightarrow AF\ (\ S\ )\ ) \quad (1)$$

We can have even more complex formulas. For example, with the requirement: "*After event Q until the arrival of event R, the end of task A leads to the end of task B.*"; we would have the following formulas in the CTL logic.

$$CTL : AG(Q \rightarrow\ !E[!R\ U\ (P\ \&\ !R\ \&\ (E[!S\ U\ R] \mid EG(!S\ \&\ !R)))]) \quad (2)$$

We may notice on this example that formulas in temporal logics can be very complex and quickly become incomprehensible to domain experts.

Property specification patterns allow domain specialists to write formal specifications that can be used for model checking. One of the best-known specification models are Dwyer's patterns [6], [7]. Dwyer et al.[8] developed a pattern system for property specification. These patterns allow people who are not experts in temporal logic to read and write formal specifications. They are divided in two major groups: order and occurrence. Each pattern has an associated scope, which represents the context in which the property must hold. With Dwyer's patterns, the formulas of equations 1 and 2 are written respectively:

$$\textit{Globaly S Responds to P} \quad (3) \qquad \textit{After Q Until R (S Responds to P)} \quad (4)$$

We thus obtain in equations 3 and 4 logical expressions that are much easier to understand by domain specialists.

2.3 OBP model checker

Heim et al.[10] address the state space explosion problem observed in the verification of industrial asynchronous systems. To meet this challenge, they proposed a new approach based on the specification of the context (the environment of the system) and an

observation engine called OBP (Observer Based Prover). They start from the idea that, given a property to verify, one does not need to explore all the possible configurations of the complete system. Among all the possible behaviors of the system, a tiny part is sufficiently representative for the property to be verified. Thus, specifying a relevant environment (a context) makes it possible to restrict the behavior of the system to the only parts where the property deserves to be checked.

The OBP model checker is also used coupled with a language interpreter to provide verification and monitoring on embedded models [2]. This capacity is based on a language interface definition between the interpreter and the model checker and also the verification of formal properties on exhaustive exploration of the embedded models. We intensively use these potentialities in our methodology.

3 Motivating example

This section introduces the case study used throughout this paper to illustrate our approach. The Car Reservation System (CRS) has been designed for a companies with a large car fleet. The CRS system presented in this article is an abstraction of a study system, coming from a collaborative project between companies and research institutions.

3.1 System Presentation

The global context of the system under study is presented by the figure 1. This figure aims to highlight the key component, the embedded system, deployed in its environment that includes malicious persons. This critical part of the system is implemented

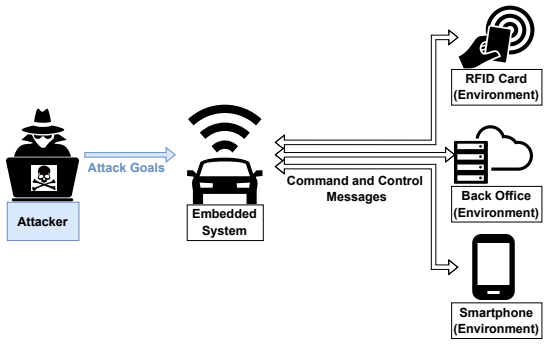


Fig. 1: System Architecture

in an embedded software on a dedicated hardware in the car. This software controls the access to the car and takes into account interactions with the user and the IT server part of the CRS system. The embedded software is the main focus of our attention for a security evaluation due to a risk analysis conduct previously [11]. During this analysis, we applied a top-down MBSE methodology based on system specification study. In this context, we have designed a specification formal model to analyze the behavior of the system with regard to the security properties obtained after the risk analysis.

In this system, each user has an ID stored in a RFID card or in a mobile application. The server or back office is a web server that is used by users to book a vehicle on dedicated day and hours. After a validation, this booking is communicated to the embedded system as a tuple booking ID and user ID. Then, a car session can start. The user can unlock the car, uses it and if necessary temporarily stops using it. The session can resume later or stop if a "stop_booking" request is received. Until the session is not ended, the user can unlock and lock the car again several times to continue the session.

The embedded system interacts with one component at a time, either the server or the ID badge or mobile application. This communication is supported by messages and embedded software process messages in a FIFO mode with a run to completion semantics. A message is dequeued from the FIFO and the effect of this message is executed before any next message consumption.

The server, the mobile and the badge are connected to the embedded system through many technologies and protocols. The system then inherits all the vulnerabilities and weaknesses of these technologies. Our purpose is not target vulnerabilities at this level but malicious persons can exploit them to access or trick the embedded system and potentially change the nominal behavior at application level of the system.

When applying the methodology discussed in the next sections, an interesting goal on our system under study would be, for example, to identify a car unlock situation after session end. The feared events would be that the attacker might want to keep the car locked or unlocked according to the desired outcome.

3.2 General approach

In our MBSE approach applied previously, we modeled the system under study and the environment entities interacting with the system. This approach is particularly relevant to take into account specific behaviors in the environment notably the attacker one.

One of the issues of this approach is the system model creation. In fact, this model is obviously an abstraction of the system and this model are usually build manually by the system designer.

In this context, in order to increase confidence in embedded software, we aim to consider the real embedded source code. But to preserve flexibility and to formalize several environment hypothesis, we keep the environment models including relevant the entities and attacker model. Figure 2 schematizes our approach grouped into 4 parts, to aim differences with classical MBSE approaches :

1. First part focuses on creating properties to verify. In our cybersecurity context, the formal properties are necessarily based on the attacker interests, in addition to the system requirements. To facilitate the property expression, we use a first level, “Abstract Formalized Properties”, to reduce the gap with textual requirements and to be adapted to several temporal logic. In this approach, the attacker succeeds in his attack if a security property is violated (for example by stealing an unlocked vehicle after the end of a reservation). This part is detailed in section 5.
2. The second part focuses on modeling the environment. Against standard approaches based on modeling all the external entities, we take into account the attacker’s behavior through a malicious model. This model can define several attacker behaviors

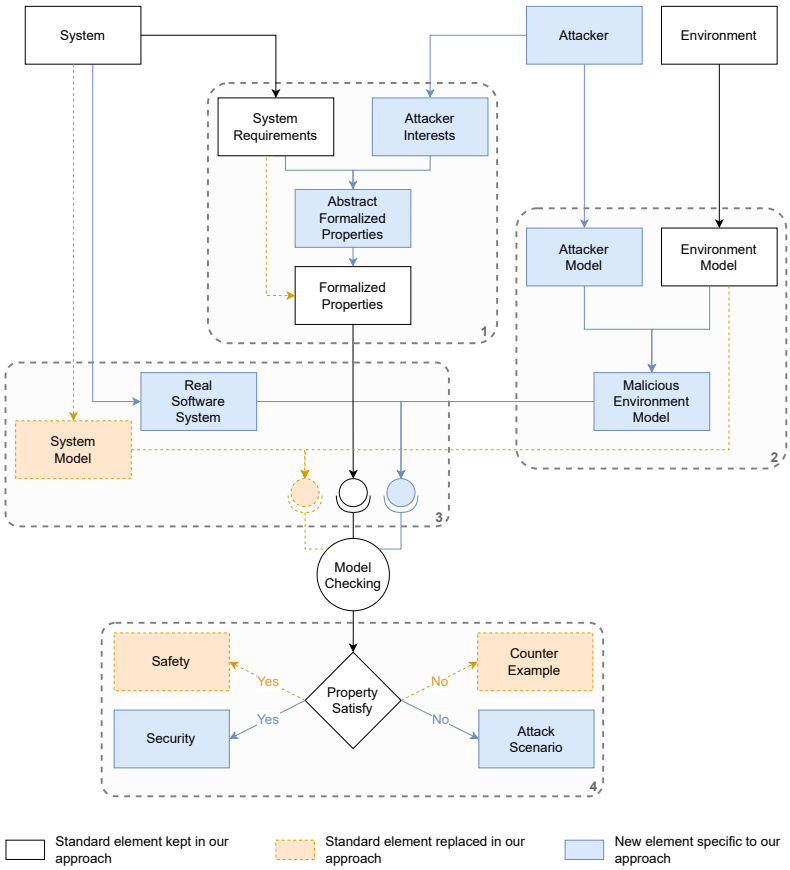


Fig. 2: General Approach

like a man in the middle attack, providing a capacity to perform any actions on the system. This environment modeling is detailed in section 4.1.

3. The third part focuses on the representation of the system. Unlike the MBSE standard approach (represented in dotted lines in figure 2), we suggest an alternative approach having the particularity of directly using the real system code instead of a model. We illustrate our approach in section 4.3 to detail the unavoidable system interface to specify and implement interactions with this system.
4. The last part focuses on property verification. In our approach we define security properties built while taking into account the malicious aspect of the attacker. The objective, for the attacker, Objective of the attacker is here to violate properties to make the system less secure. We give feedback on security property verification in section 6.

So, the main focus of this paper is to define an analysis framework based on several models, in FIACRE language for the environment and a C program for the embedded source code. The interactions between these two parts are defined and controlled through a shared interface, described in the figure 3. The contents of this interface and the link with the model checker OBP are detailed in the next section.

4 Detailed approach

4.1 Environment modeling

The general view of the link between the environment and the system is illustrated figure 3. This schema presents the environment on one side (left part of the schema)

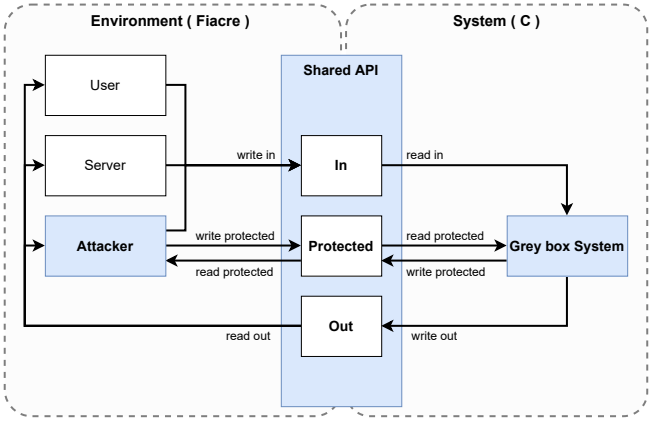


Fig. 3: View on the link between environment and system

and the embedded system on the other (right part). The environment is composed of *User*, *Server* and *Attacker* and interacts with the embedded system through an interface called Shared API. The embedded system is shown as a gray box system.

This environment written in Fiacre models the behavior of elements interacting with the embedded system. We identify three entity types:

- *User*: These model includes all the interactions that user has with the embedded system, such as: pressing a button, passing an access card, interacting with levers but also losing of GPS or GSM connection. The user interacts and produces only inputs to the system. These inputs are integrated in the "In" part of the shared API.
- *Server*: These models represent interactions with one or more servers communicating with the embedded system. Servers produce only input data for the system, and in some cases a server can take control of some system functionalities bypassing some security rules.
- *Attacker*: In order to represent an attack on the system as accurately as possible, we propose a dedicated model to represent the actions of an attacker. We take as hypothesis that attacker can perform any action on the system and at any time. The goal is to take into account every behavior, even unknown ones. These interactions are translated into accesses that can modify input data in the system ("In" part of the shared API) and also read information from the system ("Out" part of the shared API). As shown in the figure 3, we also provide the ability for the attacker to directly modify the internal data of the embedded system (contained here in the "Protected" part of the shared API). This behavior represents a direct and strong attack (rather low level) on the embedded system (memory dump, eavesdropping, sensor manipulation, etc.).

The models of the environment offer several interesting capacities relatively to the security context. First, the power of models provide a quality abstraction to start with a high level of abstraction and after, iteratively adding details into environment models to take into account more complex behaviors. This iterative approach is driven by analyzes that we want to apply on both environment and system, more precisely, according to the properties that come from requirements or attacker interests.

We can also note that modeling the environment provides the possibility, for example, to define several attacker behavior relative to the attacker skills. These skills modify the attacker behavior and specialize its behavior on some scenarios. Several attack contexts can be conceived, evaluated and capitalized through the use of models.

Once the environment has been modeled, we must define how it could behave on the system under study. To act to the system and catch its reaction, we specify a system state API in order to define the interface between environment and system.

4.2 System state and behavior

The specification of the system state API is based on the definition of variables, encoding the global state of the transition function. This function defines the evolution of the system at each execution step to take into account change of the interface values.

```c
//Structure IN
typedef struct{
  carRequest       request;
  carsharingSource originOfRequest ;
   ...

//Structure OUT
typedef struct{
  ...

//Structure Protected
typedef struct{
  bool     isApcON;
  bool     isDoorsLocked;
  bool     isImmobilizerEnabled;
  uint8_t currentBooking;
} struct_protected;

//Shared API
typedef struct{
  struct_in       in;
  struct_out      out;
  struct_protected protected;
}

//System function
int fct_run(shared_struct*);
```

(a) C Source Code

```
//Structure IN
type fstrut_in is record
  request_F           : nat,
  originOfRequest_F   : nat,
  ...

//Structure OUT
type fstrut_out is record
  ...

//Structure Protected
type fstrut_protected is record
  isApcON_F              : bool,
  isDoorsLocked_F        : bool,
  isImmobilizerEnabled_F : bool,
  currentbooking         : nat
end record

//Shared API
type fstrut is record
  c_in        : fstrut_in,
  c_out       : fstrut_out,
  c_protected : fstrut_protected
end record

//External function
extern runSystem (read write fstrut)
: int is "fct_run"
```

(b) Fiacre Code

Fig. 4: Code of the shared API in C (a) and Fiacre (b)

So, the definition of the system is based on two parts:

– *The interface state definition* : This interface includes all the variables representing the global state of the system program. This interface is shared between the system

implementation and the environment models. The interface is implemented on both sides through a C structure including standard C types (figure 4a), and a Fiacre representation (figure 4b). For example, for each external request the car receives a message implemented as an enumeration name "*carRequest*" in listing 4a line 3, and as a natural in listing 4b line 3. Even internal system variables are included in the interface. Indeed they are necessary to represent the global state of the system and can, in some cases, be modified outside the system. For example, the variable *currentBooking*, line 27 in the listing 4a and line 26 in the listing 4b, encodes the ID of the current reservation. If the value of the *currentBooking* variable is zero, it means that no booking is currently running in the vehicle. An attacker could, during an attack, modify this variable in order to harm the system (vehicle theft, cancel the reservation in progress, etc.) This two representations (listing 4a and listing 4b) implement the share interface between the environment and the embedded system.

– *The system function* : The system is viewed as a function with side effects on the shared API and the environment. This function is like a thread function implementation to execute the behavior of the system, see the declaration of the function line 38 of listing 4a. The environment calls this function as external function, see the declaration line 37 of listing 4b. The function interprets messages from the environment, processes the result relative to the current message through the completion of the resulting action, and finally gives back the result to the environment.

With these two software components, the shared API and the system function, we define an abstract transition system with its transition function which processes the system state. This system state is observable and updated from the environment to provide communication facilities and property evaluation. The properties which come from requirements and attacker interests are evaluated on the system state, based on variable values, and also through several transitions to obtain evaluation of temporal properties.

4.3 The OBP model checker

In our framework, the key component to explore all the behavior on the composition of the real system code and environment models, is the OBP model checker. In a standard model checking approach, the system and the environment are modeled like we did in a first step of our methodology [11]. Some approaches emphasize the use of model checking on real software code to help the emergence of errors during sequence of events occurrence [13]. In our approach, we want to highlight two salient points:

– We explore heterogeneous execution states constituted from states of environment models including the attacker, and states of the software system, see the left part of the figure 5
– The model checking algorithms are decoupled from the language(s) used, which provides a language independent exploration capability. In our case, we built the Label Transition System (LTS) using exchanges between the model checker and the interpreters via the runtime controller as depicted in the figure 5. This controller queries interpreters on the model checker request.

To verify properties on these heterogeneous execution states, we apply a synchronous composition between these states and the interpretation of the properties that is defined

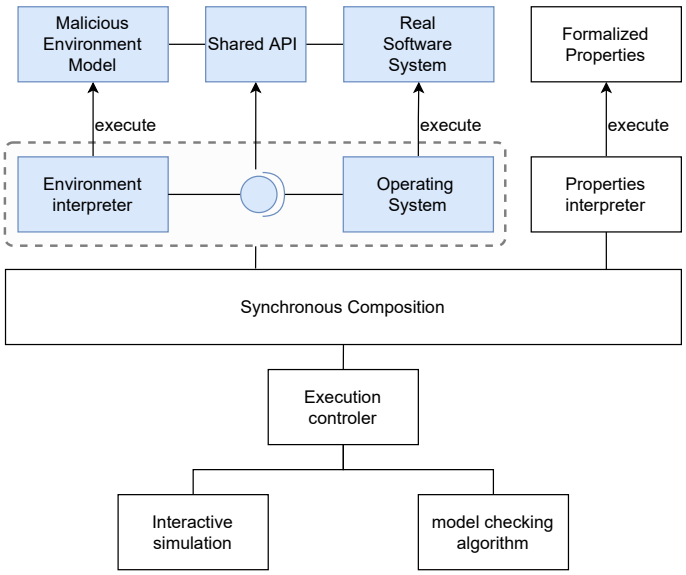


Fig. 5: Architecture of model checking

as observer automata on the execution states. At each execution step, the automata observer progresses in its behavior if the requested event is observed.

Based on this approach, properties are evaluated on the heterogeneous LTS composed of the embedded software part and environment model state. So properties take into account the software behavior dived in a malicious environment context. The exhaustive exploration providing by the model checking algorithms gives all the possible scenarios relative to a no limit attacker behavior, defined in the models.

Now the problem remains to define the properties relative to the objectives that we want to obtain during our analyses. Particularly regarding if we are able to translate the attacker interests in relevant property definitions. The goal of the next section is to present our approach relative to property definition and formalisation.

5 Security Property Modelling

In our approach, system requirements and attacker interests are formalized to ensure a formal verification with OBP model checker. The verification is achieved with LTL formulas which are composed with the heterogeneous state from system and environment. One of the problem is the semantic distance between textual requirements and LTL formulas. Many approaches are based on structured text expressions to minimize this distance. One of the main problems is to enforce a strict and constrained format by requirement engineer.

5.1 Raising abstraction level of formal security properties

In many cases, the need to formulate the requirements and attacker interests in a purely mathematical expressions creates a pragmatic barrier for requirement engineers to use

these techniques. One way to reduce the impact of this barrier would be to bridge the gap between the unstructured textual requirements and mathematical formulas of properties.

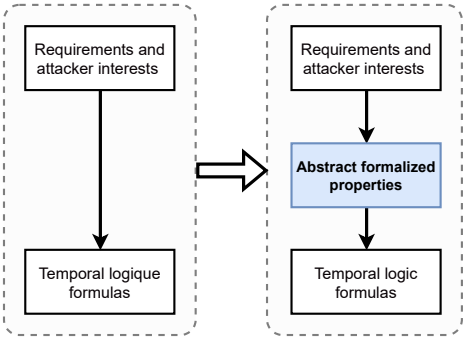


Fig. 6: Abstraction of formal properties from requirements and attacker interests.

Bridging this gap is an objective but the chosen approach must guarantee to obtain formal properties. We have therefore created an intermediate abstraction level using Dywer's patterns (the "Abstract formalized properties" in right part of the figure 6). This level facilitates requirement expression because no constrains are applied on the textual requirements. And also ensures the independence from mathematical formalism of formal properties. In our case, the bridge between the textual form of the requirements and Dywer's pattern is achieved manually, to avoid constrains for requirement engineer.

The next section presents in details the security requirements next to attacker interests and their translation into LTL formal logic, through the intermediate step of the Dywer's patterns.

5.2 From attacker interests to formal security properties

The risk analysis provides critical elements or potential attacker goals that we must take into account in the security analysis. One of the challenges is therefore to obtain formal properties regarding these attacker interests. As mentioned above, to support this step we use Dywer's patterns to create a link between the conceptual attacker goals and the LTL formulas that are evaluated on the system.

In the following, for each security requirements to be checked we write a sentence closed to the property goal to give the property intention, after we have identified the logic predicates derived from it and we use these predicates to specify the security property using Dwyer's patterns. Finally the LTL properties are derived. The security constraints having been formalized into expressions using Dwyer's patterns, the second step involves translating these expressions into formal properties in a temporal logic. Table 1 summarizes the specifications of the properties expressed using Dwyer's patterns and gives the corresponding Temporal Logic formulas in LTL.
Each block of the table has 4 elements:

1. *Ri*: describes the intention of the business security requirements to be respected and therefore to be verified on the system code;

R1	::When ending a session in the Back Office the car should be locked
proposition	::*P : (BackOffice.booking.isEnding) ∧ (System.car.unlocked)*
Dwyer	::*Globaly Absence (P)*
CTL property	::*A [] not (BackOffice.booking.isEnding and System.car.unlocked)*
R2	::If the user is driving the car, the state of the booking should not be consider as locked in the BackOffice.
proposition	::*P : (System.session.isRunning) ∧ (BackOffice.car.isLocked)*
Dwyer	::*Globaly Absence (P)*
CTL property	::*A[] not ((System.start_session or System.continue_session) and BackOffice.car_locked*
R3	::Once the car is unlocked, at some point in the future, the car should be unbooked and the car can be booked again with correct parameters.
proposition	::*P : !System.car.isLocked ⇒ System.isIdle*
Dwyer	::*Globaly Universality (P)*
CTL property	::*A[] (System.car.unlocked ⇒ System.idle)*

Table 1: mapping between analysed security requirements, properties specification in Dwyer patterns and their corresponding CTL formulas.

2. *Proposition (P)*: specifies the requirement Ri in an atomic proposition, i.e. whose value is either true or false. This proposition is formulated using predicates;
3. *Dwyer*: the specification of the property P to check using Dwyer patterns;
4. *CTL property*: the translation of this property P into an CTL formula.

Considering line 1 of table 1, the need as expressed by the domain expert is as follows: “A session that ends on the backoffice management application requires the vehicle to be locked.”. An analysis of this need makes it possible to formulate the security requirement as follows: "*R1: When ending a session in the Back Office the car should be locked*". From this requirement, it is necessary to specify the property to verify and the way to verify it. This requires using a more formal notation, so we use Dwyer’s patterns. To do this, a constraint to be checked must be defined. This is expressed in the form of an atomic proposition, that is to say a sentence with a Boolean value (true or false), and is written using predicates. From R1, we therefore define 2 predicates: *S: System.car.unlocked* (the system detects that the car is locked) and *B: BackOffice.booking.isEnding* (the state of the reservation is at “ isEnding" in the Back Office). We would not like to have the situation in which the car is unlocked and the reservation in the Back Office is over. So the situation to avoid during the execution of the system is *S AND B*, hence the proposition *P: S AND B*. Then we choose the right Dwyer pattern that corresponds to this situation: it is the *"Absence"* pattern. This allows us to write *Globaly Absence (P)*. Globaly is the scope and means that the Absence pattern must apply on P during the entire execution of the system. Finally we can deduce the property expressed in the temporal logic CTL. In this formula, *"A"* means for all execution paths and *"[]"* means during the whole execution. So the property *A [] P* with *P= not (BackOffice.booking.isEnding and System.car.unlocked)*, evaluates to true if and only if any reachable state satisfies P.

6 Property verification results analysis

In this paper, we present and test a methodology based on formal property description applied on the real embedded system. In this section, we present the lessons learned after adapting the MBSE formal verification methodology by integrating the real embedded system.

6.1 Model checking embedded system code

While integrating the embedded software code in our methodology, we have identified two main advantages relative to the use of the formal model checking techniques.

- First of all, the use of an agnostic model checker provides the possibility to analyze a composition of heterogeneous languages. In our case we have environment models in FIACRE language and system state of the embedded code. For the communication between these two languages, we must specify a shared API to take into account the system state accessible by the environment. So the entities like sensors or actuators are modeled through environment entities and interact with the system via the shared API. This shared API is also the base for security property proposition evaluations. And again, due the use of this model checker, these properties are based on heterogeneous entities (system and environment) and are applied on the composition of heterogeneous languages. One of the advantages for the security properties is to explore extended behavior for the attacker entity.
- Secondly, the major drawback of the previous MBSE approach is the semantic distance between the embedded code and the system model. In our case, the model was created manually so this distance could be reduced if we had a generation step from model to source code. But in embedded context, the source code necessarily integrates specific platform features like OS, or devices API. And in a security perspective, many vulnerabilities are coming out while on this platform deployment. So identifying the vulnerabilities at code level is more relevant instead of model level. This experiment implementing approach where the real source code is integrated in the formal verification demonstrates that the debugging phase is improved and the confidence in the developed software is increased. In our case, due to the exhaustive exploration we found that a man in middle attack leads to the system in a case where the door's car remains open after the end of a session. The attacker reaches its goal, the car can be robbed without any obstacle. Our approach based on formal exhaustive exploration provides a one step beyond on debugging phase for embedded software.

Our experiment uses industrial embedded code with all the application functionalities. In order to focus our approach on the application behavior, we have withdrew communication protocols to avoid platform complexity that has no impact on the application behavior. Indeed, all the communications are interpreted as messages received and sent by the application device communication.

6.2 Security property verification

Just before focusing on the property verification results, we analyze the semantics of these properties regarding the security perspectives. Indeed, the formulas obtained from the Dwyer's expressions have different meanings relative to the adopted viewpoint.

	System Perspective (+)	**Attacker Perspective (-)**
Property Successful (+)	Threat causes no risk (+)	Threat causes risk(-)
Property Failed (-)	Threat causes risk(-)	Threat causes no risk (+)

Table 2: Security Property Modeling.

As previously described, the formulas are based on predicates which contain expressions including the system states, in the sense that variable values translating the memory state of the program. A variable can also represent an attractive resource for an attacker like an open door to rob the car, for example.

In this context, two viewpoints are considered to analyze property results, like shown in the table 2. From the designer or system perspective, a successful property means that no risk leads to it. So the system is resilient to the risk expressed by the property. However, if the property verification fails, the system is faulty, or the attack issued was successful in changing the original behavior of the system and thus reached its goal.

Another way to consider the properties is to assume the attacker's perspective. From this point of view, properties ensure that the goals of the attack or risks are reached. So in contrary to the system perspective, a successful verification shows that the system is faulty in the sense that the attack has successfully reached its goals of altering the system's behavior. This means that the system is sensitive to the considered risk, introduced by the attack. In contrast, the failed verification implies that the attack was unsuccessful, and the system is resilient to the attack goal described by the property. Thus, there is no risk caused by this threat. Indeed if the expression of the predicate contains a state of the system favorable to the attacker like for example "the car remains open at the end of a session", the evaluation of this property to true confirms the attacker in these possibilities but does not provide to him an attack scenario. On the other hand, if the negation of this property fails, a counter example is provided by the model checker and represents an attack scenario for the threat.

Therefore, in a context of system security analysis, the expression of properties and their evaluation are elements to put in perspective according to the adopted viewpoints and also according to the objective of the user of verification formalism, designer or attacker. For example, the property R1 "*When ending a session in the Back Office the car should be locked*" of the previous section illustrates this viewpoint perspective in the sense that we adopt the attacker viewpoint and try to find an attack scenario on our system. The proposition "$P : BackOffice.booking.isEnding \wedge System.car.unlocked$" defines the stable state of the system, but the negation of this proposition expresses that we are looking for a system vulnerability and we hope to find a scenario to reach the attacker goal. Note that the Dwyer expression "*Globaly Absence (P)*" is readable and enough expressive to consider the absence of the predicate in all the futures for all the execution paths.

The evaluation of this property provides a counter example by the model checker based on a transition system with 73 870 states and a path with 2 178 transitions, for the first reached counter example. This scenario is reduced to a 4 steps scenario after manual analysis. This scenario is really an attack scenario because in this case, the attacker has the capacity via *a man in middle* attack to send a *"The car is closed"* message to the back office, and so the back office will send a *"End session"* to the car although the car is really open.

With this example we can notice a limitation of our approach which is the interpretation of the model checker results. For now, this analysis is manually accomplished with the know how on the system and the possible threats. But in case of very long scenario, we should provide a dedicated tooling to support the human interpretation.

7 Related works

Konrad et al. [12] propose a security model that can meet the development needs of secure systems. To maximize understandability, they use well-known notations such as Unified Modeling Language to represent structural and behavioral information. They modified several fields in the design template to convey more security-related information than the original template. Among these fields is the constraint field which was added in the spirit of the Dwyer pattern specification. Lamsweerde [14] offers a constructive approach to modelling, specifying and analyzing application-specific security requirements. His method is based on a goal-oriented framework to generate and resolve obstacles to goal satisfaction. The extended framework tackles malicious obstacles (called anti-objectives) put in place by attackers to threaten security objectives. Threat trees are built systematically by anti-objective refinement until leaf nodes are derived that are either software vulnerabilities observable by the attacker or anti-requirements implementable by that attacker.

Wong et al. [15] propose a model-based approach to express behavioral properties. They describe a PL property specification language for capturing a generalization of Dwyer's property specification models, and translating them into linear temporal logic. Corradini et al [4] offer a complete chain of web tools that allows modeling, verification and exploitation of the results of BPMN processes. They rely on Dwyer patterns and implement some of these patterns (like the Response pattern) in their tools. Dadeau et al. [5] proposes a property and model-based testing approach using UML/OCL models, driven by temporal property models and a tool to help formalize temporal properties. The models are expressed in the TOCL language, an adaptation of Dwyer's property models to OCL and therefore independent of the underlying temporal logic.

AUtili et al.[1] proposes a comprehensive framework, combining qualitative, real-time and probabilistic property specification models. They rely on Dwyer's patterns to systematically discover new property specification models, which would be absent to cover the three aspects mentioned above. They also offer a natural language interface to map models to chosen temporal logic (LTL, CTL, MTL, TCTL, PLTL, etc.). Gruhn et al. [9] extend Dwyer's pattern system by time-related patterns, to take into account real-time aspects in properties.

Some of the work presented ([15], [14]) has deal with the specification of security properties on systems. Others like [1] and [9], have proposed extensions of Dwyer's patterns for real time. The rest ([15], [4], [5]) provide approaches for using Dwyer patterns on BPMN and UML models.

8 Conclusion

The complexity of cyber-physical systems in general and embedded systems in particular, makes their verification less obvious than other software systems. This verification can be done by using formal methods and specifying properties from security requirements, generally defined in natural language. One of the challenges for the community is to correctly translate the security requirements into formal properties to be verified. This need can be satisfied by giving an extended and expressive formalism to the domain experts to bridge the gap between textual requirements and the formal properties.

To address this issue, we have proposed an approach to raise the abstraction level of the formal description of security properties. The methodology consists in using the Dwyer's patterns to create an intermediate level of abstraction between security requirements and logic formulas to be verified. This formalism is closer to requirement expression and provides a level-independent of temporal logics.

On the other hand, we improve the system debugging by diving the embedded software code in the environment models. The global state is obtained by composing the system state and all environment entity states. The environment includes an attacker entity which provides the possibility to define several attacker behaviors, if needed. The heterogeneity with models and embedded code is supported by the specification of a shared API between the two parts. This shared API exposes the system state description for the model checker and is the input for the synchronous composition with the property observer automaton.

This approach was evaluated on a industrial system for controlling booking and use of vehicles in large car fleet. Security properties have been defined and a vulnerability has been identified at system behavioral level.

In the future, we plan to experiment our approach on other industrial systems and we will study additional tooling to help the interpretation the analysis results.

References

[1] Marco Autili et al. “Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar”. In: *IEEE Transactions on Software Engineering* 41.7 (2015), pp. 620–638.

[2] Valentin Besnard et al. “Unified verification and monitoring of executable UML specifications”. In: *Software and Systems Modeling* 20.6 (2021), pp. 1825–1855.

[3] Hao Chen, Drew Dean, and David A Wagner. “Model Checking One Million Lines of C Code.” In: *NDSS*. Vol. 4. 2004, pp. 171–185.

[4] Flavio Corradini et al. “A formal approach for the analysis of BPMN collaboration models”. In: *Journal of Systems and Software* 180 (2021), p. 111007.

[5] Frédéric Dadeau, Elizabeta Fourneret, and Abir Bouchelaghem. “Temporal property patterns for model-based testing from UML/OCL”. In: *Software & Systems Modeling* 18.2 (2019), pp. 865–888.

[6] M. B. Dwyer. “Specification patterns web site”. In: *available at http://patterns.projects.cis.ksu.edu* (web site).

[7] Matthew B Dwyer, George S Avrunin, and James C Corbett. “Patterns in property specifications for finite-state verification”. In: *Proceedings of the 21st international conference on Software engineering*. 1999, pp. 411–420.

[8] Matthew B Dwyer, George S Avrunin, and James C Corbett. “Property specification patterns for finite-state verification”. In: *Proceedings of the second workshop on Formal methods in software practice*. 1998, pp. 7–15.

[9] Volker Gruhn and Ralf Laue. “Patterns for timed property specifications”. In: *Electronic Notes in Theoretical Computer Science* 153.2 (2006), pp. 117–133.

[10] S Heim et al. “Model checking of SCADE designed systems”. In: *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. 2016.

[11] Hiba Hnaini et al. “Security Property Modeling.” In: *ICISSP*. 2021, pp. 694–701.

[12] Sascha Konrad et al. “Using security patterns to model and analyze security requirements”. In: *Requirements Engineering for High Assurance Systems (RHAS’03)* 11 (2003).

[13] Madanlal Musuvathi et al. “CMC: A pragmatic approach to model checking real code”. In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 75–88.

[14] Axel Van Lamsweerde. “Elaborating security requirements by construction of intentional anti-models”. In: *Proceedings. 26th International Conference on Software Engineering*. IEEE. 2004, pp. 148–157.

[15] Peter YH Wong and Jeremy Gibbons. “Property specifications for workflow modelling”. In: *Science of Computer Programming* 76.10 (2011), pp. 942–967.