


ITC 3/47 Journal of Information Technology and Control Vol. 47 / No. 3 / 2018 pp. 393-405 DOI 10.5755/j01.itc.47.3.20330 © Kaunas University of Technology	Towards the Formal Development of Software Based Systems: Access Control System as a Case Study	
	Received 2018/03/11	Accepted after revision 2018/08/16
	 http://dx.doi.org/10.5755/j01.itc.47.3.20330	

Towards the Formal Development of Software Based Systems: Access Control System as a Case Study

Ammar Boucherit

Faculty of Sciences, University of Ferhat Abbas, Setif, Algeria

Department of Computer Science, University of Echahid Hamma Lakhdar, El-Oued, Algeria

Laura M. Castro

Department of Computer Science, Models and Applications of Distributed Systems Group,
Á Coruña University, Spain

Abdallah Khababa

Faculty of Sciences, University of Ferhat Abbas, Setif, Algeria

Osman Hasan

School of Electrical Engineering and Computer Science, National University of Sciences and Technology,
Islamabad, Pakistan

Corresponding author: ammar-boucherit@univ-el-oued.dz

Our daily life is increasingly becoming more and more dependent on software as they are being extensively used to control safety and mission-critical systems. This has led to very stringent verification requirements for ensuring that the software performs as intended. However, the testing based techniques cannot provide a rigorous verification due to limited computational and memory constraints and traditional formal verification techniques, like model checking and theorem proving, are not too straightforward to work with in the industrial setting. In this paper, as a first step to overcome these limitations, we describe a hybrid property based testing and model checking based technique for verifying both models and implementation of access control systems. Our approach addresses the model checking of critical properties of access control systems and aims at improving their reliability by using property based testing to analyze the corresponding software code. For illustration purposes, a simple example of an access control system is used.

KEYWORDS: Access Control System, Model-Checking, Petri nets, Property-Based Testing, Rewriting Logic.

1. Introduction

In today's world, software based systems take over more of our lives and, it becomes hard and strange to find a person or a company that doesn't use computers and software in its daily activities. Such software based systems made our daily life more easier and comfortable till the extent that we could not imagine our world as it is now; functions without software based systems. For instance, such systems are now widely present in our environment for many uses at home, bank, hospital and even for the so-called safety critical systems. Consequently, this variety of software based systems implies different quality levels that are directly proportional to the need for safety and reliability for each systems or application domain. Therefore, the use of adequate design approaches as well as rigorous analysis techniques is gaining more and more importance. Access control systems [6, 40] — as example of software based systems — are one of the most popular crucial assets of security that play a crucial role in computer security, industrial research and in finance. In addition, access control systems (ACS) are mostly used as a measure to restrict the access to sensitive data, specific areas, personal account information as well as to protect valuables. At the same time, the access control policies play an important role in distributed systems, such as the case of organizational collaboration, in order to prevent the organization's shared resources (machines and servers) from any unauthorized use or access [49]. The presence of ACS in our daily life brings forth many software security related challenges to protect both individual privacy and enterprise property through commerce and industry. This need is of paramount importance especially when it concerns safety and mission critical systems. Therefore, according to the desired security level, many techniques, such as passwords, signature or biometric access control have been used. In addition, it is essential not only to rigorously analyze these techniques to ensure that they are meeting the required security and reliability constraints but also to confirm the absence of any violations in the underlying software implementation. Traditionally, testing [30] is the oldest method and the dominant mean by which most software based systems are verified. However, testing does not allow the complete correctness of software based systems and thus cannot be used alone to rigorously ensure reliability and security aspects of ACS,

due to their complex nature and thus a huge number of possible cases. Formal methods such as model checking [50, 25], can overcome these limitations but at the cost of verification experts and a significant amount of verification effort and time. These costs make pure formal verification based analysis quite infeasible for the industry where time-to-market is very critical and human resources are usually scarce.

In our opinion, these challenges can be catered for by using a variety of verification techniques depending on the different development stages. Therefore, in this paper, we propose a novel hybrid approach combining both model checking and Property-Based Testing techniques (PBT) for verifying models and implementation of software based systems. In fact, this proposition is the enhancement, extension and combination of our previous works [2, 9], where we have used such techniques separately. The proposed approach uses Petri nets [38], which is a powerful formalism, in the modeling stage and rewriting logic [31] that represents an expressive universal logic, to give formal semantics to Petri nets models in the specification stage. Then, the Maude model checker [19] is used to check critical properties of ACS, and finally property-based testing [20, 21] as an automatic testing technique based on random generation of test sequences (i.e. test scenarios) to raise the confidence level on a given system realization.

The rest of this paper is organized as follows. Section 2 briefly describes some elementary concepts related to the presented work. Then, a literature survey is presented in Section 3. In Section 4, we present a general description about access control systems. Thereafter, we illustrate our approach by using a simple case study in Section 5 to show its feasibility and validity. The contributions of the presented approach are discussed in Section 6. Finally, Section 7 concludes the paper with a summary of the present work.

2. Preliminaries

In this section, we present the main concepts related to the rest of the paper, such as rewriting logic, Petri nets, model checking and property based testing. More details about these topics can be accessed from [34, 14, 24, 46].

2.1. Rewriting Logic and Petri Nets

Rewriting logic [31] mainly extends the equational algebraic specifications with “rewrite rules” to deal with changes, dynamics and concurrencies. In rewriting logic, concurrent systems models can be specified easily by using rewrite theory \mathcal{T} . A rewrite theory is a 4-tuple $\mathcal{T} = (\Sigma, E, L, R)$ composed from the equational theory (Σ, E) that defines the structural part of the system, L is a set of labels, and R is a set of possibly conditional rewrite rules that represent changes in the system behavior. Rewriting logic is also known as a logic of change and a unifying semantic framework in which Petri net [42] and a very wide range of concurrency models and logics can be represented [32, 47, 48, 43, 45].

Petri nets were originally introduced in 1962 by Carl Adam Petri [35] and they are still considered as very useful and reliable tool for modeling and verifying discrete and dynamic systems.

Definition 1. A Petri net is a 5-tuple $N = (P, T, F, W, M_0)$ where:

- _ P : a finite set of places connected with,
- _ T : a finite set of transitions where $P \cap T = \emptyset$,
- _ F : the flow relation that relates places and transitions by arcs, such that $F \subseteq (P \times T) \cup (T \times P)$, and
- _ $W : F \rightarrow \mathbb{N}$ is a weight function that assigns weights to each arc,
- _ $M_0 : P \rightarrow \mathbb{N}$ is the initial marking.

So, a Petri net consists of a set of places, transitions and directed arcs connecting places with transitions. Places may contain different number of tokens, which are represented by black dots. A transition has generally input places that are connected with directed arcs from each input place to the transition, and output places are connected with the transition by arcs starting from the transition. A transition is enabled if its input places contain sufficient tokens. Thereafter, the enabled transition may be unconditionally fired. The specification language chosen for our approach is Maude which is based on a rewriting logic. Maude specifications can be written using two different modules: functional modules (start with the keyword `fmod` ... `endfm`) and system modules (keyword `mod` ... `endm`).

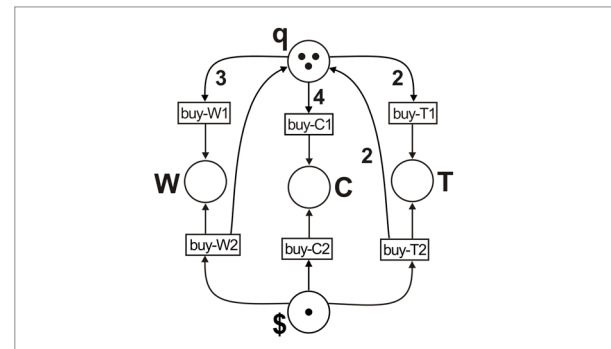
Functional modules are based on membership equational logic [33] to define basic sorts with operations on them, a set of terms with operations on them by means of equational theories, and memberships between terms and sorts.

System modules are very general rewrite theories that may have equations in addition to rewrite rules to define the dynamic part of systems. The outcome of the system module is obtained via highly divergent rewriting paths based on rewriting logic deduction rules. In this context, Petri nets have been successfully modeled in Maude as given in [42, 31]. In addition, we suggest an object oriented specification for Petri nets in order to allow designers associating data to Petri net tokens such as token type, number and color. Moreover, by adding such information in the specification of Petri nets, designers will be able to use inhibitor arcs and therefore test for zero or testing Petri nets boundedness can be implemented. This is illustrated through the following example.

Example 1. This example represents a simple vending machine to buy coffee, tea and water with a cost of 1 dollar, 2 and 3 quarters, respectively. This machine only accepts dollars and quarters. Figure 1 represents the Petri net describing the behaviour of such a machine.

Figure 1

Petri net describing the vending machine behaviour



The corresponding object oriented specification of such a machine is given in the following module:

```

mod Vending-Machine is
pr INT .
inc CONFIGURATION .
sorts service coin type .
subsorts service coin < type . op PLACE :
-> Cid .
ops tea coffe water : -> service .
ops dolar quarter : -> coin .
op tokentype :_ : type -> Attribute
[gather(&)] .
op tokennumber :_ : Int -> Attribute
[gather(&)] .

```

```

ops $ q t w c : -> Oid .
op Initial : -> Configuration .
vars x y z : Int .
crl [buy-c2] : < $ : PLACE | tokentype :
dolar , tokennumber : x > < c : PLACE |
tokentype : coffe , tokennumber : y > => < $
: PLACE | tokentype : dolar , tokennumber
: x - 1 > < c : PLACE | tokentype : coffe ,
tokennumber : y + 1 > if (x > 0) .
crl [buy-t2] : < $ : PLACE | tokentype :
dolar , tokennumber : x > < t : PLACE |
tokentype : tea , tokennumber : y > < q :
PLACE | tokentype : quarter , tokennumber
: z > => < $ : PLACE | tokentype : dolar
, tokennumber : x - 1 > < t : PLACE |
tokentype : tea , tokennumber : y + 1 > < q :
PLACE | tokentype : quarter , tokennumber
: z + 2 > if (x > 0) .
crl [buy-w2] : < $ : PLACE | tokentype :
dolar , tokennumber : x > < w : PLACE |
tokentype : water , tokennumber : y > < q :
PLACE | tokentype : quarter , tokennumber
: z > => < $ : PLACE | tokentype : dolar
, tokennumber : x - 1 > < w : PLACE |
tokentype : water , tokennumber : y + 1
> < q : PLACE | tokentype : quarter ,
tokennumber : z + 1 > if (x > 0) .
crl [buy-c1] : < q : PLACE | tokentype :
quarter , tokennumber : x > < c : PLACE |
tokentype : coffe , tokennumber : y > => < q
: PLACE | tokentype : quarter , tokennumber
: x - 4 > < c : PLACE | tokentype : coffe ,
tokennumber : y + 1 > if (x >= 4) .
crl [buy-t1] : < q : PLACE | tokentype :
quarter , tokennumber : x > < t : PLACE |
tokentype : tea , tokennumber : y > => < q :
PLACE | tokentype : quarter , tokennumber
: x - 2 > < t : PLACE | tokentype : tea ,
tokennumber : y + 1 > if (x >= 2) .
crl [buy-w1] : < q : PLACE | tokentype :
quarter , tokennumber : x > < w : PLACE |
tokentype : water , tokennumber : y > => < q
: PLACE | tokentype : quarter , tokennumber
: x - 3 > < w : PLACE | tokentype : water ,
tokennumber : y + 1 > if (x >= 3) .
endm

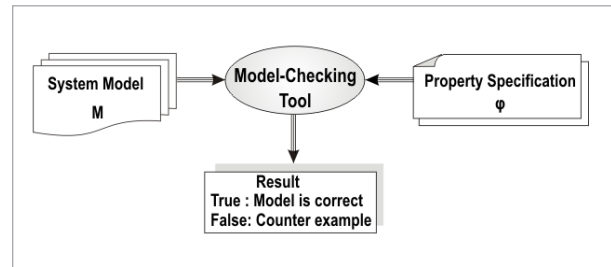
```

2.2. Model-Checking

Model checking is a formal verification technique that is particularly well suited to prove the correctness of a finite state system by analyzing its design [13, 24]. As shown in Figure 2, it generally requires the use of formal specification languages to develop a description (model) for the system under study and the specifi-

cation of property to be verified. Model checking explores all possible states of a system (or model) based on the combination of system input(s) and state(s) to determine whether or not a specified set of properties is true. If a property is not true, the model checker will produce a counter example execution trace to show why the property does not hold.

Figure 2
Model Checking Technique



The main idea behind the model checking technique—automata-theoretic approach—is to construct a Kripke structure K that is equivalent to the system model M and Büchi automaton that is equivalent to the negation of the property $B_{\neg\phi}$. Then, another Büchi automaton B' has to be constructed from K and $B_{\neg\phi}$ where: $L(B') = L(K) \cap L(B_{\neg\phi})$. Therefore, two cases are possible:

- If $L(B') = \emptyset$, then $M \models \phi$. (property ϕ is correct in M).
- Else, a counter-example is given to show where the property ϕ does not hold in M .

This principle can be expressed briefly as follows:

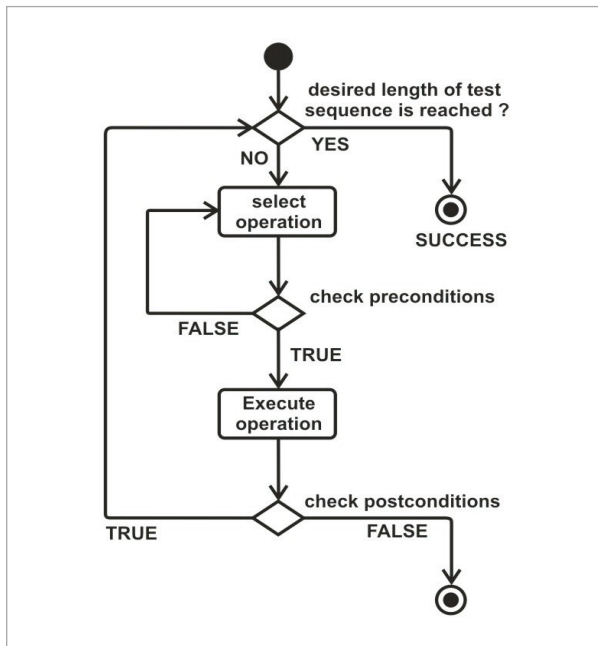
$$M \models \phi \Leftrightarrow K \models \phi \Leftrightarrow L(K) \subseteq L(\phi) \Leftrightarrow L(K) \cap L(\phi) = \emptyset \Leftrightarrow L(K) \cap L(\neg\phi) = \emptyset$$

2.3. Property-Based Testing

Even when model checking has been performed on the model of a system, its full realization needs to be tested and the developers need to manually design and implement a suite of test cases, which represent specific scenarios of interaction with the system under test (SUT). However, this technique often ignores some bugs that eventually surface during system operation. There, there is a dire need for some automated testing techniques. The Property-Based Testing (PBT) falls in this category, with further differentiation being possible on the basis of whether it is used before or after the real implementation.

Figure 3

Property-Based Testing principle



As shown in Figure 3, in Property-Based Testing, the developer has to write a universally quantified expression characterizing the behaviour of a SUT functionality. Therefore, instead of choosing specific input data, the PBT tool uses data generators to randomly produce acceptable input data, running as many concrete scenarios as desired by the developer, and diagnosing each of those scenarios offering counter examples for test failures.

The interest on PBT and PBT tools is growing rapidly, as the effort required by the developer, compared to the amount of tests that can be run in the same time, make PBT a very attractive and cost-efficient option. To date, and to the best of our knowledge, Quviq QuickCheck [37] is the most advanced and powerful PBT tool. It has been successfully used in research and industry to test complex, critical distributed and concurrent systems, implemented in Erlang and otherwise [39, 3, 10, 11].

3. Related Work

The ultimate objective of software development is to produce correct software that satisfies its require-

ments. Therefore, verification and testing are of crucial importance for software quality assurance. In the context of ACS, many techniques, like testing [36, 30] and model checking [28, 50, 25], may be used to evaluate the correctness of a software model and implementation. For instance, Schaad et al. [41] used Alloy as a modeling formalism and its analyzer as a checking tool. However, since Alloy has no built-in temporal reasoning, it is hard to code the system states and the transition relations explicitly. As results, Alloy models are too complex and their verification is usually quite inefficient. In the context of testing ACS, structural and behavioral models of Role Based Access Control (RBAC) policy specification have been developed in [30]. Then, a model-based strategy is proposed for the automated generation of a test suite from the prepared specification for testing implementations of access control systems using a well-known algorithm [12]. However, testing cannot be used to rigorously verify the reliability and security aspects of ACS, due to their complex nature and thus a huge number of possible cases. In order to overcome the shortcomings of testing, several formal specification approaches [44, 8, 27, 26, 7] have been proposed. The Z language has been used as a model-based specification language to represent the software specifications formally using mathematical models. Moreover, model checking has also been used for software testing in order to improve the quality of software specifications [1, 22]. The counter examples generated by a model checker are used to create test cases.

Based on the above-mentioned literature review, we can see that despite many attempts to ensure the safe development of software based systems, we cannot ensure a complete analysis in term of covering all development stages. However, the good news is that many of the explored techniques are complementary in nature and thus developing hybrid approaches, using these existing techniques, can certainly raise the level of completeness of verification for software systems.

In our proposed approach, the verification of Petri net model by using Maude model checker is very easy, effective and advantageous since rewriting logic on which Maude is based, is a very expressive logic allowing the formal specification of wide range of Petri nets and permitting to designers associating data to the Petri net tokens. In addition, the state-space

explosion problem in model checking technique can be somewhat overcome by the use of Maude Linear Temporal Logic of Rewriting Model Checker (LTLR) [4] that supports on-the-fly explicit-state model checking of concurrent systems.

Similarly, Maude allows using another complementary technique, i.e., a reachability-analysis tool that can explicitly check the existence/absence of critical states in the system from a pre-defined initial state, which is provided by experts in the field of study. In fact, this tool offers a very quick verification time if the best initial state is used.

4. Access Control Systems

In computer security, the term “access control” refers to the mechanisms of controlling users or processes to: 1) Perform functions included in their authorized level and restrict them from performing other unauthorized functions or 2) Access to specific resources (e.g., shared devices, networks, files, computers, databases), confidential information, or top secret information about national security, protective force and nuclear material control ... etc. However, it usually also refers in practice to door access control that is the process and mechanisms by which the entrance to a server room, building, a parking garage, or any other sensitive area is managed.

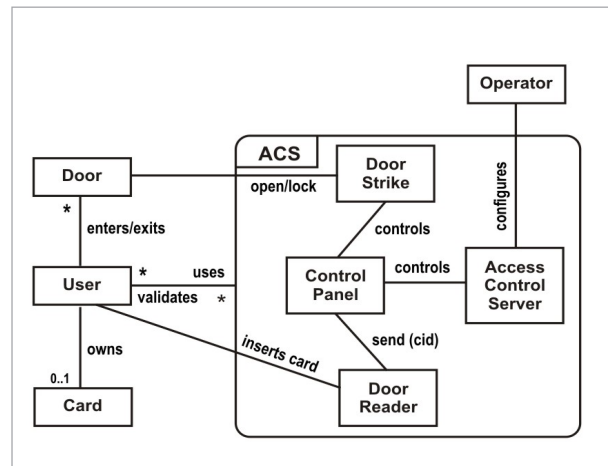
Depending to the system, the first way of access control can be achieved by a human (a guard, policeman or security agent) through mechanical means, such as locks and keys. Nevertheless, this is an extremely risky strategy since it is not possible to: (a) prohibit authorized person from duplicating keys, (b) restrict the key holder to specific times or dates, and (c) know who had entered, left and how long time they had passed in the monitored area. Instead of locks and keys, the second alternative to regulate access control and provide better security is the electronic access control systems. This new strategy can unlock doors for a predetermined time and records all failed and successful accesses. In addition, if an access card is lost, the card can easily be deactivated, replaced or removed from the list. Moreover, such systems can optionally sense the activity of forcefully a door and monitoring the entrance if it is held open for a long time after being unlocked.

4.1. System Architecture

A typical door access control system, depicted in Figure 4, consists of five main components:

- 1 Identity credential:** It may be released by a physical object (such as a key, keycard, or fingerprint) or secret information (such as a keycode or password), or combination of both that is presented to the door reader.
- 2 Door reader or keypad:** It is used to receive the information presented by the credential.
- 3 Door strike:** It is designed to hold a door closed until receiving an authorized entry request from the control panel.
- 4 Control panel:** It must be connected to the door reader, the door strike and the access control server. Its role is to authenticate the identification information and even make the final access authorization in simple ACS.
- 5 Access control server:** It is a server with registered card holder accounts and access information. It performs the verification and allows administrators to set or change access levels for each ID credential and door, view reports, and conduct audits to see who used a door at a certain time.

Figure 4
Access Control System architecture



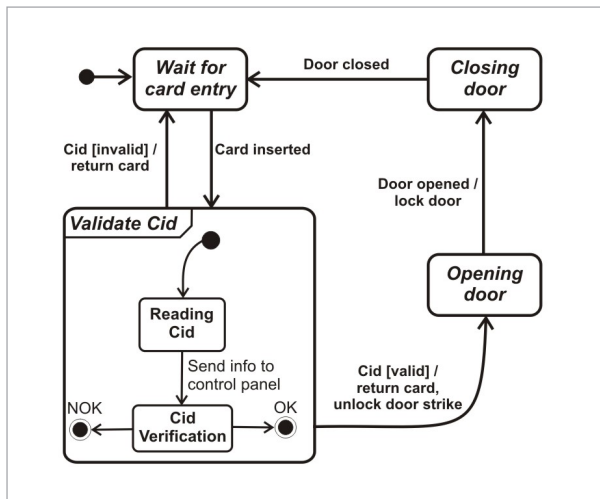
4.2. Principle

The main purpose of a door access control system is to control access of users to areas and resources in

access zones on the basis of the user identity and access rights associated with each user. Thus, only authorized users will be allowed to enter into an access zone but the other users will be denied. The process of authentication may be established by means of a card code and/or a password, PIN, entered by the user. When a user is authenticated and authorized, the door will be opened after sending a signal from the control panel to the door strike. However, if he is not authenticated, the door will remain locked.

The state machine diagram in Figure 5 illustrates this principle and describes the behaviour of the control panel at the general level. In this diagram, the state “Validate Cid” is a composite state and when the state machine enters this state it will be automatically activated. In a real scenario, a simple algorithm based on a white/black list could be used, or something more complex, like inspecting the list of other recent accesses to other parts of the building for suspicious patterns.

Figure 5
The general behaviour of the control panel



4.3. System Properties

An access control system has to satisfy the following properties:

- **Serving employees:** This property ensures that whenever an employee is present and wants access, he can use the access card reader and be served, either by being authorized or by being denied access.

- **System evolution and deadlock absence:** This property ensures that the access control system is always active whenever an employee is present and that both the access card reader and the employee will progress in their interaction, i.e., there is no deadlock.
- **Controlling access:** This property ensures that only authorized employees have access to the restricted zone.

5. Case Study

5.1. General System Description

We consider a simple door access control system in a factory that controls the access into a private zone. The employees of the factory use their access cards to enter the building. For the sake of simplicity, we assume that the privileged employees have a palindrome identification code in their cards which and for that an algorithm has to be used for verification.

Figure 6 shows the optimal Petri net describing the model of this access control system, and Table 1 presents the meaning of the Petri net places and transitions.

Figure 6
Petri net model of the access control system

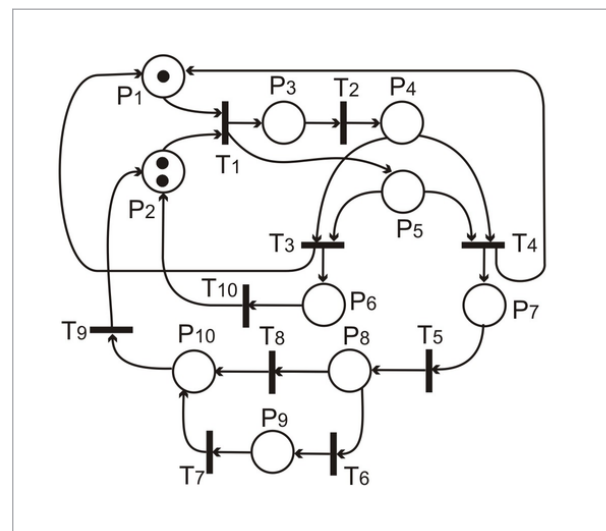


Table 1

Meaning of Petri net places and transitions

Place labels with meaning	
P1	Access card reader is ready (not is use)
P2	Employee with ID card want to access
P3	Card is inserted
P4	Card is tested
P5	Employee is waiting the card testing
P6	Employee takes his invalid card (denied)
P7	Employee takes his card and waits to access
P8	Employee is in the private area
P9	Employee is occupied
P10	Employee is out of the private area
Transition labels with meaning	
T1	Insert card into ACS card reader
T2	Testing employee card
T3	Rejecting card with negative test result
T4	Accepting card and opening access door shortly
T5	Entering to the private area
T6	Start doing job
T7	Ending job and exiting
T8	Exiting the private area without doing job
T9	Authorized employee returns to the ACS
T10	Non authorized employee returns to the ACS

5.2. System and Properties Specification

5.2.1. System Specification

The associated rewrite theory specifying the Petri net model of Figure 6 is given in the module `PN_ACS`.

```

mod PN_ACS is pr INT .
inc CONFIGURATION . op PLACE : -> Cid .
op tokens :_ : Int -> Attribute [gather(&)] .
ops P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 : -> Oid .
var x : Int .
crl [T1] : < P1 : PLACE | tokens : 1 > < P2

```

```

: PLACE | tokens : x > < P3 : PLACE | tokens
: 0 > < P5 : PLACE | tokens : 0 > => < P1 :
PLACE | tokens : 0 > < P2 : PLACE | tokens :
x - 1 > < P3 : PLACE |

```

```

tokens : 1 > < P5 : PLACE | tokens : 1 > if
(x >= 1) .

```

```

rl [T2] : < P3 : PLACE | tokens : 1 > < P4 :
PLACE | tokens : 0 > => < P3 : PLACE | tokens
: 0 > < P4 : PLACE | tokens : 1 > .

```

```

rl [T3] : < P4 : PLACE | tokens : 1 > < P5
: PLACE | tokens : 1 > < P6 : PLACE | tokens
: 0 > < P1 : PLACE | tokens : 0 > => < P4 :
PLACE | tokens : 0 > < P5 : PLACE | tokens :
0 > < P6 : PLACE | tokens : 1 > < P1 : PLACE
| tokens : 1 > .

```

```

rl [T4] : < P4 : PLACE | tokens : 1 > < P5
: PLACE | tokens : 1 > < P7 : PLACE | tokens
: 0 > < P1 : PLACE | tokens : 0 > => < P4 :
PLACE | tokens : 0 > < P5 : PLACE | tokens :
0 > < P7 : PLACE | tokens : 1 > < P1 : PLACE
| tokens : 1 > .

```

```

rl [T5] : < P7 : PLACE | tokens : 1 > < P8 :
PLACE | tokens : 0 > => < P7 : PLACE | tokens
: 0 > < P8 : PLACE | tokens : 1 > .

```

```

rl [T6] : < P8 : PLACE | tokens : 1 > < P9 :
PLACE | tokens : 0 > => < P8 : PLACE | tokens
: 0 > < P9 : PLACE | tokens : 1 > .

```

```

rl [T7] : < P9 : PLACE | tokens : 1 > < P10 :
PLACE | tokens : 0 > => < P9 : PLACE | tokens
: 0 > < P10 : PLACE | tokens : 1 > .

```

```

rl [T8] : < P8 : PLACE | tokens : 1 > < P10 :
PLACE | tokens : 0 > => < P8 : PLACE | tokens
: 0 > < P10 : PLACE | tokens : 1 > .

```

```

rl [T9] : < P10 : PLACE | tokens : 1 > < P2 :
PLACE | tokens : x > => < P10 : PLACE | tokens
: 0 > < P2 : PLACE | tokens : x + 1 > .

```

```

rl [T10] : < P6 : PLACE | tokens : 1 > < P2 :
PLACE | tokens : x > => < P6 : PLACE | tokens
: 0 > < P2 : PLACE | tokens : x + 1 > .

```

```

endm

```

In this specification, both of the static (signature) and dynamic (behavior) parts of the access control system are defined.

5.2.2. Properties Specification

After formalization, the specification of properties represents the next preparation step before the verification stage. At this level, the designer has to prepare two modules: the first one contains the set of properties that are assumed to be verified as presented in the module `PN-ACS_PREDS`. In the second one, the designer expresses the relevant properties to be checked in the system as given in the module `ACS-CHECK`.

The sample modules for our security system are given as follows:

```

mod PN-ACS_PREDS
is protecting PN_ACS .
including SATISFACTION .
subsort Configuration < State .
ops Want-Access Waiting Authorized : -> Prop .
ops Denied Enter Occupied Exiting : -> Prop .
var A : Configuration .
var y : Int .
  ***----- SOME IMPORTANT PROPERTIES -----
ceq A < P2 : PLACE | tokens : y > |= Want-
Access = true if (y >= 1) .
eq A < P5 : PLACE | tokens : 1 > |= Waiting =
true .
eq A < P7 : PLACE | tokens : 1 > |= Authorized
= true .
eq A < P6 : PLACE | tokens : 1 > |= Denied =
true .
eq A < P8 : PLACE | tokens : 1 > |= Enter =
true .
eq A < P9 : PLACE | tokens : 1 > |= Occupied
= true .
eq A < P10 : PLACE | tokens : 1 > |= Exiting
= true .
endm

mod ACS-CHECK is
inc PN-ACS_PREDS .
inc MODEL-CHECKER .
inc LTL-SIMPLIFIER .
ops initial-state : -> Configuration .
eq initial-state = < P1 : PLACE | tokens : 1
> < P2 : PLACE | tokens : 20 > < P3 : PLACE
| tokens : 0 > < P4 : PLACE | tokens : 0 >
< P5 : PLACE | tokens : 0 > < P6 : PLACE |
tokens : 0 > < P7 : PLACE | tokens : 0 > < P8
: PLACE | tokens : 0 > < P9 : PLACE | tokens
: 0 > < P10 : PLACE | tokens : 0 > .
ops no-deadlock serving : -> Prop .
eq serving = [] ((Want-Access) -> <> (Authorized
\ / Denied)) .
eq no-deadlock = [] (Want-Access \ / Authorized
\ / Denied \ / Waiting \ / Enter \ / Occupied \ /
Exiting) .
endm

```

5.2.3. Verification

Next, the two properties described in the module ACS-CHECK are verified by using Maude LTL model-checker. We have also supposed that there are 20 employees in the initial state for these properties. The following results have been tested using the Version 2.7 of Maude system with Eclipse 4.2.2.

– **Serving employees:** This property is formally specified as the following LTL formula:

```

[] ((Want-Access) -> <> (Authorized \ / Denied))

```

After verification, the LTL model checker confirms the correctness of this property as follows:

```

Maude> reduce in ACS-CHECK :
modelCheck(initial-state, serving) .

```

```

rewrites: 744 in 29843475119ms cpu (15ms
real) (~ rewrites/second)

```

```

result Bool: true

```

– **System evolution and deadlock absence:** This property is formally expressed as follows:

```

[] ( Want-Access \ / Authorized \ / Denied \ /
Waiting \ / Enter \ / Occupied \ / Exiting)

```

This property is also verified as follows:

```

Maude> reduce in ACS-CHECK :
modelCheck(initial-state, no-deadlock) .

```

```

rewrites: 857 in 41773847050ms cpu (15ms
real) (~ rewrites/second)

```

```

result Bool: true

```

5.2.4. Testing

The formalization and verification performed so far has focused on the most critical aspect of a security system, i.e., availability. Another interesting property is “controlling access” mentioned in Section 4.3. This property concerns directly the composite state “Validate Cid” of the state diagram, given in Figure 5, which is practically implemented by means of the palindrome algorithm testing as mentioned in Section 5.1. Unfortunately, this property is not expressible as a LTL formula and for that the Property-Based Testing technique is used.

The proposed C function to check whether a card code is palindrome or not is given as follows:

```

int isPalindrome(char code[])
{
  char reverse_code[];
  int i;
  int h = strlen(code);

```

```

    // inverting code string characters
    for (i = h - 1; i >= 0 ; i--)
    {
        reverse_code[h - i - 1] = code[i];
    }
    if (strcmp(code,reverse_code) == 0)
    {
        return 0; // code is palindrome
    }
    else
    {
        return 1; // code is not palindrome
    }
}

```

Formalizing this function is probably not worth the time and effort, but its property-based testing will provide much more confidence in its realization than mere unit testing. Therefore, the universally quantified expression that we use as a property is:

```

prop_privileged_employee_algorithm() ->
?FORALL(P, maybe_palindrome(), sut:validate_user(P) == is_palindrome(P)).

```

where `maybe_palindrome/0` is an input generation function, `sut:validate_user/1` is the algorithm implementation to be tested, and `is_palindrome/1` is our oracle function.

```

maybe_palindrome() -> oneof([palindrome(),
string()]).
palindrome() ->
?LET(Base, string(),
    ?LET(Middle, oneof([[], [char()]]),
        Base ++ Middle ++ lists:reverse(Base))).
string() -> list(char()).
is_palindrome(S) -> S == lists:reverse(S).

```

In other words, we use the `maybe_palindrome/0` function to generate user identifications (i.e. strings, sometimes palindromes, sometimes just lists of characters) to test whether they are considered palindromes (and thus, are granted access) by the system implementation in the same way as our oracle (`is_palindrome/1`) does. The execution of such property using QuickCheck reassures us in that this is, indeed, the case:

```

>eqc:quickcheck(prop_privileged_employee()).
.....
OK, passed 100 tests
true

And the more time we can allocate to execute more tests, the greater the confidence level is:
>eqc:quickcheck(eqc:numtests(10000,palindrome : prop_check_indices())).
..... (x10)
..... (x100)
.....
OK, passed 10000 tests
true

```

6. Discussion and Contributions

The software design process is often viewed as a sequence of phases that transforms a set of informal specifications into a detailed technical specification that can be used for the development. All the intermediate phases are characterized by a transformation from a more abstract description to a more detailed one until we have the final software product. Our approach for the development of software based systems contemplates the following phases of software development: modeling, specification, verification and testing. Note that it is not our intention to propose a methodological development lifecycle, but a systematic approach to systems analysis.

First, we start from an abstract description of the system using Petri nets that represents a well-suited theoretical model of concurrency – for both industry and academia – in which the temporal ordering and causal relationship between system components can easily be represented. However, basic Petri nets exhibit some known limitations [5, 29]. For instance, since the primary concern of Petri nets is behavior modeling and analysis [18], their most widely voiced criticism, is the lack of algebraic structuring primitives or modularity, which often leads to unstructured models and inadequate abstractions of real systems.

In addition, tokens circulating in Petri nets are undifferentiated entities and Petri nets describe only the control structure of the system without references to

its data structure. Another practical limitation concerns the number of reachable states that explodes especially when Petri nets contain a large number of tokens [17]. Moreover, a Petri nets cannot be used to express all kinds of behavior, acts and routing that are frequently encountered in real systems.

In order to cater for the above-mentioned issues, we advocate the use of Maude [16, 15] as a formal specification language based on rewriting logic [31] for describing Petri nets models of access control systems. This choice is also motivated by the expressiveness of rewriting logic, which is a unifying framework for specifying and analyzing a wide range of Petri nets [42]. In addition, its distinguishing features include the integration of the quasi-totality of OBJ3 features [23], such as parameterized programming, multiple inheritance, module instantiation, views, and a large set of programming techniques. Furthermore, the availability of an integrated rich tool set for the formal analysis, such as model checking [19] and Maude reachability tool in Maude; facilitates the overall analysis considerably.

Then, Model checking can be considered as one of the most successful and widely used verification techniques. However, it exhibits some shortcomings as well.

- 1 Model checking is generally deemed inadequate to analyze software of access control systems because of their dynamic nature and complexity, which often leads to the state-space explosion problem.
- 2 Model checking is used to automatically check whether a model meets a formal specification given by a temporal logic formula. However, certain properties of practical interest are not expressible in temporal logic.

In the proposed approach, these limitations can be somewhat overcome by the use of Maude LTLR model checker and Maude reachability-analysis tool. While the former supports on-the-fly explicit-state model checking of concurrent systems, the latter can be used to overcome the lack of expressiveness of the supported temporal logic when using LTLR model checker.

Besides the formal verification, we advocate the use

of Property-Based Testing [20] on the testing stage. Property-Based Testing is an automatic testing technique of the proposed code skeletons based on random generation of test sequences (i.e. test scenarios). Contrary to the classic testing, Property-Based Testing technique is used to test code skeletons or algorithms before implementation. Such generated tests attain a high degree from “*the whole control test coverage*” of the system implementation and, thus PBT techniques are found to be more effective in the detection of faults earlier in the life cycle of product development, allowing an efficient resource allocation and minimization of the system maintenance costs.

7. Conclusion

In this paper, we have proposed a hybrid approach for the formal development of software based systems. Therefore, a Petri net based model is prepared for the studied software based system with the corresponding rewriting logic specification. Then, the associated Maude model checker is used for the formal verification of such a Petri net model. At the last stage, property based testing technique is used to test the proposed system implementation before its realization.

The proposed approach permits us to check most of the important properties of such systems including, safety and liveness. Furthermore, some critical properties for access control system are not expressible with LTL formulas and thus, the integration of Property-Based Testing technique allows us to capture more informal conjectures about the source code and test such properties in the proposed implementation.

Acknowledgements

This work would not have been possible without the financial support No: 034/PNE/ENS/Spain/13-14 received through the Algerian ministry of higher education and scientific research.

References

1. Ammann, P. E., Black, P. E., Majurski, W. Using Model Checking to Generate Tests from Specifications. Proceedings Second International Conference on Formal Engineering Methods (Cat.No.98EX241), Bris-

- bane, Queensland, Australia, 1998, 46-54. <https://doi.org/10.1109/ICFEM.1998.730569>
2. Ammar, B., Abdallah, K. Towards the Formal Specification and Verification of multi-Agent Based Systems. *IJCSI*, 2011, 8(4), 200-210.
 3. Arts, T., Castro, L. M., Hughes, J. Testing Erlang Data Types with Quviq QuickCheck. 7th ACM SIGPLAN Workshop on ERLANG (Erlang'08), ACM, Victoria, BC, Canada, September 20-28, 2008, 1-8.
 4. Bae, K., Meseguer, J. The Linear Temporal Logic of Rewriting Maude Model Checker. *WRLA*, Springer, 2010, 208-225.
 5. Ballarini, P., Djafri, H., Duflot, M., Haddad, S., Pekergin, N. Petri Nets Compositional Modeling and Verification of Flexible Manufacturing Systems. 2011 IEEE Conference on Automation Science and Engineering (CASE), 2011, 588-593. <https://doi.org/10.1109/CASE.2011.6042488>
 6. Benantar, M. Access Control Systems: Security, Identity Management and Trust Models. Springer Science & Business Media, 2006.
 7. Bernot, G., Gaudel, M.C., Marre, B. Software Testing Based on Formal Specifications: A Theory and a Tool. *Software Engineering Journal*, 1991, 6(6), 387-405. <https://doi.org/10.1049/sej.1991.0040>
 8. Burton, S., York, H. Automated Testing from Z Specifications. Technical Report, Department of Computer Science, University of York, 2000.
 9. Castro, L. M. Advanced Management of Data Integrity: Property-Based Testing for Business Rules. *Journal of Intelligent Information Systems*, 2015, 44(3), 355-380. <https://doi.org/10.1007/s10844-014-0335-2>
 10. Castro, L. M., Arts, T. Testing Data Consistency of Data-Intensive Applications Using QuickCheck. 10th Spanish Conference on Programming and Languages (PROLE'10), Valencia, Spain, September 8- 10, 2010. Revised Selected Papers, *Electronic Notes in Theoretical Computer Science*, Elsevier Science Publishers, Amsterdam, the Netherlands, 2011, 271, 41-62. <https://doi.org/10.1016/j.entcs.2011.02.010>
 11. Castro, L. M., Francisco, M. A., Gulías, V. M. Testing Integration of Applications with QuickCheck. *International Conference on Computer Aided Systems Theory*, 2009.
 12. Chow, T. S. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 1978, SE-4(3), 178-187. <https://doi.org/10.1109/TSE.1978.231496>
 13. Clarke, E. M., Emerson, E. A. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. Springer, 1982. <https://doi.org/10.1007/BFb0025774>
 14. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J. A Maude Tutorial. Computer Science Laboratory, SRI International, 2000.
 15. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J. F. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 2002, 285(2), 187-24. [https://doi.org/10.1016/S0304-3975\(01\)00359-0](https://doi.org/10.1016/S0304-3975(01)00359-0)
 16. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. All About Maude – A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. Springer-Verlag, 2007.
 17. David, R., Alla, H. On Hybrid Petri Nets. *Discrete Event Dynamic Systems*, 2001, 11(1-2), 9-40. <https://doi.org/10.1023/A:1008330914786>
 18. Deng, Y., Lu, S., Evangelist, M. A Formal Approach for Architectural Modeling and Prototyping of Distributed Real-Time Systems. *System Sciences*, 1997, 1, 481-490. <https://doi.org/10.1109/HICSS.1997.667304>
 19. Eker, S., Meseguer, J., Sridharanarayanan, A. The Maude LTL Model Checker. *Electronic Notes in Theoretical Computer Science*, 2004, 71, 162-187. [https://doi.org/10.1016/S1571-0661\(05\)82534-4](https://doi.org/10.1016/S1571-0661(05)82534-4)
 20. Fink, G., Bishop, M. Property-Based Testing: A New Approach to Testing for Assurance. *ACM SIGSOFT Software Engineering Notes*, 1997, 22(4), 74-80. <https://doi.org/10.1145/263244.263267>
 21. Fink, G., Levitt, K. Property-Based Testing of Privileged Programs. *Proceedings 10th Annual Computer Security Applications Conference*, 1994, 154-163. <https://doi.org/10.1109/CSAC.1994.367311>
 22. Gargantini, A., Heitmeyer, C. Using Model Checking to Generate Tests from Requirements Specifications. *ACM SIGSOFT Software Engineering Notes*, Springer-Verlag, 1999, 24, 146-162. <https://doi.org/10.1145/318774.318939>
 23. Goguen, J., Kirchner, C., Kirchner, H., Mégard, A., Meseguer, J., Winkler, T. An Introduction to OBJ 3. *International Workshop on Conditional Term Rewriting Systems*, Springer, 1987, 258-263.
 24. Grumberg, O., Veith, H. 25 Years of Model Checking: History, Achievements, Perspectives. Springer, 2008, 5000. <https://doi.org/10.1007/978-3-540-69850-0>

25. Guelev, D. P., Ryan, M., Schobbens, P. Y. Model-Checking Access Control Policies. International Conference on Information Security, Springer, 2004, 219230. https://doi.org/10.1007/978-3-540-30144-8_19
26. Hierons, R. M. Testing from a Z Specification. Software Testing, Verification and Reliability, 1997, 7(1), 19-33. [https://doi.org/10.1002/\(SICI\)1099-1689\(199703\)7:1<19::AID-STVR124>3.0.CO;2-N](https://doi.org/10.1002/(SICI)1099-1689(199703)7:1<19::AID-STVR124>3.0.CO;2-N)
27. Hörcher, H. M. Improving Software Tests Using Z Specifications. International Conference of Z Users, Springer, 1995, 152-166.
28. Hu, V. C., Kuhn, D. R., Xie, T., Hwang, J. Model Checking for Verification of Mandatory Access Control Models and Properties. International Journal of Software Engineering and Knowledge Engineering, 2011, 21(1), 103-127. <https://doi.org/10.1142/S021819401100513X>
29. Kolagari, A. R. T. Transformation of Open and Algebraic High-level Petri Net Classes. Technische Universität Berlin, Fakultät IV, Elektrotechnik und Informatik, 2002.
30. Masood, A., Bhatti, R., Ghafoor, A., Mathur, A. Model-Based Testing of Access Control Systems that Employ RBAC Policies. Technical Report SERC-TR- 277, 2005.
31. Meseguer, J. Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science, 1992, 96(1), 73-155. [https://doi.org/10.1016/0304-3975\(92\)90182-F](https://doi.org/10.1016/0304-3975(92)90182-F)
32. Meseguer, J. Rewriting Logic as a Semantic Framework for Concurrency: A Progress Report. CONCUR'96: Concurrency Theory, Springer, 1996, 331-372.
33. Meseguer, J. Membership Algebra as a Logical Framework for Equational Specification. Recent Trends in Algebraic Development Techniques, Springer, 1997, 18-61.
34. Meseguer, J. Twenty Years of Rewriting Logic. The Journal of Logic and Algebraic Programming, 2012, 81(7), 721-781. <https://doi.org/10.1016/j.jlap.2012.06.003>
35. Petri, C. A. Kommunikation mit Automaten. Ph.D. Thesis, Darmstadt University of Technology, Germany, 1962.
36. Pretschner, A., Mouelhi, T., Le Traon, Y. Model-Based Tests for Access Control Policies. 1st International Conference on Software Testing, Verification, and Validation, IEEE, 2008, 338-347. <https://doi.org/10.1109/ICST.2008.44>
37. Quviq a. b. <http://www.quviq.com> (2008)
38. Reisig, W. A Primer in Petri Net Design. Springer Science & Business Media, 2012.
39. Rivas, S., Francisco, M. A., Gulías, V. M. Property Driven Development in Erlang, by Example. 5th Workshop on Automation of Software Test (ASE'10), Cape Town, South Africa, May 1-8, 2010, 75-78. <https://doi.org/10.1145/1808266.1808277>
40. Sandhu, R. S., Samarati, P. Access Control: Principle and Practice. IEEE Communications Magazine, 1994, 32(9), 40-48. <https://doi.org/10.1109/35.312842>
41. Schaad, A., Moffett, J. D. A Lightweight Approach to Specification and Analysis of Role-Based Access Control Extensions. Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies, 2002, 1322. <https://doi.org/10.1145/507711.507714>
42. Stehr, M. O., Meseguer, J., Ölveczky, P. C. Rewriting Logic as a Unifying Framework for Petri Nets. Unifying Petri Nets, 2001, 2128, 250-303. https://doi.org/10.1007/3-540-45541-8_9
43. Stehr, M. O., Talcott, C. L. Plan in Maude Specifying an Active Network Programming Language. Electronic Notes in Theoretical Computer Science, 2004, 71, 240-260. [https://doi.org/10.1016/S1571-0661\(05\)82538-1](https://doi.org/10.1016/S1571-0661(05)82538-1)
44. Stocks, P. A. Applying Formal Methods to Software Testing. Ph.D. Thesis, University of Queensland, 1993.
45. Thati, P., Sen, K., Martí-Oliet, N. An Executable Specification of Asynchronous Pi-Calculus Semantics and May Testing in Maude 2.0. Electronic Notes in Theoretical Computer Science, 2004, 71, 261-281. [https://doi.org/10.1016/S1571-0661\(05\)82539-3](https://doi.org/10.1016/S1571-0661(05)82539-3)
46. Utting, M., Legeard, B. Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, 2010.
47. Verdejo, A., Martí-Oliet, N. Executing E-LOTOS Processes in Maude. INT, Citeseer, 2000, 49-53.
48. Verdejo López, J. A., Martí Oliet, N. Executing and Verifying CCS in Maude. Technical Report, 2000, 99, 1-47.
49. Xiang, H., Xia, X., Hu, H., Wang, S., Sang, J., Ye, C. Approaches to Access Control Policy Comparison and the Inter-Domain Role Mapping Problem. Information Technology and Control, 2016, 45(3), 278-288. <https://doi.org/10.5755/j01.itc.45.3.13187>
50. Zhang, N., Ryan, M., Guelev, D. P. Evaluating Access Control Policies Through Model Checking. International Conference on Information Security, Springer, 2005, 446-460. https://doi.org/10.1007/11556992_32