Contents

Chapter 1

Modeling and Testing Secure Web Applications

1.1 Introduction

In modern networks, the heterogeneity and the increasing distribution of applications, such as telecommunication protocols, Web-based systems and real-time systems, make security management complex. These applications are more and more open and rely on networking parts of computer systems that generally make use of different solutions. In the context of the deployment of such applications and services, the security officials are led to empirically bring security solutions together. The consistency of these assemblies is difficult to achieve. Nowadays, many security features are available. We can cite for instance cryptographic protocols, management infrastructures of public keys (PKI), firewalls, control access mechanisms within operating systems and applications, intrusion detection systems or anti-viral mechanisms etc.

To ensure that these different security components are effective and that a certain level of security is always maintained, the system behavior must be restrained by a security policy. A security policy is a set of rules that regulates the nature and the context of actions that can be performed within a system, according to specific roles. As an example, such policy can tackle the interactions between a network infrastructure and Internet or manage accounts and rights toward an operating system or a database. The main objective is to ensure that security policy is well defined and that is actually implemented in the system.

To reach this aim, we usually carry out audits that focus on administrative procedures and systems configurations. Tests are then carried out to check if some known vulnerabilities would remain present. If several tools for some specific tests (such as passwords crackers) exist, there is no general solution analyzing the overall system conformance according to its security policy. Several reasons can explain these deficiencies. First, there is currently few research work on formal modeling of complete security policies, even if some aspects, such as access control security rules, have been studied further. In addition, analytical work about security checking often focuses on the verification of punctual elements, such as cryptographic protocols or code analysis. Thus, the responsible for security and all the system administrators are missing a formal solution to ensure the coherence of a system implementation with respect to its security policy, even if this last has been fairly well defined.

Most current work only concentrates on defining meta-languages in order to clearly express security policies and provide unambiguous rules. Or-BAC [Abou El Kalam *et al.* (2003)], PDL [Lobo *et al.* (1999)], Nomad [Cuppens *et al.* (2005)] and Ponder [Damianou *et al.* (2001)] are typical examples of such generic policy description models. They suggest concepts to describe the security policy independently of the system functional specification or implementation. Once the security policy is formally specified, it is essential to prove that the target system implements this policy. Indeed, if one cannot ensure this conformance, the global security cannot be guaranteed anymore.

Many solutions can be proposed to achieve this objective (the implementation conformance with respect to its security requirements). The conformance guarantee can be reached for instance by:

- formally injecting the security policy in the considered system code,
- or by formally specifying the target system to prove that it verifies the security policy it has to respect,
- or by considering several strategies of formal tests.

This last methodology will be explored in this chapter in the context of Web-based applications. Indeed, in recent years Web-based systems have become extremely popular and, nowadays, they are used in critical environments such as financial, medical, and military systems. As the use of Web applications for security-critical services has increased, the number and sophistication of attacks against these applications have grown as well.

For this reason it is essential to be able to prove that the target Web-based system implements the security requirements it should respect (described using a security policy language). Model-based testing consists in deriving a suite of test cases from a model representing the Web system behavior. Such a model can be generated from an informal specification of the system and designed by software engineers through the use of diagram manipulation tools. Moreover, Web applications can have a time dependent behavior as well as an increasing demand for security mainly due to their increased complexity and inherent distribution. Consequently, engineers developing these Web systems are not only confronted to functional requirements but also have to manage other kinds of requirements concerning security issues. Roughly speaking, by “functional requirements” we mean the services that a Web application has to offer to end users. Whereas, security rules denote the properties that a system has to fulfil so that it is always in a safe state and guarantees service quality.

To tackle this problem we rely, in this chapter, on a formal approach to integrate elaborated security rules involving time constraints into a formal specification of the system based on communicating extended timed automata [Bozga *et al.* (2004b)] supported by the IF (Intermediate Format) language [Bozga *et al.* (2002)]. The derivation of the test cases can be done automatically, providing generic test cases described in a standard language. By executing the model-based test cases, the conformance of the implemented system to its specification can be validated. More precisely, the main contributions of this chapter are:

- The specification of Web application features using the IF language. This language is well-adapted to formally describe Web systems features such as hyperlinks, sending and receiving data and client-server communications, etc.
- The definition of Web system security requirements based on the Nomad [Cuppens *et al.* (2005)] (stands for Non Atomic Actions and Deadlines Model) formal language. Nomad allows specifying, unambiguously, security rules (such as permissions, prohibitions and obligations) in specific contexts that include time constraints.
- The integration of the security rules within the functional IF model to obtain a secure specification that takes into account the security requirements.
- The automatic generation of test cases targeting security constraints. This generation is performed using TestGen-IF tool based on Hit-or-Jump algorithm [Cavalli *et al.* (1999)].
- These test cases are instantiated into TCL script language [TCL Script Language (2009)] and are applied, in an automated manner, on a real Web-based system (Travel Web application provided by France Telecom Company[1]) to check whether its behavior respects the security requirements. The application and the analysis of the designed test cases are performed by tclwebtest tool [TclWebTest Tool (2009)].

This chapter is organized as follows. In section 1.2 we discuss the related work on formal modeling and testing of secure systems with timed constraints. Section 1.3 exposes an overview of our methodology to specify and test the security of Web-based systems. In section 1.4, we apply our methodology to an industrial case study (called Travel) and formally specify its functional behavior. Section 1.5 presents our security integration methodology as well as its corresponding tool. In section 1.6 and 1.7, we automatically generate security-target test cases using TestGen-IF tool and perform them using tclwebtest tool. Finally, section 1.8 presents the conclusion and introduces future work.

1.2 Related Work

Many models are proposed in the literature for the formal specification of Web applications from their functional point of view. These models sometimes include functional time constraints. In [Syriani and Mansour (2003)] for instance, the authors present a methodology that specifies Web-based systems using the SDL formal language [Gaudin *et al.* (2007)]. This language is based on Extended Finite State Machines (EFSM) model [Lee and Yannakakis (1996)] and is well adapted for describing communicating systems.

Others studies are based on timed automata theory [Alur and Dill (1994)] and allows specifying functional system requirements with timed pre and post conditions. This paper is based on the IF language [Bozga *et al.* (2004b)] because it allows providing the main concepts to design Web-based systems with time constraints. Moreover, several tools allowing the simulation and the generation of test sequences exist and are open source. However a main issue here to guarantee the system reliability, is that this functional specification has

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

to be completed by integrating system security aspects.

To tackle this problem we have introduced in an earlier publication [Mallouli *et al.* (2007)] a formal approach that permits to augment a functional description of a system with security rules expressed with the Or-BAC language [Abou El Kalam *et al.* (2003)] (Organisationnal Based Access Control). We described security rules that specify the obligation, permission or prohibition for a user to perform some actions under given conditions called *context*. This context does not involve time aspects since we only specified rules without considering them. We applied this approach to a Weblog case study in order to validate its security and the results were satisfactory.

To be able to include timed security rules (for instance, a file system may have to specify the prohibition for a user to access to a specific document if he/she is not authenticated or if his/her session of 10 minutes has been expired), we propose in this paper to rely on the Nomad language [Cuppens *et al.* (2005)] that supports time constraints. We integrate the defined Nomad security rules that the system should respect, with its IF formal specification through the use of specific algorithms described in section 1.5. The obtained system specification is called a secure system specification.

Many tools [Jard and Jéron (2005); Vieira and Cavalli (2007)] allowing automatic test generation [Merayo *et al.* (2007)] from IF specifications exist in the literature. In this paper, we rely on our own generation tool, called TestGen-IF, that efficiently constructs tests sequences with high fault coverage. The tool avoids the state explosion and deadlock problems encountered in exhaustive or exclusively random searches used in classical tools. The execution of the generated tests are performed using tclwebtest tool [TclWebTest Tool (2009)], well adapted to check the stability and scalability of Web applications.

1.3 Testing Methodology Overview

In this framework, we present the proposed testing methodology to test security rules. This methodology manipulates three different inputs:

- A functional specification of the Web application based on the IF formal language.
- A specification of the security policy that the application has to respect (based on the Nomad language).
- And finally an implementation of the Web-based system.

The aim of this framework is to generate a new specification of the Web-based system (called secure specification) that takes into account the security policy, and then to generate a complete test suite to check whether the implementation of the system conforms to this secure functional specification.

To reach this goal we automatically integrate different types of security policy rules described using the Nomad language within the initial functional system specification. Then, we define an end-to-end methodology for testing Web applications as presented in Figure 1.1. The main steps of our testing methodology are the *specification* of a secure system based on the IF language, the *automatic test generation* based on TestGen-IF tool (gener-

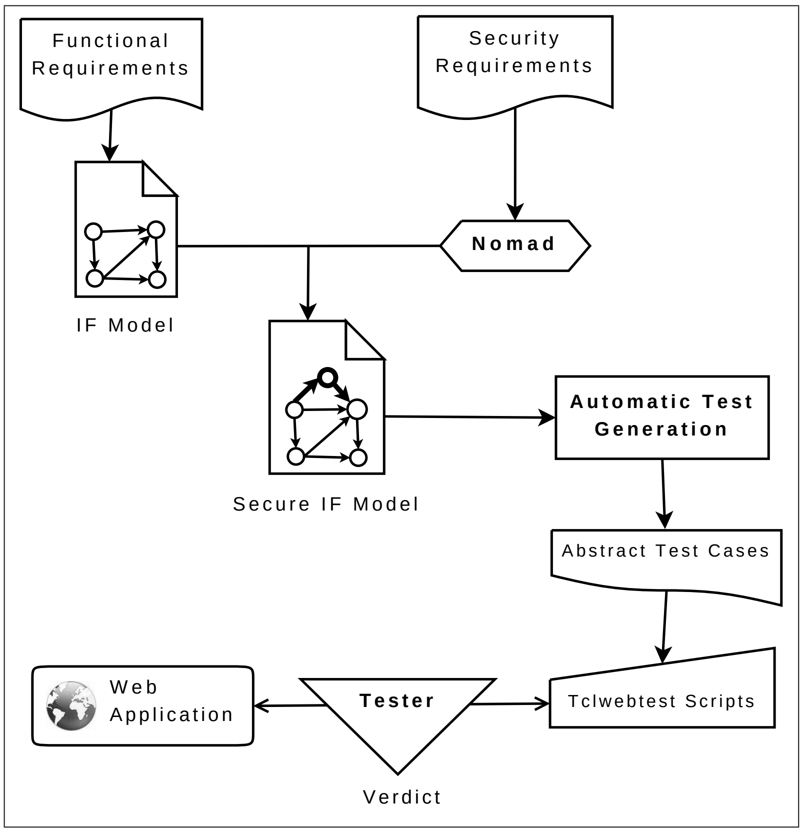

Fig. 1.1 Testing Methodology Overview

ated test cases are described in aldebaran standard notation [Fernandez *et al.* (1996)]), their instantiation to *executable test cases* and the *application* of these test cases on a real Web application using tclwebtest language. The analysis of the Web application is performed using a dedicated tool called ACS-Automated-Testing incorporated in the OpenACS platform [OpenACS Community (2009)].

1.4 Functional Specification of Web Applications using IF Language

1.4.1 *Modeling Communicating Systems*

The objective of modeling a Web-based system is to provide an operational specification of a system from the functional point of view which can include time constraints. In particular, it helps to provide a better common understanding of the system. In addition, this operational model can also be used as input to existing validation tools, such as interactive or random simulators, model-checkers or (conformance) test generation engines.

To achieve the modeling goal, we rely in this approach on TEFSM model supported by IF language [Bozga *et al.* (2004b)] because it provides the main concepts to design real-time systems. Moreover, several tools allowing its simulation and the generation of test sequences exist and are open source. A TEFSM modeling of a system consists of a set of processes, each one denotes a TEFSM that can communicate with other processes via FIFO channels.

Definition 1.1. *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.*

A 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 transition guard which is composed of predicates (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 action(s) associated with this transition occur simultaneously and take no time to complete (which is not the case of the model presented in [Merayo *et al.* (2008)]). The time progress takes place in some states before executing the selected transitions. More details about time progress can be found in [Bozga *et al.* (2004b,a, 1999)].

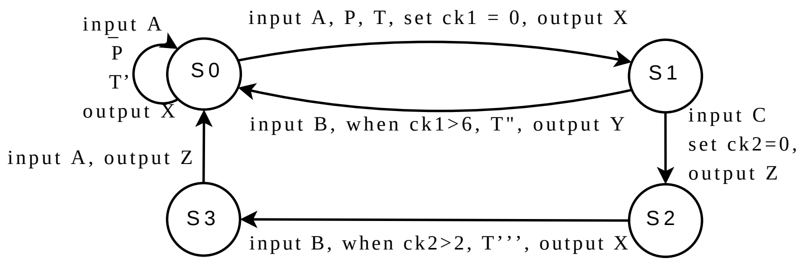
input A
$\overline{P}$
T'
output X
S0
input A, P, T, set ck1 = 0, output X
S1
input B, when ck1>6, T", output Y
input C
set ck2=0,
output Z
input A, output Z
S3
S2
input B, when ck2>2, T''', output X

Fig. 1.2 Example of a Simple TEFSM with Four States.

We illustrate the notion of TEFSM through a simple example described in Figure 1.2. This TEFSM is composed of four states (S_0, S_1, S_2 and S_3) and six transitions that are labeled with three inputs A, B and C, three outputs X, Y and Z, one guard (or predicate) P on variables, two clocks ck_1 and ck_2 and four tasks T, T', T'' and T'''.

The TEFSM operates as follows: starting from state S_0, when the input A occurs, the predicate P is checked. If the condition holds, the machine performs the task T, starts the clock ck_1, triggers the 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 ck_1 exceeds the value 6 and receives the input B. If so, task T'' is performed and output Y is triggered. On the reception of the input C, the clock ck_2 is started, the output Z is triggered and the machine moves to state S_2. Once the machine is in state S_2, it can go to state S_3 when the clock ck_2 exceeds the value 2 and receives the input B. If so, task T''' is performed and output X is triggered. In State S_3, on the reception of the input A, the machine triggers the output Z and comes back to the initial state S_0.

In the following sections, if $tr = < s_i, s_f, G, Act >$ and $a \in Act$, then we can denote Act by $(before(a);\ a;\ after(a))$ to express that action a is performed within the transition tr and that there is possibly other actions before or after a ($before(a)$ and $after(a)$ may be empty).

1.4.2 *IF Formal Language*

The Intermediate Format (IF) language can be considered as a common representation model for other existing languages. It was originally developed to sit between languages as SDL, Promela [Gallardo *et al.* (2004)] or Lotos [ISO (1989)]. It has been extended to deal with UML notation as well [Cavarra *et al.* (2002)]. IF is based on communicating timed automata TEFSM, and it is used to describe and validate asynchronous systems.

In IF, a system is a set of processes communicating asynchronously through a set of buffers. Each process is an TEFSM that models the behavior of a given component. A process can send and receive messages to and from any buffer.

The semantic of time is similar to the one of communicating extended timed automata. That is:

- A time behavior of a system can be controlled through clocks.
- The time progresses in some state before selecting and executing some transitions.
- Transitions take zero time to be executed.

In order to control the time progress or the wait time in states, IF implements the notion of urgency in the transitions. A transition may have priority over others, or may be delayed. In this context, a transition may be described as following:

```
deadline {eager, delayable, lazy};
provided <expression>;
when <constraint>;
input <signal (expression)>;
{statement};
{action};
```

```
if <expression> then {statement} endif;
while <expression> do {statement} endwhile;
nextstate <state_id>;
stop;
```

In the sample above, “Eager”, “Delayable” and “Lazy” concerns the priority of the transitions related to the progress of time, where:

- Eager: the transition has priority over the time. The time can not evolve except if the transition is fired. In other words, these transitions must be executed as soon as they are enabled and waiting is not allowed.
- Delayable: the time has priority over the transition. This means that the time may evolve until the time constraint becomes true. When it is enabled, waiting is allowed as long as time progress does not disable it.
- Lazy: the transition and the time have the same priority. In this case does not matter what comes first: the transition may be fired or the time may evolve. These transitions are never urgent. When a lazy transition is enabled, the transition may be executed or the process may wait without any restriction.

Several tools may interact with IF. Some concern the automatic transformation of system specifications into IF Format (as *SDL2IF*, or *UML2IF*). Other tools have tackled the system analysis and verification using the IF format such as TReX [Annichini *et al.* (2001)]. Other possibilities are the simulation of the system (IF-2.0[2] and IFx[1]), or even the test generation using TGV tool.

1.4.3 *Case Study: Travel Web Application*

To prove the effectiveness of our framework we carried out a case-study using a Travel application which is an internal service used by France Telecom company to manage ‘missions’ (business travels) carried out by its employees. In our case study we only consider, at first, a simple Travel application where a potential traveler can connect to the system (using a dedicated URL) to request a travel ticket and a hotel reservation during a specific period according to some business purposes (called mission). This request can be accepted or rejected by his/her hierarchical superior (called validator). In the case it is accepted, the travel ticket and hotel room are booked by contacting a travel agency. The specification of this Travel Web application is performed using the IF language.

Further, we defined some specific security rules to boost the system security. These security rules are inspired from France Telecom’s security test campaign and are formally specified using the Nomad model.

[2]http://www-omega.imag.fr/

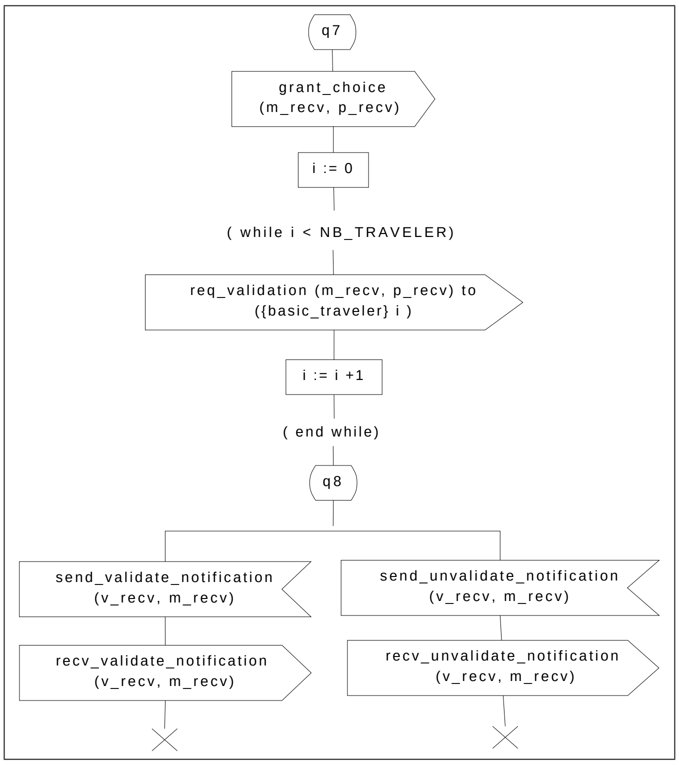
q7
grant_choice (m_recv, p_recv)
i := 0
(while i < NB_TRAVELER)
req_validation (m_recv, p_recv) to ({basic_traveler} i)
i := i +1
(end while)
q8
send_validate_notification (v_recv, m_recv)
send_unvalidate_notification (v_recv, m_recv)
recv_validate_notification (v_recv, m_recv)
recv_unvalidate_notification (v_recv, m_recv)

Fig. 1.3 An IF State in the Travel_Mission Process: q7 and q8 States

1.4.4 *Travel IF Specification*

Modeling Web applications allows software engineers to specify, understand and maintain their complex architecture in an optimized manner. To perform this formal specification, we use the IF language to model the functional behavior of the Travel communicating Web application. This specification provides the metrics in the table 1.1.

Table 1.1 IF Travel System Specification

Processes	**States**	**Transitions**	**Signals**	**Variables**
basic_traveler	5	12	13	11
traveler_mission	7	12	11	8
basic_travel	2	7	7	8
travel_mission	9	11	14	6

The IF model is composed of four processes. Each process communicates with the other using a set of signals:

- *basic_travel* and *travel_mission* are two processes that describe the Travel system behavior. *basic_travel* allows to communicate with a basic user of the system whereas *travel_mission* allows to manage the ‘missions’ requested by a potential traveler.

- *basic_traveler* and *traveler_mission* are two processes that describe the user behavior. The first process simulates a basic traveler that can change its profile, delegate its rights, and request the creation of a mission or its validation. Whereas *traveler_mission* describes a potential traveler that can choose the details of its business travel.

The IF specification of the Travel system is finite but large. For matter of space, we only present, in Figure 1.3, two states (q_7 and q_8) from the *basic_travel* process. In the state q_7, the system asks for a validation relating to a mission request (output req_validation()). State q_8 has two transitions. The input in the left transition (resp. right transition) is received from the mission validator that sent an acceptance (resp. a reject) notification to the Web-based system. This notification is transmitted to the potential traveler using the output signal (output recv_(un)validate_notification()).

1.5 Secure Specification of Web Applications

1.5.1 *Security Rules Specification Using Nomad Language*

We rely in this approach on Nomad formal language [Cuppens *et al.* (2005)] to specify, without any ambiguity, the set of security properties that the system has to respect. The choice of this language was mainly motivated by the characteristics of Nomad that provides a way to describe permissions, prohibitions and obligations related to non-atomic actions within elaborated contexts and mainly time constraints. By combining deontic and temporal logics, Nomad allows to describe conditional privileges and obligations with deadlines, 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.

1.5.1.1 *Nomad Formal Language: Syntax and Semantics*

To meet the requirements of the functional model of the system, we define an atomic action with the same concepts of TEFSM actions.

Definition 1.2. *(Atomic action) 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 a process destruction.*

Definition 1.3. *(Non-atomic action) If A and B are actions, then $(A;B)$, which means "A is followed immediately by B" is a non-atomic action.*

Definition 1.4. *(Formulae) If A is an action then $start(A)$ (starting A), and $done(A)$ (finishing A) are formula.*

- *If α and β are formulae then $\neg\alpha$, $(\alpha \wedge \beta)$ and $(\alpha \vee \beta)$ are formulae.*

- *If α is a formula then $O^d \alpha$ (α was true d units of time ago if $d \leq 0$, α will be true after d units of time if $d \geq 0$) is a formulae too.*
- *If α is a formula then $O^{<d} \alpha$ (within d units of time ago, α was possibly true if $d \leq 0$, α is possibly true within a delay of d units of time if $d \geq 0$) is a formulae.*
- *If α and γ are formulae then $(\alpha|\gamma)$ is a formula whose semantics is: in the context γ, the formula α is true.*

In the rest of the chapter, we refer to operators "*O*" and "|" by *timed* and *contextual* operators respectively. Also, we use the notation $O^{[<]d}$ to cover both cases O^d and $O^{<d}$. Notice also that using Nomad formalism, we deal with a discrete time. The choice of the unit of time can be very important and depends on the studied system. In our work, we use real time units like seconds, milliseconds or microseconds depending on the desired precision.

Definition 1.5. *(A security rule) If α and β are formulae, $\mathscr{R}$ $(\alpha \mid \beta)$ is a security rule where R denotes one of the following deontic operators: $\{\mathscr{P}, \mathscr{F}, \mathscr{O}\}$. $\mathscr{P}$ $(\alpha \mid \beta)$ (resp. $\mathscr{F}$ $(\alpha \mid \beta)$, $\mathscr{O}$ $(\alpha \mid \beta)$) means that it is permitted (resp. prohibited, mandatory) to execute α when context β holds.*

1.5.1.2 *Examples of Security Rules Specification*

We present in this section some examples of security rules specifications according to Nomad language:
Example 1:

$$\begin{gathered}\mathscr{P}(\textit{start (input ReqWrite(user,file.doc))}| \\ (O^{\leq -5s}\ \textit{done (output AuthOK(user)))} \wedge \\ (\neg\ \textit{done (output DisconnectOK(user))))}\end{gathered}$$

This rule expresses a permission granted to any user to request to write in 'file.doc', if earlier, within 5 seconds, he/she was authenticated in the system and his/her authentication is still running.
Example 2:

$$\begin{gathered}\mathscr{O}(\textit{start (output DisconnectOK(user))}\ | \\ (O^{\leq -30min} \neg\ \textit{done (input (user)))} \wedge \\ (\neg\ \textit{done(output DisconnectOK(user))))}\end{gathered}$$

According to this obligation rule, the system must disconnect a running connection of any user if this latter remains inactive for 30 minutes.
Example 3:

$$\begin{gathered}\mathscr{F}(\textit{start ((output AuthOK (user)))}\ | \\ O^{\leq -0.01s}\ \textit{done (output AuthOK(user))} \wedge \\ (\neg\ \textit{done(output DisconnectOK(user))))}\end{gathered}$$

To avoid service deny, this prohibition rule means that the system must not allow any user to get two simultaneous connections in the same millisecond.

1.5.2 *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 seeks for the rules to be applied on each transition of the TEFSM specification. Then, it modifies each transition by adding some states, transitions and clocks or by updating transition guards. This modification depends on the nature of the rule (prohibition, permission or obligation) and its syntax format.

1.5.2.1 *Integration Process Assumptions*

To integrate security rules into a TEFSM specifications, we have to make the following assumptions:

- The initial TEFSM specification representing the behavior of the system is correct. Indeed, it must be deadlock free and each state must be reachable from any other state.
- The initial TEFSM specification of the system does not take into account any security requirements. It only specifies system behavior from its functional point of view.
- The security rules to integrate are consistent. We assume that it do not contain any incoherent or redundant rules. Checking the consistency of the security policy is out of the scope of this approach. We assume that this issue has been checked. There are several techniques to achieve this goal (see for instance [Cuppens *et al.* (2006)]). Here is an example of inconsistent security policy composed of two rules $\mathscr{O}$ *(start(A)* $|$ O^{-d} *done(B))* and $\mathscr{F}$ *(start(A)* $|$ O^{-d} *done(B))*. We cannot oblige the system to perform action A in a context $C = O^{-d}$ done(B) if this action is forbidden in the same context.

1.5.2.2 *Security Rules Classification*

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 into two main classes described hereafter:

(1) *Basic Timed security rules*: we consider in this class security rules of the form $\mathscr{R}(start(A)|O^{[<]d}done(B))$ where A and B are actions. To make easier the integration of such rules, we also distinguish two subclasses:
   (a) *Basic security rules with atomic actions*: actions A and B are atomic.
   (b) *Basic security rules with decomposable actions*: A or B or both are non-atomic actions. They denote a sequential set of atomic actions.

Section 1.5.2.3 gives the algorithms we have developed to integrate such basic security rules into a TEFSM specification.

(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 or/and timed operators or/and logical connectors. In section 1.5.2.4, we show that such a rule can be decomposed and rewritten into one or several basic rules. In this way, integration algorithms developed for the first class can be reused and applied to integrate general security rules.

1.5.2.3 *Integration of Basic Timed Security Rules*

This section describes the integration of security rules of the form:

$$\mathscr{R}\ (start(A|O^{[\leq]d}done(B))$$

As stated in the previous section, we have to distinguish the cases of atomic and non-atomic actions. The first part of this section describes the algorithms that allow the integration of basic security rules that only contain atomic actions, whereas the second part deals with non atomic actions.

Basic Security Rules with Atomic Actions

A basic security rule with atomic actions has the following form: $\mathscr{R}$ *(start(A* $|$ $O^{[\leq]d}done(B))$ where $R \in \{\mathscr{F}, \mathscr{O}, \mathscr{P}\}$ and A and B are atomic actions.

Prohibitions Integration

The prohibited action relates usually to an already existing action in the initial system. Considering the TEFSM specification, action B can appear on one or several transitions. The basic idea of integrating such 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 can be performed without any rule violation. Since we deal with a timed context, we have to define a clock to manage the temporal aspect of the rule.

First Case. In the following, we present the different steps to integrate a prohibition rule in the form of $\mathscr{F}$ *(start(A)* $|$ $O^{<-d}$ *done(B))* within a TEFSM model where ($d > 0$). This rule expresses that it is forbidden to perform action A if within ($d-1$) units of time ago, B was performed. B is the context action and A is the prohibited action. Three steps are to be considered:

- The creation of a public clock Ck that can be modified by all the TEFSM model processes.
- Clock Ck is set to 0 after each occurrence of B in the TEFSM transitions. Intuitively,

Ck measures the time elapsed from the last occurrence of action *B*. Before the first occurrence of *B*, clock *Ck* is simply inactive.

- Before performing the prohibited action *A*, we verify whether clock *Ck* is already activated. If so, we check its valuation to deduce if *A* can be performed or not. If clock *Ck* is not activated, that means that the system did not perform *B* yet. In that case, *A* is allowed.

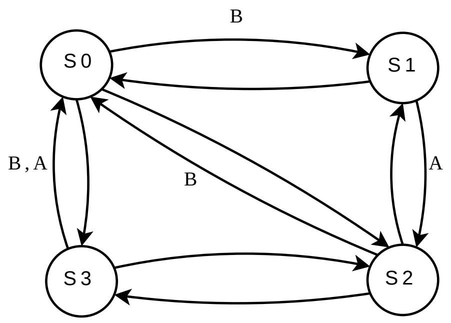

Fig. 1.4 Initial System Specification.

These steps are provided in pseudo-code in Algorithm 1.1. To illustrate this algorithm, we present an example of a prohibition rule integration in Figure 1.4 and 1.5. The initial functional system illustrated in Figure 1.4 contains several occurrences of the atomic actions *A* and *B*. We want to integrate the rule $\mathscr{F}$ *(start(A)* | $O^{<-d}$ *done(B))* that stipulates that it is forbidden to perform action *A* if within *d* units of time ago, *B* was performed. Applying Algorithm 6, we obtain the secure system depicted in Figure 1.5.

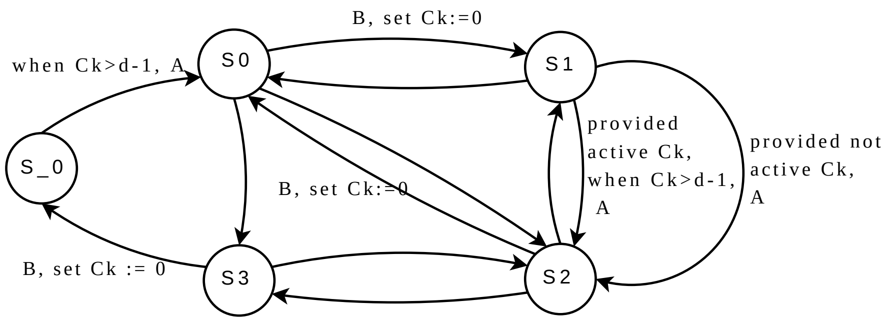

Fig. 1.5 Secure System Specification.

Algorithm 1.1 Prohibitions Integration (1/2)

```
Require: The TEFSM model $M =< S, s_0, I, O, \vec{x}, \vec{c}, Tr >$ and the prohibition security rule $\mathscr{F}$ (start(A) | $O^{<-d}$ done(B))
1:  Define a new integer variable k:=0;
2:  Define a new clock $Ck$ within $M$
3:  for each (transition $tr$ such that ($tr \in Tr \wedge tr =< S_i, S_j, G, Act >$)) do
4:    if $B \in Act$ then
5:      $tr := < S_i, S_j, G, (before(B); B; set\ ck := 0; After(B)) >$
6:      if (($A \in Act$) $\wedge$ $A \in After(B)$) then
7:        /*$tr$ is of the form $< S_i, S_j, G, (before(B); B; C; A; After(A)) >$*/
8:        Create a new state $S_{\text{-}k}$ and a new transition $Tr_k$
9:        $tr := < S_i, S_{\text{-}k}, G, (before(B); B; C) >$
10:       $tr_k := < S_{\text{-}k}, S_j, \{when\ Ck > d - 1\}, (A; After(A)) >$
11:       k++;
12:     end if
13:   else
14:     if ($A \in Act$) then
15:       Create a new transition $Tr_k$
16:       $tr := < S_i, S_j, \{G,\ provided\ not\ active\ Ck\}, (before(A); A; After(A)) >$
17:       $tr_1 = < S_i, S_j, \{G,\ provided\ active\ Ck,\ when\ Ck > d - 1\}, (before(A); A; After(A)) >$
18:       k++;
19:     end if
20:   end if
21: end for
```

Second case. This part gives the steps to follow in order to integrate, within a TEFSM specification, a prohibition rule of the form $\mathscr{F}$ *(start(A)* | O^{-d} *done(B))* where $d > 0$. This rule expresses that it is forbidden to perform action A if B was performed d units of time ago. The first solution that comes to mind consists in defining -like in the first case- a new clock Ck which is set to 0 each time action B is executed. Then, the guard of each transition that executes action A is reinforced by the guard $\{Ck \neq d\}$ to make the transition fireable only if the elapsed time from the execution of action B is different from d (It may be more or less). This solution is represented by a declination of Algorithm 6 by replacing $\{when\ Ck > d - 1\}$ with $\{when\ Ck \neq d\}$.

Table 1.2 A Transitions Sequence Example with Time Progress.

	Transition	**Arrival State S_i**	**Duration in S_i**
Tr_1	$S0 \rightarrow S2$	$S2$	2
Tr_2	$S2 \rightarrow S0$	$S0$	3
Tr_3	$S0 \rightarrow S1$	$S1$	2
Tr_4	$S1 \rightarrow S2$	$S2$	Not relevant

Figure 1.6 illustrates the application of this algorithm on the example of the initial TESM presented in Figure 1.4. However, a deep analysis of the presented solution shows that this latter is only conceivable if the interval between two successive executions of ac-

tion B is longer than d. Indeed, let us assume that TEFSM system in Figure 1.6 follows the sequences of transitions shown in table 1.2 and that clock Ck progresses after its activation in each state S_i according to a given valuation.

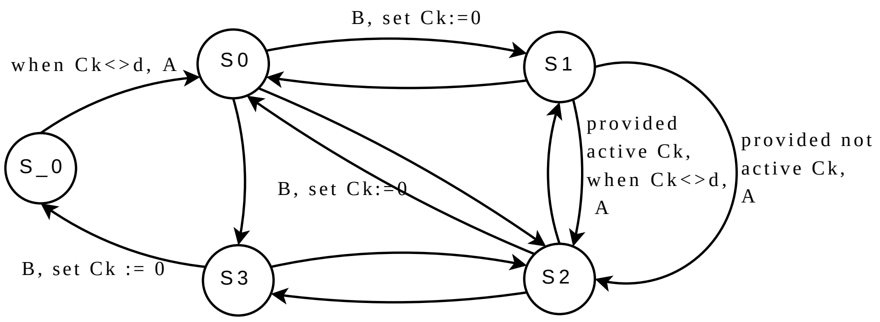

Fig. 1.6 First Intuition for Prohibition Rule Integration

Let us suppose that d is equal to 5. gck denotes a master clock that measures the system global time. The progress of the secure system is described in Table 1.3.

Table 1.3 The Secure TEFSM System Progress

Tr	**State**	*gck*	*Ck*	Note
Tr_1	$S0$	0	-1	*Ck* is not yet activated
	$S2$	0	-1	Transitions are instantaneous
Time progress (2 units of time)				
Tr_2	$S2$	2	-1	*Ck* is not yet activated
	$S0$	2	0	1st execution of *B*
Time progress (3 units of time)				
Tr_3	$S0$	5	3	Both clocks progress
	$S1$	5	0	2nd execution of *B*
Time progress (2 units of time)				
Tr_4	$S1$	7	2	Both clocks progress
	$S2$	7	2	Action *A* is performed since $Ck \neq 5$

We can notice that since Ck is not equal to 5, action A is 'wrongly' executed although the time elapsed from the first execution of action B is equal to 5. This is due to the reset action ($Ck := 0$) executed at the second occurrence of B. In other words, this re-set action erases the previous execution of B from the system memory.

To cope with this limit, we suggest the following second solution. Basically, we define a clock gck that denotes a master clock that measures the time elapsed from the beginning and an integer variable c that indicates the next moment when the execution of A is forbidden. Thus, for each execution of action B, the system creates a new process *RHP* (for Rule Handler Process) that waits during d units of time. Then, it updates the value of c to state the moment when the execution of A is forbidden, then it stops (it kills itself). The global clock gck is compared to the value of c before performing A. The algorithm 1.2 formally

Algorithm 1.2 Prohibition Integration (2/2)

```
Require: The TEFSM model $M =< S, s_0, I, O, \vec{x}, \vec{c}, Tr >$ and the prohibition security rule $\mathscr{F}$ (start(A) | $O^{-d}$ done(B))
1: Define a new integer variable k:=0;
2: In $M$, define a new public clock $gck$ and a new public integer variable $c$
3: In the initial State, set gck := 0
4: c := -1
5: for each (transition $tr$ such that ($tr \in Tr \wedge tr =< S_i, S_j, G, Act >$)) do
6:   if ($B \in Act$) then
7:     $tr := < S_i, S_j, G, (before(B); B; fork\ RHP\ ((integer)gck + d); After(B)) >$
8:     /*RHP is a new process that handles the c variable. It accepts an integer parameter*/
9:     if (($A \in Act$) $\wedge A \in After(B)$) then
10:      /*$tr$ is of the form of $< S_i, S_j, G, (before(B); B; C; A; After(A)) >$*/
11:      Create a new state $S_{-k}$ and a new transition $Tr_k$
12:      $tr := < S_i, S_{-k}, G, (before(B); B; C) >$
13:      $tr_k := < S_{-k}, S_j, \{when\ gck \neq c\}, (A; After(A)) >$
14:      k++;
15:    end if
16:  else
17:    if ($A \in Act$) then
18:      $G := \{G, when\ gck \neq c\}$
19:    end if
20:  end if
21: end for
22: for RHP (T) do
23:   In the initial state $S_0$, define a transition $Tr1$
24:   $tr1 := < S_0, \_, when\ gck = T, (c := T; stop) >$
25: end for
```

defines these steps.

Applying this algorithm on the TEFSM of Figure 1.4 gives the secured TEFSM depicted in Figure 1.7.

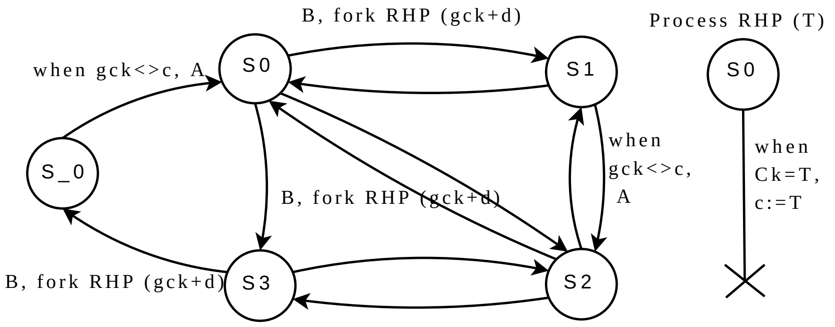

Fig. 1.7 Prohibition Rule Integration: $\mathscr{F}$ (start(A) | O^{-d} done(B)).

Permissions Integration

Like prohibitions, permission relate to actions which already exist in the initial functional system. Though even the permission to perform an action A in a given context C it is equivalent to the prohibition to execute action A when C is not verified, the permission integration algorithms are slight different form those developed for prohibition rules. We give hereafter the steps to follow to integrate permissions rules.

First case. In this section, we propose a methodology to integrate a permission rule in the form $\mathscr{P}$ *(start(A)* | $O^{<-d}$ *done(B))* within a TEFSM model where ($d \geq 0$). This rule expresses that it is permitted to perform action A if within ($d-1$) units of time ago B was performed. Like in prohibitions integration, we need to define a new public clock variable that can be modified by all the TEFSM model processes. This global clock is set to 0 after each occurrence of B. Before performing the granted action A and in case where action B is not already executed in the same transition, we verify the clock valuation (if the clock is already activated) to deduce if A can be performed or not. In fact, if action B is executed in the same transition just before action A, this latter later can be executed without violating the permission rule.The steps to integrate a permission rule are described in pseudo-code in algorithm 1.3.

Algorithm 1.3 Permission Integration (1/2)

```
Require: The TEFSM model $M =< S, s_0, I, O, \vec{x}, \vec{c}, Tr >$ and the permission security rule
    $\mathscr{P}$ (start(A) | $O^{<-d}$ done(B))
1:  Define a new clock $Ck$ within $M$
2:  for each (transition $tr$ such that ($tr \in Tr \wedge tr =< S_i, S_j, G, Act >$)) do
3:    if ($B \in Act$) then
4:      /* $tr$ is of the form $tr =< S_i, S_j, G, \{before(B); B; After(B)\} >$ */
5:      $tr := < S_i, S_j, G, \{before(B), B, set\ Ck := 0, After(B)\} >$
6:    end if
7:    if ($A \in Act$) then
8:      /* $tr$ is of the form $tr =< S_i, S_j, G, \{before(A); A; After(A)\} >$ */
9:      if ($B \notin Before(A)$) then
10:       $tr := < S_i, S_j, \{G,$ (provided not active Ck) $\vee$
                (provided active Ck, when $Ck \geq d$)$\}, \{before(A); After(A)\} >$
11:       /*Create a new transition $tr_1$*/
12:       $tr_1 = < S_i, S_j, \{G, provided\ active\ Ck, when\ Ck < d\},$
                                              $\{before(A)A, After(A)\} >$
13:     end if
14:   end if
15: end for
```

Algorithm 1.4 Permissions Integration (2/2)

```
Require: The TEFSM model $M =< S, s_0, I, O, \vec{x}, \vec{c}, Tr >$ and the permission security rule
    $\mathscr{P}$ (start(A) | $O^{-d}$ done(B))
 1: Define a new integer variable k:=0;
 2: In $M$, define a new clock public $gck$ and a new public integer variable $c$
 3: In the initial State, set gck := 0
 4: c := -1
 5: for each (transition $tr$ such that $(tr \in Tr \wedge tr =< S_i, S_j, G, Act >)$) do
 6:   if $(B \in Act)$ then
 7:     /* $tr$ is of the form $tr = < S_i, S_j, G, \{before(B); B; After(B)\} >$ */
 8:     $tr := < S_i, S_j, G, \{before(B), B,$
                            $fork\ RHP\ ((integer)gck + d), After(B)\} >$
 9:     (RHP is a new process that handles the c variable. It accepts an integer parameter)
10:     if $(A \in After(B))$ then
11:       /*$tr$ is of the form $tr = < S_i, S_j, G, \{before(B), B,$
                $fork\ RHP\ ((integer)gck + d), C, A, After(A)\} >$)*/
12:       Create a new state $S_k$ and a new transition $tr_k$
13:       $tr := < S_i, S_k, G, \{before(B), B, fork\ RHP\ ((integer)gck + d), C\} >$
14:       $tr_k := < S_k, S_j, \{when\ gck = c\}, \{A, After(A)\} >$
15:       k++;
16:     end if
17:   else
18:     if $(A \in Act)$ then
19:       /* $tr$ is of the form $tr = < S_i, S_j, G, \{before(A); A; After(A)\} >$ */
20:       $tr :=< S_i, S_j, \{G, when\ gck = c\}, Act >$
21:     end if
22:   end if
23: end for
24: for RHP (T) do
25:   In the initial state $S_0$, define a transition $tr1$
26:   $tr1 := < S_0, \_, when\ gck = T, \{c := T, stop\} >$
27: end for
```

Second case. This part gives the steps to follow in order to integrate, within a TEFSM specification, a permission rule of the form $\mathscr{P}$ *(start(A)* | O^{-d} *done(B))* where $d > 0$. This rule expresses that it is permitted to perform action A if B was performed d units of time ago. If this condition is not satisfied, A is denied. Like in the case of prohibition rule, we need to define a global clock *gck* an integer variable c that indicates the moment when the execution of A is permitted. Thus, for each execution of action B, the system creates a child process that waits state during d units of time. Then, it updates the value of c to state the moment when the execution of A is granted, then it stops. The global clock *gck* is compared the c value before performing A. The algorithm 1.4 formally defines these steps.

Obligations Integration

To integrate an obligation security rule, we rely on a new process RHP that ensures the execution of the mandatory action. If the related mandatory action is not executed by the initial specification, the process has the task to execute it itself.

First case. The integration methodology follows these steps for a rule that is in form of $\mathcal{O}$ *(start(A)* $|$ O^{-d} *done(B))* where $d > 0$:

- The definition of a new process that can be created n times by the initial functional specification. n is the maximum number of occurrences of the rule context action B that can be executed during d units of time.
- The new process has to set a clock and wait until the deadline is reached. At this moment, it performs the mandatory action A.

Algorithm 1.5 Obligations Integration (1/2)

```
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:  for each (transition $tr$ such that ($tr \in Tr \wedge tr =< S_i, S_j, G, Act >$)) do
2:      if ($B \in Act$) then
3:          $tr := < S_i, S_j, G, (before(B); B; fork\ RHP\ After(B)) >$
4:          /*RHP is a new process that handles the obligation rule*/
5:      end if
6:  end for
7:  for RHP process do
8:      Define a new clock Ck
9:      Define a new state Wait
10:     Define two transitions $Tr1$ and $Tr2$
11:     $tr1 := < S_0, Wait, \_, set\ Ck := 0 >$
12:     $tr2 := < Wait, \_, when\ Ck > d-1, (A; stop) >$
13: end for
```

Note that we assume that the initial system S is not secure, that is it does not perform the action A, d units of time after B. Otherwise, (if A is performed by S), we can easily define a boolean and public variable *var* that is set to true when A is executed at the right moment. In that case, the external process RHP performs A only if $var = false$.

In Figure 1.8, we present the integration of an obligation rule within the initial system depicted in Figure 1.4. In this functional system, we can find several occurrences of atomic action B.

Second case. To add an obligation rule of the form of $\mathcal{O}$ *(start(A)* $|$ $O^{<-d}$ *done(B))*, we have to associate with each occurrence of action B an execution of action A. This latter action has to be performed within a delay of $(d-1)$ units of time. To perform such an integration, we have to follow the steps given hereafter (see Algorithm 1.6):

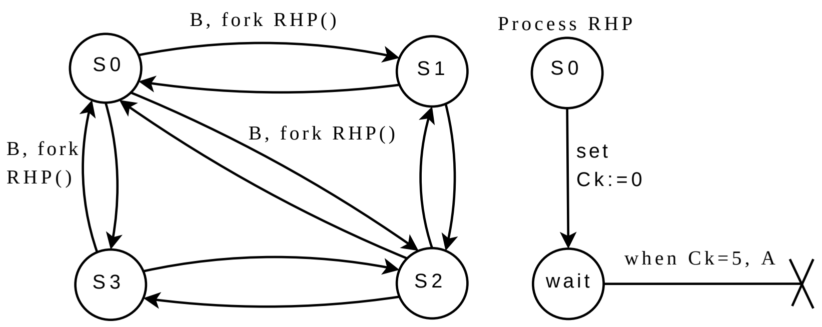

Fig. 1.8 Obligation Rule Integration : $\mathscr{O}$ (start(A) | O^{-d} done(B)).

Algorithm 1.6 Obligations Integration (2/2)

```
Require: The TEFSM model $M =< S, s_0, I, O, \vec{x}, \vec{c}, Tr >$ and the obligation security rule $\mathscr{O}$ (start(A) | $O^{<-d}$ done(B))
 1: In the initial state of $M$, $wait_A := 0$
 2: for each (transition $tr$ such that $(tr \in Tr \wedge tr =< S_i, S_j, G, Act >)$) do
 3:   if $(B \in Act)$ then
 4:     $tr := < S_i, S_j, G, (before(B); B; wait_A{+}{+}; fork\ RHP(); After(B)) >$
 5:   end if
 6:   if $(A \in Act)$ then
 7:     $tr := < S_i, S_j, G, (before(A); A; if(wait_A > 0)\ wait_A{-}{-}; After(A)) >$
 8:   end if
 9: end for
10: for RHP process do
11:   Define a new clock $Ck$
12:   Define a new state Wait
13:   Define three transitions $tr1$ and $tr2$ and $tr3$
14:   $tr1 := < S_0, Wait, \_, set\ Ck := 0 >$
15:   $tr2 := < Wait, \_, \{when\ Ck = d-1, if(wait_A > 0)\}, (A; stop) >$
16:   $tr3 := < Wait, \_, \{when\ Ck = d-1, if(wait_A = 0)\}, stop >$
17: end for
```

- We define an integer variable $wait_A$ that counts the number of occurrences of actions B that are waiting for an execution of action A.
- We define a new process *RHP* where a clock *Ck* is activated to wait $(d-1)$ units of time (till action A has to be executed). When the deadline is reached, process *RHP* checks whether we are waiting for any execution of action A ($wait_A > 0$) and executes A if necessary.
- Variable $wait_A$ is updated as follows: $wait_A$ is incremented each time action B is executed. If the value $wait_A$ is strictly positive, it is decremented each time action A is executed either by the initial specification or by process *RHP*.

Intuitively, process *RHP* has to wait for a possible execution of action A during the allowed time $(0..(d-1))$. In case where the initial specification does not execute such an action, process *RHP* must execute it.

Figure 1.9 shows the integration of obligation rule of the form $\mathcal{O}$ (start(A) | $O^{<-d}$ done(B)) within the initial system shown in Figure 1.4.

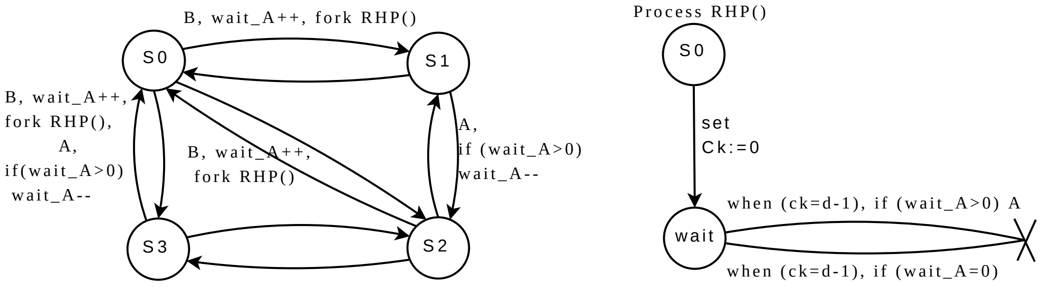
B, wait_A++, fork RHP()
S0
S1
S2
S3
B, wait_A++, fork RHP(), A, if(wait_A>0) wait_A--
B, wait_A++, fork RHP()
A, if (wait_A>0) wait_A--
Process RHP()
S0
set Ck:=0
wait
when (ck=d-1), if (wait_A>0) A
when (ck=d-1), if (wait_A=0)

Fig. 1.9 Obligation Rule Integration : $\mathcal{O}$ (start(A) | $O^{<-d}$ done(B)).

Basic Security Rules with Non-Atomic Actions

In this section, we consider the integration of security rules with non-atomic actions (see definition 1.3). A non-atomic action A is sequence of atomic actions $\{a_1;\ldots;a_{size(A)}\}$. First, we define two different categories of non-atomic actions:

Definition 1.6. *(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 it exists sequences of actions* X *and* Y *such that (*$Act = (X;A;Y)$*). Both sequences* X *and* Y *may be empty.*

Definition 1.7. *(n_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) $A \subseteq (Act(tr_1);\ldots;Act(tr_k))$. $Act(tr)$ *denotes the sequence of actions labeling transition* tr.

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 rules containing the Act in the transition tr is performed similarly to the case of atomic actions since we handle a unique transition. In other words, we can apply the algorithms we defined in section 1.5.2.3 by distinguishing the two following cases:

- $\underline{B = Act}$: the activation of the possible clocks and the creation of eventual processes are added to actions of tr immediately after the execution of action Act.

- $\underline{A = Act}$: in that case, possible guards related to the clocks are added to the guard of transition tr.

Integration of Rules with n_Tr Actions

The handling of decomposable actions can be inspired from [Mallouli and Cavalli (2007)]. In this section, we only consider n_Tr actions that are decomposable on n distinct transitions $Tr = (tr_1, \ldots, tr_n)$. We also make the assumption that the subgraph formed by transitions Tr does not contain any cycle. According to this n transitions, we introduce the following notations:

- The starting transition ST: includes the transition that the system has to follow to start the action: $ST = tr_1$.
- The ending transition ET: includes the transition that the system has to follow to end the action: $ST = tr_n$.
- The intermediate transitions set ITS: includes the remaining transitions: $IT = \{tr_2, \ldots, tr_{n-1}\}$ and may be empty.
- The outgoing transitions set OTS: includes the transitions whose initial states belong to those of transitions $(Tr - \{tr_1\})$ and whose final states are beyond the action. This set is formally defined as follows: $OTS = \{tr | \exists tr'.(tr' \in Tr - \{tr_1\} \wedge IS(tr) = IS(tr') \wedge tr \neq tr'\}$.

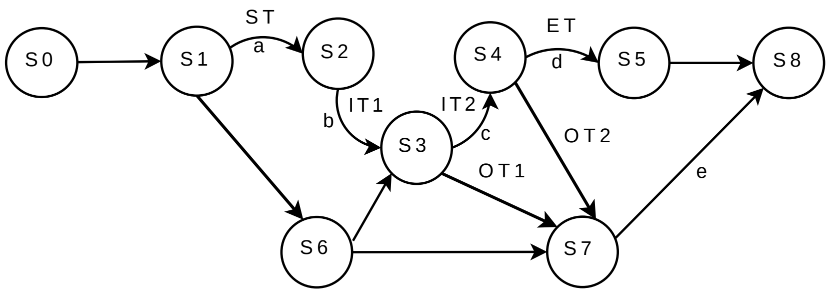

Fig. 1.10 An initial TEFSM specification with one n_Tr Action a;b;c;d.

In Figure 1.10, the 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 $OT2$).

For sake of space, this chapter only discusses the integration of a prohibition security rule of the form $\mathscr{F}(start(A)|O^{<-d}done(B))$ 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 $(C = (c_1; \ldots; c_n))$ be a decomposable action with respect to transitions $Tr = \{tr_1, \ldots, tr_n\}$. A system is executing action C if and only if it is firing transitions Tr in the good sequence order. To memorize 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.
- 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:

- *B*=*C*: the activation of the clock is done 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 (*set ck:=0*) by stating that this action is only performed when v_C is true.
- *A*=*C*: to forbid the execution of action C in a given context, we chose to skip the last action(s) in the transition tr_n when context condition holds. In our case for instance, we add condition (*(when ck ¿ d-1) or* $v_C = false$) to the ending transition ET. By this way, we state that this transition is only fired if the time constraint is fulfilled or the system is not running action C.

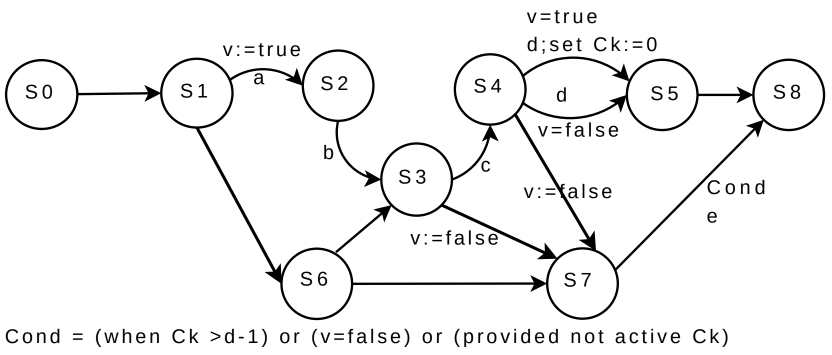

Fig. 1.11 Secure TEFSM After Integration.

Figure 1.11 shows the integration of security rule $\mathscr{F}(start(e)|O^{<-d}done(a;b;c;d))$. Variable v is added to know if the system is running or not the decomposable action $(a;b;c;d)$.

1.5.2.4 *Integration of General Security Rules*

So far, the integration algorithms we have described only deal with security rules with a unique timed operator in the context part. Moreover, these rules do not contain any logical operator. This section gives some features to extend our proposed approach by considering more elaborated rules that may contain several logical and timed operators.

Table 1.4 Rewritten Rules for Some Elaborated Security Rules

Rule	Rewrite
$\mathscr{R}(A\|C_1 \vee C_2)$	$\{\mathscr{R}(A\|C_1), \mathscr{R}(A\|C_2)\}$ where $\mathscr{R} \in \{\mathscr{F}, \mathscr{O}, \mathscr{P}\}$
$\mathscr{R}(A_1 \vee A_2\|C)$	$\{\mathscr{R}(A_1\|C), \mathscr{R}(A_2\|C)\}$ where $\mathscr{R} \in \{\mathscr{F}, \mathscr{P}\}$
$\mathscr{O}(A_1 \vee A_2\|C)$	either $\mathscr{O}(A_1\|C)$ or $\mathscr{O}(A_2\|C)$
$\mathscr{R}(A_1 \wedge A_2\|C)$	$\{\mathscr{R}(A_1\|C), \mathscr{R}(A_2\|C)\}$ where $\mathscr{R} \in \{\mathscr{F}, \mathscr{O}, \mathscr{P}\}$
$\mathscr{F}(A\|\neg B)$	$\mathscr{P}(A\|B)$
$\mathscr{P}(A\|\neg B)$	$\mathscr{F}(A\|B)$
$\mathscr{F}(\neg A\|B)$	$\mathscr{O}(A\|B)$
$\mathscr{P}(\neg A\|\neg B)$	$\mathscr{O}(A\|B)$
$\mathscr{R}(A\|O^{-d_2}O^{-d_1}C)$	$\mathscr{R}(A\|O^{-(d_1+d_2)}C)$ where $\mathscr{R} \in \{\mathscr{F}, \mathscr{O}, \mathscr{P}\}$
$\mathscr{R}(A\|O^{<-d_2}O^{<-d_1}C)$	$\mathscr{R}(A\|O^{<-(d_1+d_2)}C)$ where $\mathscr{R} \in \{\mathscr{F}, \mathscr{O}, \mathscr{P}\}$

We especially tried to show that the integration of some of the elaborated rules comes down to the integration of one or several more basic security rules.

Table 1.4 gives some examples of rewritten rules that transform a subset of complex rules into one of several simpler basic rules. These rules can be divided into two classes. The first class concerns security rules involving logical connectors, the second deals with the multiple uses of timed operators. Let us remark that the left and right parts of the third rewritten rule are not equivalent. Such a rewrite rule can be viewed as a refinement rule. In fact by integrating $\mathscr{O}(A_1|C)$ (resp. $\mathscr{O}(A_2|C)$), we will obtain a final system whose behavior is included in the behavior of a system that would verify $\mathscr{O}(A_1 \vee A_2|C)$.

1.5.3 *Correctness Proof of the Integration Approach*

To ensure the correctness of the approach we proposed to integrate security rules within a TEFSM functional model, we have to prove all the algorithm developed in section 1.5.2.3. Most of them are obvious and easily proved. We only present in this section one relevant correctness proof related to algorithms 1.1 and 1.3.

1.5.3.1 *Correctness Proof of Algorithm 1.1*

By proving the correctness of algorithm 1.1, we precisely demonstrate that the integration of prohibition rule of the form: $\mathscr{F}$ *(start(A)* $|$ $O^{<-d}$ *done(B)) where* $d > 0$ produces to a secure TEFSM specification. To achieve this goal, we define for each occurrence of action B:

- t_B as the instant of the execution action B.
- k as the time elapsed after the execution of action B.
- t is a time variable.

- $gck(t)$ is a global clock of the system. By definition $gck(t) = t$
- $ck(t)$ is the clock which is set to 0 after each occurrence of action B.

We have to prove that we can not perform the 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:

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

To prove that action A cannot be executed at the moment $(gck(t) = t_B + k)$, it is sufficient to prove that at this moment all the transitions Tr of the secure system labeled by action A are impassable. That is all the guards of the transitions Tr are false. In the secure system, the algorithm 1.1 adds the guard (*when* $ck > d - 1$) before each execution of A. Thus, we have to prove, for each positive integer k, that:

$$((k < d) \wedge (gck(t) = t_B + k)) \Rightarrow ck(t_B + k) < d \qquad (2)$$

To reach this aim, we propose to prove that:

$$((k < d) \wedge (gck(t) = t_B + k)) \Rightarrow ck(t_B + k) \leq k \qquad (3)$$

Indeed, if (3) is true which means that ($ck(t_B + k) \leq k$), and, knowing that ($k < d$) we can easily deduce equation (2). To establish goal (3) let us proceed by induction on k:

(1) Basic case ($k = 0$): $gck(t) = t_B + 0 = t_B$. We have to prove that $(ck(t_B) \leq 0)$. At the instant t_B, action B is performed and according to algorithm 1.1, we launch the clock ck (*set* $ck := 0$) after the execution of B. Thus $ck(t_B) = 0$, the goal is true since $(0 \leq 0)$.

(2) Induction hypothesis ($k = n$): let us make the assumption that the goal is true until the rank $k = n$ where ($n < (d - 1)$), that is, at the moment $(gck(t) = t_B + n)$, we have: $(ck(t_B + n) \leq n)$. Let assume that at moment $(t_B + n)$ the system is in the state S.

(3) Induction case ($k = n + 1$): let us prove that the inequality is true for $(k = n + 1)$, that is, when the global clock gck increased of one unit of time $(gck(t) = t + n + 1)$, we have $(ck(t + n + 1) \leq n + 1)$. In the period between $gck(t) = t_B + n$ and $gck(t) = t_B + n + 1$, the system has executed a transitions set TS (may be empty) without any time progress beginning from the state S and ending in the state S' (may be equal to S). Note that we consider that S' is the first state in the system when $gck(t) > t_B + n$. Let us prove that:

$$(gck(t) = t_B + n + 1) \Rightarrow ck(t_B + n + 1) \leq n + 1 \ (4)$$

Note first that the only transitions that act on clock ck are those labeled by action B. These transitions are modified by algorithm 1.1 by setting ck to 0. We call them Tr_B. Consequently, to establish (4), two cases are to be considered:

(a) $Tr_B \cap TS \neq \emptyset$: This means that at least one B action is performed and the clock ck is set to 0. Then, $ck(t_B + n) = 0$. We know that time does not progress in TS and that in state S', clocks ck and gck run simultaneously. We can then deduce that $ck(t_B + n + 1) = ck(t_B + n) + 1 = 0 + 1 = 1$. Since $1 \leq n + 1$, (4) is true.

(b) $\underline{Tr_B \cap TS = \emptyset}$: During the execution of TS, the valuation of ck does not change since no transition modifies its valuation. The clocks ck and gck run simultaneously, both of them progress in state S'. So, $ck(t_B+n+1) = ck(t_B+n)+1$. Since $ck(t_B+n) \leq n$ (induction hypothesis), we can deduce that $ck(t_B+n+1) \leq n+1$. (In particular when TS=ϕ, $\neg(\exists\ Tr \in TS$, and the system stays in the state S during one unit of time)

1.5.3.2 *Correctness Proof of Algorithm 1.3*

By proving the correctness of algorithm 1.3, we precisely demonstrate that the integration of permission rule of the form : $\mathscr{P}$ *(start(A)* | $O^{<-d}$ *done(B)) where* $d > 0$ produces a secure TEFSM specification. To achieve this goal, We have to prove that we can perform the action A only if within d units of time action B has been executed. To establish this proof, we define for each occurrence of action A:

- t_A (resp. t_B) as the instant of the execution of action A (resp. B).
- $Ck(t)$ is the clock which is set to 0 after each occurrence of action B.

Mathematically, we have to establish that for each positive integer k the following predicate holds:

$$\exists k.((k < d) \wedge (gck(t) = t_A - k) \wedge done(B)) \qquad (1)$$

Let us reason about the form of the transition tr that is triggered by the execution of action A. Since A belongs to actions of tr; tr is ,of the form $tr =< s_i, s_f, G, Before(A); A; After(A) >$. We distinguish then the following cases:

- $\underline{B \in Before(A)}$: goal (1) is satisfied for $k = 0$. Since the time does not progress during the execution of transitions, actions A and B are executed at the same time ($t_A = t_B$)even if action B has been executed before action A.
- $\underline{B \notin Before(A)}$: since transition tr contains action A and according to algorithm 1.3, transition tr has been added by the integration process. Then, its guard G is of the form $(G' \wedge$ *provided active* Ck, *when* $Ck < d)$. Goal (1) is satisfied for k equal to Ck that denotes the elapsed time from the last execution of action B. Indeed clock Ck is active because predicate (*provided active* Ck) is true. Otherwise, action A cannot be executed. For the same reason, predicate $(Ck < d)$ is true.

1.5.4 ***Travel Security Specification Using Nomad Language***

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 the Nomad language. For matter of space, we only present the following three:

- Rule 1:

$$\mathscr{F}\ (start\ (output\ req\_create\_mission(t))|$$
$$O^{\leq -2min}\ done\ (output\ req\_create\_mission(t)))$$

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

$$\mathscr{P}\ (start\ (output\ req\_proposition\_list(t,m))|$$
$$O^{\leq -10min}\ done\ (output\ req\_proposition\_list(t,m)))$$

  This permission rule expresses that a traveler can request for another list of travel propositions within a delay of 10 minutes if he/she already asked for a first list of travel propositions. This request can be performed in the *traveler_mission* process.
- Rule 3:

$$\mathscr{O}\ (start\ (output\ req\_validation())|$$
$$O^{-10080min}\ done\ (output\ req\_validation())$$
$$\wedge\ O^{\leq -10080min}\ ((\neg\ done\ (input\ recv\_validate\text{-}\ \_notification()))$$
$$\wedge\ (\neg\ done\ (input\ recv\_unvalidate\_notification())))))$$

  This obligation rule expresses that if a traveler requested for the validation of his/her mission and if he/she did not received 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.

1.5.5 *Automatic Rules Integration*

A securuity rules integration module based on Nomand formal language has been implemented using C language. This module is composed of four different sub-modules as illustrated in figure 1.12.

- Specification parsing sub-module: a communicating system described using IF language is composed of active process instances running in parallel and interacting asynchronously through shared variables and signals via communication buffers (signal routes instances) or by direct addressing. A process instance can be created and destroyed dynamically during the system execution. It has local data and a private FIFO buffer. Each IF process is described as a timed automation extended with discrete data variables, communication primitives and urgency attributes on transitions. The parsing methodology of the func-tional specification file is based on the IF language syntax. It allows to store the IF elements in a C structure. This transformation is required in order to allow the integration of the security rules within the functional specification.

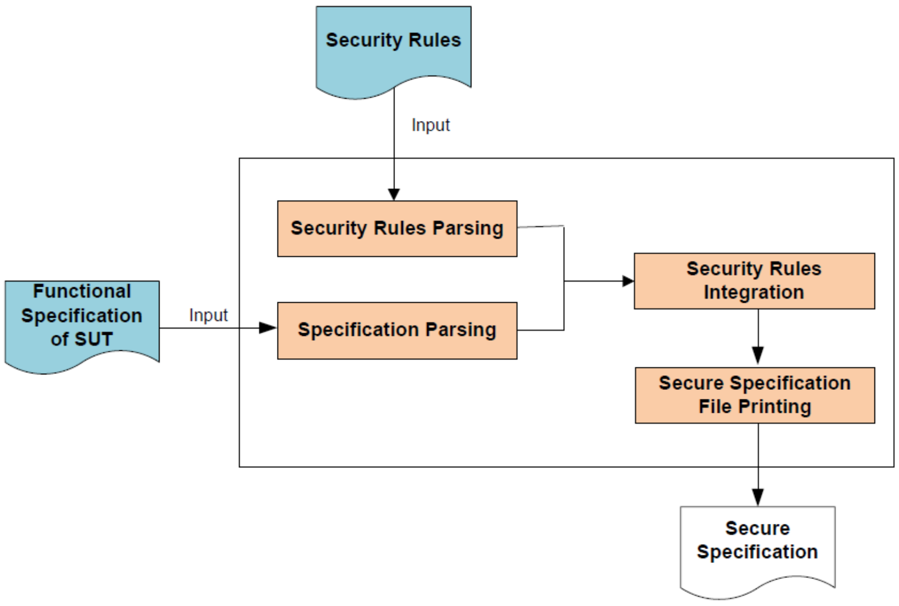
Security Rules
Input
Security Rules Parsing
Security Rules Integration
Functional Specification of SUT
Input
Specification Parsing
Secure Specification File Printing
Secure Specification

Fig. 1.12 Secure Specification Building Module.

- Security rules parsing: security rules are specified using the Nomad formal language that is suited to model permissions, prohibitions and obligations related to non-atomic actions within different contexts. To simplify the development process and to address forward compatibility issues, we defined an XML representation of the Nomad language to use within the integration module. The XML schema of such security rule is defined and the XML parsing is possible based on Gnome XML2 library. It permits to store the rules' elements within a C structure that is an input for the security rules integration sub-module.
- Security rules integration: the integration of the security rules within the IF functional specification of the studied system is done according to the algorithms presented in section 1.5.2. At first, the algorithm seeks for the rules to be applied on each state transition of the specification. Then, it integrates the security rules within the initial specification by adding or modifying guards, transitions and/or states to make the execution instance of a given action possible/mandatory under a specific context. At the end of the process, this integration will generate a new C structure describing the system specification that takes into account the security requirements.
- Secure specification file generation: this last sub-module generates, in IF syntax, the secure specification of the system under test (SUT) based on the C structure integrating the security rules within the functional specification.

1.5.6 *Rules Integration Results*

The table 1.5 shows some metrics about the modifications after the integration of some specific rules: the modified and added transitions (M&A Transitions), the added variables and clocks (Added Var & Ck), the added processes (Added Proc).

Table 1.5 IF Travel System Modifications According to Each Rule

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

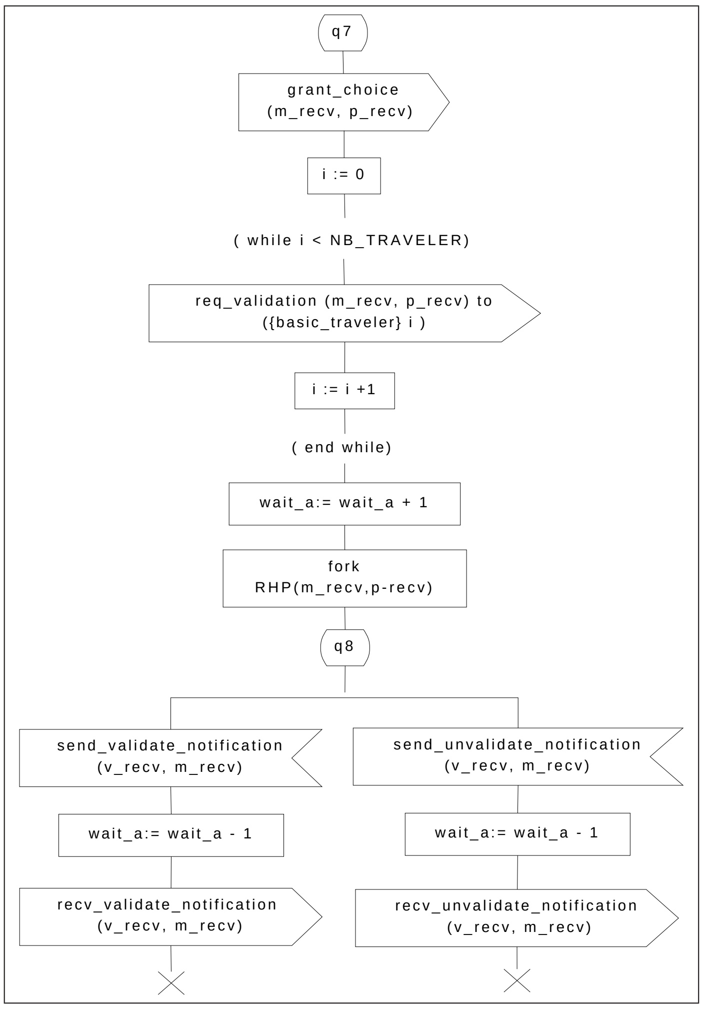
q7
grant_choice (m_recv, p_recv)
i := 0
(while i < NB_TRAVELER)
req_validation (m_recv, p_recv) to ({basic_traveler} i)
i := i +1
(end while)
wait_a:= wait_a + 1
fork RHP(m_recv,p-recv)
q8
send_validate_notification (v_recv, m_recv)
send_unvalidate_notification (v_recv, m_recv)
wait_a:= wait_a - 1
wait_a:= wait_a - 1
recv_validate_notification (v_recv, m_recv)
recv_unvalidate_notification (v_recv, m_recv)

Fig. 1.13 Resulting Transitions After Security Integration

As an example, let us consider the Travel specification part described in Figure 1.3. These transitions are modified during the integration of the third rule and leads to the creation of a new process forked in the transition triggered from state q_7. Figure 1.13 describes the resulting transitions. The new variable *wait_a* has 0 as a default value. If *wait_a* is positive, this means that the system is waiting for a mission validation. The process RHP (for Rule Handler Process) launches a clock in its first state. When the clock valuation reach the deadline of 7 days, the system verifies the *wait_a* value, if (*wait_a*¿0) the RHP process sends another validation request. Otherwise (*wait_a* ≤ 0 which means that the system got through one of the transitions of the state q_8 and already received an answer from the mission validator), the RHP process is stopped without performing any action.

1.6 Test Generation

1.6.1 *TestGen-IF tool*

To automatically generate test cases from the secure specification of Travel, we use the *TestGen-IF* test generation tool developed in our laboratory.

1.6.1.1 *Test Generation Algorithm*

TestGen-IF implements a timed test generation algorithm based on a Hit-or-Jump exploration strategy [Cavalli *et al.* (1999)]. This algorithm efficiently constructs test sequences with high fault coverage, avoiding the state explosion and deadlock problems encountered respectively in exhaustive or exclusively random searches. It allows to produce a partial accessibility graph of the system under test (SUT) specification in conjunction with the IF simulator [Bozga *et al.* (2004b)].

At any moment, a local search is conducted from the current state in a neighborhood of the reachability graph. If a state is reached, and one or more test purposes are satisfied (a Hit), the set of test purposes is updated and a new partial search is conducted from this state. Otherwise, if a search depth limit is reached without satisfying any test purpose, a partial search is performed from a random graph leaf (a Jump). This algorithm terminates when all the test purposes are satisfied or when no transition is left to explore. The test case is the path constructed on the fly from the initial state of the SUT specification containing all the hit and jump states.

1.6.1.2 *TestGen-IF Architecture*

The active testing tool is illustrated by Figure 1.14. The Properties (Test Purposes) box represents the timed system objectives to be tested (see Section 1.6.2.1). The Automatic Test Generation box represents the test generation procedure combined with the IF specification (.if file) and the test purposes (.tp file). It is up to the user to choose the exploration strategy of the generated partial graph he wants to perform during the test generation: in depth (DFS) or in breath (BFS) [Cavalli *et al.* (2006)]. During this generation, when a test purpose is satisfied, a message is displayed to inform the user. The number of test purposes already found and the number of those missing are also provided. Based on this approach, a test case is generated (represented by the Test Case box). A test suite is composed of a finite set of test cases (or scenarios) described in a standard format. It is used to stimulate the implementation under test (IUT) to validate its reaction.

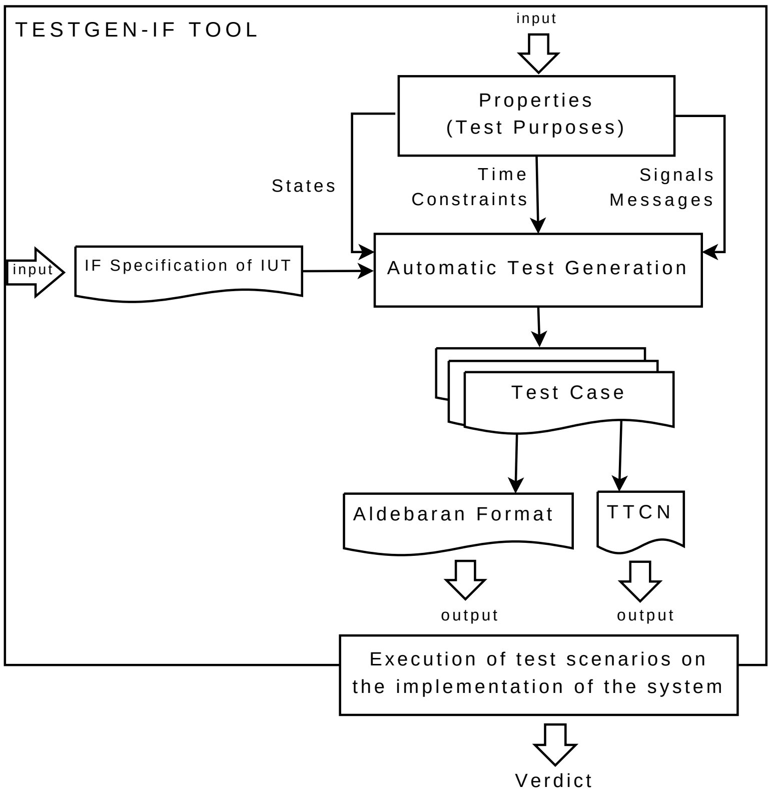

Fig. 1.14 Basic architecture of the TestGen-IF tool.

1.6.1.3 *TestGen-IF Implementation*

The *TestGen-IF* tool is based on the IF-2.0 simulator [Bozga *et al.* (2004b)] that allows to construct the accessibility graph from an IF specification. This simulator is developed by a research team at Verimag [Verimag Lab (2009)], for modeling and simulating asynchronous timed systems such as telecommunication protocols or distributed applications. *TestGen-IF* uses the IF-2.0 simulator libraries which provide some functionalities for on-the-fly state-space traversal. It is implemented in the same implementation language as the IF-2.0 simulator, i.e. C++ language.

As output of *TestGen-IF* tool, two files can be generated:

- The "*output.stat*" file containing statistics about the test generation process (number of jumps, visited states, generation duration, test case length, depth limit value, strategy exploration, etc.);
- And the "*output.sequence*" file (in Aldebaran [Fernandez *et al.* (1996)] or TTCN [J. Grabowski and Willcock (2003)] format) containing the timed test sequence. This last output is represented in Figure 1.14 by the Aldebaran format and TTCN boxes.

The test cases are generated from the output file "*output. sequence*" by filtering the generated test sequences according to the input actions, output actions and delays (i.e., progress of time) of the system under test. The test generation with *TestGen-IF* derives its benefits from *Hit-or-Jump* characteristics. It is faster that classical test generation tools and less memory consuming. In addition, it avoids the state explosion and deadlock problems.

1.6.2 *Fixing the Test Objectives*

1.6.2.1 *Test Purposes Formulation*

In order to formulate timed test purposes using TestGen-IF tool, several options are permitted:

- State constraint purposes: expressing that a system can be in a specific state;
- Action constraint purposes: corresponding, in particular, to signals actions (e.g., input signal, output signal) and describe that an action can be executed in a state (optionally) at a specific time;
- Clock constraint purposes: expressing that a clock can have a specific value, optionally in a specific state.

For instance, the constraint "$action = input\ sg\ in\ state = s\ when\ clock\ c = d$" describes that the signal sg is to be received in the state s when the clock $c = d$. Timeouts and deadlines can be usually described by clock constraint, whereas the flow requirements can be described using states and actions constraints.

1.6.2.2 *Test Purposes for the Travel Application*

The automatic test generation only targets security issues and, as a result, it is less time consuming. In this work we defined a set of test purposes describing security properties. In the following, we provide both informal and formal test purposes (according to the TestGen-IF formulation) relating to the rules described in section 1.5.4.

- Rule 1: A potential traveler wants to request for two missions. He/She is obliged to perform the two requests within a delay greater than 2 minutes. The timed test purposes of the rule 1 are formulated as:

$TP_1 = \{tp_1, tp_2\}$
  $tp_1 = \{$*signal= output "req_create_mission"*$\}$
  $tp_2 = \{$*signal= output "req_create_mission" when clock ck1 = 2*$\}$

- Rule 2: A potential traveler tries to choose his/her mission parameters (date, flight, hour, etc.) among a set of propositions provided by the Travel Web application. In the case he/she requests for more propositions, the system allows him/her to request a new list within a delay of 10 minutes. Otherwise, his/her session will expire. The timed test purposes of the rule 2 are formulated as:

$TP_2 = \{tp_1, tp_2, tp_3, tp_4\}$
  $tp_1 = \{$*signal= output "req_create_mission"*$\}$
  $tp_2 = \{$*signal= output "req_proposition_list"*$\}$
  $tp_3 = \{$*signal= input "recv_proposition_list"*$\}$
  $tp_4 = \{$*signal= informal "other_proposition_request"*$\}$

- Rule 3: Once a mission is created, it has to be validated by a specific user called validator. The system sends the mission parameters to the validator and waits for his/her acceptance/rejection. If the validator does not send any notification within a delay of 7 days, the system generates a validation request reminder. The timed test purposes of the rule 3 are formulated as:

$TP_3 = \{tp_1, tp_2, tp_3, tp_4\}$
  $tp_1 = \{$*signal= output "grant_create_mission"*$\}$
  $tp_2 = \{$*signal= input "req_proposition_list"*$\}$
  $tp_3 = \{$*signal= output "req_validation"*$\}$
  $tp_4 = \{$*signal= output "req_validation" in state relaunch*$\}$

1.6.3 *Test Generation with TestGen-IF*

Our objective is to automatically generate test sequences according to our test purposes. To reach this aim we used, for Travel test generation, two users (one being the validator) and two missions. We also defined adequate interval values for data variables in order to reduce the accessibility graph size and to avoid state explosion problems.

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. For instance, the filtered timed test case for the rule 3 is presented in the following page.

Table 1.6 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

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 TestGen-IF tool are abstract but usable; they are produced in aldebaran standard notation facilitating their portability and their automatic execution. Some metrics about this test generation relating to three rules are presented in Table 1.6.

TEST CASE FOR THE RULE 3

```
 (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}
```

1.7 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 conceived and implemented a tool called generaTCL to translate abstract test cases into executable one (in TCL scripts) and we connected this tool to tclwebtest framework [TclWebTest Tool (2009)] to apply them on Travel Web application.

1.7.1 *Tclwebtest tool*

Tclwebtest is a framework to build tests for Web applications. It provides an API for issuing http requests and processing results. It assumes specific response values, while taking care of the details such as redirects and cookies. It has the basic HTML parsing functionality to provide access to elements of the resulting HTML page that are needed for testing, mainly links and forms.

The execution of a test case written in *tclwebtest* simulates a user that is interacting with the Web application through a Web browser. By executing the instantiated test cases, it is possible to add, edit or delete data of the Web application, fill some forms or follow a specific link. Figure 1.15 illustrates the *tclwebtest* code for logging into the Travel application by requesting the register page, then filling the e-mail and password, and submitting this information.

```
::twt::do_request "http://tavel-example.org/register"
tclwebtest::form find ~n login
tclwebtest::field find ~n username
tclwebtest::field fill "user@mymail.com"
tclwebtest::field find ~n password
tclwebtest::field fill "mypassword"
tclwebtest::form submit
```

Fig. 1.15 Example of Tclwebtest Code.

1.7.2 *Test Cases Instantiation*

The test cases generated by TestGen-IF tool are provided in a standard notation and are a set of:

- Delays: a delay represents an amount of time that the tester has to wait for, before performing any input action.
- Input signals: the tester has to stimulate the Web application by applying a set of http(s) requests called inputs.
- Output signals: the tester has to access to the Web system answer to analyze it and check if it conforms to the expected reaction as described in the formal specification of the system.

1.7.2.1 *Delay Instantiation*

A delay in the test case can be easily translated in TCL script language. It is transformed directly into the code '*after n*' where *n* is the real delay in millisencond (ms). For instance, '*delay 10;*' is translated into '*after 10*1000*60;*' if we consider a delay of ten minutes. (See Algorithm 1.7, lines 3 to 5)

1.7.2.2 *Input Instantiation*

To automatically instantiate the abstract test cases provided by TestGen-IF tool, it is mandatory to know the types of HTML elements that correspond to input signals of the Web system under test (that correspond to the output signals of the tester). In this work, the Web system will be limited to receive just three types of inputs from a user via a regular browser: (i) a URL set in the address bar, (ii) a link in the body of the page or (iii) the submission of a form in the body of the page. Actions such as drag-and-drop and other Ajax functionalities are not considered in this work.

The first step of our methodology consists in mapping the signals into the three types of inputs that the Web application can receive. It is important to highlight that some IF signals can be mapped just to one single interaction with the Web application, e.g. following a link, but other signals are mapped to a set of interactions, e.g. submitting a form.

For example, considering the form submission, there are several interactions that must be performed by tclwebtest, i.e. filling text fields, selecting radio buttons, selecting checkboxes and finally submitting the form. In these cases the signal is mapped to an HTML element (e.g. a form) but also to each signal parameter (e.g. text fields).

To perform the mapping, we propose two tables containing required information of the input signals and their parameters to transform them into tclwebtest script asking to follow a link, submit a form or request a new URL. Both, the *signal_info_table* and the *parameter_info_table* are illustrated in Tables 1.7 and 1.8. To access the information they contain, we can take advantage of standard SQL queries. For each signal, the table *signal_info_table* stores the following data:

- *signal:* the name of the input signal in the IF specification,
- *html_element:* the HTML type of the element that correspond to the signal,
- *html_name:* the name or id of the HTML element. For example, in the case of a link, the name is the link caption.

Table 1.7 Signal_Info_Table Example

signal	**html_element**	**html_name**
req_connected	form	login
req_disconnect	link	logout

Then, for each parameter of an input signal, the information stored in this table 1.8 is:

- *parameter:* name of the parameter signal in the IF specification,
- *of_signal:* name of the input signal that uses this variable,
- *html_element:* the HTML type of the element that correspond to the parameter,
- *type:* the type of the variable expected by the html_element, e.g. integer, string, etc.
- *html_name:* the name or id of the html_element.

Table 1.8 Parameter_Info_Table Example

parameter	**of_signal**	**html_element**	**type**	**html_name**
user	req_connected	textfield	integer	username
password	req_connected	texfield	integer	pass

The second step of the methodology is to translate the system inputs to tclwebtest script for each test case. These inputs are built dynamically and can be divided into three categories: following a link, submitting a form or setting a URL in the browser. The inputs translation methodology is presented in pseudo-code in Algorithm 1.7 (lines 6 to 36). By performing this algorithm the following parts of the test case will be built:

- The script of the test preamble: a sequence of inputs (operations) that will lead the system to a state where the test case can be executed. During this preamble, system outputs are not analyzed. For example, to test the creation of a mission, the user needs to be authenticated by the Travel system.
- The script that will stimulate the system to test it.

Algorithm 1.7 Instantiation Methodology

Require: An abstract test case TC, signal_info_table and parameter_info_table tables. Let *act* be a delay or an observable action in TC and let $sg(d_1, d_2, ..., d_k)$ be a signal instance of $sg(x_1, x_2, ..., x_k)$. d_j is denoted in_j if sg is an input signal and out_j if sg is an output signal. ($0 < j < k+1$)

```
 1: for each ($act_i \in TC$) do
 2:     /*(where $i \in N$, $0 < i < n+1$ such that $n$ is the number of actions and delays in $TC$)*/
 3:     while ($act_i$ = delay n) do
 4:         tcl_script: after n;
 5:     end while
 6:     while ($act_i$ = input $sg_i(in_1, in_2, ..., in_k)$) do
 7:         if (html_element($sg_i$)= url) then
 8:             tcl_script: do request url;
 9:         end if
10:         if (html_element($sg_i$)= link) then
11:             tcl_script: follow link;
12:         end if
13:         if (html_element($sg_i$)= form) then
14:             tcl_script: form find ˜n html_name($sg_i$);
15:             for (each parameter $x_j$ of $sg_i$) do
16:                 /*(where $j \in N$, $0 < j < k+1$)*/
17:                 tcl_script: field find ˜n html_name($x_i$);
18:                 while (html_element($x_j$) = textfield) do
19:                     tcl_script: field fill $in_j$;
20:                 end while
21:                 while (html_element($x_j$) = textarea) do
22:                     tcl_script: field fill $in_j$;
23:                 end while
24:                 while (html_element($x_j$) = checkbox) do
25:                     if ($in_j$ = 1) then
26:                         tcl_script: field check html_name($x_j$);
27:                     else
28:                         tcl_script: field uncheck html_name($x_j$);
29:                     end if
30:                 end while
31:                 while (html_element($x_j$) = radiobutton) do
32:                     tcl_script: field select $in_j$;
33:                 end while
34:             end for
35:             tcl_script: submit form;
36:         end if
37:     end while
38:     while ($act_i$ = output $sg_i(out_1, out_2, ..., out_k)$) do
39:         if (html_element($sg_i$)= link) then
40:             tcl_script: assert {[response url] == html_name($sg_i$)};
41:         end if
42:         if (html_element($sg_i$)= form) then
43:             tcl_script: response body;
44:             tcl_script: form find ˜n html_name($sg_i$);
45:             for each (parameter $x_j$ of $sg_i$) do
46:                 tcl_script: assert {[field get_value find ˜n html_name ($x_j$)]== $out_j$} ;
47:             end for
48:         end if
49:         call deduce_verdict procedure;
50:     end while
51: end for
```

1.7.2.3 *System Output Instantiation*

The last step of the methodology consists in developing the scripts that analyzes the response (or reaction) of the Web application (Algorithm 1.7, lines 37 to 49). This script also assigns the verdict (pass or fail). Basically it checks whether the platform did what it was supposed to do. The system outputs (or reaction) can be classified into two categories:

- Observable outputs: Tclwebtest is basically dedicated for testing Web application interfaces accessible by Web user. It offers some basic HTML parsing functionalities and commands for the manipulation of the HTML elements of Web pages (in our case, on consider system output pages). The reaction of the system can be provided in one or many HTML pages of the Web-based system. In general, it is a notification message that stipulates that the desired action succeeded or failed, for instance an authentication. Sometimes, this reaction can be more difficult to discover such as when reloading the current page or navigating to another Web application page. In all cases, we need to define the two tables 1.7 and 1.8 for the output as well as their parameters and follow the same methodology developed in the input instantiation case. To analyze system Web pages, we use *response* and *find* commands to locate the HTML elements we want to test. Then we rely on *assert* command to compare the displayed Web page values and the output signal parameters and deduce the adequate verdict.
- Non observable outputs: the system may react to a user operation by performing an action which is non-observable from this user's point of view (and as a result of the tester). For example, we can consider the adding/edition/deleting of information in a specific data base or the sending a notification email to a specific user. In these cases, no automatic solution has yet been elaborated.

In the Travel case study, all our test cases consider observable system reactions that can be defined automatically.

1.7.2.4 *Test Cases Instantiation Tool: generaTCL*

The *GeneraTCL* tool, illustrated in the Figure 1.16, is used to concretize the abstract test cases translating them into an executable script able to interact with the IUT. In the concretization process, some details of the implementation (as the username and password of a real user) are added to the abstract test cases. These details are needed to perform the interaction tester-IUT.

1.7.3 *Test Cases Execution*

The test cases execution was performed on a prototype implementation of the Travel Web application (developed on OpenACS platform) to verify that the specified security requirements are respected. It is important to highlight that some time settings in this prototype have been 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 avoid waiting so long. Therefore in this case study we verify the behavior of the system concerning this rule using a delay of 3 minutes rather than using 7 days.

The execution of the test cases is performed using a dedicated testing tool proposed by the OpenACS community [OpenACS Community (2009)]. This tool is called the ACS-Automated-Testing tool that allows executing the instantiated test cases, interacting with the Web-based application under test and, also, displaying the verdict of each test case.

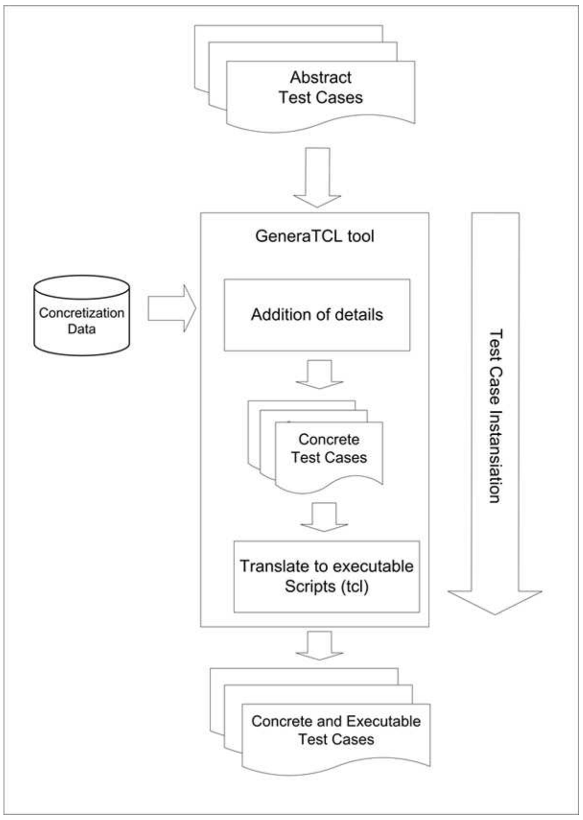

Fig. 1.16 Basic Architecture of the GeneraTCL tool

The ACS-Automated-Testing tool is, in itself, a Web application but we will refer to it just as *the tester* to avoid confusions 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 obtained positive verdicts for thirty test objectives, while, four test objectives were violated (fail verdict). For example, a problem has been detected according to the system respect to the first rule that expresses a prohibition. If a potential traveler requests for a first mission and then waits for 2 minutes, he/she is not allowed by the system to request for another mission. We analyzed the implementation of the Web-based system and noticed that a mistake was encrusted 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 four error sources. Once the mistakes corrected, all the test cases were applied again on the Web application. This time, all the verdicts were positive which demonstrates the efficiency of our methodology.

1.8 Conclusion

In this chapter, we have presented a formal approach to integrate timed security rules, expressed according to Nomad language, into a TEFSM specification of a system. Roughly speaking, a security rule denotes the prohibition, the obligation or the permission for the system to perform an action in a specific timed context. To meet this objective, we have distinguished two categories of security rules: basic rules and elaborated rules. To deal with basic rules, we have described a set of algorithms that allows to add them to a TEFSM specification describing the behavior aspect of a system. Then, we have defined a rewritten process that permits to transform an elaborated rule into one or several basic ones one which the previous integration algorithm can be reused. A proof that demonstrates the correctness of the prohibition integration algorithm is given.

Notice that our approach can be improved by minimizing the number of introduced clocks. Indeed, some dependent rules can be integrated simultaneously by using a unique clock. For instance, rules $\mathscr{P}(start(A)|O^{-5}done(B))$ and $\mathscr{P}(start(A)|O^{-3}done(B))$ can be integrated by adding a single clock Ck that the system checks whether its valuation verifies $(Ck = 3 \vee Ck = 5)$ before performing action A.

Indeed, we presented a framework for the modeling and the testing of Web applications from their security point of view. Our approach consists in automatically integrating security rules described in using the Nomad formal language within an IF specification. This integration leads to an IF secure specification that takes the system security requirements into account. Afterward, we presented an approach to derive test cases from this IF secure specification using TestGen-IF tool developed in our laboratory and to transform them into executable test cases (using TCL script language). We applied the generated test cases to an industrial Web-based system provided by France Telecom to study its respects to its security policy. Relying on our end-to-end framework, we discovered several security flaws that we were able to correct obtaining thus a secure Web system.

As future work, we want to extend the test purposes formulation by adding data constraints and complex clock constraints to express more elaborated test objectives. We also intend to adapt test generation algorithms to include these new test purposes types. In addition, we want to investigate the automatic analysis of non-observable system reactions in the context a white box testing [Tuya *et al.* (2008)].

1.9 Acknowledgements

The research leading to these results has received funding from the European Community Seventh Framework Programme (FP7/2007-2013) under grant agreement no 215995 and the French ANR project WebMov.

Bibliography

Abou El Kalam, A., Baida, R. E., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miège, A., Saurel, C. and Trouessin, G. (2003). Organization Based Access Control, in *4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy'03)* (Lake Come, Italy).

Alur, R. and Dill, D. L. (1994). A theory of timed automata, *Theoretical Computer Science* **126**, 2, pp. 183–235, URL `citeseer.ist.psu.edu/alur94theory.html`.

Annichini, A., Bouajjani, A. and Sighireanu, M. (2001). TReX: A Tool for Reachability Analysis of Complex Systems, in G. Berry, H. Comon and A. Finkel (eds.), *CAV*, *Lecture Notes in Computer Science*, Vol. 2102 (Springer), ISBN 3-540-42345-1, pp. 368–372.

Bozga, M., Fernandez, J.-C., Ghirvu, L., Graf, S., Krimm, J.-P., Mounier, L. and Sifakis, J. (1999). IF: An Intermediate Representation for SDL and its Applications, in *SDL Forum*, pp. 423–440.

Bozga, M., Graf, S. and Mounier, L. (2002). IF-2.0: A Validation Environment for Component-Based Real-Time Systems, in *CAV*, pp. 343–348.

Bozga, M., Graf, S., Mounier, L. and Ober, I. (2004a). IF Validation Environment Tutorial, in *SPIN*, pp. 306–307.

Bozga, M., Graf, S., Ober, I., Ober, I. and Sifakis, J. (2004b). The IF Toolset, in M. Bernardo and F. Corradini (eds.), *SFM*, *Lecture Notes in Computer Science*, Vol. 3185 (Springer), ISBN 3-540-23068-8, pp. 237–267.

Cavalli, A., Lee, D., Rinderknecht, C. and Zadi, F. (1999). Hit-or-Jump: An Algorithm for Embedded Testing with Applications to IN Services, in *Formal Methods for Protocol Engineering And Distributed Systems* (Beijing, China), pp. 41–56.

Cavalli, A. R., Maag, S., Mallouli, W., Marche, M. and Quemener, Y.-M. (2006). Application of Two Test Generation Tools to an Industrial Case Study. in *TestCom*, pp. 134–148.

Cavarra, Crichton, Davies, Hartman, Jeron and Mounier (2002). Using UML for Automatic Test Generation, in TACAS.

Cuppens, F., Cuppens-Boulahia, N. and Ghorbel, M. B. (2006). High-Level Conflict Management Strategies in Advanced Access Control Models, in *Workshop on Information and Computer Security (ICS)* (Timisoara, Roumania).

Cuppens, F., Cuppens-Boulahia, N. and Sans, T. (2005). Nomad: A Security Model with Non Atomic Actions and Deadlines, in *CSFW*, pp. 186–196.

Damianou, N., Dulay, N., Lupu, E. and Sloman, M. (2001). The Ponder Policy Specification Language, in *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks* (Springer-Verlag, London, UK), ISBN 3-540-41610-2, pp. 18–38.

Fernandez, J.-C., Garavel, H., Kerbat, A., R. Mateescu, L. M. and Sighireanu, M. (1996). CADP: A Protocol Validation and Verification Toolbox, in R. Alur and T. A. Henzinger (eds.), *The 8th Conference on Computer-Aided Verification, CAV'96* (Springer Verlag, New Jersey, USA).

Gallardo, M. D. M., Merino, P. and Pimentel, E. (2004). A Generalized Semantics of PROMELA for Abstract Model Checking, *Formal Asp. Comput.* **16**, 3, pp. 166–193.

Gaudin, E., Najm, E. and Reed, R. (eds.) (2007). *SDL 2007: Design for Dependable Systems, 13th International SDL Forum, Paris, France, September 18-21, 2007, Proceedings*, *Lecture Notes in Computer Science*, Vol. 4745 (Springer), ISBN 978-3-540-74983-7.

ISO (1989). *Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, International Standard IS-880.

J. Grabowski, G. R. I. S. A. W., D. Hogrefe and Willcock, C. (2003). An Introduction to The Testing and Test Control Notation (TTCN-3), in *Computer Networks 42(3)*, pp. 375–403.

Jard, C. and Jéron, T. (2005). TGV: Theory, Principles and Algorithms, *STTT* **7**, 4, pp. 297–315.

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

Lobo, J., Bhatia, R. and Naqvi, S. A. (1999). A Policy Description Language, in *AAAI/IAAI*, pp. 291–298.

Mallouli, W. and Cavalli, A. R. (2007). Testing Security Rules with Decomposable Activities, in *the 10th IEEE International Symposium on High Assurance Systems Engineering (HASE)* (Dallas, Texas, USA), pp. 149–155.

Mallouli, W., Orset, J.-M., Cavalli, A., Cuppens, N. and Cuppens, F. (2007). A Formal Approach for Testing Security Rules. in *SACMAT* (Nice, France).

Merayo, M. G., Núñez, M. and Rodríguez, I. (2007). Generation of Optimal Finite Test Suites for Timed Systems, in *TASE*, pp. 149–158.

Merayo, M. G., Núñez, M. and Rodríguez, I. (2008). Formal Testing from Timed Finite State Machines, *Computer Networks* **52**, 2, pp. 432–460.

OpenACS Community (2009). http://www.openacs.org/, .

Syriani, J. A. and Mansour, N. (2003). Modeling Web Systems Using SDL, in A. Yazici and C. Sener (eds.), *ISCIS*, *Lecture Notes in Computer Science*, Vol. 2869 (Springer), ISBN 3-540-20409-1, pp. 1019–1026.

TCL Script Language (2009). http://www.tcl.tk/, .

TclWebTest Tool (2009). http://tclwebtest.sourceforge.net/, .

Tuya, J., Dolado, J. J., Cabal, M. J. S. and Riva, C. D. L. (2008). A Controlled Experiment on White-Box Database Testing, *ACM SIGSOFT Software Engineering Notes* **11**, 1.

Verimag Lab (2009). http://www-verimag.imag.fr/ ˜ async/if/, .

Vieira, E. R. and Cavalli, A. (2007). Toward Test Suite Automatic Generation with Delayable Transitions and Timing-Fault Detection, in *RTCSA*.