Modeling Web Applications by the Multiple Levels of Integrity Policy

G. Amato [1]

Dipartimento di Scienze
Università degli Studi "G. d'Annunzio", Italy

M. Coppola, S. Gnesi [2]

Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
CNR Pisa, Italy

F. Scozzari, L. Semini [3]

Dipartimento di Informatica
Università di Pisa, Italy

Abstract

We propose a formal method to validate the reliability of a web application, by modeling interactions among its constituent objects. Modeling exploits the recent "Multiple Levels of Integrity" mechanism which allows objects with dynamically changing reliability to cooperate within the application. The novelty of the method is the ability to describe systems where objects can modify their own integrity level, and react to such changes in other objects. The model is formalized with a process algebra, properties are expressed using the ACTL temporal logic, and can be verified by means of a model checker. Any instance of the above model inherits both the established properties and the proof techniques. To substantiate our proposal we consider several case-studies of web applications, showing how to express specific useful properties, and their validation schemata. Examples range from on-line travel agencies, inverted Turing test to detect malicious web-bots, to content cross-validation in peer to peer systems.

Key words: Formal Methods, Model Checking, Process Algebra, Temporal Logic.

[1] Email: amato@sci.unich.it
[2] Email: {coppola,gnesi}@isti.cnr.it
[3] Email: {scozzari,semini}@di.unipi.it

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

1 Introduction

Formal methods are increasingly being used to validate the design of distributed systems and have already proved successful in specifying commercial and safety-critical software and in verifying protocol standards and hardware design [4,8]. It is increasingly accepted that the adoption of formal methods in the life cycle development of systems would guarantee higher levels of dependability and greatly increase the understanding of a system by revealing, right from the earliest phases of the software development, inconsistencies, ambiguities and incompletenesses, which could cause subsequent faults. In particular model checking techniques [6,7] have emerged as successful formal verification techniques. They have been defined to automatically check the truth of system properties, expressed as temporal logic formulae, on the finite state model representing the behavior of a system. Model checkers can easily be used by non–expert users too. For this reason model checking has often been preferred in industries to other verification tools, and many efficient verification environments are currently available, based on model checking algorithms [5,11,15].

In the last few years distributed applications over the WEB have gained wider popularity. Several systems have led to an increasing demand of evolutionary paradigms to design and control the development of applications over the WEB. The main advantages of exploiting the WEB as underlying platform can be summarized as follows. The WEB provides uniform mechanisms to handle computing problems which involve a large number of heterogeneous components that are physically distributed and (inter)operate autonomously. Conceptually, WEB services are stand-alone components that reside over the nodes of the network. Each WEB service has an interface which is network accessible through standard network protocols and describes the interaction capabilities of the service. Applications over the WEB are developed by combining and integrating together WEB services. Web applications show the same verification problems of classical distributed systems. We may hence extend techniques and tool used for their verification also in the case of Web applications.

The formalization framework that we propose in this paper is based on some results presented in [14], where the formal validation of an interaction policy between communicating objects was carried out. The policy is the Multiple Levels of Integrity policy, defined in the context of the design of fault tolerant systems to enhance systems dependability. The original definition of the policy simply consists of a set of declarative rules: it can be operationally realized defining a communication protocol. The protocol which carries out the integrity policy is formally specified as a collection of interacting processes in a process algebra. We consider specific interaction patterns, which subsume the most complex interaction schemata, and check on them temporal logic formulae expressing the non-violation of integrity rules.

2 The Multiple Levels of Integrity policy

Integrity policies are defined in the field of fault tolerant systems. The design of fault tolerant systems usually includes the modeling of faults and failures or the definition of fault tolerant schemata. At the software architecture level, a fault tolerant schema usually describes a set of components and their interactions. A component that is part of a fault tolerant system is said to be *critical* if its failure can seriously affect the reliability of the overall system. Fault tolerant schemata, and in particular integrity policies, are defined to prevent failure propagation from non critical to critical components. An integrity policy assigns a *level of integrity*, ranging over a finite set of natural values, to each system component, and states the communication patterns. Components that may be critical are assigned a high integrity level.

The *Multiple Levels of Integrity* policy has been defined within an object–oriented framework, to provide flexible fault tolerant schemata. Instead of forbidding data flow from low level to high level objects, this policy permits some objects to receive low level data, by decreasing their integrity level. The policy is based on the following concepts:

Integrity levels (il) range from 0, the lowest, to 3, the highest. Data are assigned the integrity level of the object which produced them.

Single Level Objects (SLO) are objects whose integrity level does not change during computations. Consequently, an SLO of level n is only allowed to receive data from objects of level $\geq n$.

Multiple Level Objects (MLO) are the core of the policy: their integrity level can be dynamically modified, since they are allowed to receive low level data. To this purpose, an MLO is assigned three values:

maxil which represents the maximum integrity level that the MLO can have. It is also called the *intrinsic level* of the MLO, since it is assigned during the design of the application. It is a static value.

minil which represents the minimum value the integrity level of the MLO can reach while interacting with other objects. It is set at invocation time, on the bases of the invocation level. No memory of it is kept after the answer to the invocation is returned: *minil* is local to an invocation.

il which is the current integrity level. It is set at invocation time to a value ranging between *maxil* and *minil* and decreases if lower level data are received during the computation to serve the invocation. Also *il* is local to each invocation.

The policy requires a new MLO instance to be created every time the MLO is invoked. As a consequence, an MLO cannot be used to implement a component which has to store some data. This means that an MLO, from a functional point of view, is a stateless object: only SLOs can store data. In Fig. 1, we provide an example of the evolution of an MLO in response to an invocation: when an MLO with $maxil = 3$ receives a read request of level 1, it

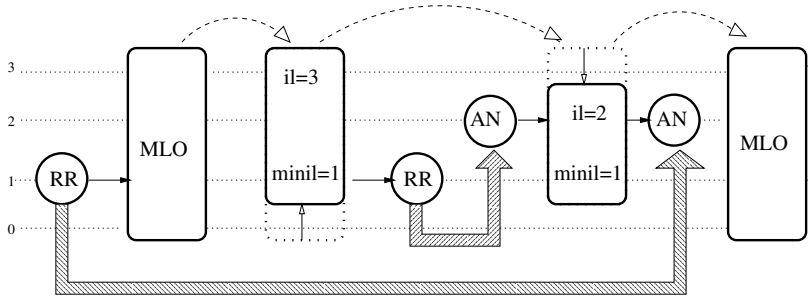

Fig. 1. Behaviour of an MLO: dotted arrows follow the MLO's evolution, thick arrows bind requests to the corresponding answers.

sets its *minil*: no answer with integrity level smaller than 1 can be returned. The value of *il* equals *maxil*: a read request does not corrupt the integrity level of the MLO. Suppose the MLO needs to delegate part of the answer construction, sending another read request to a third object. The level assigned to the request equals *minil*: an answer to this request is accepted if greater or equal to *minil*. Since the integrity level of the answer is 2, the MLO can accept it but *il* is decreased to level 2. Finally, an answer to the first request is provided, whose level equals the current *il*, and the MLO restores its initial state.

Validation Objects (VO) are used to extract reliable data from low level objects and to provide information at a fixed level of integrity. In real systems, it is sometimes necessary to get data from unreliable sources, such as sensors, and use them in critical tasks. However, this use could either lower the level of the entire system or violate the integrity policy. Validation Objects represent a safe way to upgrade the integrity level of these data. An example of Validation Object is the one that uses a redundant number of data sources, and filters them with appropriate algorithms. For instance, a voting policy can be used. These policies are well known in the literature, in particular in the distributed fault tolerant community. Among them, we recall the solutions to the Byzantine Generals problem [16], where an agreement among multiple nodes is sought in the presence of faults. To validate a voting algorithm we can apply the results presented in [2].

A set of **rules** is given, describing all the possible communication patterns among pairs of objects, depending on the respective integrity levels. We list them in Table 1: we call A and B the invoking and the invoked objects, respectively. The first part of the table considers invocation conditions. The invocation is refused if the specified condition is not satisfied. If it is accepted, the invoked object (if an MLO) might have to change its integrity level, as shown in the second part of the table, where invocation effects are considered. In the case of read or read–write invocation, an answer is returned at the end of the method execution. If the invoking object was an MLO, then the returned data may decrease its integrity level as follows: $il(A) := min(il(A), il(B))$.

The **communication model** is based on the notion of method invocation.

Conditions	$A\&B$ SLOs	A SLO, B MLO	A MLO, B SLO	$A\&B$ MLOs
A reads B	$il(A) \leq il(B)$	$il(A) \leq maxil(B)$	$minil(A) \leq il(B)$	$minil(A) \leq maxil(B)$
A writes B	$il(B) \leq il(A)$	$always$	$il(B) \leq il(A)$	$always$
A r-w B	$il(A) = il(B)$	$il(A) \leq maxil(B)$	$minil(A) \leq il(B) \leq il(A)$	$minil(A) \leq maxil(B)$

Effect	A SLO, B MLO	$A\&B$ MLOs
A reads B	$minil(B) := il(A);$ $il(B) := maxil(B)$	$minil(B) := minil(A);$ $il(B) := maxil(B)$
A writes B	$il(B) := min(il(A), maxil(B))$	$il(B) := min(il(A), maxil(B))$
A r-w B	$minil(B), il(B) := il(A)$	$minil(B) := minil(A);$ $il(B) := min(il(A), maxil(B))$

Table 1
Conditions to be satisfied for a method invocation to be accepted, and the effect on the level of objects after acceptance.

Method invocations are assigned an integrity level too. In particular, *read*, *write* and *read–write requests* are considered as abstractions of any method, with respect to the effect on the state of objects. The level of a write request corresponds to the level of the data which are written, the level of a read request corresponds to the minimum acceptable level of the data to be read. Read–write requests are assigned two integrity levels, one for read and one for write.

3 Formal Validation Methodology

The Multiple Levels of Integrity policy has been validated according to the following steps. We follow the same methodology to validate the case studies.

- Formal specification of the mechanism using the CCS process algebra [17]. Process algebras are based on a simple syntax and are provided with a rigorous semantics defined in terms of Labeled Transition Systems (LTSs). Process algebras are well suited to describing interaction policies, since a policy definition abstracts from the functionalities of the objects, and the relevant events to be specified are the object invocations (the actions) which may change the object integrity level (the state). In Table 2 we present the subset of the CCS operators used in the following.
- Use of the ACTL temporal logic [10] to describe the desired properties. ACTL is a branching-time temporal logic whose interpretation domains are LTSs. It is the action based version of CTL [13] and is well suited to expressing the properties of a system in terms of the actions it performs. We use a fragment of ACTL, given by the following grammar, where ϕ denotes a state property:

$a : P$	Action prefix	Action a is performed, and then process P is executed. Action a is in Act_τ
$P + Q$	Nondeterministic choice	Alternative choice between the behaviour of process P and that of process Q
$P \parallel Q$	Parallel composition	Interleaved executions of processes P and Q. The two processes synchronize on complementary input and output actions (i.e. actions with the same name but a different suffix)
$P \setminus a$	Action restriction	The action a can only be performed within a synchronization
$P = P'$	Process definition	It includes recursion

Table 2
A fragment of CCS syntax

$$\phi ::= true \mid \phi \,\&\, \phi' \mid [\mu]\phi \mid AG\,\phi \mid A[\phi\{\mu\}U\{\mu'\}\phi']$$

In the above rules μ is an action formula defined by:

$$\mu ::= true \mid a \mid \mu \vee \mu \mid \sim\mu \qquad \text{for } a \in Act$$

We provide here an informal description of the semantics of ACTL operators. The formal semantics is given in [10]. Any state satisfies *true*. A state satisfies ϕ & ϕ' if and only if it satisfies both ϕ and ϕ'. A state satisfies $[a]\phi$ if for all next states reachable with a, ϕ is true. The meaning of $AG\,\phi$ is that ϕ is true now and *always* in the future.

A state P satisfies $A[\phi\{\mu\}U\{\mu'\}\phi']$ if and only if in each path exiting from P, μ' will eventually be executed. It is also required that ϕ' holds after μ', and all the intermediate states satisfy ϕ; finally, before μ' only μ or τ actions can be executed. A useful formula is $A[\phi\{true\}U\{\mu'\}\phi']$ where the first action formula is *true*: this means that any action can be executed before μ'.

– Generation of the (finite state) model. To this end, we use the tools of the JACK (*Just Another Concurrency Kit*) verification environment [3], which is based on the use of process algebras, LTSs, and temporal logic formalisms, and supports many phases of the systems development process.
– Model checking of the ACTL formulae against the model, using the model checker for ACTL available in JACK, FMC.

3.1 Validation of the Multiple Levels of Integrity policy

The Multiple Levels of Integrity policy has to guarantee that the interaction among different components does not affect the overall confidence of the application, i.e., that a non–critical component does not corrupt a critical one. In particular, data of a low integrity level cannot flow to a higher integrity level

(unless through a Validation Object). This condition should hold for isolated objects and in any schema of interaction among objects. In [14], the following properties have been validated:

(i) An object with intrinsic level i cannot provide answers of level $j > i$.

(ii) An object with intrinsic level i does not accept read requests of level $j > i$.

(iii) If an MLO with intrinsic level i receives a read request of level $j \leq i$, and, to serve the request, it invokes with a read request a third object of intrinsic level *maxil* smaller than j, then it cannot answer the initial request. Indeed, its level is decreased to the *maxil* value of the third object because of the new data received.

(iv) If an MLO with intrinsic level i receives a read request of level $j \leq i$, and then a write request of level $k < j$, then it can still answer the read request. In other words, its level is not decreased by the concurrent invocation.

4 A concept of interface

The model proposed in [14] assumes that all the components of a system and their relationships are known. This assumption cannot be satisfied in the case of web site specification, since in most cases we only analyze a piece of the system, while of the remaining components we only know the interface toward the components of interest. We therefore need to define a formal concept of interface for a component of a system expressed in the CCS process algebra. This is accomplished using the restriction operator together with a dummy process which simulates the rest of the world. To be more precise, let P be a process over the set of actions Act_P. We could image to have a process W describing the rest of the world, thus we would like to verify the overall system $P \parallel W$. Of course, this is not possible, since we cannot specify all the possible components. Actually, since we are not interested in other communications than those among our process P and the rest of the world, we can restrict ourselves to study the process $(P \parallel W) \setminus (Act_W \smallsetminus Act_P)$, where Act_W is the set of actions of W. But this is equivalent to considering the process $P \parallel (W \setminus (Act_W \smallsetminus Act_P))$. Our idea is to consider, instead of the process $W \setminus (Act_W \smallsetminus Act_P)$ its interface toward P. To this aim, we need to introduce the notion of *dummy process*, that we use to separate the proper interface of W we are interested in. Let $D_{W,P}$ be the dummy process

$$\mathtt{D_{W,P}} = \mathtt{a_1} : \mathtt{D_{W,P}} + \mathtt{a_2} : \mathtt{D_{W,P}} + \ldots + \mathtt{a_n} : \mathtt{D_{W,P}} \tag{1}$$

where $\{a_1, \ldots, a_n\} = Act_W \setminus Act_P$. We define the *interface of W w.r.t. P* the process $W_P = (W||D_{W,P}) \setminus (Act_W \smallsetminus Act_P)$. Actually, for our purpose, any process trace-equivalent to W_P would suffice, that is any process which exhibits the same behaviour w.r.t. P when we observe only the traces of the

system. In the following, we call *interface* any of such processes. Thus, given any interface I of W w.r.t. P, we simply consider the system $P||I$.

For example, given $\mathtt{P} = ?\mathtt{request} :!\mathtt{ask\_google} :?\mathtt{read\_google} :!\mathtt{reply}$ we do not want to observe the interaction of the Google web site with the rest of the world, then we may choose $?\mathtt{ask\_google}$ and $!\mathtt{read\_google}$ as the only actions we are interested in, and which should be described in the interface of Google.

Our aim is to verify ACTL formulas on processes defined by CCS agents. Since we adopt the above concept of interface, we are particularly interested in those formulas such that, once proved for $P \parallel I$, where I is any interface of W w.r.t. P, they also hold for $P||W$. It is well–known that every formula in ACTL which does not contain any existential path quantifier E and negation $\sim$, enjoys the above property, since we can observe only the traces of the system. This motivates our choice of the ACTL fragment, as presented in Section 3.

5 Case Study: the Travel Agency

Our first case study concerns the modeling and analyzing of the architecture of the subset of the Web, which is of interest for a user willing to organize a travel by booking flights and hotels. The user interacts with an on–line travel agency. The travel agency, in turn, accesses the web sites of air companies, tour operators, single hotels as well as other travel agencies specialized in hotel booking, and so on. Here, we consider a simplified scenario, with two reliable sites, namely those of the travel agency and the air company Alitalia[4], and a fairly reliable hotel booking site.

We model Alitalia and the travel agency as MLOs with *maxil* 3, called $\mathtt{ALITALIA_3}$ and $\mathtt{TA_3}$, respectively, and the hotel booking site as HOTELSEEK, an MLO with *maxil* 2. All these sites are supposed to interact and receive data from many distinguished other components. We want them to perform their job even if the received data have a lower integrity level. At the same time, we recall that MLOs cannot store data: we can imagine that these components interact with some private SLOs, where to store the information of any finalized reservation. To exemplify this, we specify the SLOs of the travel agency, and call them disks. Since the travel agency is an MLO of level 3, it may be at any level when accessing its disks with a write request. We hence introduce 4 disks, one for each integrity level. They are specified parametrically by the process $\mathtt{DISK_x}$. We also need a disk manager, specified as an MLO of level 3, in order to choose the right $\mathtt{DISK_x}$ according to the integrity level.

[4] Disclaimer: The company names and the integrity level we use, are freely introduced for the purposes of the example, and have no correspondence with the reliability of the actual sites, when they exists.

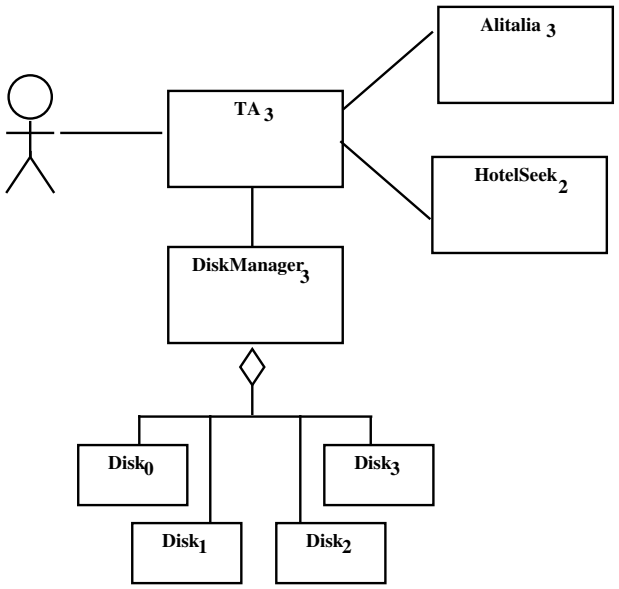

Fig. 2. The travel agency architecture.

The architecture of the resulting system is described in Figure 2. The full specification is given below, by instantiating the process defined in [14]. A disk can receive a read request when the travel agency needs to access previous reservations. $\texttt{read\_request}_{\texttt{x}}$ is a read request action of level `x`. In general, this means that the invocation was issued either by an SLO with `x` as *il* or by an MLO with `x` as *minil*. A disk can receive a write request too, when the travel agency needs to store a new reservation. Only requests at level `x` are served. A write request at a different level will be served by another disk.

```
DISK_MANAGER(3) =
     ?read_data(y). !read_disk(y).!answer_data(y).DISK_MANAGER(3) +
     ?write_data(y).!write_disk(y).DISK_MANAGER(3)

DISK_0 = ?read_disk(0).DISK_0 +
         ?write_disk(0).DISK_0
DISK_1 = ?read_disk(1).DISK_1 +
         ?write_disk(1).DISK_1
DISK_2 = ?read_disk(2).DISK_2 +
         ?write_disk(2).DISK_2
DISK_3 = ?read_disk(3).DISK_3 +
         ?write_disk(3).DISK_3
```

The agent `HOTELSEEK` accepts read–write hotel reservation requests, and write–only confirmation requests. $\texttt{r\_w\_hreserve}_{\texttt{y,z}}$ denotes a request issued either by an MLO with `y` as *minil* and `z` as *il* or by an SLO with $il = \texttt{y} = \texttt{z}$. Variable `y` denotes the read level of the request, variable `z` denotes the write level. $\texttt{w\_confirm}_{\texttt{y}}$ denotes a write request of level `y`, issued by an object with `y` as *il*. Hotel reservation requests are served as specified by process `HOTEL_RES`.

```
HOTELSEEK(2) = ?r_w_hreserve(y,z).!hotel.(
```

```
                    ( [y <= z] [z <= 2] HOTEL_RES(y,z,2) ) +
                    (   [y <= 2] [2 <= z] HOTEL_RES(y,2,2) ) +
                    (   [y > 2] !answer_hres(-1). HOTELSEEK(2) ) ) +
                ?w_confirm(y).  HOTELSEEK(2)

HOTEL_RES(min,il,max) =
      ( [min <= 0] [0 <= il]  !answer_hres(0). HOTELSEEK(2) )  +
      ( [min <= 1] [1 <= il]  !answer_hres(1). HOTELSEEK(2) )  +
      ( [min <= 2] [2 <= il]  !answer_hres(2). HOTELSEEK(2) )  +
      ( [min <= 3] [3 <= il]  !answer_hres(3). HOTELSEEK(2) )  +
      !answer_hres(-1). HOTELSEEK(2)
```

The Alitalia specification is very simple. A web site such as the Alitalia one can be implemented using a groupware protocol. These protocols address, among others, the concurrency control problems that arise in systems with multiple users (namely, groupware systems [1,12]) whose actions may be conflicting. A typical example is to reserve the same seat to two or more users that are concurrently booking a flight. The high integrity level of the Alitalia site can be guaranteed by formally specifying the protocol and by proving the non interference properties of interest. Validation can be done by model checking using, for instance, the results given in [18] where some properties of a public subscribe groupware protocol have been proved.

```
ALITALIA(3) = ?r_w_freserve(y,z). !flight.
                      [y <= z] !answer_fres(z). ALITALIA(3) +
                ?w_confirm(y). ALITALIA(3)
```

Finally, the travel agency.

```
TA(3) = ?r_w_booktravel(y,z). [y <= z] TA_BOOK(y,z,3) +
        ?r_infotravel(y). TA_INFO(y,3,3)

TA_BOOK(min,il,3) = F_BOOK(min,il,3) +
                    H_BOOK(min,il,3) +
                    F_BOOK(min,il,3).H_BOOK(min,il,3)

F_BOOK(min,il,3) = !r_w_freserve(min,il). ?answer_fres(x).
                      ( [x < min] !answer_booktravel(-1). TA(3) +
                        [min <= x] [x <= il] TA_FINALIZE(min,x,3) +
                        [il <= x]  TA_FINALIZE(min,il,3) )

H_BOOK(min,il,3) = !r_w_hreserve(min,il). ?answer_hres(x).
                      ( [x < min] !answer_booktravel(-1). TA(3) +
                        [min <= x] [x <= il] TA_FINALIZE(min,x,3) +
                        [il <= x]  TA_FINALIZE(min,il,3) )

TA_FINALIZE(min,il,3) = !write_data(il). !w_confirm(il).
                        !answer_booktravel(il). TA(3)

TA_INFO(min,3,3) = !read_data(min). ?answer_data(x).
```

```
                    ( [x < min] !answer_info(-1). TA(3) +
                      [x >= min] !answer_info(x). TA(3) )
```

We also need to specify a generic user of the system, which can ask for information or book a travel.

```
User(x) = !info. ( ( !r_infotravel(x). ?answer_info(y).
                   (  ( [y < 0  ] !failure. User(x) ) +
                      ( [y >= 0 ] !success. User(x) ) ) ) +
          !book. ( !r_w_booktravel(0,x). ?answer_booktravel(y).
                      (  ( [y < 0  ] !failure. User(x) ) +
                      ( [y >= 0 ] !success. User(x) ) ) )
```

In our test, we use a generic process consisting of the travel agency, the air company, the hotel booking site, a generic user of level 2 and the disks.

```
( HOTELSEEK(2) || ALITALIA(3) || TA(3) || User(2) || DISK_MANAGER(3)
  || DISK_0 || DISK_1 || DISK_2 || DISK_3 ) \read_data \answer_data
 \write_data \answer_hres \r_w_hreserve \w_confirm \r_w_freserve
 \answer_fres \r_w_booktravel \r_infotravel \answer_booktravel
 \answer_info \read_disk \write_disk
```

The only non restricted actions are `info`, `book`, `hotel` and `flight`. Therefore we will use them when specifying the ACTL formula. As a first example, we want to prove that, if a client requires a booking service (action !`book`), the travel agency will either book an hotel (action !`hotel`) or a flight (action !`flight`) before any positive answer (action !`success`). Formally, we require to verify the following ACTL formula:

```
AG [ !book ] A [ true { ~ !success } U { !hotel | !flight } true ]
```

The result of the model checker is that the formula is true and that 153 states has been observed. The following formula:

```
AG [ !info ] A [ true { ~ !success } U { !hotel | !flight } true ]
```

states that at any request of information will follow the booking of an hotel or a flight. Of course, in this case the result of the model checker is that the formula is false.

6 Case Study: Peer to Peer Validation Service

We here describe a peer to peer architecture that a user can query to download a video. This is a simplified instance of the concrete problem of identifying remote file content before downloading in peer to peer systems, where some or all of the peers are untrusted, or content-based access control has to be enforced. In the example we assume that two peers exist, at level 1 and 2 respectively. Moreover, the system includes two refutation lists which collect information of help to know whether the expected content of a file corresponds to the file name. The download is filtered by a Validation object that first looks for the video with a `read_video` request, and then validates the answer by querying the refutation lists. The system is described in Figure 3.

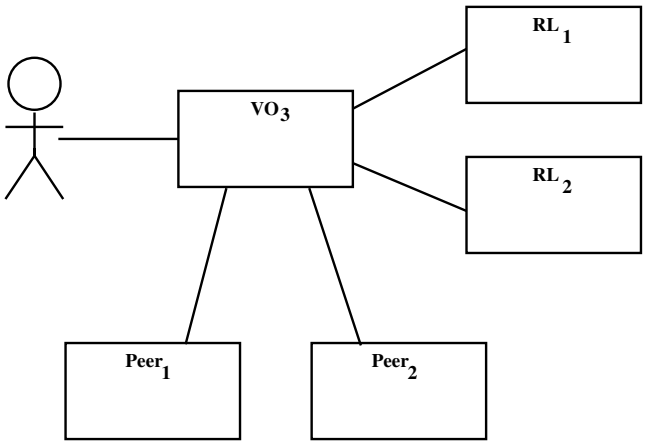

Fig. 3. The Peer to Peer Validation Service architecture.

A peer's answer to a `read_video` request carries two values: the peer integrity level, and an integer holding -1 if the peer does not have the video, a different value otherwise. If the video is not found from both peers `P`, the validator `VO` sends a negative answer to the user, otherwise it validates the video content with the help of the clauses of the agents `VAL` and `VAL2`. This involves querying one or more of the refutation lists processes `RL`.

In the example, we abstract from actual validation algorithms in `VAL` and `VAL2`, and show a completely non-deterministic behaviour that can be refined in any concrete solution. Our validation approach is compositional: to prove the correctness of the final system, we only need to validate the refinement step. Indeed, the abstract behaviour of the `VO` specified here corresponds to the interface of any actual validation object, with a specific validation algorithm demanded to the `VAL` agent.

To complete the example description, we assume that peers perform a visible action `video` when the video is available, and the user performs the visible actions `start` at search beginning, then `success`, or `failure`. The last two actions discriminate the cases where a valid video was found from the cases where either no video was found, or the video content was not correct.

```
P(x) = ?read_video(y). ( ( [y <= x] (!video. !answer_video(x,x). P(x) +
                                     !answer_video(x,-1). P(x) )       ) +
                         ( [y > x] !answer_video(-1,-1). P(x)          ) )

RL(x) = ?query_video(y). ( ( [y <= x]  !query_answer(x). RL(x)  ) +
                           ( [y > x]   !query_answer(-1). RL(x) )  )

VO(x) = ?user_req(y). ( ( [y <= x]  !read_video(0). ?answer_video(z,w).
                          ( ( [z = -1]  !user_answer(-1). VO(x) ) +
                            ( [z >=0]    VAL(x,w)               )  ) +
                      ( [y > x]   !user_answer(-1).VO(x)            )  )  )

VAL(x,w) = [w = -1] !user_answer(-1). VO(x) +
           [w >= 0] ( !query_video(0). ?query_answer(y).
```

```
                                        ( !user_answer(x). VO(x)  +
                                          !user_answer(-1). VO(x) +
                                           VAL2(x)                )  )

VAL2(x) = !query_video(0). ?query_answer(y). ( !user_answer(x). VO(x) +
                                               !user_answer(-1). VO(x)  )

User(x) = !start. !user_req(x). ?user_answer(y).
              ( ( [y < 0  ] !failure. User(x) ) +
                ( [y >= 0 ] !success. User(x) )  )

net Net = ( VO(3) || P(1) || P(2) || RL(1) || RL(2) || User(1) )
        \read_video \query_video \user_req \answer_video \query_answer
        \user_answer
```

The validation process has lead to the generation of a model with 524 states, against which the following properties have been checked, returning the expected results.

```
AG [ !start ] A [ true { ~ !start } U { !failure | !video } true ]
--        The formula is TRUE              --

AG [ !start ] A [ true { true } U { !video } true ]
--        The formula is FALSE             --
```

7 The inverted Turing Test

The Inverted Turing test, proposed by Watt[19] as an alternative to the conventional Turing Test for artificial intelligence, requires:

- to put a system in the role of the observer;
- the observer to discriminate between humans and machines.

The machine that wants to mimic humans should show naive psychology, that faculty which predisposes us to anthropomorphism and enables us to ascribe intelligence to others. An example test is the one that asks many questions like "how close is a building to a house, how close is a hotel to a house, how close is a lodge to a house, how close is a cavern to a house", with answers in a finite range, say 1–100. The observer compares the answers to a table obtained by making the same questions to a sufficiently large population.

A variant of the Inverted Turing Test is the Editing Test [9], often used to discriminate humans from machines when assigning a new e-mail address. It is based on the so-called interpretative asymmetry, that is the asymmetry of the skillful way in which humans "repair" deficiencies in speech, written texts, handwriting, etc., and the failure of computers to achieve the same interpretative competence. For instance, an optical sequence of characters like the one in Figure 4 is printed on the screen, and the observed entity is asked to type the characters with the keyboard.

Fig. 4. The editing test: only humans are supposed to read the sequence of characters.

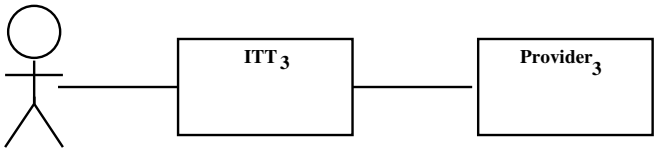

Fig. 5. The architecture of the subset of the WEB including an Inverted Turing test.

The component implementing the Inverted Turing test can be modeled in our framework as a validation object. The architecture of the subset of the WEB of interest can be modeled as described in Figure 5: the validation object intercepts the interactions between the entity (human or machine) asking for an e-mail address.

8 Conclusion

We have proposed a formal method to describe web applications by means of a process algebra which can be automatically verified by a model checker. By considering a fragment of the ACTL logic which does not contain negation and existential path quantification, we can introduce a formal notion of interface which allows us to prove properties expressed by temporal formulae in a modular way. We exploit the notion of validation object proposed in fault tolerant system verification and show examples of web applications where validation objects play a fundamental role. We describe in details two case studies validating some formulae with the help of the FMC model checker. Moreover, we briefly sketch another example where a commonly used web application can be easily modeled as a validation object. As a future work, we intend to investigate the possibility to separately verify different parts of the system and to compose the results.

References

[1] Baecker, R., editor, "Readings in Groupware and Computer Supported Cooperation Work–Assisting Human-Human Collaboration," 1992.

[2] Bernardeschi, C., A. Fantechi and S. Gnesi, *Formal validation of fault–tolerance*

mechanisms inside Guards, Reliability, Engineering and System Safety (RE&SS) **71** (2001), pp. 261–270, elsevier.

[3] Bouali, A., S. Gnesi and S. Larosa, *JACK: Just another concurrency kit*, Bulletin of the European Association for Theoretical Computer Science **54** (1994), pp. 207–224.

[4] Bowen, J. and M. Hinchey, *Seven more myths of formal methods*, IEEE Software **12** (1995), pp. 34–41.

[5] Burch, J., E.M.Clarke, K. McMillan, D. Dill and J. Hwang., *Symbolic Model Checking* 10^{20} *states and beyond*, in: *Proceedings of Symposium on Logics in Computer Science*, 1990.

[6] Clarke, E., E. Emerson and A. Sistla, *Automatic Verification of Finite–State Concurrent Systems Using Temporal Logic Specification*, ACM Transaction on Programming Languages and Systems **8** (1986), pp. 244–263.

[7] Clarke, E., O. Grumberg and D.Peled, "Model Checking," MIT Press, 1999.

[8] Clarke, E. and J. Wing, *Formal methods: state of the Art and Future Directions*, ACM Computing Surveys **28** (1996), pp. 627–643.

[9] Collins, H., *The editing test for the deep problem of AI*, Psycoloquy **8** (1997).

[10] De Nicola, R. and F. Vaandrager, *Action versus State based Logics for Transition Systems*, in: *Proceedings Ecole de Printemps on Semantics of Concurrency*, Lecture Notes in Computer Science **469** (1990), pp. 407–419.

[11] Dill, D., A. Drexler, A. Hu and C. H. Yang, *Protocol Verification as a Hardware Design Aid*, in: *IEEE International Conference on Computer Design: VLSI in Computers and Processors* (1992), pp. 522–525.

[12] Ellis, C. and S. Gibbs, *Concurrency control in groupware systems*, in: *In Proc. SIGMOD'89* (1989), pp. 399–407.

[13] Emerson, E. and J. Halpern, *Sometimes and Not Never Revisited: on Branching Time versus Linear Time Temporal Logic*, Journal of ACM **33** (1986), pp. 151–178.

[14] Fantechi, A., S. Gnesi and L. Semini, *Formal Description and Validation for an Integrity Policy Supporting Multiple Levels of Criticality*, in: C. Weinstock and J. Rushby, editors, *Proc. DCCA–7, Seventh IFIP International Conference on Dependable Computing for Critical Applications* (1999), pp. 129–146.

[15] Holzmann, G., *The Model Checker SPIN*, IEEE Transaction on Software Engineering **5** (1997), pp. 279–295.

[16] Lamport, L., R. Shostack and M. Pease, *The Byzantine Generals problem*, ACM Transactions on Programming Languages and Systems **4** (1982), pp. 282–401.

[17] Milner, R., "A Calculus of Communicating Systems," Lecture Notes in Computer Science **92**, Springer-Verlag, Berlin, 1980.

[18] ter Beek, M., M. Massink, D. Latella and S. Gnesi, *Model checking groupware protocols.*, in: F. Darses, R. Dieng, C. Simone and M. Zacklad, editors, *Cooperative Systems Design - Scenario-Based Design of Collaborative Systems*, Frontiers in Artificial Intelligence and Applications **107** (2004), pp. 179–194.

[19] Watt, S., *Naive-psychology and the inverted turing test*, Psycoloquy **7** (1996).