A Systematic Approach to Integrate Common Timed Security Rules within a TEFSM-Based System Specification

Amel Mammar

Institut Telecom SudParis, CNRS/SAMOVAR, France
amel.mammar@it-sudparis.eu

Wissam Mallouli

Montimage EURL, 39 rue Bobillot, 75013, Paris, France
wissam.mallouli@montimage.com

Ana Cavalli

Institut Telecom SudParis, CNRS/SAMOVAR, France
ana.cavalli@it-sudparis.eu

Abstract

Context: Formal methods are very useful in the software industry and are becoming of paramount importance in practical engineering techniques. They involve the design and modeling of various system aspects expressed usually through different paradigms. These different formalisms make the verification of global developed systems more difficult.

Objective: In this paper, we propose to combine two modeling formalisms, in order to express both functional and security timed requirements of a system to obtain all the requirements expressed in a unique formalism.

Method: First, the system behavior is specified according to its functional requirements using TEFSM (Timed Extended Finite State Machine) formalism. Second, this model is augmented by applying a set of dedicated algorithms to integrate timed security requirements specified in Nomad language. This language is adapted to express security properties such as permissions, prohibitions and obligations with time considerations.

Results: The proposed algorithms produce a global TEFSM specification of the system that includes both its functional and security timed requirements.

Conclusion: It is concluded that it is possible to merge several requirement aspects described with different formalisms into a global specification that can be used for several purposes such as code generation, specification correctness proof, model checking or automatic test generation. In this paper, we applied our approach to a France Telecom *Travel* service to demonstrate its scalability and feasibility.

Key words: Formal Methods, Timed Extended Finite State Machines, Nomad Language, Test Generation.

1. Introduction

1.1. Context and motivations

Security and reliability are important issues in designing and building systems with time constraints because the consequence of any security failure on a user's business and/or environment may be irreversible. Currently, software engineers developing systems with time constraints are not only confronted with their functional requirements but also have to manage other aspects concerning security issues. By "functional requirements", we mean the services that a system provides to end-users. Security rules denote the properties (restrictions) that a system has to fulfill to be constantly in a safe state. For instance, a file system may have to specify an access prohibition to a specific document for a user if he/she is not authenticated or if his/her 10 minute session has expired. More generally, a security rule expresses the obligation, permission or interdiction to perform an action under given conditions called *context*.

Complex systems are often incrementally designed and are evolutionary. New requirements appearing during the life cycle have to be integrated into the initial specification. In this paper, we deal with particular kinds of requirements denoting security properties. Security rules are typical examples of such requirements. At France Telecom[1] for instance, there are many systems running for which new security requirements appear due to security breaches discovered during their life cycles. An example of such systems that we use to illustrate our approach is the *Travel* Web application which is a management application that deals with business travel (called 'missions') of the different employees of the company. During the deployment of this application, France Telecom needs to ensure, among other things, that two successive mission requests are separated by at least 2 minutes. This requirement prevents *denial of service* attacks for example. Redoing the development of these systems to take these new security requirements into account would be very expensive and time consuming. Furthermore, in general the formal specifications of such systems already exist and can be updated to include the additional security requirements. The present paper addresses this need by providing a formal approach to integrate security rules involving time constraints into a system specification based on communicating extended timed automata called TEFSM specification [1].

[1]France Telecom is the main telecommunication company in France

The analysis of security issues led researchers and security experts to define a large number of security languages and models that provide a formal representation of the system's security policies. With the great majority of these models, security rules are specified using modalities like permission, prohibition and obligation that express some constraints on the system behavior. Among these models, we can mention for instance the Policy Description Language (PDL) [2], Ponder [3], RBAC (stands for Role Based Access Control) [4, 5] and Nomad [6] (stands for Non atomic actions and deadlines). This latter is the one we have chosen since it allows security policies with time constraints to be easily expressed.

1.2. Contribution with respect to previous work

In the literature, previous works dealing with secure system specifications are based on integration methodology. The authors of [7] and [8] for instance proposed formal procedures that make it possible to augment a functional description of a system with security rules expressed with OrBAC language [9] (stands for Organizationnal Based Access Control language). In both work, they described security rules that specify the obligation, permission or interdiction of a user to perform certain actions under a given context. However, this context does not involve time aspects. In fact, they only specified rules without time considerations. Another work presented in [10] proposed to translate security rules (still without time constraints) into observers that can communicate with the functional specification of a system specified in EFSM formalism [11] (stands for Extended Finite State Machine) to regulate its behavior.

To overcome the limit of the previous work in dealing with time aspects, we have presented in [12, 13, 14] an approach that goes further by considering time constraints which are relevant in the modern applications that are time-dependent. The approach consists in defining integration algorithms that allow Nomad security rules to be added into a TEFSM specification of the functional requirements of a system. However, the main drawback of this proposal is the large number of clocks it introduces. Indeed, a new clock is defined for each Nomad security rule that the underlying system must fulfill. Since a large number of clocks may lead to a state space explosion when we apply the testing based approach to the resulting TEFSM specification, this paper presents new algorithms that use a unique master clock to integrate any number of Nomad rules. In addition to these new algorithms, the contributions of this paper with respect to [12, 14] are as follows:

- We define algorithms not only to deal with basic rules that include atomic action, but also those involving sequential actions.
- The correctness proof of one integration algorithm which demonstrates the accuracy of our approach.
- An application of the proposed approach to an industrial case study provided by France Telecom.

This paper is organized as follows. Section 2 provides the basic concepts used for the modeling of system behavior from both a functional and security point of view. In section 3, we present an overview of the approach we have developed to augment a TEFSM specification with timed security rules. Then, we describe the integration algorithms in sections 4 and 5. The proof of one of the proposed algorithms is presented in section 6. To demonstrate the feasibility of the developed approach, in section 7 we present an industrial case study that deals with a business travel reservation web application provided by France Telecom. Finally, section 8 concludes the paper.

2. Preliminaries

2.1. Modeling communicating systems using TEFSM model

The objective of modeling a system is to provide an operational specification of a system. This operational specification may include time constraints and can be used as an input to existing validation tools such as interactive or random simulators, model-checkers or test generation engines. In our case, we rely in this paper on a TEFSM model supported by IF language [1] because it includes the main concepts needed to design real-time systems. Moreover, several tools allowing the simulation and the generation of test sequences exist and are open source. A TEFSM modeling of a system consists of a set of IF processes (i.e entities), each one denotes a TEFSM that can communicate with other processes via FIFO channels.

Definition 1. *A TEFSM M is a 7-tuple* $M = < S,\ s_0,\ I,\ O,\ \vec{x},\ \vec{c},\ Tr >$ *where S is a finite set of states,* s_0 *is the initial state, I is a finite set of input symbols (messages possibly with parameters), O is a finite set of output symbols (messages possibly with parameters),* $\vec{x}$ *is a vector denoting a finite set of variables,* $\vec{c}$ *is a vector denoting a finite set of clocks and Tr is a finite set of transitions. Each transition tr is a 4-tuple* $tr =< s_i, s_f, G, Act >$ *where* s_i *and* s_f *are respectively the initial and final state of the transition, G is the guard which denotes a predicate (boolean expression) on variables* $\vec{x}$ *and clocks* $\vec{c}$ *and Act is an ordered set (sequence) of atomic actions including inputs, outputs, variable assignments, clock setting, process creation and destruction.*

The execution of any transition is spontaneous i.e. the actions associated with this transition occur simultaneously and take no time to complete. The *time progress* takes place in some states before executing the selected transitions. More details about *time progress* can be found in [15].

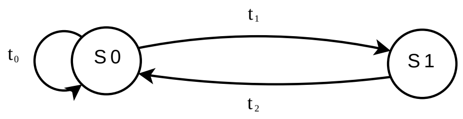


Figure 1: Example of a simple TEFSM with two states.

$t_0 =< S_0, S_0, \bar{P}, (input\ a;\ T'\ ;output\ x) >$
$t_1 =< S_0, S_1, P, (input\ a;\ T\ ;set\ Ck := 0\ ;output\ x) >$
$t_2 =< S_1, S_0, when\ Ck > 6, (input\ b;\ T"\ ;output\ y) >$

We illustrate the notion of TEFSM through the simple example shown in Figure 1. This TEFSM is composed of two states S_0, S_1 and three transitions that are labeled with two inputs a and b, two outputs x and y, one guard (or predicate) P on variables, one clock Ck and three tasks T, T' and T''. The TEFSM operates as follows: starting from state S_0, when input a occurs, predicate P is checked. If the condition holds, the machine performs task T, starts clock Ck, triggers output x and moves to state S_1. Otherwise, the same output x is triggered but it is action T' that is performed and the state loops on itself. Once the machine is in state S_1, it can come back to state S_0 when the clock exceeds the value 6 and receives input b. If so, task T'' is performed and output y is triggered.

Notations: In the remainder of the paper, we need the following notations. For each action a that belongs to the sequence of actions Act:

- $before(a)$ denotes the actions of *Act* that are executed before action a. $before(a)$ is empty if action a is the first action of *Act*.
- $after(a)$ denotes the actions of *Act* that are executed after action a. $after(a)$ is empty if action a is the last action of *Act*.
- The sequence of *Act* actions can then be denoted by: $(before(a); a; after(a))$.

For instance, the action of transition t_1, of Figure 1, can be denoted by:

$$Act = (before(T); T; after(T))$$

where $(before(T) = input\ a)$ and $(after(T) = (set\ Ck := 0; output\ x))$.

2.2. Specification of security rules using Nomad

We use the Nomad formal language to specify without ambiguity the set of security properties that the system has to respect. The choice of this language is mainly determined by the Nomad features that provide a way of describing permission, prohibition and obligation related to non-atomic actions within elaborated contexts that take into account time constraints. By combining deontic and temporal logics, Nomad allows conditional privileges and obligations with deadlines to be described, thanks to the time concept it supports. Finally, it can also formally analyze how privileges on non atomic actions can be decomposed into more basic privileges on elementary actions.

2.2.1. Nomad syntax and semantics

To meet the requirements of the functional model of the system, we define an atomic action in Nomad using the same concepts as for TEFSM actions.

Definition 2. *We define an atomic action as one of the following actions: a variable assignment, a clock setting, an input action, an output action, a process creation or destruction.*

Definition 3. *A non-atomic action is inductively defined as follows:*

- *If A and B are atomic actions, then $(A; B)$, which means that "A is followed immediately by B", is a non-atomic action.*
- *If A and B are atomic actions, then $(A \| B)$, which means that "A and B are performed simultaneously", is a non-atomic action.*

- *If A or B (or both) is a non-atomic action, then $(A; B)$ and $(A \| B)$ are non-atomic actions.*

Definition 4. *(Formula) Let t_c denotes the current instant of the system.*

- *If A is an action then $start(A)$ (starting A), and $done(A)$ (finishing A) are formulae.*
- *If α and β are formulae then $\neg\alpha$, $(\alpha \wedge \beta)$ and $(\alpha \vee \beta)$ are formulae.*
- *If α is a formula and $(d > 0)$, then $O^{-d}\alpha$ is also a formula. $O^{-d}\alpha$ means that α was true at moment $(t_c - d)$.*
- *If α is a formula and $(d > 0)$, then $O^{<-d}\alpha$ is also a formula. $O^{<-d}\alpha$ means that it exists a moment t' where α was true such that $(t_c - d \leq t' < t_c)$.*
- *If α and γ are formulae then $(\alpha|\gamma)$ is a formula whose semantics is: in context γ, formula α is true.*

Remarks. In the remainder of the paper, we respectively refer to operators "O" and "|" by timed and contextual operators, and we use the notation $O^{[<]-d}$ to cover both cases O^{-d} and $O^{<-d}$ with $(d > 0)$. Notice also that using Nomad formalism, we deal with discrete time. In our work, we use real time units like seconds, milliseconds or microseconds depending on the precision desired.

Definition 5. *(A security rule) If α and β are formulae, $\mathcal{R}$ $(\alpha \mid \beta)$ is a security rule where R denotes one of the following deontic operators: $\{\mathcal{P}, \mathcal{F}, \mathcal{O}\}$. The security rule $\mathcal{P}$ $(\alpha \mid \beta)$ (resp. $\mathcal{F}$ $(\alpha \mid \beta)$, $\mathcal{O}$ $(\alpha \mid \beta)$) means that it is permitted (resp. prohibited, mandatory) to have α true when context β holds.*

In the sequel of the paper, formula α denotes a formula of the form $start(\gamma_1)$ where γ_1 is the execution of an actions (rep. a sequence of actions), whereas formula β denotes the context of the form $O^{[<]-d}done(\gamma_2)$ with γ_2 representing the execution of an actions (rep. a sequence of actions). According to definition 5, security rule $\mathcal{P}$ *$(\alpha \mid \beta)$* implies that it is forbidden to start the execution of γ_1 in context β holds, that is: $\mathcal{P}$ *$(\alpha \mid \beta)$*$=$ $\mathcal{F}$ *$(\alpha \mid \neg\beta)$.* In practice, a security policy can be open or closes, that is, permission or prohibition rules are specified. Indeed to avoid the negation in a context, it would be better to write $\mathcal{P}$ *(α | β)* instead of $\mathcal{F}$ *(α | $\neg\beta$).* This is why we propose integration algorithms for both cases (permissions and prohibitions integration) even if they are very similar.

2.2.2. Examples of security rule specifications

We present in this section some examples of security rule specifications expressed in Nomad:

Example 1:

$$\mathcal{P}(start\ (input\ ReqDownload(user, file.doc))| \\ \mathrm{O}^{<-30days}\ (done\ (output\ SubscriptionOK(user))))$$

This rule expresses a permission granted to any user to request to download a file 'file.doc', if he/she has paid his/her subscription. The subscription must have been made during the last month.

Example 2:

$$\mathcal{O}(start\ (output\ DisconnectOK(user))| \\ \neg\ \mathrm{O}^{<-30min}(done\ (input\ Message(user))))$$

According to this obligation rule, the system must disconnect a running connection of any user who is inactive for 30 minutes, that is, the system does not receive any input from him/her.

Example 3:

$$\mathcal{F}(start\ (input\ AuthReq(user))| \\ \mathrm{O}^{<-0.01s}\ (done\ (input\ AuthReq(user))))$$

This prohibition rule forbids the system to accept more than one user authentication request in the same millisecond.

3. Security integration methodology

The integration of security rules into a TEFSM model describing the behavioral aspects of a system leads to a TEFSM specification that takes the security policy into account: we call it *secure functional specification.* The integration process is twofold. At first, the algorithm searches for the rules

to be applied on each transition of the TEFSM specification. Then, it adds some states, transitions or updates the guard of the related transition. These modifications depend on the nature of the rule (prohibition, permission or obligation) and its syntax format. To integrate security rules into a TEFSM specification, we have to make the following assumption: the security rules to be integrated are consistent. We assume that the security rules do not contain any incoherent rules. The consistency of the security policy is out of the scope of this paper and we assume that it has already been checked using different techniques (see for instance [16]). This is an example of an inconsistent security policy composed of two rules $\mathcal{O}(start(A)|O^{-30}done(B))$ and $\mathcal{F}(start(A)|O^{-30}done(B))$. We cannot force the system to perform action A in a context $(C = O^{-30}done(B))$ if this action is forbidden in the same context. Note that no hypothesis is made on the non-deterministic feature of the TEFSM. Indeed, the TEFSM may be non deterministic if for instance two outgoing transitions have their guard true at the same time. In that case, one of them, chosen randomly, is fired. The proposed algorithms remain applicable.
According to the Nomad syntax, there are several possible forms for security rules. It would obviously be tedious to deal separately with each of these forms. Consequently, we classify the Nomad security rules in two main classes described as follows:

(1) Basic security rules: in this class we consider security rules of the form $\mathcal{R}(start(A)|O^{[<]-d}done(B))$ where A and B are actions, $\mathcal{R} \in \{\mathcal{F}, \mathcal{O}, \mathcal{P}\}$ and $(d > 0)$. To make the integration of such rules easier, we also distinguish two subclasses:

   – Basic security rules with atomic actions: actions A and B are atomic.

   – Basic security rules with decomposable actions. A or B or both are non-atomic actions.

(2) General security rules: a general security rule denotes any rule that does not fit into the first class. This means that the rule may contain several contextual and/or timed operators and/or logical connectors.

As a first step of our research work, this paper deals with the first class of security rules.

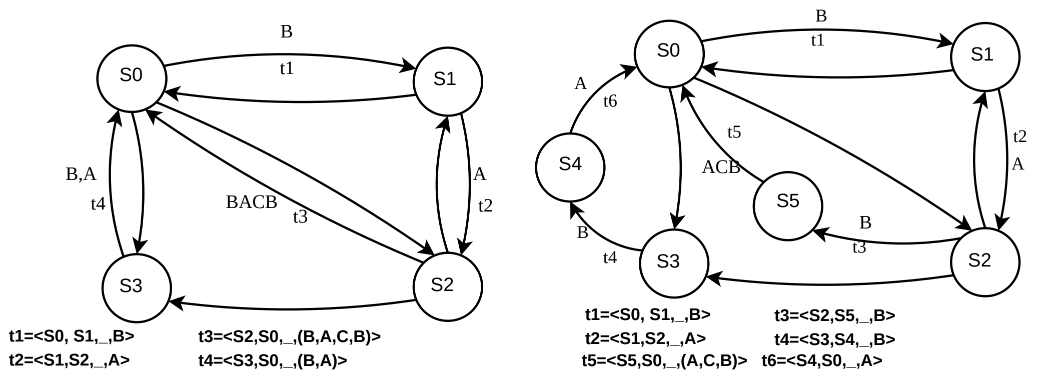


Figure 2: Transition decomposition

4. Integration of basic security rules involving atomic actions

This section describes the integration of basic security rules of the form $\mathcal{R}(start(A|O^{[<]-d}done(B))$ where $\mathcal{R} \in \{\mathcal{F}, \mathcal{O}, \mathcal{P}\}$, A and B denote atomic actions, and $(d > 0)$. Since we deal with a timed context, we need to define a global clock gck to manage the temporal aspect of the rules.

4.1. Prohibition integration: $\mathcal{F}(start(A)|O^{[<]-d}done(B))$

The key idea of integrating such a prohibition rule in a TEFSM model is to check the rule context before performing the prohibited action. If this context is verified, the prohibited action A must be skipped. Otherwise, if the context is not valid, the action is performed without any rule violation.

To achieve this goal, we have to construct a table $Prohib$ to store all the instants (or moments) where it is prohibited to trigger a given transition since it contains the forbidden action A. These instants are denoted by a predicate on clock gck which is a global clock for the system launched in its initial state. For instance, a predicate $((gck < 10) \vee (gck = 15))$ means that the execution of a transition tr is prohibited till the tenth unit of time and also at the fifteenth since it contains a prohibited action. We also define the function $val(gck)$ that provides the clock value gck at a specific moment. Table $Prohib(tr)$ is updated as follows (tr denotes a transition where action A appears):

- After each occurrence of B in the TEFSM transitions, the value of $Prohib(tr)$ is updated by adding a predicate on the instant(s) when it

Algorithm 1 Prohibitions Integration

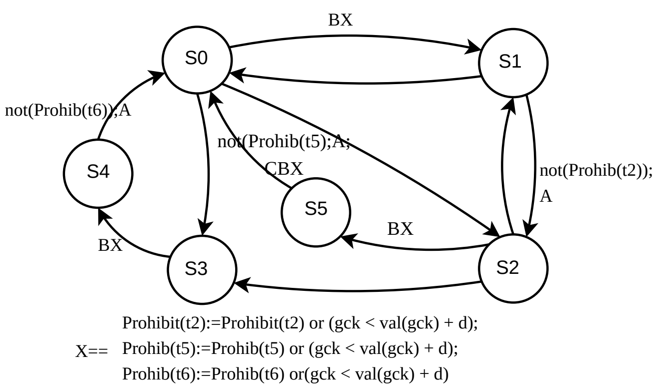
```
Require: The TEFSM model $M =< S, s_0, I, O, \vec{x}, \vec{c}, Tr >$ and the prohibition security
    rule $\mathcal{F}$ (start(A) | $O^{[<]-d}$ done(B))
 1: Let $Tr_{BA}$ be the set of transitions where action $A$ appears after action $B$.
 2: while ($Tr_{BA} \neq \emptyset$) do
 3:     for each (transition $tr =< S_i, S_j, G, Act >$ such that ($tr \in Tr_{BA}$)) do
 4:         /*$tr = < S_i, S_j, G, \{before(B), B,\ after(B) \cap before(A),\ A, after(A)\} >$*/
 5:         /*Create a new state $S'$ and a new transition $tr'$*/
 6:         $tr := < S_i, S', G, \{before(B), B, after(B) \cap before(A)\} >$;
 7:         $tr' := < S', S_j, \_, \{A, after(A)\} >$;
 8:     end for
 9:     $Update(Tr_{BA})$
10: end while
    /*At this point, there is no transition where $A$ appears after $B$*/
11: Let $Tr_A$ be the set of transitions where action $A$ appears
12: Let $Tr_B$ be the set of transitions where action $B$ appears
13: for each (transition $tr =< S_i, S_j, G, Act >$ such that ($tr \in Tr_B$)) do
14:     $tr := < S_i, S_j, G, \{before(B), B,\ Update_{tr' \in Tr_A}(Prohib(tr')),\ after(B)\} >$;
15: end for
16: for each (transition $tr =< S_i, S_j, G, Act >$ such that ($tr \in Tr_A$)) do
17:     $tr := < S_i, S_j, G \wedge \{when\ (\neg Prohib(tr))\}, Act >$;
18: end for
```

is prohibited to trigger tr since it contains action A. The new value of $Prohib(tr)$ is defined by:

$$Prohib(tr) = \begin{cases} Prohib(tr) \vee (gck < val(gck) + d), \\ \qquad\qquad for \mathcal{F}(start(A)|O^{<-d}done(B)) \\ Prohib(tr) \vee (gck = val(gck) + d), \\ \qquad\qquad for \mathcal{F}(start(A)|O^{-d}done(B)) \end{cases}$$

- Before triggering the prohibited transition tr, we check whether the value $val(gck)$ satisfies ($Prohib(tr)$) to deduce if tr can be fired or not.

Notice that $Prohib(tr)$ can be updated according to all the security rules that deny the execution of any action A that may be executed in tr. Depending on each context, the update of $Prohib(tr)$ is performed as mentioned above.



BX
S0
S1
not(Prohib(t6));A
not(Prohib(t5);A;
CBX
S4
S5
not(Prohib(t2));
A
BX
BX
S3
S2
Prohibit(t2):=Prohibit(t2) or (gck < val(gck) + d);
X== Prohib(t5):=Prohib(t5) or (gck < val(gck) + d);
Prohib(t6):=Prohib(t6) or(gck < val(gck) + d)

Figure 3: Prohibition Rule Integration : $\mathcal{F}$ (start(A) | $O^{<-d}$ done(B))

The prohibition integration methodology is described in pseudo-code in Algorithm 1, which performs a primary processing of the initial TEFSM specification, so that no transition would contain the prohibited action A after action B. Figure 2 illustrates this primary phase by decomposing the transitions where prohibited action A appears after action B. Transition $(t_4 =< S_2, S_0, \_, (B, A, C, B) >)$ for instance has been split into two transitions $(t_4 =< S_2, S_5, \_, B >)$ and $(t_6 =< S_5, S_0, \_, (A, C, B) >)$ by introducing the new state S_5. In the resulting specification, we want to integrate the rule $\mathcal{F}$ *(start(A)* | $O^{<-d}$ *done(B))* which stipulates that it is forbidden to perform action A within d units of time of B being performed. The application of algorithm 1 produces the secure system depicted in Figure 3.

4.2. Permission integration: $\mathcal{P}(start(A)|O^{[<]-d}done(B))$

The algorithm to integrate a permission rule is very similar to Algorithm 1. We define a variable $Permis(tr)$ for each transition that contains an action A appearing in a rule of the form $\mathcal{P}(start(A)|O^{[<]-d}done(B))$. $Permis(tr)$ stores all the instants where it is permitted to trigger tr; it is updated exactly as $Prohib(tr)$ each time action B occurs. Finally, we have to check that predicate $Permis(tr)$ is satisfied before triggering transition tr.

4.3. Obligation integration

Since different obligation rules related to action A may be defined, we have to take into account the possible dependencies that may exist between them. In fact let us consider for instances rules $\mathcal{O}$ *(start(A)* | O^{-5} *done(B))* and $\mathcal{O}$ *(start(A)* | $O^{<-10}$ *done(C))* where action C is executed 3 units of times (less than 5 units of times) after the execution of B. The execution of action A five units of time after the execution of B satisifies both rules at the same time. The idea behind this simple example is to check if it necessary to execute the mandatory rule for each execution of the context action. To integrate an obligation security rule in the TEFSM based system specification, we rely on a new process RHP which ensures the execution of the mandatory action. If the related mandatory action is not executed by the initial specification, process RHP then has the task to execute it itself. The integration methodology follows these steps for a rule in the form of $\mathcal{O}$ *(start(A)* | $O^{[<]-d}$ *done(B))* where $d > 0$:

- A boolean variable $wait_A$ is defined. It checks whether we are waiting for the execution of an instance of action A or not. This variable is set to *true* at the execution of each action B for which an obligation rule $\mathcal{O}$ *(start(A)* | $O^{[<]-d}$ *done(B))*, and set to *false* when action A is executed.
- The definition of a new process that can be created (forked) n times by the initial functional specification, where n is the maximum number of simultaneous executions of action B in the initial TEFSM specification. This new process has two parameters. The first parameter, equal to $(val(gck) + d)$ (resp. $(val(gck) + (d - 1)))^2$, states the instant when (resp. before which) action A should be executed if we deal with the O^{-d} timed operator (resp. $O^{<-d}$). The second parameter $exactTime$ is a boolean variable which determines whether action A must be executed at $(val(gck) + d)$ (for a rule of the form $\mathcal{O}$ *(start(A)* | O^{-d} *done(B))*) units of time or before this moment (for a rule of the form $\mathcal{O}$ *(start(A)* | $O^{<-d}$ *done(B))*). The process has to wait until the execution deadline of action A is reached. Also, we give it a lower priority with respect to the main process in order to let this last executes its actions at first. To do that, we declare the transitions of this new process as delayable. A delayable transition allows time progress unless time progress disables it, in that case it is taken. The deadline and the actions to perform $(deadline, Action)$ depend on the type of rule and

[2]Term 1 in $(val(gck) + (d - 1))$ denotes a unit of time which may be a second, a millisecond, etc.

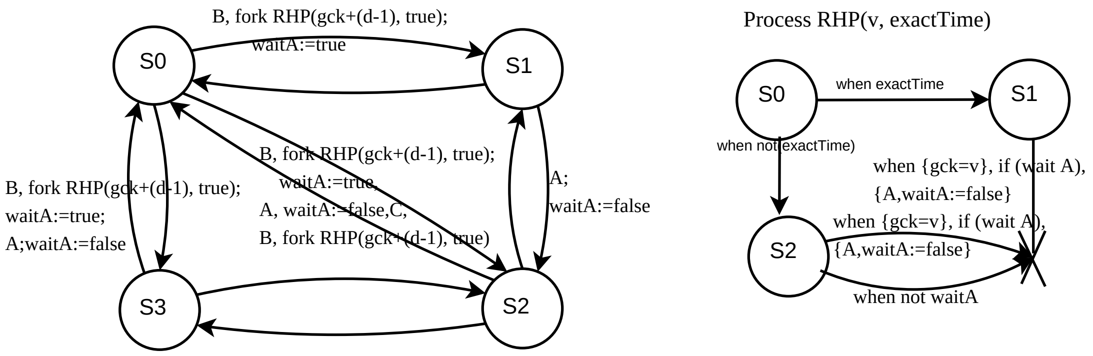

B, fork RHP(gck+(d-1), true);
waitA:=true
S0
S1
B, fork RHP(gck+(d-1), true);
waitA:=true,
A, waitA:=false,C,
B, fork RHP(gck+(d-1), true)
A;
waitA:=false
B, fork RHP(gck+(d-1), true);
waitA:=true;
A;waitA:=false
S3
S2
Process RHP(v, exactTime)
S0
when exactTime
S1
when not(exactTime)
when {gck=v}, if (wait A),
{A,waitA:=false}
S2
when {gck=v}, if (wait A),
{A,waitA:=false}
when not waitA

Figure 4: Obligation Rule Integration : $\mathcal{O}$ (start(A) | $O^{<-d}$ done(B)).

whether we are waiting for an execution of action A:

$$(deadline, Action) = \begin{cases} (gck = val(gck) + d, A; (wait_A := false); stop), \\ \qquad\qquad for\ \mathcal{O}(start(A)|O^{-d}done(B)) \\ ((gck = val(gck) + (d-1)) \vee \neg wait_A, if\ wait_A\ then\ (A; stop)), \\ \qquad\qquad for\ \mathcal{O}(start(A)|O^{<-d}done(B)) \end{cases}$$

The complete algorithm that permits an obligation rule of the form $\mathcal{O}$ *(start(A)* | O^{-d} *done(B))*[3] to be integrated is as follows:

[3]For an obligation rule of the form $\mathcal{O}$ *(start(A)* | $O^{<-d}$ *done(B))*, we have only to replace the second parameter of process RHP with the value *false*.

Algorithm 2 Obligations Integration

```
Require: The TEFSM model $M =< S, s_0, I, O, \vec{x}, \vec{c}, Tr >$ and the obligation security
    rule $\mathcal{O}$ (start(A) | $O^{<-d}$ done(B))
1: In the initial state of $M$, $wait_A := 0$
2: for each (transition $tr =< S_i, S_j, G, Act >$ such that $(tr \in Tr)$) do
3:     if $(B \in Act)$ then
4:         $tr := < S_i, S_j, G, (before(B); B; wait_A := true;$
                   $fork\ RHP(gck + (d - 1), true);\ after(B)) >$
5:     end if
6:     if $(A \in Act)$ then
7:         $tr := < S_i, S_j, G, (before(A); A; wait_A := false; after(A)) >$
8:     end if
9: end for
```

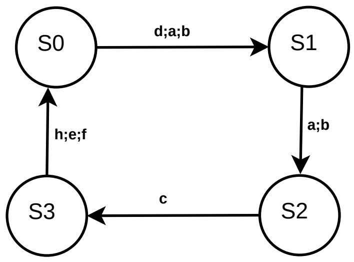
Process RHP used by Algorithm 2 is defined by algorithm 3. In Figure 4, we present the integration of an obligation rule within the initial system depicted in Figure 2. In this functional system, we can find several occurrences of the atomic action B.

Algorithm 3 The RHP process

```
1: for RHP process (v, exactTime) do
2:     Define two states $S_1$ and $S_2$ and five transitions $tr_1$, $tr_2$, $tr_3$, $tr_4$ and $tr_5$
3:     $tr_1 := < S_0, S_1, \{when\ exactTime\}, \_ >$
4:     $tr_2 := < S_0, S_2, \{when\ not\ exactTime\}, \_) >$
       /*We should execute action $A$ even if an other instance of this action has already
       been executed */
5:     $tr_3 := < S_1, \_, \{when\ gck = v\}$,if $(wait_A)$ $(wait_A := false; A; stop) >$
       /*We should execute action $A$ only if no instance has been executed before $(v =$
       $gck + d)$*/
6:     $tr_4 := < S_2, \_, \{when\ gck = v\}$, if $(wait_A)$ $\{wait_A := false; A; stop\} >$
7:     $tr_5 := < S_2, \_, \{when\ not(wait_A)\}, stop >$
8:     make transitions $tr_{i(i=1..5)}$ as delayable.
9: end for
```

5. Basic security rules with non-atomic actions

In this section, we consider the integration of security rules with non-atomic actions (see Definition 3). A non-atomic action A may contain several sequential (";") and parallel ("$\|$") operators. We only provide here integration algorithms that deal with security rules including the sequential operator. First, we define two different non-atomic action categories.



t1=<S0, S1,_,(d;a;b)>
t2=<S1,S2,_,(a;b)>
t3=<S2,S3,_,c>
t4=<S3,S0,_,(h;e;f)>

Figure 5: Examples of non-atomic actions: 1_Tr (a;b) and 2_Tr (a;b;c)

Definition 6. *(1_Tr actions) a non atomic action A is 1_Tr action with respect to a transition $tr =< s_i, s_f, G, Act >$ if and only if $A \subseteq Act$. That means that there exists sequences of actions X and Y such that ($Act = (X; A; Y)$). Both sequences X and Y may be empty.*

Definition 7. *(k_Tr actions) a non atomic action A is k_Tr action with respect to the ordered set of transitions $Tr = \{tr_1, \ldots, tr_k\}$ if and only if the execution of A needs the triggering of all the transitions of Tr in their order in Tr. More formally:*

1. *$\forall i.(1 \leq i \leq (k-1) \Rightarrow FS(tr_i) = IS(tr_{(i+1)}))$ where $IS(tr)$ (resp. $FS(tr)$) denotes the initial state (resp. Final state) of transition tr.*
2. *action A is a sub-action of $(Act(tr_1); \ldots; Act(tr_k))$ where $Act(tr)$ denotes the sequence of actions labeling transition tr. In other words, there are two sequences of actions X and Y such that $((Act(tr_1); \ldots; Act(tr_k)) = (X; A; Y))$.*

For instance in Figure 5, $(a; b)$ is a 1_Tr action with respect to transition from S_0 to S_1, whereas action $(a; b; c)$ is a 2_Tr action with respect to transitions $\{$from S_1 to S_2 then from S_2 to $S_3\}$.

In the sequel of the paper, the time of execution of non-atomic actions denotes the instant at which the execution of the last basic action is achieved.

5.1. Integration of Rules with 1_Tr Actions

Let us consider a 1_Tr non atomic action Act with respect to a transition tr, and a security rule of the form $R(A|O^{[<]-d}B)$. The integration of security

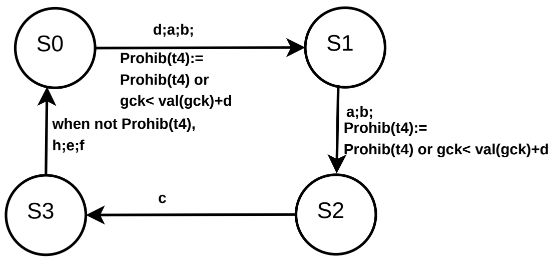


Figure 6: Prohibition Rule Integration with $1\_Tractions$: $\mathcal{F}(start(e;f)|O^{<-d}done(a;b))$

rules containing the *Act* in the transition *tr* is performed in a manner similar to that in the case of atomic actions since we are handling a unique transition. In other words, we can apply the algorithms we have defined in section 4 by distinguishing the two following cases:

- If *Act* is *B*:
  - For prohibition (resp. permission) rules, action $Update_{tr}(Prohib(tr))$ (resp. $Update_{tr}(Permis(tr))$) is added to the action of *tr* immediately after the execution of action *Act*.
  - For obligation rules, action $(wait_{Act} := true; fork(vak(gcd) + (d-1), v))$ is added to the action of *tr* immediately after the execution of action *Act*.
- If *Act* is *A* :
  - For prohibition (resp. permission) rules, guard $\{when(\neg Prohib(tr))\}$ (resp. $\{when(Permis(tr))\}$) is added is added to each transition *tr* immediately before the execution of action *Act*.
  - For obligation rules, action $(wait_{Act} := true; fork(gcd + d - 1, v))$ is added to the action of *tr* immediately after the execution of action *Act*.

Figure 6 illustrates the integration of the prohibition rule $\mathcal{F}(start(e;f)|O^{<-d} done(a;b))$ into the specification depicted in Figure 5.

5.2. Integration of Rules with n_Tr Actions

The handling of decomposable actions are inspired by [17]. Let *Act* be k_Tr action decomposable on the distinct transitions $Tr = (tr_1, \ldots, tr_k)$. According to these k transitions, we introduce the following notations:

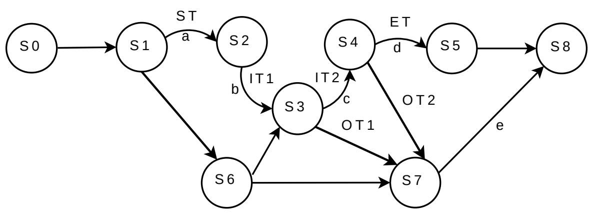


Figure 7: Example of 4_Tr decomposable action with its starting/intermediate/outgoing/ending transitions

- The starting transition ST: denotes the transition that the system has to follow to start the action: $ST = tr_1$.
- The ending transition ET: denotes the transition that the system has to follow to end the action: $ST = tr_k$.
- The intermediate transitions set ITS: includes the remaining transitions: $IT = \{tr_2, \dots, tr_{k-1}\}$ and may be empty.
- The outgoing transitions set OTS: includes the transitions that don't belong to Tr and whose initial states belong to those of transitions $(Tr - \{tr_1\})$. This set is formally defined as follows: $OTS = \{tr | \exists tr'.(tr' \in Tr - \{tr_1\} \wedge IS(tr) = IS(tr') \wedge tr \neq tr'\}$.

In Figure 7, action $(a; b; c; d)$ is a 4_Tr decomposable action. We have one starting transition (ST), two intermediate transitions ($IT1$ and $IT2$), one ending transitions (ET) and two outgoing transitions ($OT1$ and $OT2$).

This paper only discusses the integration of a prohibition security rule of the form $\mathcal{F}(start(A)|O^{<-d}done(B))$ with A and/or B denoting decomposable actions. The other cases can be deduced based on the same methodology. To integrate a security rule with decomposable actions we have to know whether the underlying system is executing any decomposable action. Let $(C = (c_1; \dots; c_n))$ be a decomposable action with respect to transitions $Tr = \{tr_1, \dots, tr_k\}$ $(k \leq n)$. A system is executing action C if and only if it is firing transitions Tr in the right sequence order. To record such a state, we define a variable v_C that is initialized to false and updated in the TEFSM as follows:

- The action $(v_C := true)$ is added to the starting transition ST.

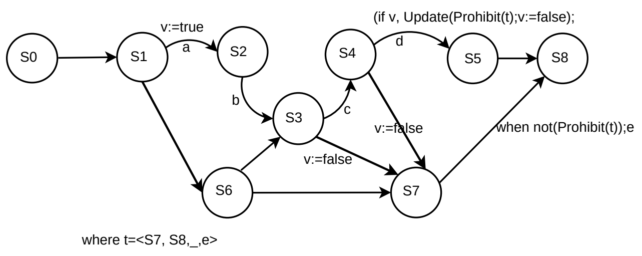


Figure 8: Prohibition Rule Integration with n_Tr actions: $\mathcal{F}(start(e)|O^{<-d}done(a;b;c;d))$

- The action $(v_C := false)$ is added to each outgoing transition $tr \in OTS$. In fact if the system fires a transition belonging to OTS, this means that the possible execution of action C is interrupted.

Variable v_C being defined for each decomposable action, the integration process proceeds as follows:

- if C is B: variable $Prohib(tr)$ is updated when the whole decomposable action is performed, that is, when the system is firing the ending transition ET and while v_C is true. So, we add action (*Update(Prohib(tr))*) by stating that this action is only performed when v_C is true.

  if C is A: to forbid the execution of action C in a given context, we chose to skip the last action(s) in the transition tr_k when a context condition holds. In our case for instance, we add the condition ((*when not* $Prohib(tr_k) \vee$ not(v_C)) to the ending transition ET. In this way, we state that this transition is only fired if it is not prohibited and we are executing the prohibited action. Also, we set variable v_C to false to express the end of the decomposable action.

Figure 8 shows the integration of security rule $\mathcal{F}(start(e)|O^{<-d}done(a;b;c;d))$. Variable v is added to check if the system is running the decomposable action $(a;b;c;d)$ or not.

Note that in our approach, sets $\{ST, ET, IT, ITS\}$ must be mutually disjoint such that each transition of an n_Tr non atomic action belongs to only one set. Indeed, transitions of each set are modified in a specific way. If this assumption is not satisfied (e.g. n_Tr non atomic actions containing cycles), another approach has to be defined to integrate the security rule. This constitutes one of the limitations of our work.

6. Correctness proof of the integration approach

To ensure the correctness of the proposed approach to integrating security rules, we have to prove all the algorithms presented in the previous sections. This paper only presents the correctness of algorithm 1 for rules of the form $\mathcal{F}$ *(start(A)* | $O^{<-d}$ *done(B))*. Other correctness proofs are similar and can be found in [18].

By proving the correctness of algorithm 1, we demonstrate precisely that the integration of prohibition rule of the form $\mathcal{F}$ *(start(A)* | $O^{[<]-d}$ *done(B))*, where $(d > 0)$, produces a secure TEFSM specification. To achieve this goal, we define for each occurrence of action B:

- t_B is the instant of the execution action B.
- k is the time elapsed since the last execution of action B.
- gck is a global clock of the system that gives the current time.

We have to prove that we cannot perform action A within d units of time after the execution of action B. Mathematically, we have to establish that for each positive integer k the following predicate holds:

$$((k < d) \wedge (val(gck) = t_B + k)) \Rightarrow \neg start(A) \qquad (1)$$

To prove that action A cannot be executed at the moment $(val(gck) = t_B+k)$, it is sufficient to prove that at this moment all the transitions Tr of the secure system labeled by action A cannot be traversed, that is, that all the guards of the transitions Tr are false. In the secure system, the algorithm 1 adds the guard $(when\ (\neg Prohib(tr)))$ before each execution of A. Thus, we have to prove, for each positive integer k, that:

$$((k < d) \wedge (val(gck) = t_B + k)) \Rightarrow Prohib(tr) \qquad (2)$$

Predicate $Prohib(tr)$ is a disjunction of predicates to which predicate $(gck < val(gck) + d)$ has been added when action B has been executed at instance t_B. The added predicate is equivalent to $(gck < t_B + d)$ (See algorithm 1). Consequently, it is sufficient to establish that:

$$((k < d) \wedge (val(gck) = t_B + k)) \Rightarrow val(gck) < t_B + d \qquad (3)$$

which is obviously true.

7. Formal testing of security rules

In [19], we presented a passive testing based monitoring[20] approach to check an implementation against a set of Nomad security rules. A passive testing approach consists in observing, during execution, whether the system behavior conforms to a set of formal properties. Following this approach, a user has no control of the collected traces which may be insignificant with respect to the properties to be checked. Such an approach is especially useful when the abstract formal specification of the implementation we would like to verify is not available. However, an active based approach becomes more interesting in the presence of such a formal specification from which significant test scenarios can be derived to check specific behaviors of the system. Indeed, these test scenarios are used to simulate the implementation and to observe how it behaves with respect to the properties to be checked. Our approach takes the following three elements as input:

- a TEFSM functional description of the system,
- a set of security rules described using Nomad language,
- the existing implementation of the system.

The objective is to check whether the existing implementation verifies the security rules. It proceeds in four steps as shown in Figure 9.

1. The security rules are integrated into the TEFSM specification according to the different algorithms we presented in the previous sections.
2. Abstract test cases are automatically generated from the secured TEFSM specification obtained in the first step. We use TestGenIF tool that implements a formal test case generation approach based on the Hit-or-Jump algorithm [21]. This tool accepts a TEFSM specification encoded in the IF textual formalism [22].
3. The abstract test cases are transformed into an executable script capable of communicating via http (or https) with the implementation under test.
4. The concrete test cases obtained from the instantiation are executed on the implementation under test to check whether it verifies the security rules.

In this paper, we focus on the two first steps that are illustrated through an industrial case study. More details about the others can be found in [23].

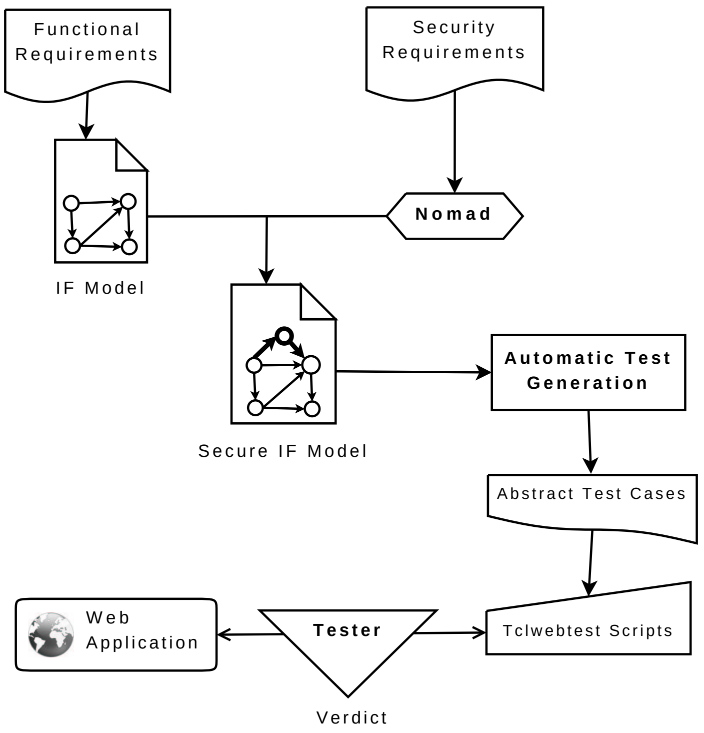


Figure 9: Testing Methodology Overview

7.1. Description of the case study: the France Telecom travel application

To demonstrate the feasibility of our approach, we applied it to different case studies, among them a real application used at France Telecom. We present the results obtained. This application manages 'missions' (business travel) of the company employees. To facilitate the presentation, this paper considers a simplified version of this real-size application where a potential traveler can connect to the system (using a dedicated URL) to request a travel ticket and a hotel reservation for a specific period according to some business purposes (called a mission). This request can be accepted or rejected by his/her hierarchical superior (called a validator). In the case of an acceptance, the travel ticket and hotel room are booked by contacting a travel agency.

The main functionalities of this system are represented by the TEFSM of Figure 10 where transitions $t_{i(i=0..9)}$ are defined as follows:

$t_0 =< S0, S0, \_, input\ req\_create\_mission(v\_recv, m\_recv), output\ error >$
$t_1 =< S0, S1, \_, input\ req\_create\_mission(v\_recv, m\_recv),$

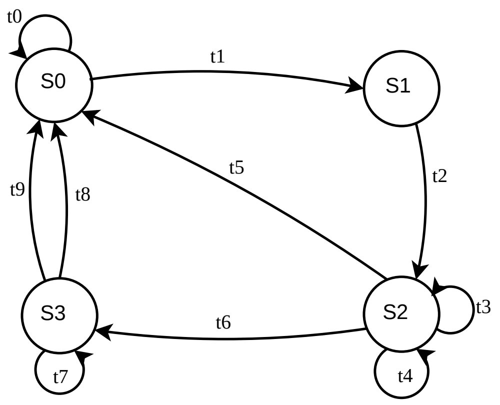


Figure 10: TEFSM specification of the case study

$$
\begin{aligned}
&\quad\quad\quad\quad output\ grant\_create\_mission(v\_recv, m\_recv) >\\
t_2 &=< S1, S2, \_, input\ req\_prop\_list(v\_recv, m\_recv),\\
&\quad\quad\quad\quad output\ recv\_prop\_list(v\_recv, m\_recv) >\\
t_3 &=< S2, S2, \_, input\ req\_prop\_list(v\_recv, m\_recv),\\
&\quad\quad\quad\quad output\ recv\_prop\_list(v\_recv, m\_recv) >\\
t_4 &=< S2, S2, \_, input\ req\_choice(v\_recv, m\_recv),\\
&\quad\quad\quad\quad output\ unvalid\_choice >\\
t_5 &=< S2, S2, \_, input\ req\_prop\_list(v\_recv, m\_recv), output\ empty\_list >\\
t_6 &=< S2, S3, \_, input\ req\_choice(v\_recv, m\_recv),\\
&\quad\quad\quad\quad output\ grant\_choice(v\_recv, m\_recv) >\\
t_7 &=< S3, S3, \_, input\ req\_validation(v\_recv, p\_recv) >\\
t_8 &=< S3, S0, \_, input\ send\_unvalidate\_notif(v\_recv, m\_recv),\\
&\quad\quad\quad\quad output\ recv\_validate\_notif(v\_recv, m\_recv) >\\
t_9 &=< S3, S0, \_, input\ send\_validate\_notif(v\_recv, m\_recv),\\
&\quad\quad\quad\quad output\ recv\_validate\_notif(v\_recv, m\_recv) >
\end{aligned}
$$

In the initial state $S0$, the system receives an input message *req_create_mission* from a given user to create a new mission. Depending on specific conditions, the system may reject (in the case of an internal *error*) or accept (*grant_create_mission*) the creation of the requested mission. In the second case, the user can ask for a list of possibilities (different departure dates for instance) for his/her mission (*req_prop_list*). After receiving the first proposal from the system (*recv_prop_list*), the user can request other proposals if those given are not suitable for him/her (loop on state S2 with message (*req_prop_list*)). If there is no possible proposal (*empty_propal_list*), the system moves into its initial state. After selecting the desired proposal (*req_choice*) and depending on the current conditions, the choice is either rejected (*unvalid_choice*) or accepted (*grant_choice*). If accepted, the system has to ask for the validation of the hierarchical person in charge. Finally, the system informs the user about the validation decision and returns to the initial state.

France Telecom proposed a preliminary version of the case study *Travel*, in which some informal security requirements are provided. Based on these requirements, we formally specified a set of 34 security rules using Nomad. In this paper, we only present three of them:

- Rule 1:

$$\begin{gathered}\mathcal{F}\ (start\ (output\ req\_create\_mission(t))| \\ O^{<-2min}\ done\ (output\ req\_create\_mission(t)))\end{gathered}$$

This first prohibition rule states that two missions requests by the same traveler must be separated by at least 2 minutes. This request can be performed in the *basic_traveler* process.

- Rule 2:

$$\begin{gathered}\mathcal{P}\ (start\ (output\ req\_proposition\_list(t,m))| \\ O^{<-10min}\ done\ (output\ req\_proposition\_list(t,m)))\end{gathered}$$

This permission rule expresses that a traveler can request for another list of travel propositions within a delay of 10 minutes if he/she has already asked for a first list of travel propositions. This request can be performed in the *traveler_mission* process.

- Rule 3:

$$\begin{gathered}\mathcal{O}\ (start\ (output\ req\_validation())| \\ O^{-10080min}\ done\ (output\ req\_validation()) \wedge \\ \neg\ O^{<-10080min}\ (done\ (input\ recv\_validate\text{-}\ \_notification()) \\ \vee\ done\ (input\ recv\_unvalidate\_notification())))\end{gathered}$$

This obligation rule states that if a traveler requested validation of his/her mission and if he/she did not receive an answer, the system must send, as a reminder, another request to the potential mission validator. This reminder is sent within a delay of (10080 min = 7 days). The requests and answers are made in the *travel_mission* process.

7.2. Security rules integration

To take the security requirements presented in the previous section into account, the TEFSM specification depicted in Figure 10 was augmented by the Nomad rules that model them. This is achieved by applying the different algorithms provided in this article which were implemented using C language. Table 1 shows some measurements concerning the modified and

Table 1: IF Travel System Modifications According to Each Rule

Rule	**M&A Transitions**	**Added Var**	**Added Proc**
1	1+1	1	0
2	2+1	1	0
3	4+3	1	1

added transitions (M&A Transitions), the added variables (Added Var), the added processes (Added Proc).

As we can see in this case study but also from the various experiments we have achieved, the modifications inserted in the formal specification (modified and added transitions) are about 20% of the initial specification. Thus, we can conclude that the integration of security rules does not generate any explosion of number of states/transitions which demonstrates its scalability.

7.3. Test Generation

To generate test cases automatically from the secure specification of Travel, we use TestGen-IF [13] a test generation tool. This tool implements a timed test generation algorithm based on a Hit-or-Jump exploration strategy [21]. This algorithm constructs test sequences efficiently with high fault coverage, avoiding state explosion and deadlock problems encountered respectively in exhaustive or exclusively random searches. It allows a set of test scenarios to be produced according to a set of test purposes. A security test purpose is a necessary property to be checked in the system studied. It can be derived from the set of security rules that the system has to respect. The automatic test generation only targets security issues and, as a result, it is less time consuming. No performance or stress tests are generated (out of scope). Our main objective is to covers all possible scenarios. To reach this objective we used, for Travel test generation, two users and two missions:

- the user can be either a traveler or a validator
- the mission can be either "created and not validated yet" or "created and validated".

The generation of test cases for only two users and two missions is thus sound and valid and covers all relevant scenarios. We also defined adequate interval values for data variables in order to reduce the accessibility graph size and avoid state explosion problems.

Table 2: Some Test Generation Metrics

Rule	**Strategy**	**Maxdepth**	**Jumps**	**Test Case Length**	**Visited States**	**Duration**
1	BFS	10	0	9	291	0.2s
2	BFS	10	1	16	7844	10s
3	BFS	10	2	23	26552	1m25s

A set of timed test cases are generated based on the IF specification of Travel Web application and the timed test purposes for each rule, using TestGen-IF. These test cases are then filtered according to the observable actions (input, output and delays) relating to the system under test. Some measurements concerning this test generation relating to three rules are presented in Table 2. This table shows the characteristics of TestGen-IF according to the main standard criteria used in the testing field to compare test generation tools. Table 2 indicates that our test generation tool TestGen-IF has good performance according to these criteria. The choice of maximum depth should be as small as possible without having a big number of jumps which is the case.
For instance, the filtered timed test case for rule 3 is presented in Figure 11 where "?" and "!" are standard notations for respectively input and output messages, and "/" a separator. Each message has some parameters defined in the formal specification of the studied formal (described in IF language). For instance, message $request\_connect(0, 0)$ means that the user with ($id = 0$) and password ($pwd = 0$) wants to connect to system. Notice that the input/output signals described in each test case are relative to the system under test. In our case it is the Travel system designed by the two processes *basic_travel* and *travel_mission*. The test cases generated by the TestGen-IF tool are abstract but usable; they are produced in aldebaran standard notation facilitating their portability and their automatic execution.

7.4. Test Cases Instantiation and Execution

In order to execute the generated test cases to a real Web application, they need to be transformed into an executable script capable of communicating via http (or https) with the implementation under test. In this work, we translated the abstract test case automatically into the tcl script used by the tclwebtest tool[4] to execute the designed tests.

[4]TCLWEBTEST Tool, http://tclwebtest.sourceforge.net

```
 1. ?give_traveler_id{0} / !req_connect{0,0}
 2. ?req_connect{0,0} / !grant_connect{0,0}
 3. ?grant_connect{0,0}
 4. !req_create_mission{0}
 5. ?req_create_mission{0}
 6. !grant_create_mission{{0,0,0,{traveler_mission}0,{travel_mission}0}}
 7. ?grant_create_mission{{0,0,0,{traveler_mission}0,{travel_mission}0}}
 8. !req_proposition_list{0,{0,0,0,{traveler_mission}0,{travel_mission}0}}
 9. ?req_proposition_list{0,{0,0,0,{traveler_mission}0,{travel_mission}0}}
10. ?give_choice_list{{{0},1}} /
    !recv_proposition_list{0,{0,0,0,{traveler_mission}0,{travel_mission}0},{0},1}
11. ?recv_proposition_list{0,{0,0,0,{traveler_mission}0,{travel_mission}0},{0},1}
12. !req_choice{0,{0,0,0,{traveler_mission}0,{travel_mission}0},0}
13. ?req_choice{0,{0,0,0,{traveler_mission}0,{travel_mission}0},0}
14. !grant_choice{{0,0,0,{traveler_mission}0,{travel_mission}0},0}
    !req_validation{{0,0,0,{traveler_mission}0,{travel_mission}0},0}
15. ?grant_choice{{0,0,0,{traveler_mission}0,{travel_mission}0},0}
16. delay = 10080
17. !req_validation{{0,0,0,{traveler_mission}0,{travel_mission}0},0}
```

Figure 11: Test Case for the Rule 3

The test cases execution was performed on a prototype implementation of the Travel Web application (developed on the OpenACS platform) to verify that the specified security requirements are respected. It is important to highlight that some time settings in this prototype were changed so that the application of the tests where faster than in the real system. For example, we changed 10080 minutes (7 days) in the third rule to 3 minutes to reduce waiting time. Therefore in this case study we verify the behavior of the system concerning this rule using a delay of 3 minutes rather than 7 days.

The execution of the test cases is performed using a dedicated testing tool proposed by the OpenACS community [5]. This tool is called the ACS-Automated-Testing tool that allows the instantiated test cases to be executed, interacting with the Web-based application under test and, also, displaying the verdict of each test case. The ACS-Automated-Testing tool is, in itself, a Web application but we will refer to it just as *the tester* to avoid any confusion between this system and the Web application to be tested.

As a result of the execution of the designed test cases on the prototype, we

[5]OpenACS Community, http://www.openacs.or

obtained positive verdicts for thirty test objectives, while, four test objectives were violated (fail verdict). For example, a problem was detected with the first rule, which expresses a prohibition. If a traveler requests a first mission and then waits for 2 minutes, he/she is not allowed by the system to request another mission. We analyzed the implementation of the Web-based system and noticed a mistake in the code. Instead of 2 minutes, the Travel system waited much longer before allowing a second mission request.

The Travel application was analyzed to detect the source of the four errors. Once the mistakes were corrected, all the test cases were applied again to the Web application. This time, all the verdicts were positive which demonstrates the efficiency of our methodology.

7.5. Results analysis

Several criteria can be defined to characterize the quality of the proposed integration approach described in the paper. According the CSE (Scientific Council of Evaluation), quality assessment can be summed up in five criteria: Utility-relevance, reliability, objectivity, scalability (possibility of generalization) and transparency. We applied these criteria to our methodology in the context of an experiment dealing with a real life case study provided by France Telecom. This case study is an internal Web application used by the company employees to manage their business travel (transport, accommodation, etc.). The evaluation of our approach according to the defined criteria is as follows:

1. *Utility-relevance:* The integration methodology can be applied in several domains and for different purposes. The chosen experiment was intended to use the derived formal specification including both functional and security requirement as a starting point to automatically generate test cases in order to check the security of the studied Web application. The automatic generation was based on classical model-based testing techniques. The experimentation showed that our methodology dealing with the integration of security rules within a TEFSM specification, allowed a correct and complete specification to be derived. This specification can be used without any modification to generate test cases targeting functional but also security objectives. The generation using a dedicated tool TestGen-IF was possible and the test cases produced were good enough to be able to detect defects in the Web application studied. Based on these results, we can conclude that

our approach is useful and relevant and has at least one possible application, which is, testing. Using our method, generating test cases for security checking becomes possible, and we are even able to generate timed test cases based on the time constraints that can be defined within Nomad security rules. We believe that the integration approach has other applications like model checking or code generation. Further experimentations are planned to confirm our theory.

2. *Reliability:* The different proposed algorithms for the integration of security rules within a TEFSM based specification were proved correct (see section 7). These mathematical proofs demonstrate that the result of the integration is a formal specification of the system studied that takes into account the designed security rules. Thus, the test cases derived from this specification (in this case, we used the specification for a testing purpose) are obviously secure and do not violate any security rule.
3. *Objectivity:* This criteria means that the evaluation results are not influenced by personal preferences or institutional positions of evaluation responsibility or at least that these preferences were adequately explained or controlled so that we can assume that another evaluation answering the same questions and using the same methods leads to similar conclusions. This criterion was particularly respected for our experiment since we mainly relied on quantitative results (number of security rules, number of generated test cases, number of detected defects etc.). This demonstrates the fairness of the work accomplished.
4. *Scalability:* The integration methodology has been applied to a real life case study provided by a known telecommunication operator: France Telecom. The size of the application studied is big enough and we believe that our methodology is scalable since the modifications introduced to the functional specification in order to derive a secure specification are minor (almost 20% of the size of the specification). The integration methodology may be extended to other situations and contexts, even under different security policies.
5. *Transparency:* In addition to the requirement of a complete and rigorous methodology, this criterion includes the idea that the experiment must be clear enough to be able to determine its steps and limitations. This was the case in our testing methodology where we clearly defined the different experiment inputs and outcomes of each stage. We also provided quantitative results to be able to express the results and the

limitations of the methodology.

8. Conclusion

In this paper, we have presented a formal approach to integrating timed security rules, expressed according to Nomad, into a TEFSM specification of a system. Roughly speaking, a security rule denotes the prohibition, obligation or permission for the system to perform an action in a specific timed context. To meet this objective, we have described a set of algorithms that allows them to be added to a TEFSM specification describing the behavior aspect of a system. A proof that demonstrates the correctness of the prohibition integration algorithms is given. These algorithms are implemented in a tool and the methodology was applied to several real-size applications that gave very promising results. Finally, the complexity of the algorithms presented in this paper are linear (in $O(n)$) with n denoting the number of rules in the policy. In practical experience, the integration of security rules into a system does not produce important modifications since in general the number of rules is not huge. In most transitions, only some changes in predicates are applied.

The integration of security rules into a TEFSM functional specification of a system is not an end in itself. In this paper, we have shown how the secured specification obtained is used to check the correctness of an implementation with respect to a set of security rules thanks to a testing technique. Doing so, we discovered errors in the implementation and corrected them.

Currently, we are working of the definition of integration rules to deal with elaborated and general rules that may involve several timed/logical operators. Our aim is to define rewritten rules that allow decomposing such rules into more simple rules on which it becomes possible to apply the algorithms presented here. For instance, it is possible apply the proposed algorithm on rule $\mathcal{F}(A|C_1 \vee C_2)$ by rewriting it $\mathcal{F}(A|C_1 \vee C_2)$ into $\{\mathcal{F}(A|C_1), \mathcal{F}(A|C_2)\}$.

9. Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Program (FP 7/2007-203) under the grant agreement number 215995 (http://www.shields-project.eu/).

References

[1] M. Bozga, S. Graf, I. Ober, I. Ober, J. Sifakis, The IF Toolset, in: M. Bernardo, F. Corradini (Eds.), Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems(SFM-RT), Vol. 3185 of Lecture Notes in Computer Science, Springer, 2004, pp. 237–267.

[2] J. Lobo, R. Bhatia, S. A. Naqvi, A Policy Description Language, in: AAAI '99/IAAI '99: Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence, American Association for Artificial Intelligence, 1999, pp. 291–298.

[3] N. Damianou, N. Dulay, E. Lupu, M. Sloman, Ponder: An object-oriented language for specifying security and management policies, in: 10th Workshop for PhD Students in Object-Oriented Systems (PhDOOS), 2000.

[4] J. Barkley, K. Beznosov, J. Uppal, Supporting relationships in access control using role based access control, in: ACM Workshop on Role-Based Access Control, 1999, pp. 55–65.

[5] R. Ferrini, E. Bertino, Supporting RBAC with XACML+OWL.

[6] F. Cuppens, N. Cuppens-Boulahia, T. Sans, Nomad: A Security Model with Non Atomic Actions and Deadlines, in: 18th IEEE Computer Security Foundations Workshop(CSFW), IEEE Computer Society, 2005, pp. 186–196.

[7] W. Mallouli, J.-M. Orset, A. Cavalli, N. Cuppens, F. Cuppens, A Formal Approach for Testing Security Rules, in: V. Lotz, B. Thuraisingham (Eds.), 12th ACM Symposium on Access Control Models and Technologies,(SACMAT), 2007, pp. 127–132.

[8] N. Benaïssa, D. Cansell, D. Méry, Integration of Security Policy into System Modeling, in: J. Julliand, O. Kouchnarenko (Eds.), Formal Specification and Development in B, 7th International Conference of B Users

(B 2007), Vol. 4355 of Lecture Notes in Computer Science, Springer, 2007, pp. 232–247.

[9] A. Abou El Kalam, R. E. Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, G. Trouessin, Organization Based Access Control, in: International Workshop on Policies for Distributed Systems and Networks (Policy), IEEE Computer Society, 2003, p. 120.

[10] K. Li, L. Mounier, R. Groz, Test Generation from Security Policies Specified in Or-BAC, in: 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), IEEE Computer Society, 2007, pp. 255–260.

[11] D. Lee, M. Yannakakis, Principles and Methods of Testing Finite State Machines - A Survey, in: Proceedings of the IEEE, Vol. 84, 1996, pp. 1090–1126.

[12] W. Mallouli, A. Mammar, A. Cavalli, Modeling system security rules with time constraints using timed extended finite state machines, in: D. Roberts, A. El-Saddik, A. Ferscha (Eds.), 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications (DS-RT), IEEE Computer Society, 2008, pp. 173–180.

[13] A.Cavalli, E. M. de Oca, W. Mallouli, M. Lallali, Two complementary tools for the formal testing of distributed systems with time constraints, in: D. Roberts, A. El-Saddik, A. Ferscha (Eds.), 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications (DS-RT), IEEE Computer Society, 2008, pp. 315–318.

[14] W. Mallouli, M. et Amel Amel Mammar, A. R. Cavalli, A formal framework to integrate timed security rules within une spcification de systme base tefsm, in: APSEC, IEEE Computer Society, 2009, pp. 489–496.

[15] M. Bozga, S. Graf, L. Mounier, I. Ober, IF Validation Environment Tutorial, in: S. Graf, L. Mounier (Eds.), Model Checking Software, 11th International SPIN Workshop,(SPIN), Vol. 2989 of Lecture Notes in Computer Science, Springer, 2004, pp. 306–307.

[16] L. Cholvy, F. Cuppens, Analyzing consistency of security policies, in: IEEE Symposium on Security and Privacy, IEEE Computer Society, 1997, pp. 103–112.

[17] W. Mallouli, A. Cavalli, Testing Security Rules with Decomposable Activities, in: Tenth IEEE International Symposium on High Assurance Systems Engineering (HASE 2007), IEEE Computer Society, 2007, pp. 149–155.

[18] W. Mallouli, A. Mammar, A. Cavalli, Integration of Timed Security Policies within a TEFSM Specification, Technical Report TI-PU-08-868, Telecom SudParis (2008).

[19] W. Mallouli, F. Bessayah, A. Cavalli, A. Benameur, Security rules specification and analysis based on passive testing, in: the Global Communications Conference (GLOBECOM), IEEE, 2008, pp. 2078–2083.

[20] E. Bayse, A. Cavalli, M. Núñez, F. Zaïdi, A passive testing approach based on invariants: application to the wap, Computer Networks 48 (2) (2005) 235–245.

[21] A. Cavalli, D. Lee, C. Rinderknecht, F. Zaïdi, Hit-or-jump: An algorithm for embedded testing with applications to in services, in: J. Wu, S. Chanson, Q. Gao (Eds.), Formal Methods for Protocol Engineering and Distributed Systems(FORTE), Vol. 156 of IFIP Conference Proceedings, Kluwer, 1999, pp. 41–56.

[22] M. Bozga, J. Fernandez, L. Ghirvu, S. Graf, J. Krimm, L. Mounier, J. Sifakis, IF: An intermediate representation for SDL and its applications, in: Proceedings of SDL Forum, Elsevier, 1999.

[23] W. Mallouli, A Formal Approach for Testing Security Policies, Ph.D. thesis, Telecom and Management SudParis Evry-France (December 2008).