HAL
open science

A Design by Contract Approach to Verify Access Control Policies

Hakim Belhaouari, Pierre Konopacki, Régine Laleau, Marc Frappier

To cite this version:

Hakim Belhaouari, Pierre Konopacki, Régine Laleau, Marc Frappier. A Design by Contract Approach to Verify Access Control Policies. IEEE Computer Society, 2012, pp.263-272. 10.1109/ICECCS.2012.4 . hal-00724267

HAL Id: hal-00724267
https://hal.science/hal-00724267

Submitted on 20 Feb 2013

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

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

A Design by Contract Approach to Verify Access Control Policies

Hakim Belhaouari*, Pierre Konopacki†, Régine Laleau§ and Marc Frappier‡
*XLIM-SIC, Université de Poitiers, France
hakim.belhaouari@sic.univ-poitiers.fr
†‡GRIL, Département d'informatique, Université de Sherbrooke, Canada
{pierre.konopacki,marc.frappier}@usherbrooke.ca
§LACL, Université Paris-Est Créteil, France
laleau@u-pec.fr

***Abstract*—In the security domain, access control (AC) consists in specifying who can access to what and how, with the four well-known concepts of permission, prohibition, obligation and separation of duty. In this paper, we focus on role-based access control (RBAC) models and more precisely on the verification of formal RBAC models. We propose a solution for this verification issue, based on the use of the Tamago platform. In Tamago, functional contracts can be defined with pre/postconditions and deterministic automata. The Tamago platform provides tools for static verifications of these contracts, generation of test scenarios from the abstract contracts and monitoring facilities for dynamic analyses. We have extended the platform to take into account AC aspects. AC rules, expressed in a subset of EB^{3}SEC, a process algebra-based language, are translated into pre and post conditions of new security contracts. We have also adapted the test case generator to derive suitable test scenarios and the monitoring framework by adding a new security component.**

***Keywords*-Access control specification; verification; Design by Contract**

I. Introduction

Nowadays, organisations are so dependent on the proper functioning of their information systems (IS) that corruption, loss of data or breaches in confidentiality, may have serious consequences. One of the facets of IS security is concerned with access control (AC) that defines who can access to what and how. A number of access control models have already been proposed. In this paper, we focus on role-based access control models (RBAC) [1]. In such models, AC rules can be described through the use of four concepts: permission, prohibition, Separation of Duty (SoD) and obligation. Some of these models make use of formal languages, allowing verifications to be carried out at an abstract level, before any implementation. We have developed the EB^{3}SEC language [2], based on process algebra, that allows designers to specify the four kinds of AC constraints in the same model. By now, the most critical point concerns verification of AC constraints. Different techniques of verification exist to prove properties on systems in general. Theorem proving allows invariant properties to be verified but it is not easy to prove dynamic properties, such as temporal or ordering constraints. For these properties, model checking is more appropriate but faces the problem of combinatorial explosion, especially in information systems where there is a large amount of entities [3]. An alternative for verification is to carry out tests and the possibility to generate test scenarios from formal models.

In this paper, we propose a solution for the verification issue, based on the use of the Tamago platform [4]. The Tamago language allows for the definition of functional contracts described, on one hand, by pre and post conditions expressed in first order logic, and on the other hand by deterministic automata. The Tamago platform provides tools for static verifications of these contracts, generation of test scenarios from the abstract contracts and monitoring facilities for dynamic analyses. We have extended the platform to take into account AC aspects. AC rules, expressed in a subset of EB^{3}SEC, are translated into pre and post conditions of new security contracts. We have also adapted the test case generator to derive suitable test scenarios and the monitoring framework by adding a new security component.

In Section II, we describe the sub-language of EB^{3}SEC, called ACA (Access Control lAnguage), adapted to Tamago. Section III presents the different components of the Tamago platform. Section IV details the extensions we have developed in the Tamago platform to consider the different kinds of AC rules described in Section II. Section V is dedicated to related works and Section VI draws the conclusions of this work and the perspectives.

II. The Access Control Language ACA

An AC policy is part of the security policy of a system that deals with who can access to what and how. Generally, AC constraints are classified into four concepts :

- **Permission** allows the execution of an action.
- **Prohibition** forbids the execution of an action.
- **Obligation** links two actions. Once a person performed one of them, he must execute the second action [5].
- **Separation of duty** links two actions. Once a person performed one of them, he is not allowed to perform the second action.

With regard to verification purpose, it is also interesting to differentiate them according to their static/dynamic feature.

- **Static AC constraints** are constraints that do not depend on the state or the evolution of the system. Typically, static AC constraints are permissions and prohibitions.
- **Dynamic AC constraints** depends on the state or the evolution of the system. This category contains constraints of SoD and obligation.

Contrary to existing AC modelling languages [1], [6], our approach, called EB^{3}SEC [2], allows designers to express the four kinds of AC constraints in the same model. In this paper, we present ACA (Access Control lAnguage), a subset of EB^{3}SEC, adapted to the input language of the Tamago platform. It also contains the four kinds of AC constraints but contextual constraints are excluded since they require to extend the Tamago platform. Contextual constraints are predicates defined on the state of the system that reduce the range of validity of AC constraints. For instance, in a bank, users holding role *clerk* can execute a validate between 8.00am and 5.00pm.

The rest of the section presents the ACA language through the description of the case study which consists of modelling an AC policy of a bank that wants to secure its check deposit process. This business process can be described as follows :

1) The action deposit must be executed for the check.
2) The actions check and register must be executed for the check, but in any order.
3) The action cancel must be executed, or, the actions validate and validate_dir must be executed, but in any order.

To model this AC policy, we choose an AC model derived from RBAC [1]: our model adds to the RBAC model the concept of organisation introduced by the OrBAC [7] method. In ACA, designers must, at first, declare the sets containing the users, the roles, the organisations and the actions as follows:

```
users:=alphonse,boris,catherine,damien,elise,franck;
roles:=customer,clerk,banker,director;
organisations:=montreal,toronto;
actions:=deposit,cancel,check,validate,
         validate_dir,register;
```

The set users defines active entities evolving in the system (*i.e.,* users that interact with the system). The set roles stands for the roles that users can hold in the system. For instance, in a bank, roles can be *customer*, *clerk*, *banker* or *director*. The set organisations represents physical branches of the bank. Indeed, in a corporation, it is rather common that the AC policy can slightly differ from branches to branches. Rather than duplicating roles to consider the different branches, we prefer to add a new element, organisations. The set actions specifies actions that can be executed in the system and that need to be protected by the AC policy. Note that business parameters of the actions do not appear in the ACA model since they are not useful to express AC constraints.

ACA allows designers to specify an AC policy that can be applied on each instance of a business process, thus, business parameters of actions are not required in the AC rules. If an action appears in several AC rules, its execution must satisfy all the AC rules where it appears.

The next element deals with the relation that links sets users, roles and organisations, called play. It is similar to the association *User-Role assignment* existing in the RBAC model, but enhanced with the concept of organisation. The relation play specifies who can hold which role in which organisation. For instance, the user *alphonse* holds the role *customer* in the branch of *montreal*. In our case study, the relation play is filled with :

```
play :=
  <alphonse,customer,montreal>,
  <boris,customer,montreal>,
  <boris,clerk,montreal>,
  <catherine,director,montreal>,
  <catherine,customer,toronto>,
  <damien, banker,montreal>,
  <elise,clerk,toronto>,
  <franck,director,toronto>;
```

The next element is the definition of the relation permission. This relation links the sets users, roles, organisations and actions.Thanks to this association, designers can express which role a user must hold to execute an action in an organisation. For instance, it could help to model that in every branches, any user who holds the role of *clerk* can execute the action *deposit*. In our case study, the permissions are declared as follows :

```
permissions :=
  <_,clerk,_,deposit>,
  <_,clerk,_,register>,
  <_,banker,_,deposit>,
  <_,banker,_,register>,
  <_,banker,_,cancel>,
  <_,banker,_,validate>,
  <_,director,_,cancel>,
  <_,director,_,validate>;
  <_,director,_,validate_dir>;
```

The symbol _ means that this parameter can take any value. To compute effective permissions, we join the relations permissions and play. The join is calculated on the explicitly declared parameters (parameters not represented by an _). In our example, the computation of the effective permissions for the permission `<_,clerk,_,deposit>` gives :

```
effPerm1 :=
  <boris,clerk,montreal,deposit>,
  <elise,clerk,toronto,deposit>,
```

The next declaration deals with the relation prohibitions. It also links the sets users, roles, organisations and actions. Opposite to permissions, prohibitions forbid people who hold a given role to execute an action in an organisation. For instance, in every branches the user *elise*, holding any role, is forbidden to execute a *cancel*. The symbol ! before a parameter denotes the negation operator.

```
prohibitions :=
  <!elise,_,_,cancel>,
  <_,_,!Toronto,validate>,
  <_,!customer,_,deposit>,
  <_,!customer,_,cancel>,
  <_,!customer,_,validate>;
```

As for permissions, effective prohibitions are computed by joining the relations prohibitions and play on explicitly declared parameters. Effective prohibitions for the prohibition `<_,_,!Toronto,validate>` are :

```
effPro1 :=
  <catherine,customer,toronto,validate>,
  <elise,clerk,toronto,validate>,
  <franck,director,toronto,validate>;
```

The next declaration deals with obligations. Contrary to permissions and prohibitions, the concept of obligation is not precisely defined in the literature. In our AC model, an obligation defines an ordering constraint between two actions, linked by the same value of one AC parameter. In the case study, we have two obligations that force actions to be executed by the same user.

```
obligations :=
OBL(user,<user,_,_,deposit>,<user,_,_,register>),
OBL(user,<user,_,_,deposit>,<user,_,_,cancel>);
```

The first one states that the same user must execute actions deposit and register, holding any role in any organisation. Recall that it is for the same instance of the business process, that is for a given check. The second obligation stipulates that the same user must execute the actions deposit and cancel, holding any role in any organisation.

The last declaration of an ACA model deals with SoD constraints. A SoD also defines an ordering constraint between two actions, but, contrary to an obligation, the value of one AC parameter must be different. In our example, we have two SoD.

```
separations :=
SOD(user,<user,_,_,deposit>,<!user,_,_,validate>),
SOD(user,<user,_,_,validate>
        ,<!user,_,_,validate_dir>);
```

The first SoD forbids the same user to execute the actions deposit and validate for a given check. This means that these actions must be executed by two different users whatever the role they are holding in any organisation.

The second SoD constraint prohibits the same user to execute validate and validate_dir for a given check.

In ACA, when an action is executed, it must comply with all AC rules that secure it. First, the action must be permitted by at least one permission. Indeed, permissions give the set of values that can be used by security parameters of an action. Thus the security parameters of an action to be executed must comply with at least one permission. Prohibitions give the set of values that cannot be used by security parameters of an action. Thus, to be executed, an action must not have security parameters that match values given by the prohibitions. Finally, an action can appear zero or several times in *obligation* and *SoD*. Then, to be executed, an action must also comply with all obligation and SoD constraints where it appears.

III. The Tamago platform

The Tamago platform allows rapid verifications of systems. The main idea is to add information to existing applications in order to verify static and dynamic properties on these applications. Up to now, Tamago considers Java applications. The Tamago platform follows a component-based system engineering (CBSE) approach combined with a design by contract (DbC) approach. A Tamago project is composed of two levels. The lower level, called business level, contains the Java code of the application to be verified. The upper level, called functional level, provides contracts that must be satisfied by the business level. These contracts are described by different kinds of entities, using a specification language, called Tamago Contract Description Language (CDL). This paper is only concerned by two kinds of entities: service and component.

A. The CDL language

The CDL language is based on a model of co-algebraic observable properties with first-order logic assertions (preconditions, postconditions and invariants) and descriptions of *service behaviours* [8], [9], based on finite-state automata with conditional transitions.

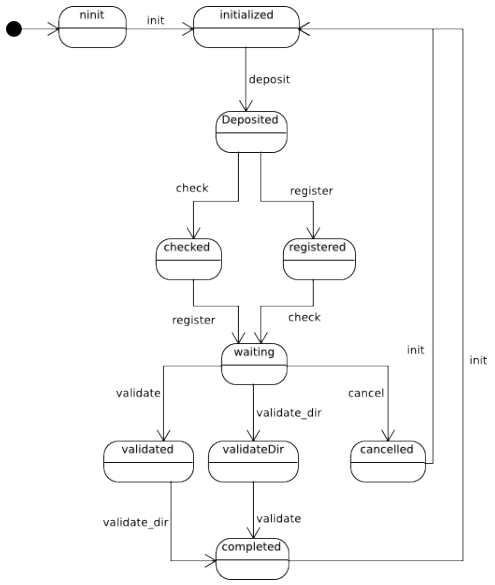
The two kinds of entities depict two levels of contract. Each level supports the verification of distinct kinds of properties. In the rest of this section, we present the service contract that specifies the functional semantics that can be shared between several components and the component contract for the description of the interaction between required services.

1) Service contract: A service contract defines the functional semantics of a service as a set of observable properties and a set of methods that any provider (*i.e.,* component) must implement. In addition to its signature, a method can be annotated with pre/postconditions. We also have an inheritance mechanism in order to preserve the behavioural subtyping for services (marked by the keyword `refine`). Furthermore, invariants can also be expressed on observable properties and/or methods. Finally, a finite state machine (FSM) can be expressed in order to increase the expressiveness and reliability of analyses. This FSM describes the service behaviour, that is, the correct ordering of methods.

Figure 1 illustrates the service DepositService written in the CDL language. Figure 1a describes the complete functional information of the service contract. The first line references a refinement of an existing service (called ACASecurity). This service comes from the core package of the ACA extension (cf. section IV) and acts as a marker in the Tamago-ACA mechanism during the creation of a secure component. Then, the property (called opNumber)

```
service DepositService {
    refine service tamago.aca.core.ACASecurity;
    property read int opNumber;
    method void init() {
        id init;
        post opNumber < 0;
    }
    method void deposit(tamago.ext.aca2.ACA aca) {
        id deposit;
    }
    method void cancel(tamago.ext.aca2.ACA aca) {
        id cancel;
    }
    method void validate(tamago.ext.aca2.ACA aca) {
        id validate;
    }
    method void validate_director(tamago.ext.aca2.ACA aca) {
        id validate_dir;
    }
    method void check(tamago.ext.aca2.ACA aca) {
        id check;
        post @isInState("waiting") ==> (opNumber > 0);
    }
    method void register(tamago.ext.aca2.ACA aca) {
        id register;
        post @isInState("waiting") ==> (opNumber > 0);
    }
    // the automaton goes here
}
```

(a) The functional part



(b) The automaton part

Figure 1: A service contract for the DepositService service

corresponds to an inner number used inside the bank agency (imposed by the business layer). Initially the unique property has a negative value, but when the state waiting is reached (*i.e.,* after calling the check and register methods), the contract verifies if the property has been set to a positive value. It will be used later in the validation process.

All the methods defined for the service are then specified. Finally, the service behaviour is illustrated in Figure 1b. The FSM describes the ordering of methods. In Tamago, the state ninit denotes the initial state in all FSM. Method init is used to initialise properties.

2) Component contract: The component contract includes part of a service contract: methods, properties and a FSM. The main difference with a service contract is the definition of dependencies between the provided services and the required services that comes from external components. Consequently, assertions in methods can give semantics of the interactions between the current component and the required services.

Figure 2 presents an extract of the component CheckDeposit. It provides the DepositService service and requires three services. It is also possible to be more precise in the specification of methods. For example, pre and postconditions can be added in the check method. The precondition guarantees that the client is creditworthy and that the amount of the check is positive before calling the business code of the check method (*i.e.,* real treatment). The postcondition ensures that the operation is correctly treated at the bank level (by the bank service). The postcondition of the register method ensures that the current operation is correctly registered in the audit trails of the bank.

These examples show the interest of the separation of concerns and the possibility offered by Tamago to organise the design by contract for each kind of contracts. Obviously, we could put all contracts directly in the component by providing the core service ACASecurity. However, the service DepositService can be used in another situation without the threesome (bank, user and check), for example in the case of a money deposit that corresponds to another component.

B. Architecture and tools

The Tamago platform provides a set of tools that can assist developers during the design and the implementation of software. The platform follows a Model-Driven System

```
component CheckDeposit{
    provide service tamago.aca.bank.DepositService;
    require service tamago.aca.bank.Client label user;
    require service tamago.aca.bank.Check label check;
    require service tamago.aca.bank.Bank label bank;
    ...
    method void check(tamago.ext.aca2.ACA aca) {
        id check;
        pre user.isCreditworthy() ∧ check.getAmount() > 0;
        post bank.isOperationChecked(this);
    }
    method void register(tamago.ext.aca2.ACA aca) {
        id register;
        post bank.getHistory().contains(this);
    }
}
```

Figure 2: An extract of the component CheckDeposit

Engineering [10] approach as it can generate verification code, skeletons of the business code, static analyses and test cases generation from a CDL contract. Figure 3 summarizes the typical development cycle for Tamago-based software. Initially, the designer writes the CDL contract of the application, then the first analysis allows inconsistencies to be detected. When all inconsistencies are corrected, code can be produced for the implementation step. With the contract compiler, skeletons and verification code can be produced, and with the test case generator, many test cases can be automatically generated. When the business code is written, tests can be executed in order to detect errors, failures and so on in the code. Finally when the code is reliable enough, the verification code (produced by Tamago-CC) monitors the execution of the business code to catch the last undetectable failures that were missed during the static analysis done by Tamago-Check and Tamago-Test.

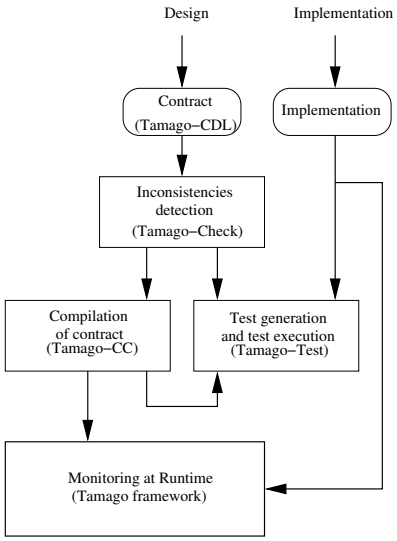


Figure 3: Summary of the Tamago platform

1) Static verification: In Figure 3, static analysis is achieved by two tools: Tamago-Check and Tamago-Test. The first one detects inconsistencies exclusively in contracts whereas the second one finds errors in the implementation of contracts before their deployment.

Tamago-Check takes a contract and checks syntactic elements (*i.e.,* all IDs are defined, assertions used in pre/post conditions call existing properties, methods and so on). Then, it uses model checking symbolic techniques [11] by simulating all the correct behaviours defined by the automaton. The final output is a report with three kinds of terminations for each behaviour: an error occurs with explanation, the simulation finds a fix-point or the simulation has been interrupted by reaching a limit (*i.e.,* predefined limit for integer or an hardware limit as a low memory).

Tamago-Test [4] assumes that a contract is correct (even if sometimes it can detect some new inconsistencies). As stated previously, it relies on simulation techniques. It selects a test scenario extracted from the automaton and computes exact test data for this scenario. In order to produce such a scenario, Tamago-Test makes use of test strategies to choose the scenario. By now, five strategies are proposed:

- *nominal strategy* randomly chooses a scenario and random correct values.
- *boundary strategy* randomly chooses a scenario and select boundary values (if possible).
- *outbound strategy* randomly chooses a scenario as previously, but for the last step, it generates a wrong security parameter in order to test the robustness of the system.
- *bad scenario strategy* randomly chooses a scenario with correct values, but at some point, it chooses an invalid action (*i.e.,* a missing transition) in order to see how the implemention reacts.
- *all transitions strategy* tries to cover all transitions of the automaton with one or many scenarios by taking into account the partitionning of the assertions (pre/-postconditions) in the contracts.

We have also developed a new constraint solver component, called Tamago Constraint Satisfaction Problem (Tamago-CSP), that can reason on standard enumerated types and more complex types (*e.g.,* the String type [12]).

2) Dynamic verification: The Tamago Contract Compiler (Tamago-CC) produces a Java interface from a service contract and verification code or skeletons from a component contract. In the paper, we focus on component contracts that are the only ones useful for code generation. The originality is the absence of usual weaving techniques by using wrappers with a hierarchical structure (called *containers*). Roughly speaking, a container is a Java class that acts as a filter to execute dynamic verifications on the associated business class.

Figure 4 shows the class diagram of the generated architecture for our case study.

The service DepositService becomes the interface on the top. The component contract is divided into several entities. The first one is the component interface CheckDeposit that corresponds to the syntactic part of the contract (name and signatures of the methods). Any business code will implement this interface, as the class CDepositBusiness. The last two entities are containers. Interface CheckDepositContainer is the root interface of all container implementations dedicated to this component. This container has a delegate link with the component interface with respect to the design pattern decorator [13]. On the figure, the verification code of the functional part is the class CheckDepositContainer_plugin. The verification code implements the container interface. It is generated by Tamago-CC from the specification of the component contract (that includes the specification of the provided services, cf. Fig. 1 and 2). At the runtime, the Tamago platform manages the

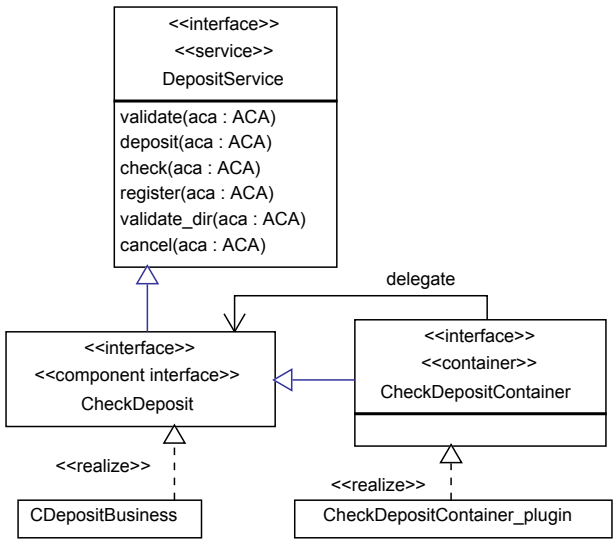


Figure 4: Generated architecture for the DepositService example

correct construction of a component where the delegate in the verification code is an instance of the business code.

IV. The extension Tamago-ACA

Tamago-ACA is the extension of the Tamago framework for access control purpose. It includes three parts. The first one is the translator from ACA specifications into CDL contracts. The second one deals with static verifications and generation of relevant test scenarios. The last one, for dynamic verifications, relies on the existing monitoring architecture of Tamago, in order to check AC rules at runtime.

A. CDL contracts from ACA specifications

This section presents the automatic translation from ACA specifications into CDL contracts. The translation process creates a new service contract for each functional service that needs to be secured. Initially the service contract is empty and is automatically enriched by the syntactic part of the functional contract (methods and properties) and the automaton. Furthermore, the translation adds two properties: the first one for the play table of the ACA specification and the second one, called history, to store all the methods already executed. This property is required to manage SoD and Obligation ACA rules.

Each method in the functional service contract corresponds to an action in an ACA specification. The idea is to complete the Tamago method to take into account AC rules expressed in ACA. For each method, we need to extract from the ACA specification the elements that concern the corresponding action. For example, the register action is involved in the permissions `<_,clerk,_,register>` and in the obligation `OBL(user,<user,_,_,deposit> ,<user,_,_,register>)`. These extracted elements are then translated into the pre/post conditions of the method, using the following template:

```
method <type> <functionality>(...,ACA aca) {
    pre (check aca parameters)
        ∧ (check permissions and prohibitions constraints)
        ∧ (check sod and obl constraints)
    post (add this method call to the history)
}
```

The ACA parameter stands for the tuple (user,role,organisation). The precondition contains three parts. The first part verifies that the ACA parameter is correct with regard to the play table. The second part checks the permissions and prohibitions related to the current method. The third part takes into account the history of the contract in order to check the SoD and obligation constraints related to the method. The postcondition ensures the correct evolution of the history property in order to be able to verify SoD and obligation constraints.

```
service GenDepositSecurity {
    refine service tamago.aca.bank.DepositService;
    property read int opNumber;
    property read History history;
    property read Play play;
    method void init() {
        id init;
        post history ≠ null ∧ history.size() = 0
            ∧ play.size() = 6
            ∧ play.contains("alphonse","customer","montreal") ∧ ...;
    }
    method void deposit(tamago.ext.aca2.ACA aca) {
        id deposit;
        pre play.isCorrectACA(aca) ∧ ((aca.role = "clerk")
            ∧ (aca.role ≠ "customer") ∧ ...);
        post history.lastSecuParam() = aca
            ∧ history.lastAction() = "deposit";
    }
    method void register(tamago.ext.aca2.ACA aca) {
        id register;
        pre play.isCorrectACA(aca) ∧ ... ∧ history.mustDone("deposit")
            ∧ history.getSecuParamFromID("deposit").getUser() = aca.user;
        post history.lastSecuParam() = aca ∧ history.lastAction() = "register";
    }
    ...
}
```

Figure 5: Extract of the generated CDL security contract

Figure 5 presents an extract of the generated security service that secures the DepositService service.

The *refine* line links the security contract to its functional contract DepositService. In the Init method, the postcondition ensures that the history is not null but its size is zero. It also initialises the play property with the content of the play table of the ACA specification. Actually this content is just copied but we plan to store it in a database. In the deposit method, the precondition checks that: (i) the ACA parameters are contained in the play property; (ii) these parameters match the values defined in the permissions and don't appear in the values defined in the

prohibitions. The postcondition ensures that the last method contained in the history property is really a deposit. This is necessary because deposit is involved in SoD and obligation constraints. In the register method, the precondition checks not only that the method is permitted and not prohibited for the values of the security parameters, but also that a deposit has already been executed with the same value for the user parameter.

B. Static verification of AC rules

From the generated security contract, some verifications can be achieved on the Tamago platform. A new tool has been developed and integrated in the Tamago platform. From the sets of effective permissions and prohibitions, it checks for each action that it is always possible to find an instantiation of the tuple (user,role,organisation) such that the action can be executed.

The second kind of verifications concerns the generation of test scenarios. Recall that the Tamago-Test tool produces test scenarios with respect to a given strategy. The final output of this activity is the production of a harness which contains essential elements for the test (preamble, data values, oracle, ...). Consequently, we can translate it into a concrete implementation: by now, we use the JUnit format for the Java prototype, where a test method represents a full scenario.

Tamago-Test takes into account a functional service and its associated security service. It extracts from the functional automaton a correct sequence of method calls in order to build the final scenario. Existing testing tools [14], [15], [16] generate random sequences without semantic links between each step of a sequence, that is why we use the term "scenario" in Tamago-Test. The way the automaton is visited is controlled by the test strategy. For security contracts, we can use the *nominal*, the *bad scenario* and the *all transitions* strategies. Table I gives examples of scenarios generated for a given strategy. It gives a summary of the scenarios, the complete files can be consulted at http://code.google.com/p/tamago/source/browse/trunk/TamagoACA2/ICECCS.

The first two lines of the table give some scenarios obtained with the *nominal* strategy. Obviously, this strategy can generate more scenarios and some parameters can be set inside the strategy (for instance the length of the scenario, the number of scenarios, ...). Each scenario follows the functional automaton, but scenarios have some differences in the ordering of actions (*e.g.,* in the first scenario, register comes before check). We will detail this scenario at the end of the section.

The third line in Table I shows an example of the *bad scenario* strategy. This strategy uses the automaton and at the second step, it chooses a nonexistent transition but with an existing method: it calls register before deposit, which is not allows by the automaton. If the pre/post conditions of methods have been well designed, the execution of the method is not possible. In our example, it is effectively detected since the precondition of register checks that a deposit has already been executed. This strategy allows designers to test the robustness of the application especially for SoD and obligation constraints.

The last lines of the table are dedicated to the *all transitions* strategy. This strategy tries to cover exhaustively the automaton. Thus, nothing is needed (excepted the maximal length of the scenario) for this strategy which computes automatically the set of scenarios. In our example, it finds 6 scenarios.

Figure 6 illustrates an extract of the final code generated by the testing tool for the first nominal scenario. Lines 7, 13 and 28 give the method that will be executed, with the test preamble (empty for Step 0). Each step of the scenario consists in setting the test data (empty for Step 0; for Step 1, lines 15 to 20; for Step 2, lines 30 to 33), calling the business code (line 22 for Step 1, line 35 for Step 2) and testing the oracle (lines 23-25 for Step 1, lines 36-38 for Step 2). When executing the test, the tester has to provide the value of aca parameters. Thus, wrong instantiations can be detected. Obligation and SoD constraints violations can also be detected if the correct value of aca parameters are not provided.

For example, consider the following obligation `OBL(user,<user,_,_,deposit> ,<user,_,_,register>)`, with the values of the relations permissions and play given in Section II. The instantiation `<boris,clerk,montreal>` of the aca parameters for the deposit method followed by the instantiation `<boris,clerk,montreal>` of the aca parameters for the register method is correct and will be accepted by the test. On the contrary, the instantiation `<boris,clerk,montreal>` of the aca parameters for the deposit method following by the instantiation `<damien,banker,montreal>` of the aca parameters for the register method is not correct and will be rejected by the test.

Note that both the functional and the AC models are tested together. This allows us to check that AC constraints, specifically SoD and obligations constraints, prevent actions to be executed and then do not block the system.

C. Dynamic verifications of AC rules

The ACA extension of the Tamago platform is composed of a library for the dynamic framework and a generator, based on the architecture of Tamago-CC. This generator produces the code for the implementation of the new container, called CheckDepositContainer_ACA, that implements the interface CheckDepositContainer (see Figure 4). The code is automatically generated from the GenDepositSecurity service.

Figure 7 illustrates the workflow in the hierarchy of containers in order to monitor access controls and the functional contract. On the left, when the user makes an action,

Strategy	Scenarios					
	0	1	2	3	4	5
Nominal	init	deposit	register	check	validate_dir	validate
	init	deposit	check	register	validate	validate_dir
Bad scenario	init	register				
All transitions	init	deposit	register	check	cancel	
	init	deposit	register	check	validate_dir	validate
	init	deposit	register	check	validate	validate_dir
	init	deposit	check	register	cancel	
	init	deposit	check	register	validate_dir	validate
	init	deposit	check	register	validate	validate_dir

Table I: Summary of test scenario produced by Tamago-Test

```
package tamago.aca.bank;
public class TestDepositSecurityNominal extends junit.framework.TestCase {
    // Members Variables
    private DepositSecurity code;
        ...
    public void testScenario0(){
        // Step: 0 Method: init Transition: ninit—{init}—>initialized
        try {code.init();
            boolean __tamagotest_oracle = ((code.getOpNumber() < 0) && acaInitialised);
            assertTrue(__tamagotest_oracle);}
        catch(Exception exc) {
            fail(("Test failed at step 0. " + exc.getMessage()));}
        // Step: 1 Method: deposit Transition: initialized—{deposit}—>deposed
        try {// TODO fulfill the parameter: aca
            tamago.ext.aca2.ACA aca;
            if(!(code.getPlay().isCorrectACA(aca)
                && (("alphonse".equals(aca.user) || ...)
                    && ("customer".equals(aca.role) || ...) && ...)
                && "clerk".equals(aca.role) && "banker".equals(aca.role)
                && !"customer".equals(aca.role)))
            {    fail("Wrong Test");}
            code.deposit(aca);
            boolean __tamagotest_oracle = ("deposit".equals(code.getHistoric().lastAction())
                    && (aca == code.getHistoric().lastSecuParam()));
            assertTrue(__tamagotest_oracle);}
        catch(Exception exc) {
            fail(("Test failed at step 1. " + exc.getMessage()));}
        // Step: 2 Method: register Transition: deposed—{register}—>registered
        try {// TODO fulfill the parameter: aca
            tamago.ext.aca2.ACA aca;
            if(!(code.getPlay().isCorrectACA(aca) && (("alphonse".equals(aca.user) || ...) && ...)
                && code.getHistoric().mustDone("deposit")
                && (code.getHistoric().getSecuParamFromID("deposit").getUser() == aca.user)))
            {    fail("Wrong Test");}
            code.register(aca);
            boolean __tamagotest_oracle = ((code.getOpNumber() > 0) && ("register".equals(code.getHistoric().lastAction())
                    && (aca == code.getHistoric().lastSecuParam())));
            assertTrue(__tamagotest_oracle);}
        catch(Exception exc) {fail(("Test failed at step 2. " + exc.getMessage()));}
        ...}}
```

Figure 6: Extract of the test case for the first nominal scenario

the access control container CheckDepositContainer_ACA catches the indirection and checks the preconditions of the deposit method in the GenDepositSecurity service. Then it calls the delegate CheckDepositContainer_plugin that checks the preconditions of the deposit method in the DepositService service. If everything is okay, then the business code of the action is executed. At the end, the CheckDepositContainer_plugin container verifies that the effect of the business code conforms with the postconditions of the deposit method in the DepositService service. Finally, the CheckDepositContainer_ACA container verifies that the postconditions of the deposit method in the DepositService service are satisfied and gives back the answer to the user. If an error occurs at the access control level or the functional level, the CheckDepositContainer_ACA container throws an exception with details of the problem.

V. Related works

A substantial overview of verification methods applied to formal AC models can be found in [17]. A number of AC modelling methods are based on UML (UMLSec [18], secureUML [10]) since UML is widely used in IS design. Contextual constraints can be expressed in a formal way with OCL. Some tools have been designed to verify AC

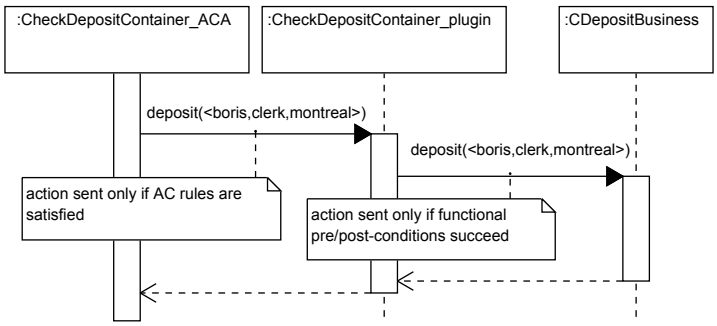


Figure 7: Tamago-ACA sequence diagram

policies expressed in UML/OCL. The USE tool (UML-based Specification Environment) [19] was designed to verify predefined properties on the model. During its verifications, the USE tool does not take into account the evolution of the functional parts of the IS. The SecureMOVA [20] tool is designed with the same goal but takes into account the evolution of the functional part of the IS.

Other works also aim at verifying AC policies. The difference, between these works and USE or SecureMOVA, is that they are based on the translation of AC models into formal languages, such as Z or CSP, which are equipped with model checking or theorem proving tools. In [21], an RBAC model enhanced with SoD constraints is translated into Alloy. No prohibition and no obligation can be expressed in the model. Properties can be verified on AC policies, but the functional part of the IS is not taken into account. In [22], the model used to express AC policies is an RBAC model with role hierarchy constraints. Policies expressed with this model are translated into Alloy. Properties can then be verified on the AC model only. In [23], an RBAC model enhanced with dynamic SoD constraints is used to express AC policies. These policies are translated into Z for verification without taking into account the functional part of the IS. In [24], the AC policies are expressed with an RBAC model extended with contextual constraints and SoD. The AC policy and the functional part of the IS are translated into Z. Verifications are made on the static part of the model and simulation techniques are used to test the accuracy of the dynamic part. Verification and simulation are made with the Jaza tool. In [25], an RBAC model and SoD constraints are translated into CSP. SoD constraints are expressed in SoDA [6]. In addition to the AC policy, the functional part of the IS is translated into CSP. Then, properties can be checked on the AC policy and the functional part of the IS. In [7], [26], AC policies can be expressed with the OrBAC extended with SoD constraints. Verifications can be made on the model, taking into account the functional part of the IS. All these works use model checking techniques and then face the problem of combinatorial explosion.

In comparison, in our approach, ACA allows designers to express AC policies with permissions, prohibitions, SoD and obligations. Inconsistencies between permissions and prohibitions can be detected [27]. We use the Tamago platform to generate test scenarios to simulate the behaviour of the dynamic part of an AC policy. For the simulation of the dynamic part, the functional part of the IS is also taken into account.

VI. Conclusion and Future Work

This paper has addressed the problem of verification of AC policies. We have proposed to equip the AC language, called ACA, with verification techniques. ACA is based on RBAC and allows the specification of permissions, prohibitions, SoD and obligations in the same model. Different kinds of verification are then possible by using the Tamago platform, extended to consider AC properties. Extensions of the platform have been easy to achieve since its architecture is based on components organised in hierarchical layers. We have added an external layer to deal with AC constraints. Furthermore, the Tamago language allows the definition of contracts described by pre and post conditions and automata. AC rules expressed in ACA have been automatically translated into new security contracts. Static analyses allows for the detection of inconsistencies between AC constraints. Tamago-Test has been adapted to generate test scenarios specific for testing AC rules. Finally dynamic verifications can now be carried out thanks to the definition of a new security component in the monitoring framework. It is interesting to note that the last two points (test and monitoring) take into account the functional and the AC models to check the consistency of the whole system.

The next step is to provide more powerful static analyses. Two venues are considered. First, we want to exploit the *type builder* in Tamago-Test in order to teach to the underlying constraint satisfaction solver the semantics of the properties introduced in our translation, the play and history properties. Consequently, Tamago-Test could directly produce concrete values for the generated tests and the right instantiation in the final test code. The second venue consists in specifying a new test strategy dedicated to AC.

Finally, a number of long-term perspectives can be considered. The first one is to extend the ACA language to support contextual constraints in AC rules. Another one deals with traceability between the ACA rules and their Tamago specification. Indeed, if an ACA rule is violated after Tamago verifications, it could be useful to send back the error in the ACA model to correct it.

References

[1] *Role-Based Access Control*, 359th ed., INCITS, ANSI, Februar 2004.

[2] P. Konopacki, M. Frappier, and R. Laleau, "Expressing access control policies with an event-based approach," in *CAiSE Workshops*, ser. LNBIP, C. Salinesi and O. Pastor, Eds., vol. 83. Springer, 2011, pp. 607–621.

[3] M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, and M. Ouenzar, "Comparison of model checking tools for information systems," in *ICFEM*, ser. Lecture Notes in Computer Science, J. S. Dong and H. Zhu, Eds., vol. 6447. Springer, 2010, pp. 581–596.

[4] H. Belhaouari and F. Peschanski, "Automated generation of test cases from contract-oriented specifications: A csp-based approach," in *11th IEEE High Assurance Systems Engineering (HASE)*. IEEE, 2008, pp. 219–228.

[5] Q. Ni, E. Bertino, and J. Lobo, "An obligation model bridging access control policies and privacy policies," in *Proceedings of the 13th ACM symposium on Access control models and technologies*, ser. SACMAT '08, 2008, pp. 133–142.

[6] N. Li and Q. Wang, "Beyond separation of duty: An algebra for specifying high-level security policies," *J. ACM*, vol. 55, no. 3, pp. 1–46, 2008.

[7] A. A. E. Kalam, S. Benferhat, A. Miège, R. E. Baida, F. Cuppens, C. Saurel, P. Balbiani, Y. Deswarte, and G. Trouessin, "Organization based access control," in *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, ser. POLICY '03, 2003, pp. 120–130.

[8] A. Brown, S. Johnston, and K. Kelly, "Using service-oriented architecture and component-based development to build web service application," *Rational Software white paper*, 2002.

[9] R. Reussner, I. Poernomo, and H. W. Schmidt, "Reasoning about software architectures with contractually specified components." in *Component-Based Software Quality*, ser. LNCS, vol. 2693. Springer, 2003, pp. 287–325.

[10] T. Lodderstedt, D. A. Basin, and J. Doser, "Secureuml: A uml-based modeling language for model-driven security," in *Proceedings of the 5th International Conference on The Unified Modeling Language*, 2002, pp. 426–441.

[11] K. L. McMillan, "Symbolic model checking," Ph.D. dissertation, Carnegie Mellon University, 1992.

[12] H. Belhaouari and F. Peschanki, "A constraint logic programming approach to automated testing," in *ICLP*, ser. LNCS, vol. 5366. Springer, 2008, pp. 754–758.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Boston, MA: Addison-Wesley, January 1995.

[14] C. Oriat, "Jartege: A tool for random generation of unit tests for java classes," in *2nd International Workshop on Software Quality - SOQUA'05*, vol. 3712. Erfurt, Allemagne: LNCS, Sept 2005, pp. 242–256.

[15] Y. Cheon, "Automated random testing to detect specification-code inconsistencies," in *Proceedings of the 2007 International Conference on Software Engineering Theory and Practice, July 9-12, 2007, Orlando, Florida, U.S.A*, Jul. 2007, pp. 112–119.

[16] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed Random Testing for Java," in *OOPSLA 2007 Companion, Montreal, Canada*. ACM, Oct 2007.

[17] Y. Ledru, A. Idani, J. Milhau, N. Qamar, R. Laleau, J.-L. Richier, and M.-A. Labiadh, "Taking into account functional models in the validation of is security policies," in *1st International Workshop on Information Systems Security Engineering host by CAISE*, 2011.

[18] J. Juerjens, *Secure Systems Development with UML*. SpringerVerlag, 2003.

[19] M. Gogolla, F. Büttner, and M. Richters, "Use: A uml-based specification environment for validating uml and ocl," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 27–34, 2007.

[20] D. A. Basin, M. Clavel, J. Doser, and M. Egea, "Automated analysis of security-design models," *Information & Software Technology*, vol. 51, no. 5, pp. 815–831, 2009.

[21] A. Schaad and J. D. Moffett, "A lightweight approach to specification and analysis of role-based access control extensions," in *Proceedings of the seventh ACM symposium on Access control models and technologies*, 2002, pp. 13–22.

[22] J. Zao, H. Wee, J. Chu, and D. Jackson, "Rbac schema verification using lightweight formal model and constraint analysis," in *Proceedings of the 8th ACM symposium on Access control models and technologies*, 2003.

[23] C. Yuan, Y. He, J. He, and Z. Zhou, "A verifiable formal specification for rbac model with constraints of separation of duty," in *Information Security and Cryptology*, ser. LNCS, H. Lipmaa, M. Yung, and D. Lin, Eds., 2006, vol. 4318.

[24] Y. Ledru, N. Qamar, A. Idani, J.-L. Richier, and M.-A. Labiadh, "Validation of security policies by the animation of z specifications," in *SACMAT*, R. Breu, J. Crampton, and J. Lobo, Eds. ACM, 2011, pp. 155–164.

[25] D. A. Basin, S. J. Burri, and G. Karjoth, "Dynamic enforcement of abstract separation of duty constraints," in *ESORICS*, ser. Lecture Notes in Computer Science, M. Backes and P. Ning, Eds., vol. 5789. Springer, 2009, pp. 250–267.

[26] S. Ayed, N. Cuppens-Boulahia, and F. Cuppens, "Managing access and flow control requirements in distributed workflows," in *Proceedings of the 2008 IEEE/ACS International Conference on Computer Systems and Applications*, pp. 702–710.

[27] P. Konopacki, H. Belhaouari, M. Frappier, and R. Laleau, "Specification and verification of access control policies in eb3sec: Work in progress," in *FPS*, ser. LNCS, J. García-Alfaro and P. Lafourcade, Eds., vol. 6888, 2011, pp. 227–233.