SGAC: a Multi-Layered Access Control Model with Conflict Resolution Strategy

Nghi Huynh [a,b], Marc Frappier[a], Herman Pooda[a], Amel Mammar[c], Régine Laleau[b]

[a] *Université de Sherbrooke, Québec, Canada*
[b] *Université Paris Est-Créteil, Val de Marne, France*
[c] *SAMOVAR, Télécom SudParis, CNRS, Université Paris-Saclay, EVRY, France*

Abstract

This paper presents SGAC (*Solution de Gestion Automatisée du Consentement / automated consent management solution*), a new healthcare access control model and its support tool, that manages patient wishes regarding access to their electronic health record (EHR). This paper also presents the verification of access control policies for SGAC using two first-order-logic model checkers based on distinct technologies, Alloy and ProB. The development of SGAC has been achieved within the scope of a project with the University of Sherbrooke Hospital (CHUS), and thus has been adapted to take into account regional laws and regulations applicable in Québec and Canada, as they set bound to patient wishes: for safety reasons, under strictly defined context, patient consent can be overriden to protect his/her life (break-the-glass rules). Since, patient wishes and those regulations can be in conflict, SGAC provides a mechanism to address this problem based on priority, specificity and modality. In order to protect patient privacy while ensuring effective caregiving in a safety-critical situations, we check four types of properties: accessibility, availability, contextuality and rule effectivity. We conducted performance tests comparison: implementation of SGAC versus an implementation of another access control model, XACML, and property verification with Alloy versus ProB. The performance results show that SGAC holds better performance than XACML and that ProB outperforms Alloy with two order of magnitude thanks to its programmable approach to constraint solving.

Keywords: healthcare, access control, consent management, formal model, verification, Alloy, ProB,

1. Introduction

Patient data are nowadays not anymore stored physically: the data is digitalised and stored directly in the electronic health records of the patient. The use of Electronic Health Records (EHR) is widely spreading and each hospital can have its own EHR for its patients. The trend is to gather data across several EHRs into one unique EHR for each patient: this way, data sharing between different health workers is easier and the effectiveness of the caregiving increased. In Québec, access to the former health paper record is managed by specially trained staff, the archivists, who are responsible for enforcing the laws and regulations on access requests.

Laws and regulations allow patient to decide who can or cannot access their data, as long as they are not endangering themselves or others by hiding crucial data. Access control applied to healthcare knows two major concerns that can be in conflict: patient privacy and patient safety. The former is about non-disclosure of data, that the owner would judge confidential, to any unauthorised party, and the latter is about keeping patients alive and healthy by providing them suitable care, which cannot be done when the access control is too restrictive and prevents health workers from accessing all the necessary informations to provide care.

Having patients choosing who can or cannot access their records (thus expressing their consent) is a way to address the first concern. The second is taken care of by laws and regulations which specify strict contexts under which patient consent can be overridden, in order to save the patient's life for instance. That is the case when a patient is in a life threatening condition: the context of emergency allows physicians to

access the patient's record without restriction. But this approach raises multiple problems: the laws and regulations are fitted for exceptional cases where a patient's life is directly threatened but they do not take into account everyday care. This can lead to serious complications when the patient is hiding important data such as medicinal allergies. Furthermore, conflicts may arise between hospital rules, which define the regular access of each health worker to any patient records under a "normal" context, patient rules, and break-the-glass rules, which provide full access for the health workers in strictly defined contexts.

In this paper, we present an access control method named SGAC (*Solution de gestion automatisée du consentement* / Automated consent management solution) which offers a resolution mechanism to the different conflicts that may occur between the different sources. This method allows for formal verification in order to detect cases where suitable care cannot be given. In order to be used in real life, this verification has to be done automatically. Thus we also present the translation of SGAC into B and Alloy, two tools that have model-finding and verification capabilities.

This paper is structured as follows: Section 2 presents our access control method, our motivation and the requirements we meet. Notation and data structures are introduced to describe the behaviour of SGAC. The formalisation of SGAC is introduced in Section 3. We also discuss how we model SGAC, the choices we made to capture all required concepts efficiently and the use of this formalisation to detect potential dangers. This detection can be automated with the use of tools like ProB and Alloy, presented in Section 4. We provide in Section 5 performance tests of two aspects of the SGAC method. First, we compare the response time of an implementation of SGAC developed by CHUS and GRIL with Balana, an open source implementation of XACML, an industry standard for access control. Second, we compare the verification time of four types of properties (accessibility, availability, contextuality and rule effectivity) in Alloy and ProB for SGAC policies. Section 6 compares our findings with similar work on property verification on access control models. We conclude this paper with an appraisal of our work in Section 7.

2. SGAC: handling patient consent

This section introduces the SGAC access control model, and the motivations in its design. This work has been done in partnership with the University of Sherbrooke Hospital (CHUS — Centre Hospitalier de l'Université de Sherbrooke). CHUS wanted to take into account patient consent in accessing EHR. This provided a great opportunity with real needs and requirements.

2.1. Access Control Requirements at CHUS

Our access control model has been designed to meet the requirements of CHUS within the context of applicable laws on privacy protection in Québec and Canada. We believe these requirements are sufficiently general to be applicable in other countries as well.

Req. 1: The patient's consent must be obtained in order to provide access to his/her electronic health record (EHR).

Req. 2: A patient can grant or deny access to any part of his/her EHR to any person of the hospital staff.

Req. 3: As required by the laws of Québec, when the patient's life is in danger, the medical staff must have access to his/her EHR, without regards to his/her consent. Other conditions, like a court order, can also override the patient's rules.

Req. 4: Rules can be specified for a single person or a group of persons. Persons can be grouped according to any criteria, like functional role, work group, departments, care unit, etc.

Req. 5: Rules can be specified for a single record or a group of records. Records can be grouped according to the taxonomy commonly used for EHR.

Req. 6: When several rules are applicable for a user request, they must be ordered according to the following priority to determine which rule prevails: the rules prescribed by laws override the patient's rules; the rules of the patient override the rules of the hospital.

Req. 7: For two rules with the same level of priority, a rule which targets a group of person G_1 has precedence over a rule targeting a less specific group of persons G_2, (ie, when $G_1 \subset G_2$).

Req. 8: For two rules with the same level of priority, when neither of the two groups of persons is more specific than the other (*i.e.*, when $\neg(G_1 \subset G_2 \vee G_2 \subset G_1)$), a prohibition rule overrides a permission rule.

Req. 9: Each rule has a condition that determines its applicability. This condition can refer to any attribute that can be computed using the context of the clinical system (*e.g.*, the state of the patient, the presence of the patient in the hospital, etc).

Req. 10: The access control system shall be able to handle a very large volume of data, hundreds of thousands of patients and rules.

To illustrate some of these requirements, we provide the following scenarios. In these scenarios, Anna and Sam are patients, Alice is a nurse and Bob is a doctor. For each scenario, we refer to the requirements that it illustrates.

Scenario 1 - Group prohibition

Anna wants to deny access to her psychiatric records to the entire hospital staff.
Requirements: Req. 2, Req. 4 and Req. 5.

Scenario 2 - Record taxonomy

Sam has two laboratory results, *lab1* and *lab2*. He authorises hospital staff to access all his laboratory results. Later, Sam receives a third laboratory result, *lab3*.
Expected behaviour: all requests from hospital staff to access Sam's laboratory results, including *lab3* should be permitted.
Requirements: Req. 5.

Scenario 3 - Priority

Sam wishes to grant all hospital staff access to his blood tests, DNA tests and psychiatric records. However, there is a law that restricts access to psychiatric records to psychiatrists only.
Expected behaviour: all requests from hospital staff, other than psychiatrists, to access Sam's psychiatric records are denied; all requests of hospital staff to access Sam's blood and DNA record are permitted.
Requirements: Req. 6.

Scenario 4 - Specificity

Anna wants to deny Alice access to her laboratory results. Anna also has a rule granting nurses access to her laboratory results.
Expected behaviour: all requests from nurses, except Alice, to access Anna's laboratory results are permitted; Alice can't access Anna's laboratory results.
Requirements: Req. 7.

Scenario 5 - User group specificity

Anna specifies two rules: the first rule denies emergency staff access to her EHR; the second rule grants general practitioners access to her EHR. Bob, working in both department, requests access to Anna's EHR.
Expected behaviour: Bob's request should be denied, since the group of general practitioners is not more

specific than the group of emergency staff, and vice-versa.
Requirements: Req. 8.

Scenario 6 - Condition

Sam wants to specify rules that are valid in certain contexts: he want access to his EHR to be allowed only when he is hospitalised; when he is not hospitalised, Sam wants to deny access to his EHR to all hospital staff.
Requirements: Req. 9.

2.2. Data structures

This section presents a formal model of SGAC and the different data structures needed to specify rules and requests. Conflict resolution is then illustrated by different examples. Notations are first introduced.

2.2.1. Notation

For the rest of the paper, we introduce the following notations drawn, in most cases, from the B notation [1].

Set Theory. Let A, B, C be sets.

- For a n-tuple $a = (a_1, \ldots, a_n)$, we denote by $a.a_k$ the component of a named a_k.
- $\mathcal{P}(A) = \{X \mid X \subseteq A\}$, called the power set of A, is the set of all subsets of A.
- $A \times B = \{x \mapsto y \mid x \in A \wedge y \in B\}$ is the Cartesian product; it is a set of ordered pairs $x \mapsto y$.
- A relation R from A to B is a subset of $A \times B$.
- $A \leftrightarrow B = \mathcal{P}(A \times B)$ denotes the set of relations between A and B.
- $\mathsf{id}(A) = \{x \mapsto x \mid x \in A\}$ denotes the identity relation on A, i.e. the relation that associates each element of A to itself.
- $\mathsf{dom}(R) = \{x \in A | \exists y \in B \bullet x \mapsto y \in R\}$ denotes the domain of R.
- $R[C] = \{y \mid y \in B \wedge \exists x \in C \bullet x \mapsto y \in R)\}$ denotes the image set of C by relation $R \in A \leftrightarrow B$.
- $A \nrightarrow B$ denotes the set of (partial) functions from A to B. A partial function f from A to B is a relation such that $|f[\{x\}]| \leq 1$ for $x \in A$.
- $A \rightarrow B$ denotes the set of total functions from A to B. A total function f is a function such that $\mathsf{dom}(f) = A$.
- $R^{-1} = \{x \mapsto y \mid y \mapsto x \in R\}$ is the inverse of relation R.
- $R_1 \fatsemi R_2 = \{x \mapsto z \mid \exists y \in B \bullet x \mapsto y \in R_1 \wedge y \mapsto z \in R_2\}$ is the relational compositon of $R_1 \in A \leftrightarrow B$ and $R_2 \in B \leftrightarrow C$.
- Let $R \in A \leftrightarrow A$. R^n denotes the composition of R with itself n times ($n \geq 0$), with $R^{n+1} = R \fatsemi R^n$ and $R^0 = \mathsf{id}(A)$.
- $R^+ = \bigcup_{n \geq 1} R^n$ denotes the transitive closure of R, *i.e.*, the smallest transitive relation which contains R.
- Let $R \in A \leftrightarrow A$. $R^* = R^+ \cup \mathsf{id}(A)$ denotes the transitive and reflexive closure of R, *i.e.*, the smallest transitive and reflexive relation which contains R.

Graph. A *directed graph* is an ordered pair $G = (V, E)$ where V is the set of *vertices* and E is the set of *edges*, such that $E \in V \leftrightarrow V$. G is said *acyclic* iff $G.E^+ \cap \mathsf{id}(G.V) = \emptyset$. In an edge $x \mapsto y$, y is called a *successor* of x and x a *predecessor* of y. In an edge $x \mapsto y$ of $G.E^+$, *i.e.*, the transitive closure of $G.E$, x is an ancestor of y and y is a descendant of x. A vertex without any successor is called a *sink* and sinks reachable from (*i.e.*, descendant of) a vertex v in a graph G are denoted by $sink(G, v) = G.E^*[\{v\}] - \mathsf{dom}(G.E)$. All the sinks of a graph G are denoted by $sink(G) = G.V - \mathsf{dom}(G.E)$.

2.3. Subject and resource type graphs

In SGAC, two directed acyclic graphs (DAG) are needed in order to specify rules and requests:

- the subject graph represents the hierarchy which mirrors the functional organisation chart or any grouping of users relevant for access control;
- the resource type graph represents the taxonomy of EHR and their organisation in the healthcare facilities.

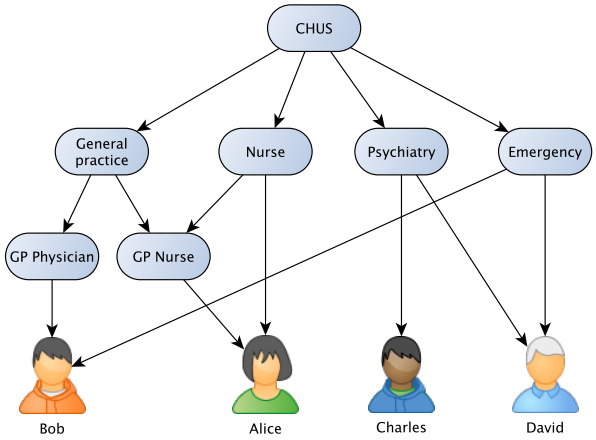
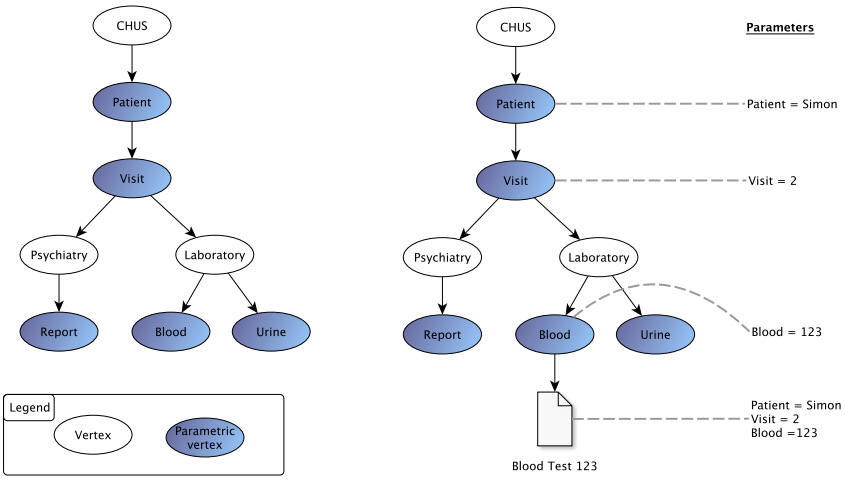


Figure 1: Subject graph example

Figure 1 illustrates a subject graph. The graph includes people and subjects as vertices. A subject represents a person or a set of people. The hierarchy works as follows: a rule on subject s is inherited by all the successors of s in $G.E^+$. For instance in Figure 1, if a permission is given to the *General Practice* department then this permission is inherited by *GP Physician* and *GP Nurse*, *Bob* and *Alice*.

Figure 2a illustrates the resource type graph. We distinguish between *resources types* and *documents*. Medical records are structured into a taxonomy which is represented by a graph of resource types. A document is an actual medical record of a patient. The sinks of the resource type graph are called *document types*. A document is an instance of a particular document type. Documents are not vertices of the resource type graph; thus the resource type graph does not grow with the creation of documents. A non-sink vertex can be *parametric*; it is denoted by a blue vertex in the graph. A parametric vertex is inherited by its descendant vertices, including document types. It is also an attribute that can be valued for a document and tested in a condition of a rule. A document type (*i.e.*, a sink vertex) is always parametric; its value, called the *document identifier*, uniquely identifies a document. There is a functional dependency between the document type identifier and the other attributes, making the *document identifier* a key which is sufficient to retrieve a document, and all its attributes. A resource type graph can be instantiated by adding documents as successors of their document types. Figure 2b illustrates the resource type graph instantiated with the document Blood Test 123. This document has three attributes: Patient, Visit and Blood, inherited from the ancestors of its document type. The values of these attributes respectively are Simon, 2 and 123.



(a) Resource type graph (b) Instantiated resource type graph

Figure 2: resource type graph example

The resource type graph sinks are document types, and the non-sink vertices represents aggregations of these document types. For instance, the vertex patient of Figure 2a represents all the document types of all patients and can be instantiated with a parameter to target all the documents of a particular patient.

The subject graph and the resource type graph define the basis on which rules and requests are defined. An instantiated resource type graph is used to evaluate a request.

2.4. Rule and request specification

A rule allows to specify control over the access to a document. It is defined by:

- a *subject*: a person or a group of people to control;
- a *resource*: the documents to be protected;
- a *resource condition*: a condition on the document attributes;
- an *action*: the operation the *subject* wants to do on the *document*;
- a *priority*: a number which defines the priority of the rule;
- a *modality*: an authorisation or a prohibition which defines the effect of the rule;
- a *condition*: a formula which determines the applicability of the rule. It can be evaluated at run time by functions checking for instance information stored in a database. For the sake of simplicity, we describe rule conditions in natural language, but they can easily be represented by a logical formula in SGAC.

A request is the demand the *subject* issues in order to execute an *action* on a *document*. It has the following attributes:

- a *subject*: the request initiator;
- a *document*: a document the request initiator wants access to;
- an *action*: the operation the *subject* wants to do on the *document*.

A rule applies to a request iff:

- the request subject is a descendant of the rule subject, and
- the request resource is a descendant of the rule resource, and
- the request document's attribute values satisfies the rule resource condition, and
- the request action is the same as the rule action, and
- the rule condition holds.

2.5. Conflict resolution

When more than one rule apply to a request, and if they have different modalities, a situation, typically called a *conflict* in the literature, arises. To decide whether access is granted or denied, we define an ordering (a precedence) on rules. The rule with the "highest" precedence determines the access decision. Let r_1, r_2 be two applicable rules for a request.

1. If r_1 has a smaller priority than r_2, we say that r_1 has precedence over r_2.
2. If r_1 and r_2 have the same priority, and if the subject of r_1 is more specific than the subject of r_2 (*i.e.*, the subject of r_1 is a descendant of the subject of r_2 in the subject graph), then r_1 has precedence over r_2.
3. If r_1 and r_2 have the same priority, and neither of their subjects is more specific than the other, then a prohibition has precedence over a permission.

This ordering is not total. There may be two rules r_1, r_2 such that neither of them precedes the other. However, in such a case, r_1 and r_2 have the same modality, thus there is no conflict and the decision is the modality of these elements with highest precedence. The formal definition of this ordering in Section 3 shall clarify the third clause in some subtle cases, to avoid any ambiguity in its interpretation.

This conflict resolution method is absolutely autonomous and does not require the intervention of an external actor.

2.6. Examples

This section illustrates the behaviour of SGAC rules with three examples. For the sake of simplicity, we illustrate *read* requests. The same approach applies for any other action.

2.6.1. Example 1: basics

Let's model scenario 1. Modelling Anna's rule consists in prohibiting access to the documents descending from the vertex Psychiatry in the resource type graph. The parametric vertex Patient allows to identify Anna's documents. The vertex CHUS in the subject graph (Figure 1) represents all the personnel from the hospital. By convention, patient rules are of priority 2. When no condition is specified, the rule condition is set to TRUE. A prohibition is represented by symbol "−", whereas a permission is represented by symbol "+". The rule is presented in Table 1; rule actions are typically omitted in our examples for the sake of concision, because they are easy to understand and deal with. The resource of the rule is Psychiatry, which is a descendant of parametric vertex Patient in the resource type graph. Thus, resource Psychiatry inherits parameter Patient, and the value of this parameter can be tested for equality in the resource condition of the rule. Note that this test might as well have been represented in the condition part of the rule; we have made the design choice in SGAC to separate resource conditions from the rest of the rule condition, for readability and understandability reasons.

If Bob requests an access to Anna's psychiatric report no. 20, then SGAC will first determine the applicable rules. Rule r_1 is applicable because: $r_1.subject$ is an ancestor of the request subject, $r_1.resource$ is an ancestor of the requested resource, the rule resource condition is satisfied, the action matches and the condition is verified. If this is the only rule applicable, then the system returns prohibition. We only described the rule issued by Anna's consent for the sake of simplicity in this example. In the case where no rules from laws and regulations are applicable, if Anna's rule is among the other rules applicable to a request, then this request is denied.

Rule	Resource	Resource condition	Subject	Pri.	Mod.	Cond.
r_1	Psychiatry	Patient = Anna	CHUS	2	–	TRUE

Table 1: Scenario 1 rule (for the action read)

2.6.2. Example 2: let's get started

In this example, the rule base is as follow:

- the laws and regulations allow emergency physicians to access the data of any patient who is in a life-threatening situation;
- the hospital allows general physicians to access data for any patient under their care;
- the hospital allows nurses to access vitals of a patient at any time.

Rule	Resource	Subject	Pri.	Mod.	Cond.
r_1	Patient	Emergency	1	+	patient life is threatened
r_2	Patient	GP Physician	3	+	the subject is the attending physician
r_3	Vitals	Nurses	3	+	TRUE

Table 2: Example 2 rules

This can be represented by the rule base presented in Table 2. By convention for these examples, the priority of a rule is determined by the entity issuing the rule: if the rule is from a healthcare facility, then it is set to 3, if it is from the patient, then priority is set to 2, and if the rule is from laws and regulations, priority is set to 1. The lower the value a rule priority has, the higher precedence the rule gets. This reflects the wanted behaviour: laws and regulation have precedence over patient rules, which have precedence over healthcare facility rules.

Rule r_1 represents the fact that any subject in the Emergency department can access a record if its owner's life is threatened: an authorisation given to the vertex Emergency to access all documents from Patient, under the specified condition. The priority is set to 1 since the rule stems from the laws and regulations.

The rule r_2 represents the fact that a physician is allowed to access the data of the patients under his/her care, i.e. the physician has to be the patient's attending physician: an authorisation given to the vertex GP Physician to access all documents from Patient, under the condition that the physician is the attending physician of the patient. The priority of this rule is set to 3 since the rule stems from the hospital.

Finally, the rule r_3 represents the fact that a nurse is allowed to access the vitals of any patient, at any time. Since, the nurse can access the Vitals of any Patient in any condition, the condition of r_3 is set to TRUE. The priority is also set to 3 since the rule stems from the hospital too.

In order to have a better understanding of the rules, the subject graph, the resource type graph and the rules are presented in the same picture in Figure 3.

Now let's say that patient Anna is treated for some light mental disorder by Charles, a psychiatrist. and physician. Since Charles is Anna's attending physician, he can access her records while others can't except Alice who can read Anna's vitals. The access rights are summed up in Table 3.

Staff	Pulse	Blood Pressure	Report	Blood	Urine
Alice	✓	✓	×	×	×
Bob	×	×	×	×	×
Charles	✓	✓	✓	✓	✓
David	×	×	×	×	×

Table 3: Example 2: Access of the CHUS personnel to Anna's Record, wrt Figure 3

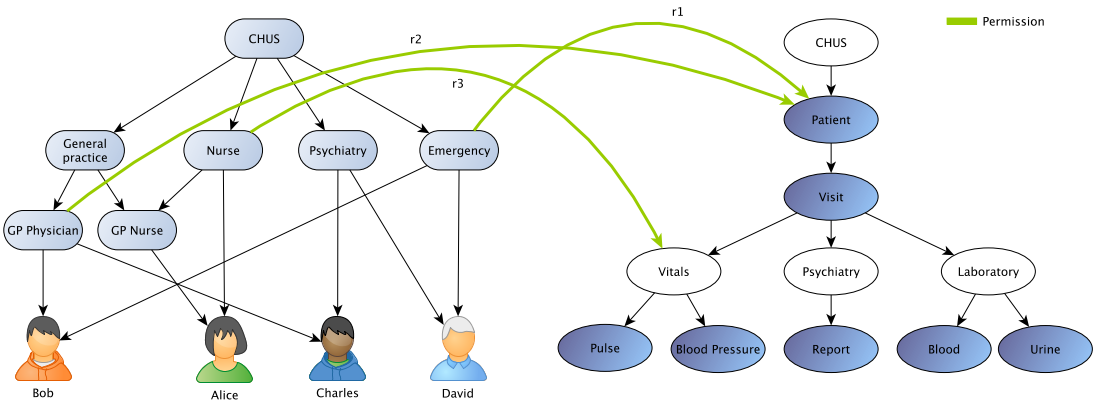


Figure 3: Example 2 graphs with rules

Then comes Sam, badly hurt, unconscious in the Emergency department. Since, Bob and David are working in the Emergency department and that Sam's life is threatened, both have access to his records. Alice still can read Sam's vitals while Charles does not have any access to Sam's data. The resulting accesses are presented in Table 4.

Staff	Pulse	Blood Pressure	Report	Blood	Urine
Alice	✓	✓	×	×	×
Bob	✓	✓	✓	✓	✓
Charles	×	×	×	×	×
David	✓	✓	✓	✓	✓

Table 4: Example 2: Access of the CHUS personnel to Sam's Record, wrt Figure 3

Finally, in the case of a patient who has no attending physician, and whose life is not threatened, the only person who can access this patient's records is Alice, who is allowed to read the vitals.

2.6.3. Example 3: adding consent

When a patient is admitted at CHUS, he/she signs a form that says he/she accepts the default consent and access control policy of the hospital. However, a patient can add his/her own rules to override the default policy. In this example, we take the same initial rule base (Table 2), and we add some consent rules. Let's say Anna personally knows Bob and does not want him to access her records (rule r_4). This rule targets directly Bob and Anna's data, and is applicable at any time. Since r_4 is directly issued by a patient, its priority is set to 2. At this point, even if Anna is under Bob's care, Bob won't have access to Anna's records because of r_4, unless there is an emergency context where Anna's life is threatened. In that case, r_1 would allow him to access the data.

Then, Anna is hospitalised and gets on with the staff of the Emergency department. When she has to undergo rehabilitation, she decides to allow the whole Emergency department to access her vitals data in order to let her new friends follow her progress (rule r_5). In this situation, there is a conflict between r_4 and r_5 when Bob wants to access Anna's vitals. Bob still can't access any data of Anna, since r_4 is considered to have precedence over r_5 since the target of r_4 is more specific than the target of r_5, but David who is also under the scope of r_5 can access Anna's vitals. The accesses at this point are presented in Table 5.

Finally, Anna decides to share her vitals to Bob and she adds a new rule, r_6, to do so, but forgets to remove r_4. These two rules contradict each other: they have the same priority, and one is not more specific than the other. In that case, a prohibition has precedence over a permission. Bob's access is unchanged: he

Staff	Pulse	Blood Pressure	Report	Blood	Urine
Alice	✓	✓	×	×	×
Bob	×	×	×	×	×
Charles	×	×	×	×	×
David	✓	✓	×	×	×

Table 5: Example 3: Access of the CHUS personnel to Anna's Record

Rule	Resource	Subject	Pri.	Mod.	Cond.
r_1	Patient	Emergency	1	+	patient life is threatened
r_2	Patient	GP Physician	3	+	the subject is the attending physician
r_3	Vitals	Nurses	3	+	TRUE
r_4	Patient = Anna	Bob	2	−	TRUE
r_5	Vitals	Emergency	2	+	TRUE
r_6	Vitals = Anna	Bob	2	+	TRUE

Table 6: Example 3 rules

can't access Anna's data, unless there in an emergency context where Anna's life is threatened. The final rule base of this example is presented in Table 6 and with the graphs in Figure 4.

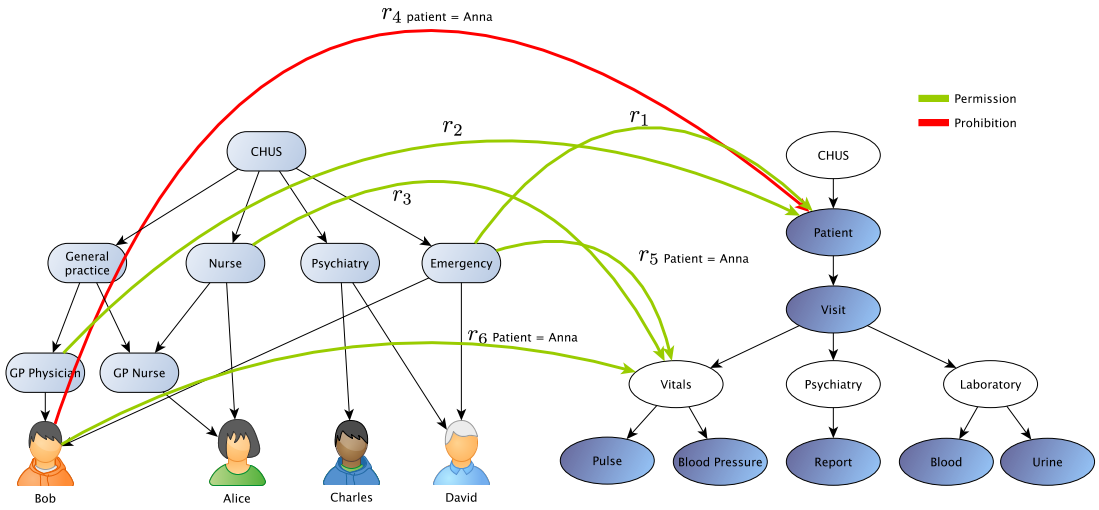


Figure 4: Example 3 graphs with rules

3. Formal model

In this section, our formalisation of SGAC is presented. This formalisation provides a way to evaluate requests for a given set of rules, and a way to verify properties.

3.1. Subject graph

The subject graph is denoted by S. We denote by $SUBJECT$ the set of all subjects and by $PERSON$ the set of all persons. Formally, S is described by $S = (Vertices, Edges, Persons)$ with:

- $(Vertices, Edges)$ is a DAG;
- $Vertices \subseteq SUBJECT$ is the set of the subjects;
- $Persons = Vertices \cap PERSON$ represents the persons, and elements of $Vertices - Persons$ are entities which represent groups of persons;
- $Persons \subseteq sink(S)$ since a person is a sink of S, as it cannot include any other subject.

$S.Edges$ represents the inheritance relation: recall that a rule on a subject s is inherited by all the successors of s in $S.Edges^+$ There are two types of subject: persons and entities. A sink of S can be either a person or an entity, but a person is by definition a sink. A non-sink vertex is then an entity.

3.2. Resource type graph

As mentioned in Section 2.3, data have been abstracted by types into a resource type graph. Recall that an atomic element of data is called a document and can be for instance a prescription, a radiography, etc. We introduce the notion of parametric directed acyclic graph (PDAG) representing a resource type graph R as follows: $R = (Graph, Param)$ where $Graph = (Vertices, Edges)$ is a DAG and $Param = (Pvertices, Pvalues, Documents, Dtype, Pvaluation)$ denotes constraints linking the DAG $Graph$ to the documents. These symbols are described as follows:

- $Pvertices$ is the set of the parametric vertices;
- $Pvalues$ denotes all the accepted values for the parameters;
- $Documents$ denotes the patient documents;
- $Dtype$ denotes the type of a document;
- $Pvaluation$ is a valuation of parameters of the documents.

We denote by $DOCUMENT$ the set of all possible documents. These symbols are subject to the following constraints.

- $Pvertices$ denotes parametric vertices and they are called *parametric groups*. The elements of $Vertices - Pvertices$ are called *groups*. Parametric groups introduce exactly one parameter, like Patient, Visit etc.

$$Pvertices \subseteq Vertices$$

- Sinks of R are document types, so they are parametric groups since a type of document requires an identifier.

$$sink(R) = sink(R.Graph)$$

- $Documents$ denotes the set of all patient documents, and $Dtype$ the type of a document. Each document has exactly one type, so

$$Dtype \in Documents \to sink(R);$$
$$Documents \subseteq DOCUMENT$$

- $Pvaluation$ denotes a valuation of parameters of the documents. The parameter valuation $Pvaluation$ is defined for each document and associates a document with a (partial) function between *parametric groups* and parameter values. It is a partial function since a document does not use all the parameters of the graph, but only those of its ancestors. Since each document has unique attributes, valuation of parameters is defined for all documents, thus we have

$$Pvaluation \in Documents \to (Pvertices \nrightarrow Pvalues);$$

- $Pvaluation$ is defined for each parameter inherited by a document,

$$\forall d \in Documents \bullet \mathsf{dom}(Pvaluation(d)) = Edges^{-1*}[\{Dtype(d)\}] \cap Pvertices$$

3.3. Rule

We denote by $MODALITY = \{\mathsf{permission}, \mathsf{prohibition}\}$ the set of modalities, $ACTION$ the set of all actions and $CONDITION$ the set of all formulas based on predicates that depend on the language used for representing rule conditions. A rule l is a septuplet consisting of:

- a modality $mod \in MODALITY$;
- a resource $res \in R.Graph.Vertices$;
- a resource condition represented by a valuation of its resource parameters;
- an action $act \in ACTION$;
- a subject $sub \in S.Vertices$;
- a priority $pri \in \mathbb{R}^+$;
- a condition $con \in CONDITION$.

We denote by $RULE$ the sets of all rules. Since a rule depends on a subject graph and a resource type graph, we introduce the notion of a policy $P = (S, R, L)$ composed of a subject graph S, a resource type graph R, and a set of rules L. We denote by $POLICY$ the set of all policies. Formally, we have:

- $S = (Vertices, Edges, Persons)$ a subject DAG;
- $R = (Graph, Param)$ a resource type PDAG;
- $L \subseteq RULE$ the set of rules of the policy.

The rules of a policy P are related to the graphs by the following constraints:

- $\forall l \in P.L \bullet l.sub \in P.S.Vertices$
- $\forall l \in P.L \bullet l.res \in P.R.Graph.Vertices$
- $\forall l \in P.L \bullet l.val \in P.R.Graph.Edges^{-1^*}[\{l.res\}] \nrightarrow P.R.Param.Pvalues$

We also introduce the function $documents(R, v, K)$ which returns all documents reachable from vertex v in PDAG R with document parameter valuation K. Formally:

$$documents(R, v, K) = \{d \mid d \in R.Param.Documents \wedge R.Param.Dtype(d) \in sink(R, v) \wedge K \subseteq R.Param.Pvaluation(d)\}.$$

For example, $documents(R, \mathsf{Visit}, \{\mathsf{Patient} \mapsto \mathsf{Simon}\})$ denotes the set of all documents issued during any visit of patient Simon. The blood test of the example of Figure 2b denoted by bt has the following associated parameters:

$$\begin{array}{ll} & R.Param.Pvaluation(bt) \\ = & \{\mathsf{Patient} \mapsto \mathsf{Simon}, \mathsf{Visit} \mapsto 2, \mathsf{Blood} \mapsto 123\} \\ \supseteq & \{\mathsf{Patient} \mapsto \mathsf{Simon}\} \\ = & K, \end{array}$$

Thus $bt \in documents(R, \mathsf{Visit}, \{\mathsf{Patient} \mapsto \mathsf{Simon}\})$.

Note that $documents(R, v, \emptyset)$ returns all documents associated to vertex v. For instance $documents(R, \mathsf{Patient}, \emptyset)$ denotes all the documents from all patients. We then have then property that function $documents$ is antitone:

$$\begin{array}{l} \forall P \in POLICY \bullet \\ \quad \forall \mathsf{Val}_1, \mathsf{Val}_2 \subset Pvaluation \bullet \\ \quad\quad \forall v \in P.R.Vertices \\ \quad\quad\quad \mathsf{Val}_1 \subseteq \mathsf{Val}_2 \\ \quad\quad\quad\quad \Rightarrow \\ \quad\quad\quad documents(P.R, v, \mathsf{Val}_2) \subseteq documents(P.R, v, \mathsf{Val}_1) \end{array}$$

3.4. Request

We define a request q as a triplet (sub, act, doc) where:

- $sub \in PERSON$ is the person initiator of the request;
- $act \in ACTION$ is the action sub wants to do;
- $doc \in DOCUMENT$ is the document targeted by the action act.

Do note that a request is made by one person and targets a single document at a time. Then formally, the set of all possible requests $REQUEST$ from a policy P is defined as follows:

$$REQUEST(P) = P.S.Persons \times ACTION \times P.R.Param.Documents$$

3.5. Request evaluation

The approach to evaluate a request is the following:

- extract all rules applicable to the request;
- sort extracted rules and represent them by a rule DAG;
- evaluate the request from the rule DAG.

3.5.1. Rule extraction

To this end, we introduce the function $Rules(P, q)$ for a policy P and a request q:

$$Rules(P, q) = \{l \mid l \in P.L \wedge matching\_sub(P, l, q) \wedge matching\_act(l, q) \wedge matching\_doc(P, l, q)\}$$

where:

- $matching\_sub(P, l, q)$ represents the condition that the request q share a common subject with the rule l within the policy P;

$$matching\_sub(P, l, q) := q.sub \in P.S.Edges^*[\{l.sub\}]$$

- $matching\_act(l, q)$ represents the condition that the request q and the rule l share the same action ;

$$matching\_act\_con(l, q) := (l.act = q.act)$$

- $matching\_doc(P, l, q)$ represents the condition that the documents targeted by the request q are a subset of those targeted by the rule l within the policy P.

$$matching\_doc(P, l, q) := \\ P.R.Param.Dtype(q.doc) \in P.S.Edges^*[l.res] \wedge l.val \subseteq P.R.Param.Pvaluation(q.doc)$$

Then for a policy P and a request q, $Rules(P, q)$ denotes all rules of $P.L$ such that:

- action corresponds to $q.act$;
- subject is $q.sub$ or an ancestor of $q.sub$;
- reachable documents contains $q.doc$.

Note that rule conditions are taken into account later in the graph analysis step in Section 3.5.3, to optimise the verification of properties of a policy.

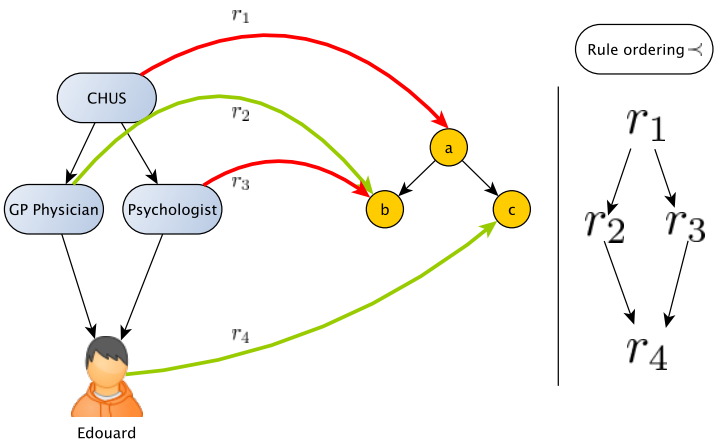


Figure 5: Illustration of the partial order relation $\prec$

3.5.2. Rule ordering

Once we have all applicable rules, we need to sort them. We therefore introduce a partial order relation $\prec$ defined as follows :

$$
\begin{array}{l}
\forall P \in POLICY \bullet \\
\quad \forall x, y \in \mathsf{P.L} \bullet \\
\qquad x \prec y \\
\qquad \Leftrightarrow \\
\qquad\quad y.pri < x.pri \\
\qquad\quad \vee \ (x.pri = y.pri \wedge y.sub \in P.S.Edges^{+}[\{x.sub\}])
\end{array}
$$

This relation $\prec$ captures the fact that precedence is given to the rule with a lower priority value or, at equal priority, to the rule targeting the lower subject (inclusion-wise). This order does not take into account the resources targeted by a rule. For instance, in Figure 5, we suppose that r_1, r_2, r_3 and r_4 share the same priority. We have then $r_1 \prec r_2$, $r_1 \prec r_3$, $r_2 \prec r_4$, $r_3 \prec r_4$ and finally $r_1 \prec r_4$. Note that r_2 and r_3 cannot be compared with $\prec$. If r_1, r_2, r_3 and r_4 are the only applicable rules then precedence over the other rules would be given to r_4 since it is the only maximal element (there is no other rule r' such that $r_4 \prec r'$). But what happens when there are more than one maximal element?

Let's take the previous example, and remove r_4. We have $r_1 \prec r_2$ and $r_1 \prec r_3$, but r_2 and r_3 still can't be compared. We then define another partial order on rules, noted "$<$". The set of maximal elements of the set $Rules(P, q)$ with the relation $\prec$ is denoted by $\mathsf{max}_{\prec}(P, q)$. Formally,

$$\mathsf{max}_{\prec}(P, q) = \{x \in Rules(P, q) \mid \neg \exists y \in Rules(P, q) \bullet x \prec y\}$$

We define $<$ on rules as follows:

$$
\begin{array}{l}
\forall P \in POLICY \bullet \\
\quad \forall q \in REQUEST(P) \bullet \\
\qquad \forall x, y \in Rules(P, q) \bullet \\
\qquad\quad x < y \\
\qquad\quad \Leftrightarrow \\
\qquad\qquad x \prec y \\
\qquad\qquad \vee \ (\quad x, y \in \mathsf{max}_{\prec}(P, q) \\
\qquad\qquad\quad \wedge \ y.mod = prohibition \\
\qquad\qquad\quad \wedge \ x.mod \neq y.mod \\
\qquad\qquad\quad)
\end{array}
$$

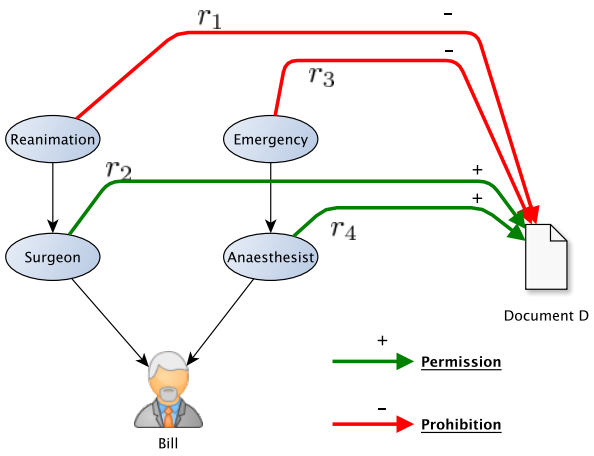


Figure 6: Rule ordering example

The partial order $<$ extends $\prec$: in the case there are more than one maximal element, precedence over the permissions are given to the prohibitions. Thus ordered rules can be represented in the DAG $Rg(P, q)$ which is calculated from a request q in the policy P:

- $Rg(P, q).Vertices = Rules(P, q)$;
- $Rg(P, q).Edges$ is the covering relation of $<$, which is in our case equal to the transitive reduction of $<$, in order to find the immediate successor precedence-wise. Formally, we have:

$$Rg(P, q).Edges = \{(x, y) | x < y \wedge \neg \exists z \bullet x < z \wedge z < y\}$$

The requirement that only maximal elements of $\prec$ can be compared by their modality is crucial: for instance, if we take the example of Figure 6, we have $r_1 \prec r_2$ and $r_3 \prec r_4$ but r_2 and r_4 cannot be compared with $\prec$. If we remove the maximal element condition from the definition of $<$ then we would have $r_1 < r_2$, $r_3 < r_4$, $r_2 < r_3$ and $r_4 < r_1$, forming a cycle and destroying the ordering.

3.5.3. Rule graph analysis

The rule graph $Rg(P, q)$ contains all rules applicable to a request q in a policy P ordered by precedence. Thus the rules from $sink(Rg(P, q))$ have precedence over the others. We denote by the predicate $eval(P, q, c)$ the evaluation of the request q in the policy P within the context c; it holds iff q is permitted. In order to determine *eval*, we proceed as follow:

1. we determine first all applicable rules by calculating $Rules(P, q)$;
2. we create the DAG $Rg(P, q)$ and order the rules with respect to $<$;
3. we verify that the rules that have precedence over all the other rules are permissions, and that there is at least one applicable rule. If no rule is applicable or if there is at least one prohibition, then request is rejected.

The tricky part of this procedure is that the request evaluation must take contexts into account. Our previous approach considered the rule conditions as a selection criteria: a rule was extracted if the subject, the resource, and the action match and if its condition was evaluated to TRUE in the context of the request. This approach makes us create a rule graph for each request/context combination, with only applicable rules that are active in the given context. This computational burden can be avoided by analysing a rule

Rule	Resource	Subject	Pri.	Mod.	Cond.
r_1	Laboratory (Patient = Anna)	Nurse	2	+	*TRUE*
r_2	Laboratory (Patient = Anna)	Alice	2	−	*TRUE*
r_3	Laboratory (Patient = Anna)	GP Physician	2	+	*TRUE*
r_4	Patient	Emergency	3	+	*the subject is the attending physician of the data owner*
r_5	Patient = Anna	Emergency	2	−	*TRUE*
r_6	Patient	Emergency	1	+	*data owner's life is threatened*

Table 7: Rule base of Example 4

graph for a request containing all the rules without regards to their active/inactive status. Then, instead of looking for the sinks of the rule graph to determine the result of the request, we look for *pseudo-sinks* of the rule graph, *i.e.* the vertices that, given a *context*, are active while all their successors are not. A context denote a state of the environment where a rule condition is evaluated, and we denote by $c \models con$ the fact that condition *con* is satisfied in the context c. We define the function $\textit{pseudo-sink}(P, q, c)$ for a policy P, a request q and a context c as follows:

$$\begin{aligned}&\textit{pseudo-sink}(P, q, c) = \\ &\quad \{r_1 \in Rg(P, q).\mathit{Vertices} \mid \quad c \models r_1.\mathit{con} \\ &\qquad\qquad\qquad\qquad \wedge \quad \forall r_2 \in Rg(P, q).\mathit{Edges}^+[\{r_1\}] \bullet c \models \neg r_2.\mathit{con} \\ &\quad \}\end{aligned}$$

Predicate *eval* is formally defined as follows:

$$\textit{eval}(P, q, c) \quad \Leftrightarrow \quad \textit{pseudo-sink}(P, q, c) \neq \emptyset \wedge \forall r \in \textit{pseudo-sink}(P, q, c) \bullet r.\mathit{mod} = \mathsf{permission}$$

There must be at least one pseudo-sink and all pseudo-sinks must be permissions.

As an interesting property, $<$ ensures that all sinks of Rg have the same modality. Indeed, let $r_1, r_2 \in \textit{sink}(Rg)$ with $r_1.\mathsf{mod} \neq r_2.\mathsf{mod}$. Two cases can be distinguished:

- $r_1.\mathit{pri} = r_2.\mathit{pri}$: according to the definition of $<$, we have $r_1 < r_2$ or $r_2 < r_1$, which contradicts $r_1, r_2 \in \textit{sink}(Rg)$.
- $r_1.\mathit{pri} \neq r_2.\mathit{pri}$ then we have $r_1 < r_2$ or $r_2 < r_1$, which also contradicts $r_1, r_2 \in \textit{sink}(Rg)$.

3.6. Example 4

Let's say that the patient Anna is to be hospitalised in the CHUS. She worked there as a nurse and had befriended most of her former colleagues, but also had some rivals like Alice. Anna decided to share her laboratory data to her nurse friends except for Alice and general practice physicians. She is aware that in the emergency department, physicians can access the all records of the patient, while that patient is under their care. Since she knows personally some of these physicians, she decides to prevent the department from accessing her records. But in the case her life is threatened, regulations and laws permit emergency physicians to access her records in order to provide faster and better medical care. We denote by P_1 the policy containing all the previous rules, the subjects and resources. We have the following rules presented in the Table 7 as $P_1.L$. We use Figure 1 as the subject graph $P_1.S$ and Figure 2a as the resource type graph $P_1.R$.

We suppose that Anna's EHR only contain two blood tests bt_1, bt_2 and a psychiatry report pr_1. bt_1 has been issued during the first visit, and bt_2 and pr_1 during the second visit. For this example, $P_1.R$ only contains Anna's documents. Formally, we introduce the documents in $P_1.R.\mathit{Param}$, which we simply denote by *Param* in the sequel:

- $Param.Documents = \{bt_1, bt_2, pr_1\}$;
- $Param.Dtype = \{bt_1 \mapsto \mathsf{Blood}, bt_2 \mapsto \mathsf{Blood}, pr_1 \mapsto \mathsf{Report}\}$;
- $Param.Pvaluation =$
  $\{bt_1 \mapsto \{\mathsf{Patient} \mapsto \mathsf{Anna}, \mathsf{Visit} \mapsto 1, \mathsf{Blood} \mapsto 1\}$,
  $bt_2 \mapsto \{\mathsf{Patient} \mapsto \mathsf{Anna}, \mathsf{Visit} \mapsto 2, \mathsf{Blood} \mapsto 2\}$,
  $pt_1 \mapsto \{\mathsf{Patient} \mapsto \mathsf{Anna}, \mathsf{Visit} \mapsto 2, \mathsf{Report} \mapsto 1\}\}$.

We then have:

$$documents(P_1.R, \mathsf{Patient}, \mathsf{Patient} \mapsto \mathsf{Anna}) = \{bt_1, bt_2, pt_1\}.$$

We distinguish two specific conditions among the rule base example: $r_4.con$ and $r_6.con$. This leads to four different possible contexts :

- the requester was the patient's attending physician and the patient is fine (context c_1);
- the requester is not the patient's attending physician and the patient is fine (context c_2);
- the requester is the patient's attending physician and the patient is in a life-threatening condition (context c_3);
- the requester is not the patient's attending physician and the patient is in a life-threatening condition (context c_4).

Let's assume that Alice wants to access bt_1. Let's denote by q_1 the request $(\mathsf{Alice}, \mathsf{read}, bt_1)$.

$$Rules(P_1, q_1) = \{r_1, r_2\}$$

We then calculate $Rg(P_1, q_1).Edges = \{r_1 \mapsto r_2\}$ since Alice belongs to the subject Nurse.
Since, those rules do not have specific conditions, the pseudo-sinks for any context are the sinks of $Rg(P_1, q_1)$ and $sink(Rg(P_1, q_1)) = \{r_2\}$. Thus:

$$\forall c \in CONTEXT \bullet eval(P_1, q_1, c) = false$$

Thus, in all possible contexts, Alice can't access Anna's data. Let's consider another request: now Bob wants to access bt_2, $q_2 = (\mathsf{Bob}, \mathsf{read}, bt_2)$. The applicable rules are:

$$Rules(P_1, q_2) = \{r_3, r_4, r_5, r_6\}$$

Since Bob belongs to GP Physician and Emergency, he is affected by any rules targeting one of the two entities. The calculus of $Rg(P_1, q_2).Edges$ is a bit trickier than before: we have $r_4 < r_3$ and $r_4 < r_5$, because $r_4 \prec r_3$ and $r_4 \prec r_5$ but r_3 and r_5 are incomparable with $\prec$. In fact, $r_3 < r_5$ since they are both maximal elements and r_5 is a prohibition and r_3 is a permission. We also have $r_3 \prec r_6$, $r_4 \prec r_6$, $r_5 \prec r_6$ because of r_6 priority. Then we take the transitive reduction of $<$, thus $Rg(P_1, q_2).Edges = \{r_4 \mapsto r_3, r_3 \mapsto r_5, r_5 \mapsto r_6\}$.

Figure 7 illlustrates the simplification of rule conditions into contexts and the use of pseudo-sinks: the four first rule graphs ($R1$, $R2$, $R3$, $R4$) contain only active rules in the different contexts, and the last rule graph ($R0$) shows $Rg(P_1, q_2)$ with the contexts satisfying each rule condition. The sinks of the first four graphs can be found as pseudo-sinks in the latest: for instance, the sink of $R1$, that is r_5, is the deepest active rule in the context c_1 of $R0$. We can express the rules using those contexts, as presented in Table 8.

If the request q_2 is made under the context c_2 (patient is not in a life threatening condition and the requester is not the attending physician), we have:

$$pseudo\text{-}sink(P_1, q_2, c_2) = \{r_5\}$$

Thus, since r_5 is a prohibition, q_2 is denied. However if the request q_2 is made under the context c_4 (patient is in a life threatening situation and the requester is not the attending physician) then:

$$pseudo\text{-}sink(P_1, q_2, c_4) = \{r_6\}$$

The request is then permitted since r_6 is a permission.

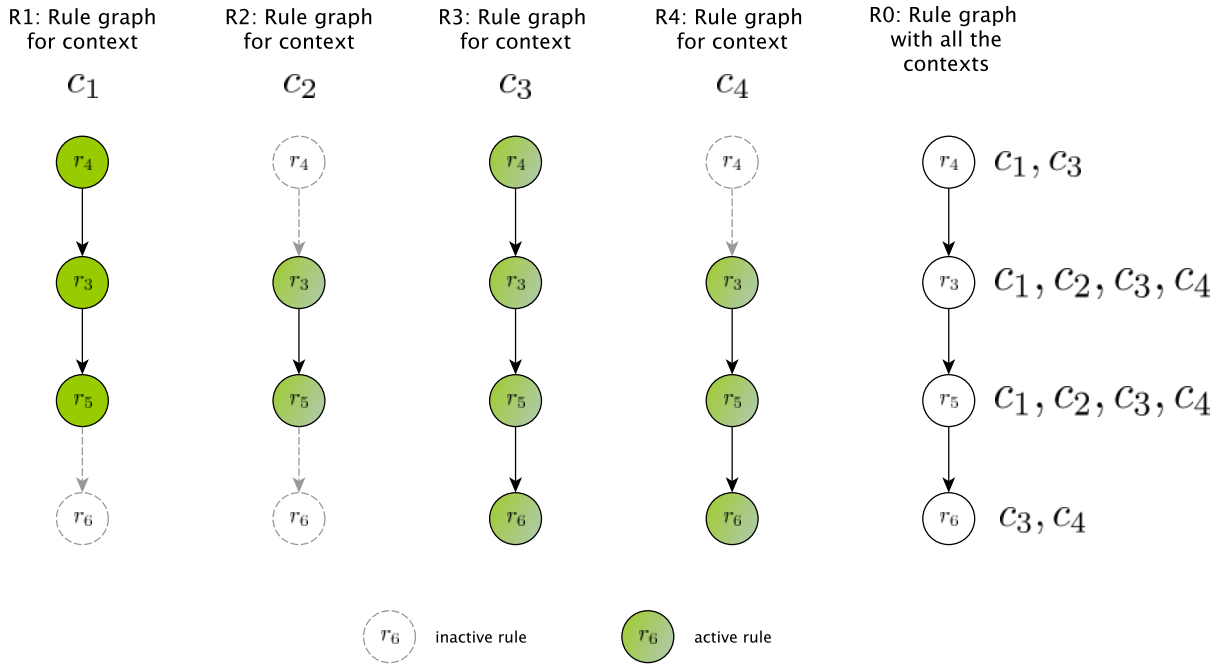


Figure 7: Rule graphs under the different contexts and context independent rule graph

Rule	Resource	Subject	Pri.	Mod.	Contexts
r_1	Laboratory (Patient = Anna)	Nurse	2	+	c_1, c_2, c_3, c_4
r_2	Laboratory (Patient = Anna)	Alice	2	−	c_1, c_2, c_3, c_4
r_3	Laboratory (Patient = Anna)	GP Physician	2	+	c_1, c_2, c_3, c_4
r_4	Patient	Emergency	3	+	c_1, c_3
r_5	Patient = Anna	Emergency	2	−	c_1, c_2, c_3, c_4
r_6	Patient	Emergency	1	+	c_3, c_4

Table 8: Rule base of Example 4 with contexts satisfying the rule conditions

3.7. Potential danger detection

In order to detect potential danger, we use property verification. A potential dangerous situation occurs, for instance, when the patient hides important data from the medical staff. In order to avoid such a situation, the following property must hold for the patient with the ID p within the policy P in the context c:

$$\forall d \in documents(P.R, \mathsf{Patient}, \{\mathsf{Patient} \mapsto p\}) \bullet \exists i \in (P.S).Persons \bullet eval(P, (i, \mathsf{read}, d), c)$$

This property can be generalised for all patients, and thus detect any patient with potential danger. In order to quantify on all the patients, we use the fact that the parametric vertex Patient is be associated with identifiers of each patient. Formally for a policy P and a context c:

$$\begin{array}{l} \forall p \in \mathsf{ran}(P.R.Param.Pvaluation)[\{\mathsf{Patient}\}] \bullet \\ \quad \forall d \in documents(P.R, \mathsf{Patient}, \{\mathsf{Patient} \mapsto p\}) \bullet \\ \quad\quad \exists i \in (P.S).Persons \bullet \\ \quad\quad\quad eval(P, (i, \mathsf{read}, d), c) \end{array}$$

Using the fact that if a parametric vertex is given no parameters then the vertex denotes all the possibles instances, this property can be simplified into:

$$\forall d \in documents(P.R, \mathsf{Patient}, \{\}) \bullet \exists i \in (P.S).Persons \bullet \mathsf{Prop}(P, (i, \mathsf{read}, d), c)$$

Our formalisation of SGAC allows for access verification and the following additional properties:

- *Determination of necessary and sufficient conditions for a subject to access a resource*: it is possible to determine a formula which must hold in order to authorise a request.
- *Redundant rule detection*: a rule is said redundant within a policy if the requests accepted by the policy is the same with and without the rule.
- *Determination of the data accessible by a subject*: since we can determine the result of a request, we can determine all accessible documents for a given subject.

So far, we have described some of the possibilities our formalisation offers. However, we need to check if our formal model is amenable to formal verification of these properties. This is why we explored automated verification using model checkers, presented in the following section.

4. Evaluation of SGAC verification with tools

In this section, we present an overview of Alloy and B languages, then the formalisation of SGAC using their associated model checkers, Alloy and ProB. There are three main classes of model checkers for first-order logic: SAT-based approaches like Alloy [12], constraint-based approaches like ProB [7], and SMT-based approaches like CVC4 [5], Yices [6] and Z3 [4]. ProB and Alloy are easier to use to model SGAC policies and they have both been shown to be useful in solving graph problems and complex first-order constraints like those used in SGAC [10, 8, 14, 19].

Basically, we evaluate the applicability of Alloy and ProB for checking four basic properties of SGAC policies:

1. *accessibility*: verify if a user has access to a document in a given context;
2. *availability*: verify that for a given context, each document of a patient is accessible by some user (*i.e.*, nothing is completely hidden);
3. *contextuality*: enumerate the contexts that provides access to a patient's data for a given user;
4. *rule effectivity*: identify the rules that are ineffective, *i.e.*, rules that are always overridden by other rules, and thus have no effect on the access granting decisions. Such rules may denote misrepresented safety/privacy requirements.

In order to make the model-checker tools workable on the SGAC model, we have to ensure :

- the ease of modelling SGAC policies;
- the capability to deal with the access control policy of a patient, which includes graphs and rules.

Indeed, an easy modelling of the SGAC policies would facilitate the detection of error and their correction. Moreover, SGAC policies may be huge by including millions of rules and hundreds of millions of documents. We don't expect any model checker to be able to handle such large data. Thus, the verification of policies must be done on a patient-by-patient basis, by extracting the rules applicable to a patient, and for a subset of a patient's document, those which are most critical for the patient safety and privacy.

In order to conduct tractable analysis with a simpler model, we perform some simplifications:

- in the subject graph, all sinks are persons (*i.e.*there is no empty group of persons);
- action is abstracted, since it does not affect the way rules are ordered;
- since the verification is done on a patient-by-patient basis, each verification is made with only one policy (one set of rules, one resource type graph and one subject graph);
- the parametric resource type graph is seen as a simple directed acyclic graph.

4.1. Alloy

Alloy is a formal language for describing relational structures. Relations are declared using an object-oriented syntax. All variables of an Alloy specification are n-ary relations, with $n \leq 5$. Alloy is supported by a tool, the Alloy Analyzer, for analysing and exploring the relational specification. It is a first-order logic model finder: the solver takes the constraints of a specification and finds instances that satisfy them, thus it bears some similarities with model checking. The Alloy Analyzer is used to explore a specification by generating sample *instances* of it, to check its properties by generating counterexamples in case of property violations. An instance is an assignment of values to the symbols of the Alloy specification; an instance is typically called "a model " in the logic literature. Alloy offers a customisable graphical interface and an evaluator which improves the user experience and greatly helps in understanding the model and counterexamples. Its graphical interface is particularly convenient when handling complex graphs with several vertices and edges.

Alloy is a relational language, where relations are declared using an object-oriented syntax. The semantics of an Alloy specification is represented by a set of *n-ary* relations. Sets are represented as unary relations. Elements of a set are represented by singleton sets. Set membership is represented by set inclusion.

4.1.1. Instances and signatures

The specification is described with structures, called signatures. These signatures can have fields, similarly to classes in object-oriented programming. Accessing field r of an instance a of a signature A is denoted by $a.r$. In Figure 8 we define an abstract signature `A` denoting a set of instances of type `A` (Line 1). The field `r` of type `set A` of the structure `A` is represented by a relation from `A` to `A`. An abstract signature has no proper instance. A type can also be a cartesian product of signatures, represented by the operator `->`. A type is decorated by the multiplicity constraint `set`, which says that the value of `a.r` is a set. Multiplicity can be specified as a constraint in fields, but also in formulas and signature declarations:

- `one`: the structure is instantiated exactly once.
- `lone`: the structure can be instantiated at most once.
- `some`: the structure must be instantiated at least once.

```
1 abstract sig A{
2   r : set A
3 }
4
5 sig B extends A {}{
6   #r>2
7 }
8
9 fact {
10   all a:A | #a.r < 4
11 }
```

Figure 8: Example of Alloy declarations of signatures and constraints

The signature `B` that extends `A` (Line 5) can be instantiated since it is not abstract. Signature `A` declares a field `r` of type `A`. Relations can be composed with the "." (or *join*) operator, an extension of the relational composition to n-ary relations with $n \geq 1$, as follows:

$$\forall p, q \bullet p.q = \{(p_1, \ldots, p_{n-1}, q_2, \ldots q_m) \mid (p_1, \ldots, p_n) \in p \wedge (p_n, q_2, \ldots, q_m) \in q\}$$

In particular, the join operator can be used on instances or on a signature. For example, `a.r` returns the value of `r` for the instance `a`, `A.r` returns the codomain of `r`, *i.e.*, the set of values of `r` for all the instances of `A`, and `r.A` returns the domain of `r`, *i.e.*, the set of all the instances of `A` that are mapped to the values of `r`.

To specify constraints on a signature, two options are provided:

- Declare a constraint specifically bound to a signature: this constraint is present in the signature declaration, or it follows its definition and it always refers to this signature fields. This is the case for the constraint on line 6 of Figure 8.
- Declare a constraint as a **`fact`**: this constraint is declared outside a signature. This is the case for the constraint on line 9 of Figure 8.

Expression `#`e returns the number of elements of the set e. For instance, in Figure 8, the signature `B` that extends `A` (line 5) has the constraint that each instance of `B` is related by `r` to more than two instances of `A` (line 6). The fact constraint on line 9 ensures that each instance of `A` is related to less than four instances of `A`. An instance of an Alloy specification must verify all the constraints and facts. For the specification of Figure 8, there can only be instances of `B`, and there are at least three of them, since each instance of `B` is related to exactly three instances of `B`, by the conjunction of constraints of lines 6 and 9 in Figure 8.

4.1.2. Predicate, function and assertion

The following list describes three other constructs of the Alloy language:

- **`pred`**, which declares a predicate,
- **`fun`**, which declares a function, and
- **`assert`**, which declares an assertion, that is, a formula that should hold on all instances of the Alloy specification. An assertion is similar to a theorem of a theory.

4.1.3. Generating specification instances

Once all signatures and constraints are specified, the specification exploration can start: Alloy tries to find a specification instance satisfying all the constraints and displays it. One can also use the evaluator to manually check predicates, and functions, or ask Alloy to find another instance. At this point, the user can run the specification or check some constraints by executing the following commands:

```
1 run {some B} for 4 int
2
3 assert assertion1{ all b:B | #b.r<=1 }
4
5 check assertion1 for 4 int
```

Figure 9: Using Alloy commands

- `run` *p* `for` *scope*: this command searches for an instance satisfying the signatures, the facts and the predicate *p* within the *scope*. A scope determines the number of instances for each signature. The predicate *p* and the *scope* are optional.
- `check` *p* `for` *scope*, or `check` *name* `for` *scope*: checks that the assertion *p*, or the assertion declared as *name*, holds on all the instances of the specification for the *scope*. If it does not, Alloy provides a counterexample, that is, an instance where *p* is violated.

Figure 9 illustrates how to use these commands.

- `run` at line 1 finds an instance of the model for up to 4 instances of each signature. The analyser finds an instance satisfying all the constraints of Figure 8 and exhibits it.
- `assert` at line 3 declares an assertion: each instance of `B` is related by `r` to at most one instance.
- `check` at line 5 verifies the assertion *assertion1* previously defined. Keyword `int` defines the number of bits (*i.e.*bitwidth) used to store integers. If a counterexample is found, it is shown, and that is obviously the case here, since the specification allows an instance of `B` to be related to exactly three instances. Assertion *assertion1* is violated.

4.1.4. Formalisation of SGAC in Alloy

Introduction of the basic types. Figure 10 shows the Alloy specification for the declaration of the signature, plus the predicate `evalRuleCond[r,c]` (line 29) which holds when the rule *r* is active in the context *c*. It contains the following signatures:

- `Subject`, that represent the users with their graph;
- `Resource`, that represent the data with their graph;
- `Modality`, with instances Prohibition and Permission;
- `Context`, that represents different conditions where a rule can apply;
- `Rule`, that represents the elements of the access control policy.

From these signatures, we can define the edges of the subject and resource type by defining fields `subSucc` and `resSucc` in signatures `Subject` and `Resource` (lines 2 & 6). Signature `rule` maps a rule to one `subject`, one `resource`, one `modality`, one integer (`priority`) and a set of `context`, which represent a rule condition by the set of contexts that satisfies it.

Adding request type and rule ordering. We then add the *REQUEST* type that basically is the Cartesian product of the sinks of the subject and resource type graphs. To order the rules and determine which rule will have precedence over the others, we have to create a rule graph. Since the ordering must be made from rules applicable to a given request, the rule ordering must be somehow linked to the signature request. If we take into account the rule condition when ordering the rule, then the ordering would be a quarternary relationship between rules, rules, requests and contexts. Since we came up with an alternative to avoid computing a

```
sig Subject {
  subSucc: set Subject
}

sig Resource {
  resSucc: set Resource
}

fact{
  acyclic[subSucc,Subject]
  acyclic[resSucc,Resource]
}

abstract sig Modality {}
one sig prohibition, permission extends  Modality {}

sig Context {}

sig Rule {
  p : one Int,
  s : one Subject,
  t : one Resource,
  m : one Modality,
  ct : set Context
}{
  p >= 0
}

pred evalRuleCond[r:Rule,c:Context]{
c in r.ct
}
```

Figure 10: Alloy declaration of the Subject and Resource type graphs and Rules signature

```
1 sig Request{
2 sub: one Subject,
3 res: one Resource,
4 ruleSucc:  Rule -> set Rule
5 }{
6 no sub.subSucc
7 no res.resSucc
8 Rule.ruleSucc in appRules[this]
9 ruleSucc.Rule in appRules[this]
10 }
11
12 fun appRules[req:Request]: set Rule{
13 {r: Rule | req.sub in (r.s).*subSucc
14 and req.res in (r.t).*resSucc}
15 }
```

Figure 11: Alloy declaration of the request

rule graph for each couple (context, request), the rule ordering is a ternary relationship between rules, rules and requests. Then, the rule ordering is encoded as a relation `ruleSucc` within the signature `Request` of type: `Rule -> set Rule` (as we did for the graphs).

The resulting Alloy code for the request signature declaration is presented in Figure 11: the constraints in lines 6-7 compel the request to target only sinks of the graphs and constraints in lines 8-9 define the rule graph only among the rules applicable to the request. We define for this purpose the function appRules (from line 12) which returns the set of rules applicable to a given request: a rule r is applicable to a request req iff the subject of req is a successor (or is the subject) of r and the resource of req is also a successor (or the resource) of r.

Ordering the rules. The next step is to specify how the rules are ordered in the rule graph within the set of the applicable rules. First we define the predicate `lessSpecific` that compares two rules with their priority, and in the case they share the same priority, with their subject. It is exactly the relation "$\prec$" already defined in Section 3.5.2. A rule r_1 is `lessSpecific` than a rule r_2 when either r_1 has a higher priority value or r_1's subject is a predecessor (strict) of r_2's.

We then define predicate `isPrecededBy` (which represents relation "$<$" of 3.5.2) using predicate `lessSpecific`; when the rules are not comparable by `lessSpecific` and that those rules are maximal elements of `lessSpecific`, their modality is compared; precedence is given to prohibitions. Note that in our graphical representations, maximal elements appear at the bottom of the graphs.

Finally, since relation $<$ is transitive, we use a fact to set attribute `ruleSucc` to contain the transitive kernel of relation isPrecededBy, in order to minimise the rule graph and to make it easier to understand (see line 27 of Figure 12).

Scope of `Request`. When doing a run or a check, one must specify a scope for each signature. Scope determination is sometimes a bit tricky in Alloy, as signatures are related by fields. Signature `Request` constrains its instances to sinks of the subject graph and the resource graph. In a naive approach, one could specific the scope of signature `Request` as the product $k_s * k_r$ of the number sinks in the subject and resource graphs, but this raises two major concerns:

- this product is hard to estimate;
- Alloy takes a long time to compute rule ordering for each request.

However, a scope of 1 for `Request` is sufficient for formulas of the form " $\forall req \in REQUEST \bullet$ Property " or " $\exists req \in REQUEST \bullet$ Property ", because Alloy will cover all possible instances (*i.e.* all possible values)

```
pred lessSpecific[r1,r2: Rule]{
      (r2.p < r1.p)
  or (      r2.p = r1.p
        and r2.s in (r1.s).^subSucc)
}

pred maximal[r:Rule]{
no r1 : Rule | lessSpecific[r,r1]
}

pred isPrecededBy[r1,r2:Rule]{
  (
        lessSpecific[r1,r2]
    or
    (
          not lessSpecific[r2,r1]
      and maximal[r1]
      and maximal[r2]
      and r2.m = prohibition
      and r1.m != r2.m
    )
  )
}

fact {
all rq: Request | all r1,r2: appRules[rq] |
    r1 in req.ruleSucc.r2
  <=>
    (   isPrecededBy[r1,r2]
    and not some r3 : appRules[rq] |
              isPrecededBy[r1,r3]
          and isPrecededBy[r3,r2]
    )
}
```

Figure 12: Alloy declaration of the predicates and function needed to sort the rules

of field `sub`, `res` and `ruleSucc` for that given element of `Request`, covering all possible combination of sinks of the subject and resource graphs. If we have to verify a property of the form $\forall r \in \mathsf{RESOURCE} \bullet \exists req \in REQUEST \bullet \ldots$, then we must consider instances that contains $k_s * k_r$ requests, so that each value of r can find it corresponding value req in a given instance.

4.2. The B-Method and the ProB tool

The B language [1] relies on first order logic and set theory. It is used to specify systems in terms of state variables and operations that modify these state variables. A system in B is described using the concept of a *machine*. A machine contains:

- a clause **SETS** containing the declaration of user sets and types;
- a clause **CONSTANTS** containing the declaration of all constants;
- a clause **PROPERTIES** containing all properties binding constants, such as their type and their valuation;
- a clause **VARIABLES** containing the declaration of all variables defining the state of the system;

- a clause **INVARIANT** containing all the invariants the variables must satisfy at all time, such as typing, etc. ;
- a clause **INITIALISATION** containing the initial valuation of all variables;
- a clause **OPERATIONS** containing the operations that can modify the variables. In B, an operation is defined by preconditions and postconditions on variables, defined using the generalised substitution language, and can return a value.

4.2.1. Additional notation

We add the following notation needed to translate SGAC into B. Let A, B be sets. Let $R \in A \leftrightarrow A$.

- $\mathsf{closure1}(R) = R^+ = \bigcup_{n \geq 1} R^n =$ denotes the transitive closure of R, *i.e.*, the smallest transitive relation which contains R.
- $\mathsf{closure}(R) = R^* = R^+ \cup \mathsf{id}(A)$ denotes the transitive and reflexive closure of R, *i.e.*, the smallest transitive and reflexive relation which contains R.
- Let $prj1(A, B)$ denotes the first projection relation of $A \times B$ in A. Let $prj2(A, B)$ denotes the second projection relation of $A \times B$ in B. Thus,
  $prj1(A, B) = \{x, y, z \mid x, y, z \in A \times B \times A \wedge z = x\}$.
  $prj2(A, B) = \{x, y, z \mid x, y, z \in A \times B \times B \wedge z = y\}$.
- Let λ denotes the operator for lambda expression. Let $\lambda x.(x \in A \wedge P \mid e(x))$ denotes the function that maps the element x of A satisfying formula P to the expression $e(x)$.
- Let $S = \mathsf{struct}(Id_1 : T_1, \ldots, Id_n : T_n)$ denote the type (*i.e.*, set) of records; the elements of this set have *fields*, denoted by Id_i, of type T_i. An element e of S can be created using $\mathsf{rec}(Id_1 : v_1, \ldots, Id_n : v_n)$, where v_i is the value of field Id_i; expression $e'Id_i$ returns value v_i.

4.2.2. Small example

Presented in Figure 13, this small machine represents a stash in which elements can be put with the following constraints:

1. there are no duplicates in the stash;
2. the stash has a maximum capacity.

Thus we introduce:

- the type ELEMENT captured by the set ELEMENTS;
- the maximum stash size captured by the constant stashsize which is a strictly positive integer;
- the stash captured by the variable stash which is a subset of ELEMENTS, and which has a number of elements lesser or equal than stashsize;
- the constraint that the capacity of the stash is not exceeded;
- the operation add which adds an element to the stash if the element is not already in the stash and there still room left;
- the operation del which removes an element from the stash if the element is already in the stash.

MACHINE
stash_machine
SETS
ELEMENTS
CONSTANTS
stashsize
PROPERTIES
$stashsize \in \mathbf{INT} \wedge$
$stashsize > 0$
VARIABLES
stash
INVARIANT
$stash \subseteq ELEMENTS \wedge$
$\mathbf{card}(stash) < stashsize + 1$
INITIALISATION
$stash := \emptyset$
OPERATIONS
$\mathbf{add}(ele)=$
PRE
$ele \in ELEMENTS \wedge$
$ele \notin stash \wedge$
$\mathbf{card}(stash)<stashsize$
THEN
$stash := stash \cup \{ele\}$
END;

$\mathbf{del}(ele)=$
PRE
$ele \in ELEMENTS \wedge$
$ele \in stash$
THEN
$stash := stash - \{ele\}$
END
END

Figure 13: Stash machine

4.2.3. ProB

ProB is a tool for the B-Method that can animate, model check and solve constraints: the animation of a B specification is fully automatic, and the constraint-solving capabilities of ProB can be used for model finding, deadlock checking and test-case generation. Figure 14 illustrates ProB animating the previous stash machine. The values of the variables in the current state are displayed on the left side, the enabled operations for the current state in the middle and the trace of executed operations on the right. After having added 3 elements, we see that only element removal is allowed due to the stash size restriction. If the stash size

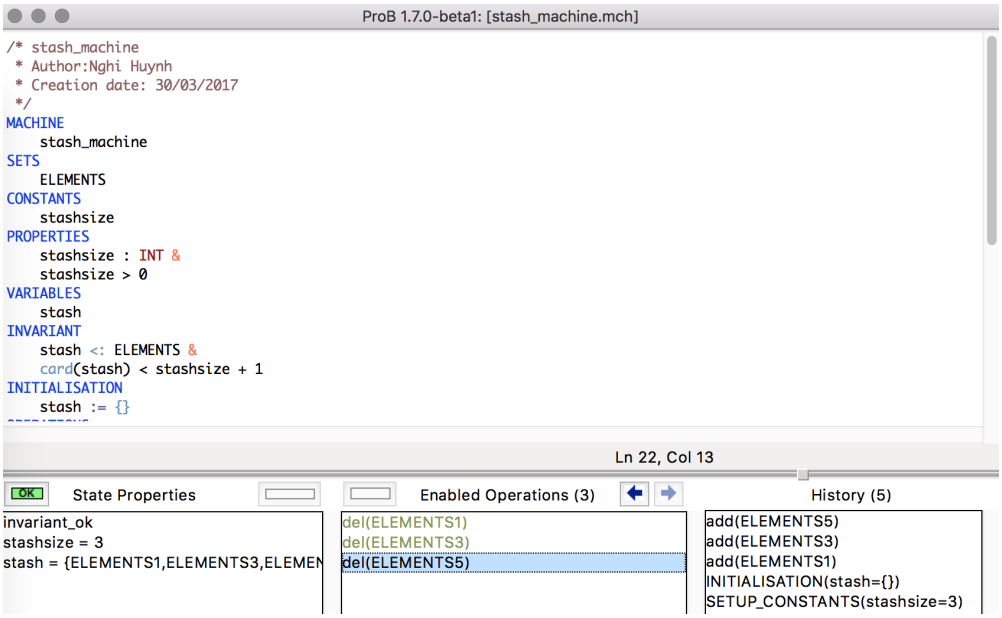


Figure 14: Using ProB to animate the stash machine

check is removed from the addition operation, then using the model checking abilities of ProB an invariant violation is found: the capacity of the stash is exceeded, see Figure 15. ProB exhibits a trace leading to the faulty state: in our case, adding four elements for a 3-capacity stash violates the invariant.

4.2.4. Formalisation of SGAC in B

In order to model SGAC in B, we use the model finder and constraint solving features of ProB. We tried two approaches. In the first approach, we model the SGAC policies and the properties using solely sets, constants and properties clauses of a B machine, in a way pretty similar to the Alloy specification. Here is how the machine for the first approach looks like in a nutshell:

- the SETS clause contains all declaration of the basic types and the enumeration of the values;
- the CONSTANTS clause contains the declaration of the other types (structures made of the basic types), and that also introduces constants that will be computed mainly for rule ordering.
- the PROPERTIES clause set the constraints binding all constants, by defining the constants type, their value (for setting up the graphs, computing the rule ordering) and also contains the property we want to verify.

Given this, ProB has three options:

1. it succeeds in finding a model that meets all constraints we put up, then the property holds;
2. it fails in finding a model that meets the constraints, then the property does not hold;

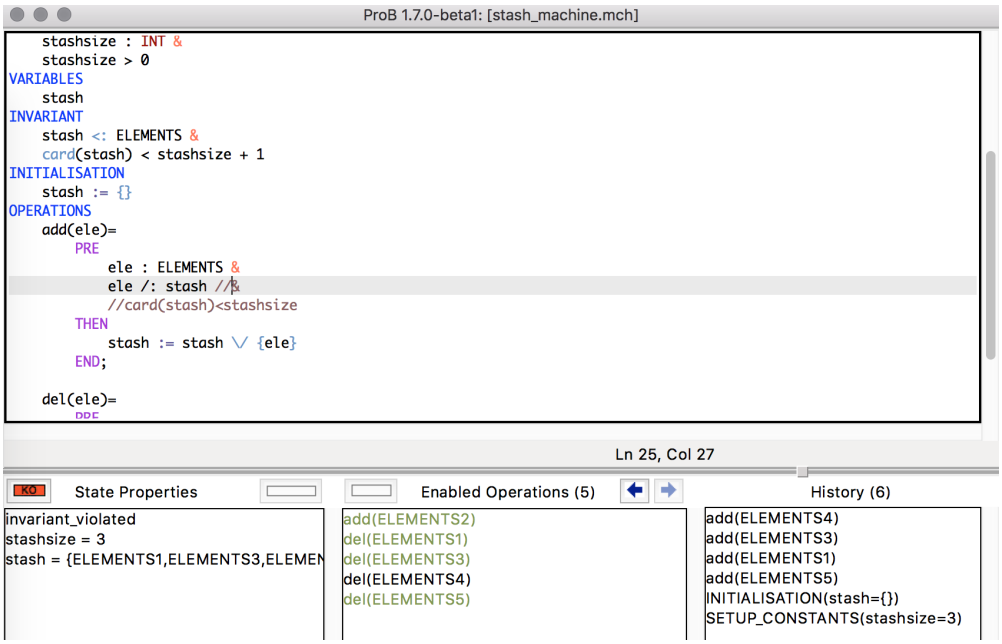


Figure 15: Model Checking with ProB

SETS

$CONTEXT$={ $c0$, $c1$, $c2$, ... };
V_SUB = {$s0$, $s1$, $s2$, ... };
V_RES = {$r0$, $r1$, $r2$, ... };
$RULE\_T$ = {$rule0$, $rule1$, $rule2$...};
$MODALITY$ = {per,pro}

Figure 16: **SETS** clause of the B specification of SGAC

3. it times out, then the result is unknown.

This approach which has the advantage of having a B model close to the Alloy one, knows several limits. The first one is that for one model, you can have only one property checked at a time. The second one is that ProB was unable to solve the constraints in an efficient manner and ended up most of the time timing out for a small model where Alloy performed good.

Those poor performances oriented us towards a second approach using the other possibilities ProB offers. In the second approach, we use variables and operations to impose an order in which the constraints can be efficiently solved by ProB. Thus, some of the data is represented as constants and properties, and the others are represented as state variables which are computed in a specific order in the initialisation clause of the machine. The operations are used to compute values of the properties to check. This approach is highly successful and represents a decisive advantage for ProB in comparison with Alloy. Our final model is structured as follows:

1. Declaration of the basic types in the clause **SETS**, presented in Figure 16. Those are the same basic types as in Alloy, except that we add rule identifier as a basic type.

2. Declaration of the other definitions such as the request type in the clause **CONSTANTS** except for

those related to rule ordering. The constants we use are constrained in Figure 17.

PROPERTIES

$$
\begin{array}{l}
\quad // \text{ *** Types ***} \\
e\_sub \in V\_SUB \leftrightarrow V\_SUB \wedge \\
e\_res \in V\_RES \leftrightarrow V\_RES \wedge \\
REQUEST\_T = (V\_SUB\text{-}\mathbf{dom}(e\_sub)) \times (V\_RES\text{-}\mathbf{dom}(e\_res)) \quad \wedge \\
rules \in RULE\_T \rightarrow \mathbf{struct}(su : V\_SUB,\ re : V\_RES,\ mo : MODALITY, \\
\qquad\qquad\qquad\qquad\qquad pr : \mathcal{Z},\ co : \mathcal{P}(CONTEXT)) \wedge \\
lessSpecific \in RULE\_T \leftrightarrow RULE\_T \wedge \\
\\
\quad // \text{ *** Closures ***} \\
\\
cl1\_e\_sub = \mathbf{closure1}(e\_sub) \wedge \\
cl\_e\_sub = \mathbf{closure}(e\_sub) \wedge \\
cl1\_e\_res = \mathbf{closure1}(e\_res) \wedge \\
cl\_e\_res = \mathbf{closure}(e\_res) \wedge \\
\\
\quad // \text{ *** Acyclicity of the graphs ***} \\
\\
cl1\_e\_sub \cap \mathbf{id}(V\_SUB) = \emptyset \wedge \\
cl1\_e\_res \cap \mathbf{id}(V\_RES) = \emptyset \wedge \\
\\
\quad // \text{ *** rule ordering: lessSpecific ***} \\
\\
lessSpecific = \{xx,yy \mid xx \in \mathbf{dom}(rules) \wedge yy \in \mathbf{dom}(rules) \wedge \\
\qquad\qquad (\\
\qquad\qquad\quad ((rules(xx))'pr > (rules(yy))'pr) \\
\qquad\qquad\quad \vee \\
\qquad\qquad\qquad (\\
\qquad\qquad\qquad ((rules(xx))'pr = (rules(yy))'pr) \\
\qquad\qquad\qquad \wedge \\
\qquad\qquad\qquad (rules(yy))'su \in cl1\_e\_sub[\{(rules(xx))'su\}] \\
\qquad\qquad\quad) \\
\qquad\qquad) \\
\qquad\qquad \} \wedge \\
\\
\quad // \text{ ***Setup of the graphs and ruleset***} \\
\\
e\_sub = \{s0 \mapsto s1,\ s0 \mapsto s2,\ \ldots\} \wedge \\
e\_res = \{r0 \mapsto r1,\ r0 \mapsto r2,\ \ldots\} \wedge \\
rules = \{ \\
\quad rule0 \mapsto (\mathbf{rec}(su{:}s1,\ re{:}r0,\ mo{:}per,\ pr{:}\ 4,\ co{:}\ \emptyset\ )), \\
\quad rule1 \mapsto (\mathbf{rec}(su{:}s0,\ re{:}r0,\ mo{:}pro,\ pr{:}\ 3,\ co{:}\ \{c1,c3\}\ )), \\
\quad \ldots\} \wedge
\end{array}
$$

Figure 17: **PROPERTIES** clause of the B specification of SGAC

3. Declaration of the types of the constants in **PROPERTIES**, and initialisation of the two graphs and

VARIABLES
applicable,
conRule,
isPrecededBy,
ruleSucc,
cl1_ruleSucc,
pseudoSink

INVARIANT
$applicable \in \quad REQUEST\_T \rightarrow \mathcal{P}(RULE\_T) \wedge$
$conRule \in CONTEXT \rightarrow \mathcal{P}(RULE\_T) \wedge$
$isPrecededBy \in REQUEST\_T \rightarrow (RULE\_T \leftrightarrow RULE\_T) \wedge$
$ruleSucc \in REQUEST\_T \rightarrow (RULE\_T \leftrightarrow RULE\_T) \wedge$
$cl1\_ruleSucc \in REQUEST\_T \rightarrow (RULE\_T \leftrightarrow RULE\_T) \wedge$
$pseudoSink : (REQUEST\_T \times CONTEXT) \rightarrow \mathcal{P}(RULE\_T)$

Figure 18: **VARIABLES** and **INVARIANT** clauses of the B specification of SGAC

the rule base, presented in Figure 17:

- *e_sub* and *e_res* denote the edges of the subject and resource type graphs, respectively;
- *REQUEST_T* denotes the type of a request, being the Cartesian product of the sinks of the graphs;
- *rules* denotes a function associating a *RULE_T* to a record containing a vertex of each graph (subject & resource), an integer (priority), a *MODALITY* and a set of *CONTEXT*s;
- we set constants containing the closures of the different graphs, *cl_e_sub* for the closure of *e_sub* etc...;
- we check that the graphs are acyclic with $cl\_e\_sub \cap \mathsf{id}(V\_SUB) = \emptyset$;
- we set the values of the constants as follow:
  - the constant *lessSpecific* is valued (same semantic as the relation in Alloy) by comparing each rule with their priority and their subject;
  - graphs and the rules are then valued.

4. Declaration of the rule ordering variables in **VARIABLES**, plus some variables to avoid repeating some heavy computations such as closure of the rule graph:
   - *applicable* denotes a function that returns the set of rules applicable to a given request;
   - *conRule* denotes a function that returns the set of rules that contain a given context;
   - *isPrecededBy* denotes a function that returns a relation that maps rules to rules given a request (same semantic as the alloy *isPrecededBy*);
   - *ruleSucc* denotes a function that, given a request req, returns the transitive reduction of $isPrecededBy(req)$ (readability purpose);
   - *cl1_ruleSucc* denotes the transitive closure of *ruleSucc*;
   - *pseudoSink* denotes a function that, given a request and a context, returns the pseudosinks of the rule graph of that request, with that context.

5. Declaration of the types of the variables and the constraints binding the variables in the **INVARIANT**, presented in Figure 18.

INITIALISATION
BEGIN
$applicable := \lambda\ rr.(rr \in REQUEST\_T \mid applicable\_def(rr))$;
$conRule := \lambda\ con.(con \in CONTEXT \mid \{cc \mid cc \in \mathbf{dom}(rules) \wedge con : (rules(cc))'co\})$;
$isPrecededBy := \lambda\ xx.(xx \in REQUEST\_T \mid$
$\{yy, zz \mid$
$yy \in applicable(xx) \wedge$
$zz \in applicable(xx) \wedge$
$yy \neq zz \wedge$
$($
$yy \mapsto zz \in lessSpecific$
$\vee$
$($
$\{yy,zz\} \subseteq maxElem(xx) \wedge$
$(rules(yy))'mo = per \wedge$
$(rules(zz))'mo = pro$
$)$
$)$
$\})$;
$ruleSucc := \lambda\ xx.(xx \in REQUEST\_T \mid$
$\{yy,zz \mid$
$yy \in applicable(xx) \wedge$
$zz \in applicable(xx) \wedge$
$yy \mapsto zz \in isPrecededBy(xx) \wedge$
$\neg (\exists\ uu.($
$uu \in RULE\_T \wedge$
$yy \mapsto uu \in isPrecededBy(xx) \wedge$
$uu \mapsto zz \in isPrecededBy(xx) \wedge$
$uu \neq yy \wedge uu \neq zz$
$))$
$\})$;
$cl1\_ruleSucc := \lambda\ xx.(xx \in REQUEST\_T \mid$
$\mathbf{closure1}(ruleSucc(xx))$
) **END ;**
$pseudoSink := \lambda\ (req,con).(req \in REQUEST\_T \wedge con \in \mathbf{dom}(conRule)$
$\mid \{ ru \mid ru \in applicable(req) \wedge$
$ru \in conRule(con) \wedge$
$\forall\ subr.($
$(subr \in cl1\_ruleSucc(req)[\{ru\}]) \implies \neg (subr \in conRule(con))$
$)$ $\})$

Figure 19: **INITIALISATION** clause of the B specification of SGAC

DEFINITIONS

$applicable\_def(req) ==$
$\{rul \mid is\_applicable(req,rul)\ \};$

$is\_applicable(req,rul) ==$
$(\ rul \in RULE\_T \wedge$
$\mathbf{dom}(\{req\}) \subseteq cl\_e\_sub[\{(rules(rul))'su\}] \cup \{(rules(rul))'su\} \wedge$
$\mathbf{ran}(\{req\}) \subseteq cl\_e\_res[\{(rules(rul))'re\}] \cup \{(rules(rul))'re\}$
$);$

$maxElem(req) == \{$
$rul \mid rul \in applicable(req)$
$\wedge$
$\neg\ ($
$\exists\ rul2.($
$rul2 \in applicable(req) \wedge rul \mapsto rul2 \in lessSpecific$
$)$
$)$
$\};$

$access\_def(req,con) ==$
$($
$\forall\ rsinks.(rsinks \in pseudoSink(req,con) \implies (rules(rsinks))'mo = per\ )$
$\wedge$
$pseudoSink(req,con) \neq \emptyset$
$)$

Figure 20: **DEFINITIONS** clause of the B specification of SGAC

6. Initialisation of the variables with a sequence, with regards to the dependency in Fig 19: for instance, *isPrecededBy* has to be set before *ruleSucc*, since it is its transitive reduction. Thus, we set first *applicable*, then *isPrecededBy*, then *ruleSucc*, then *cl1_ruleSucc* and then *pseudoSink*. Since *ruleCon* has no dependency link with the other variables it can be initialised in any order.

   Afterwards we can set *cl1_ruleSucc* then *pseudoSink*. ProB is having a very hard time computing *isPrecededBy* and *ruleSucc* in parallel, not inferring the dependency: that is the main reason to use variables and constant, to force ProB to set them in the adequate order.

7. Declaration of the operations which represent the different properties we want to verify.

In B, we can use macros defined in the **DEFINITIONS** clause: the macros are used in the same manner as #define of the C language. The macros we use are presented in Figure 20: we define here the definitions for *applicable*, *isPrecededBy* with *applicable_def* and *maxElem*, and the definition of *access_def*.

4.3. Properties verification

Once SGAC translated in B and Alloy, the four previous properties can be translated and checked.

4.3.1. Accessibility

In order to verify that a user u can access the document d under the context *con*, we check that the pseudo-sinks of the rule graph of the request defined by (u, d) are all permissions. If there is at least a

prohibition among them or no applicable rule is found, the request is denied. This is illustrated for the Alloy specification in Figure 21 and in Figure 22 for ProB.

In Alloy, we introduce the function `pseudoSinkRule` that retrieves the pseudo sinks of the rule graph for given context and request. We then declare the predicate `accessCondition` which returns true iff the given request is granted in the given condition. Then in order to verify that a user u can access the document d under the context *con*, we check that Alloy Analyzer (line 22) can find an instance of a model where: the pseudo-sinks of the rule graph of the request defined by (u, d) are all permissions. If there is at least a prohibition among them or no applicable rule is found, the request is denied. For efficiency, we try to keep the number of explicitly declared signatures to the minimum for each run. Thus for each request evaluation, each request is explicitly declared in a separate Alloy file.

```
1  fun pseudoSinkRule[req: Request,c:Context]: set Rule{
2    {r : appRules[req] |evalRuleCond[r,c] and
3         all ru : r.^(req.ruleSucc) |
4                not evalRuleCond[ru,c]
5    }
6  }
7
8  pred accessCondition
9    [req:Request,c:Context]{
10   (
11     no  r:pseudoSinkRule[req,c] |
12     r.m=prohibition
13   ) and
14   some r:appRules[req] | evalRuleCond[r,c]
15 }
16
17 one sig req0 extends Request{}{
18   sub=s1
19   res=r0
20 }
21
22 run accessReq0_c1 {accessCondition[req0,c1]} for 4
```

Figure 21: Checking Access with Alloy

The formalisation in B of the same property is provided in Figure 22 and is similar to the Alloy definition: we check that all pseudo-sinks are permissions and that at least one rule applies to grant the given request in the given context. Here, the result of the operation CheckAccess is returned in the variable access as a boolean. The body of the operation is very close to the Alloy specification.

$$
\begin{array}{l}
access \leftarrow \textbf{CheckAccess}(req,con) = \\
\textbf{PRE} \\
\quad req \in REQUEST\_T \wedge \\
\quad con \in CONTEXT \\
\textbf{THEN} \\
\quad access := \textbf{bool}(access\_def(req,con)) \\
\textbf{END};
\end{array}
$$

Figure 22: Checking access with ProB

```
1 fun documentsG[]: set Resource{
2   { rt : Resource | no rt.resSucc}
3 }
4
5 pred HiddenDocument[reso:Resource,c:Context]{
6   no req: Request | (req.res = reso and
7   access_condition[req,c])
8 }
9
10 check HiddenDocument_doc1_c0{
11   HiddenDocument[doc1,c0]
12 } for 4
```

Figure 23: Detection of hidden documents with Alloy

hidden ← **HiddenDocument**(*con*) =
PRE
  con ∈ *CONTEXT*
THEN
  hidden := **bool**(∃ (*hdoc*).(
    hdoc ∈ *V_RES* - **dom**(*e_res*) ∧
    ∀ *req*.(*req* ∈ *REQUEST_T* ∧
      prj2(*V_SUB*, *V_RES*)(*req*)=*hdoc*
      ⇒
      ¬ *access_def*(*req*,*con*))))
END;

Figure 24: Detection of hidden documents with B

4.3.2. Availability and contextuality

Once we are able to check that someone has access or not to a document, we can find hidden data in order to warn the patient that in some contexts, some data will be out of everyone's reach. A document is defined unreachable or hidden under the context *con* if all the requests that target it are denied under *con*. The formalisation in Alloy of this property is presented in Figure 23. We define the predicate `HiddenDocument` as follows: it returns true if the given document is not reachable under the given context. We then check that predicate with the command `check` (line 10) as discussed in Section 4.1.4.

We do the same in B: the operation HiddenDocument presented in Figure 24 checks if there is a document under the context *con* that cannot be accessed by anyone.

We can in the same manner determine contexts which make a given request granted. We introduce the signature `GrantingContext` containing all contexts that make a given request granted. We then need the predicate `grantingContextDet` binding all contexts that make a given request granted to `GrantingContext`. The Alloy formalisation is presented in Figure 25. We then ask the Analyzer to find an instance that satisfies the predicate `grantingContextDet` (line 10).

In B, we do not need to introduce an object that will wrap up the contexts since we can return the set of granting contexts. The operation GrantinContexts, presented in Figure 26, returns all contexts that make the given request granted.

```
one sig GrantingContext{
  acc: set Context
}{}

pred grantingContextDet[req:Request]{
    all c: Context | accessCondition[req,c]
  <=> c in GrantingContext.acc
}

run grantingContextDetermination{
  grantingContextDet[req1]
} for 5
```

Figure 25: Determination of the contexts that make a request accepted with Alloy Analyzer

granting ← **GrantingContexts**(*req*) =
PRE
  req ∈ *REQUEST_T*
THEN
  granting :=
    { *con* | *con* ∈ *CONTEXT* ∧
      access_def(*req*,*con*)
    }
END;

Figure 26: Determination of the granting contexts in B

```
1 pred ineffectiveRule[r:Rule]{
2   no rq : Request | (
3         r in appRules[rq]
4     and some cr : r.c | (
5       r in pseudoSinkRule[rq,cr]
6       and
7       (no ru : pseudoSinkRule[rq,cr]-r |
8         r.m = ru.m)
9       and
10      (r.m = permission implies
11         no (pseudoSinkRule[rq,cr]-r))
12    )
13  )
14 }
15
16 check ineffectiveRule_rule3{
17   ineffectiveRule[rule3]
18 }
```

Figure 27: Determination of the ineffective rules in Alloy

4.3.3. Rule effectivity

A rule is considered ineffective if it can never be determinant for the evaluation of a request. For instance, if we take two rules which only differ in their priorities, one of them is ineffective since the one with the lowest priority will always have precedence over the other. Formally, the criteria for a rule r to be effective are:

- if it's a prohibition: there is at least one couple request-context where r is a pseudo-sink of the rule graph, and r is the only prohibition among the pseudo-sinks;
- if it's a permission: there is at least one couple request-context where r is the only pseudo-sinks.

Indeed, if the rule is a prohibition, then it is effective in the case where the prohibition is the only pseudo-sinks with this modality. It does not matter if there are permissions among the pseudo-sinks, since prohibitions will have precedence. In the case of a permission, it is slightly different since no prohibitions must be among the pseudo-sinks, *i.e.*the permission is the only pseudo-sink.

We introduce the predicate `ineffectiveRule`, in Figure 27, which returns TRUE iff there is no request and no context for which:

- the given rule `r` is a pseudo-sink of the rule graph;
- there is no other pseudo-sink of the same modality as the given rule;
- if the given rule `r` is a permission, there is no prohibition among the pseudo-sinks, *i.e.* `r` is the only pseudo-sink .

Then to verify that a rule is not ineffective, we use the command `check` (line 16 of Figure 27). We do not use a signature wrapping up all ineffective rules as we did with `GrantingContext`, because this would require to build instances that contain all possible requests, which is too expensive.

In B, all the ineffective rules are returned by executing the operation IneffectiveRuleSet presented Figure 28.

The set of ineffective rules cannot be removed all at once: for instance, let r_1 and r_2 be two permissions that share the same attributes and condition. Let's suppose that there is a request req_1 for which only r_1 and r_2 apply. They both are flagged to be ineffective since each is a copy of the other rule. However if both are removed, then req_1 that was previously granted would be denied, because no rule would apply.

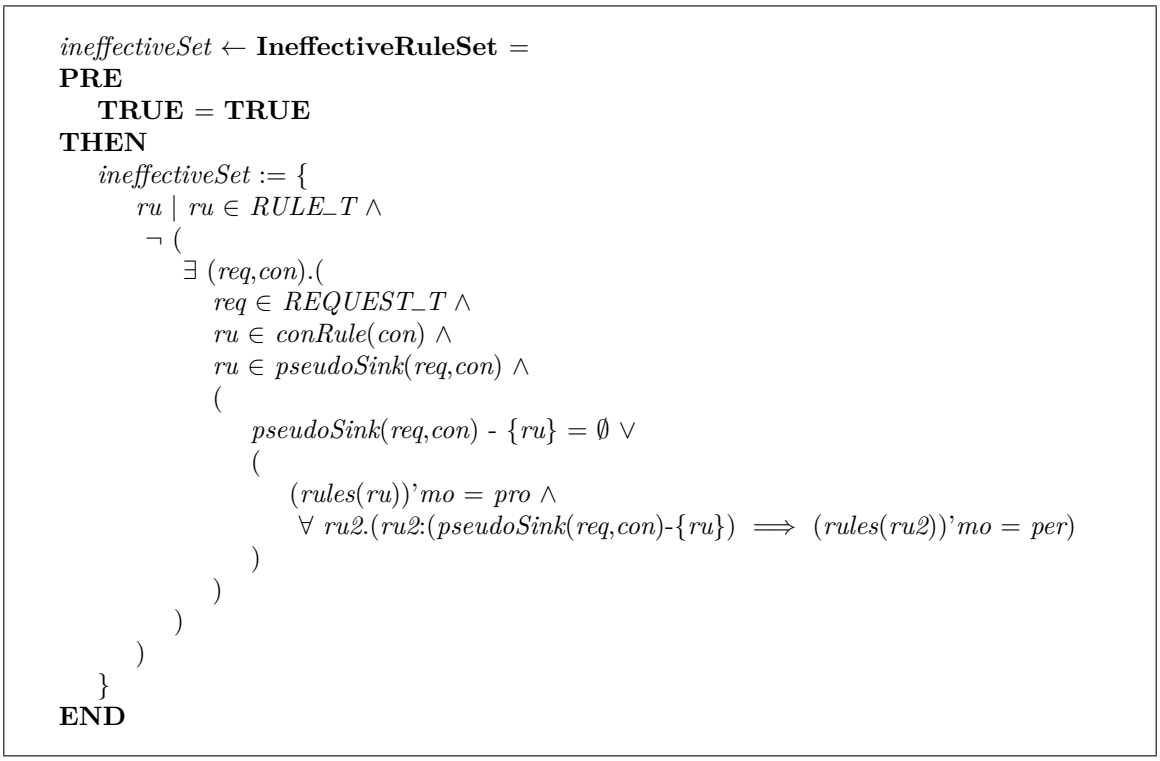


Figure 28: Detection of ineffective rules in B

5. Performance tests

In this section, we present the results of the performance tests we conducted. We have tested two aspects of the SGAC method: the implementation of SGAC and the property verification.

5.1. SGAC implementation

Since XACML is an industrial standard and is very close to satisfying SGAC requirements, we tried to simulate SGAC policies in XACML using paths and the rule combining algorithm first-applicable. The other rule combination algorithms do not fit the SGAC requirements.

To simulate SGAC policies in XACML, we proceed as follows. We define three XACML policies, sets of rules with combining algorithm, one for each level of priority (law, patient and hospital). We use first-applicable as the policy combination algorithm. Within an XACML policy, we order rules according to the subject hierarchy and modality, enumerating the subject graph in a post-order fashion (*i.e.*. bottom-up). We use first-applicable as the rule combination algorithm of a policy. SGAC rule subjects are translated as a regular expression of the form "$*s*$". A request $q = (s_i, a, r)$ is rewritten using the XACML context handler as $q = (s_1/\ldots/s_i, a, r)$, where $s_1/\ldots/s_i$ is the path from the root of the subject graph to the vertex s_i targeted by the request. Of course, this only works when the subject graph is a tree, in which case there is a single path from the root to s_i. A request can then match any rule that applies to any ancestor subject of s_i, since rule subjects are expressed as regular expressions matching any path that contains the rule subject.

To compare the performance of XACML with SGAC, we have used Balana [20], an open-source implementation of XACML based on Sun's XACML implementation. The tests were performed on a server running a virtual machine (Intel(R) CPU 2.67 GHz, 4.00 GB RAM). Balana is written in Java. SGAC is written in NodeJS.

We have generated SGAC policies in a random fashion using a program that generates a subject tree and a resource tree with depth h and node branching factor b, which gives a tree of size $(b^h - 1)/(b - 1)$.

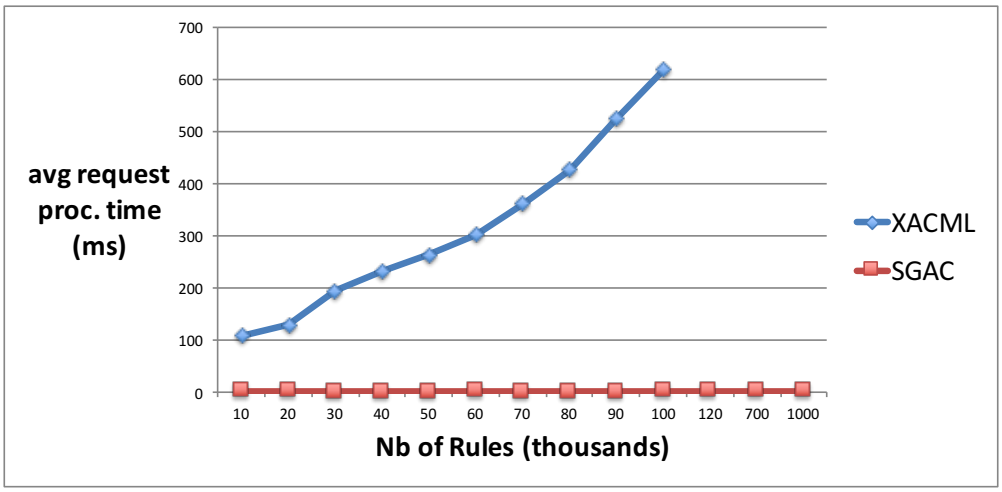


Figure 29: SGAC versus XACML performances summary

Rules are randomly generated. The size of the trees ranged from 1093 to 21845 vertices. Fig. 29 shows the average request processing time in miliseconds versus the number of rules given in thousands. Here are the conclusions we drew from these results.

- Request processing time with XACML is significantly longer than SGAC's to evaluate the same request. When the number of rules is important (*e.g.*, 100 000 rules), SGAC is in average 300 times faster.
- Request processing time with XACML increases linearly with the number of rules whereas SGAC's is near constant (≤ 2 ms in average for up to 1M rules, and a maximum of 7 ms). SGAC uses an $n \log n$ algorithm for indexing rules at system initialisation, where n is the size of the subject graph, and a hash table provides a near constant time for fetching rules applicable to a request.
- When the number of rules is high (200 000 rules for instance), XACML cannot load the file containing the policies: the error returned refers to insufficient Java heap space, which remained even after increasing the memory to 12 GB on a 64-bit architecture. SGAC could process all tests on a 4GB virtual machine with a 32-bit architecture. XACML policies are written in XML, and they are quite verbose.

5.2. Property verification

To test our models, we randomly generate graphs and requests. We vary the following parameters: the number of vertices in the graphs, the number of contexts, the number of rules and the number of requests. We check all four basic properties by varying only one parameter at a time. For a given value of the parameters, we generate at least 6 models and compute the average execution time for checking the properties of the models. For Alloy, each property is verified by running a check or run command. For ProB, one execution can verify all the properties. Thoses tests were performed on a virtual server (Intel Xeon 3.10GHz) using Java 1.7 with 12GB of RAM.

5.2.1. Varying the number of contexts

In this experiment, the number of vertices is set to 30 in each graph, and the number of rules to 12. The results, presented in Figure 30, show that the solving time is quite constant and seems to be independent of the number of contexts.

5.2.2. Varying the number of vertices

For this second test in Figure 31, the number of rules is set to 13, the number of contexts to 10. Alloy grows linearly. ProB is also linear with a slower growth speed than Alloy

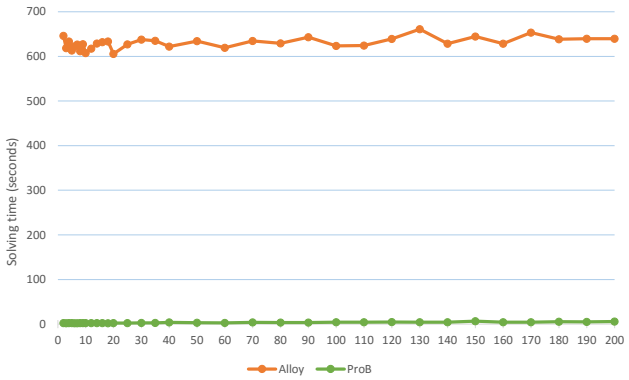


Figure 30: Varying the number of contexts versus the solving time (30 vertices, 12 rules)

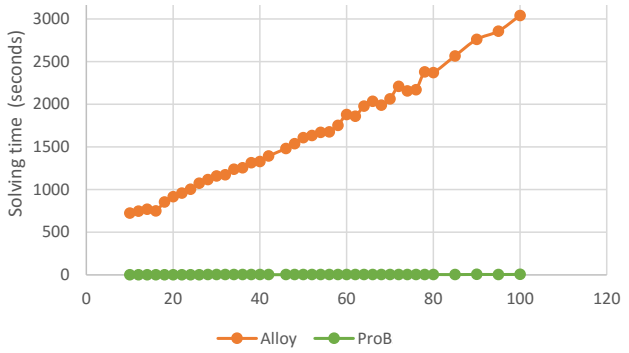


Figure 31: Varying the number of vertices versus the solving time (13 rules, 10 contexts)

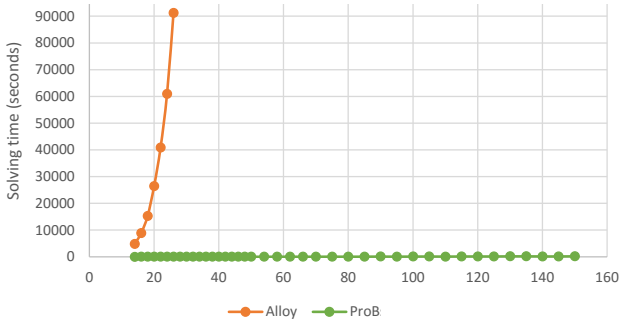


Figure 32: Varying the number of rules versus the solving time (100 vertices, 30 contexts)

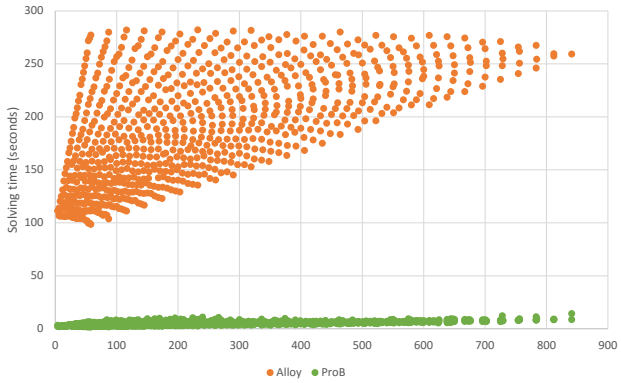


Figure 33: Varying the number of requests versus the solving time (30 vertices, 20 rules and 10 contexts)

5.2.3. Varying the number of rules

For this test, depicted in Figure 32, the number of vertices in each graph is set to 100 and the number of contexts to 30. For 26 rules, Alloy takes more than 16 minutes to give the result of one run command while ProB takes 16 seconds for the whole verification. We observe that increasing the number of rules is the fastest way to increase the solving time in both Alloy and ProB. We also repeated the experience with smaller graphs and number of contexts (40 and 20, 20 and 3), the result is the same: rule number has the greatest impact on solving time, greater than the number of vertices.

5.2.4. Varying the number of possible requests

The previous tests highlight that another parameter in the graphs may affect the solving time: the number of sinks of the graphs. In this fourth experiment, we generate graphs in order to control the number of sinks, which determines the number of possible requests. In Figure 33, we compare Alloy and ProB for 30 vertices, 20 rules and 10 contexts when varying the number of sinks (requests) in each graph. A cloud of points appears, because for a given number of requests, there are several combinations of sinks in the subject graph and the resource type graph (*e.g.*, $12 = 6 \times 2 = 4 \times 3$ possible requests from 2 resources sinks and 6 subjects, or 4 resources and 3 subjects).

5.2.5. Upper bounds

We managed to reach 300 vertices, 160 rules, 100 contexts with 200 requests in about 15 minutes with ProB. This could be sufficient to use ProB on real verification cases, since property verification is done per patient.

6. Related Work

RBAC (*Role Based Access control*) [9, 18], is a classic access control model which uses the notions of user, role, operation, object and session. In order to gain privileges, which are represented by a pair (operation, object), the user must have activated one of her/his roles in a session that has the privileges needed. There are two additional features: role hierarchy allows for privilege inheritance among roles, and separation of duty constraints prevent a user from activating/being assigned to specified combinations of roles. Formalisation of RBAC has been done in Z [15] and in B [11]. Properties verified on those formalisation are:

- role activation: a role can be activated only if it is assigned to the user;
- role hierarchy: a role properly passes assigned privileges to its children, and the role hierarchy is acyclic;
- separation of duty: all constraints of separation of duty hold.

RBAC allows privilege grouping, thanks to roles and role inheritance, but it does not support prohibition, conditions, priority, and resource inheritance. This makes the management of complex fine-grain policies quite difficult. Thus, RBAC does not satisfy the requirements of SGAC.

OrBAC [13] (*Organisation-Based Access Control*), is a logic-based access control model which takes into account RBAC weaknesses and fixes some of them. It reuses the notions of role, user, action, object, and adds some new concepts: i) activity, an abstraction of actions, ii) view, an abstraction of objects, iii) contexts, which allow for the expression of complex rule conditions, iv) prohibition, v) priority in order to manage conflicts, and vi) organisation. The concept of organisation is used to parameterize assignment of roles to users, of views to objects, and of subjects to roles. It supports two kinds of rules: organisational rules that use abstract notions, and concrete rules that use concrete notions. Conflicts are detectable by static checking with the Prolog-based tool MotOrBAC [3]. If two organisational rules with different modalities are applicable to the same abstract concepts, then a potential conflict is detected. This conflict is only potential since there may not exist a common concrete entity (subject, action or object) for which the two organisational rules apply. The user can solve a potential conflict by modifying the priority or the rules, by adding separation constraints, or by just ignoring the conflict when the user knows that there is no concrete entity for which the two organisational rules simultaneously apply. Inheritance among roles or views can be specified by using logic rules. OrBAC is powerful enough to satisfy the SGAC requirements, but its logic-based approach may suffer from performance problem for a very large number of rules, since its execution engine is based on a prolog-like language. Conflict management also requires manual intervention, whereas SGAC uses an ordering that forbids conflicts.

Ponder [17] has a domain hierarchy which contains resources and subjects in the same graph. A rule in Ponder has a subject, a resource, an action, a modality and a condition. It can also be marked as final to have precedence over another rule not marked as such. In case both are/are not marked final, if their subjects are comparable, then precedence is given to the rule with the more specific subject, and if their subjects are the same, then precedence is given to the rule with the more specific resource. Finally, if their subject are not comparable, rules marked as final become normal and if there still is a modal conflict then Ponder returns a prohibition. Ponder does not include a rule priority attribute, and it uses a single graph to represent both subjects and resources, which cannot be used in our case where there is a huge number of resources and subjects. Moreover, its conflict management is not adapted to the SGAC requirements.

XACML [16] (*eXtensible Access Control Markup Language*) is an attribute-based access control language. A rule has a target defined by a subject, an action, a resource, a condition, an effect which can be either *permit* or *deny*. There is no native inheritance among subjects or resources. Tree-like inheritance can be simulated by using paths for resources and subjects identifiers. Precedence among rules is managed by using a rule combination algorithm. The basic rule combination algorithms are:

- permit-overrides: it returns permit if at least one applicable rule returns permit;
- deny-overrides: it returns deny if at least one applicable rule returns deny;

Model	Native Subject Hierarchy	Native Resource Hierarchy	Dynamic Rules	Explicit prohibition	Autonomous Conflict Management	Ease of Rule Expression
RBAC	✓	×	×	×	✓	✓
OrBAC	✓	✓	✓	✓	×	✓
XACML	×	×	✓	✓	✓	×
SGAC	✓	✓	✓	✓	✓	✓

Table 9: Comparison between access control models having a formal model

- first applicable: it returns the effect of the first applicable rule.

XACML satisfies most of the SGAC requirements, but its weak support of inheritance and its management of conflicts make it difficult to manage large security policies. It also suffers from poor performance when a large number of rules are used. Brians [2] formalizes XACML with CSP in order to simulate policies. Using CSP has some drawbacks: conditions are not handled, properties can not be always specified in CSP and our own combining algorithms can't be added easily.

Table 9 summarises the difference between the different models for which a formal model exists.

7. Conclusion

We have proposed SGAC, an innovative access control method, to meet the EHR access control and consent management requirements of a large hospital in Canada (CHUS). SGAC uses an intuitive ordering on rules to manage rule conflicts. This ordering uses priority to manage the different providers of rules and their precedence according to the applicable laws. Subject specificity and modalities are used to order rules of the same priority. SGAC's implementation can manage large policies (at least 1M rules) and large subject and resource type graphs. Its implementation performs significantly better than Balana, an open-source implementation of XACML. SGAC's access control model offers flexibility in managing policies and in satisfying various laws on privacy in Canada. It should be applicable to other legislations in other countries, and to other application domains, like banking, insurance, social networks, government services, etc.

In order to prevent the patient from endangering himself/herself by hiding crucial data, we also have proposed a formal model of SGAC policies to enable automated analysis of policy properties.

We have presented an approach to verify four types of crucial properties (accessibility, availability, contextuality and rule effectivity) for SGAC access control policies using Alloy and ProB. ProB performs significantly better (at least two orders of magnitude) than Alloy for all properties, thanks to the ability to control the solving process in ProB by using B operations which allows one to determine an optimal order in which the constraints are solved, and also by storing frequently needed results into state variables of a B machine. Performance results are promising enough to consider ProB for the verification of real SGAC patient policies. The verification process is completely automatic.

In future work, we plan to investigate SMT solvers and compare their efficiency to ProB. Those like Z3 offering programmable tactics may also offer a good performance; it may allow us to simulate the ordering of the constraint solving process like we did with ProB. However, it is not clear yet whether SMT solvers are good at solving constraints on concrete data sets like our graphs. In addition, SMT solvers may also facilitate the representation of rule conditions. In this paper, we have abstracted from conditions by representing them with sets of contexts. A better approach would be to use the real predicates, which would constitute some kind of higher-order specification where rules are treated as objects and predicates are represented by Boolean functions which must evaluate to true for a rule to apply.

To the extent of our knowledge, this is the first experiment that uses ProB on large data sets, uses rules in constraint solving, and uses B operations to guide the solving process. ProB has been previously used for verifying large data sets of railway parameters, but for some simpler formulas [14], and for university time tables [10]. Treating rules as objects for a constraint solver is a quite challenging task, as illustrated by the heavy computation times of Alloy.

Acknowledgements We would like to thank M. Leuschel for his help and reactive support provided for ProB. This research was funded in part by CIUSSS-Estrie and by NSERC (Natural Sciences and Engineering Research Council of Canada). In particular, the authors would like to thank Hassan Diab, of CIUSSS-Estrie, and Mohammed Ouenzar, of Université de Sherbrooke, for their contribution in defining SGAC, and all the SGAC development team at UdeS and CIUSSS-Estrie.

References

[1] Jean-Raymond Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 2005.

[2] Jeremy Bryans. Reasoning about XACML policies using CSP. In *SWS '05: Proceedings of the 2005 workshop on Secure web services*, pages 28–35. ACM Press, 2005.

[3] Frédéric Cuppens, Nora Cuppens-Boulahia, and Céline Coma. MotOrBAC: un outil d'administration et de simulation de politiques de sécurité. In *Security in Network Architectures (SAR) and Security of Information Systems (SSI), First Joint Conference*, pages 6–9, 2006.

[4] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[5] Morgan Deters, Andrew Reynolds, Tim King, Clark W. Barrett, and Cesare Tinelli. A tour of CVC4: how it works, and how to use it. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, page 7. IEEE, 2014.

[6] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.

[7] Michael Leuschel *et al.* ProB. `http://www.stups.uni-duesseldorf.de/ProB`.

[8] Jerome Falampin, Hung Le-Dang, Michael Leuschel, Mikael Mokrani, and Daniel Plagge. Improving railway data validation with ProB. In *Industrial Deployment of System Engineering Methods*, pages 27–43. Springer, 2013.

[9] David F. Ferraiolo. *Role-Based Access Control, Second Edition.* Artech House, 2006.

[10] Dominik Hansen, David Schneider, and Michael Leuschel. Using B and ProB for Data Validation Projects. In *Proceedings ABZ 2016*, volume 9675 of *LNCS*, pages 167–182. Springer, 2016.

[11] Nghi Huynh, Marc Frappier, Amel Mammar, Régine Laleau, and Jules Desharnais. Validating the RBAC ANSI 2012 standard using B. In Yamine Aït Ameur and Klaus-Dieter Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science*, pages 255–270. Springer, 2014.

[12] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.

[13] Anas Abou El Kalam, Salem Benferhat, Alexandre Miège, Rania El Baida, Frédéric Cuppens, Claire Saurel, Philippe Balbiani, Yves Deswarte, and Gilles Trouessin. Organization based access control. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 120–131, June 2003.

[14] Michael Leuschel, Jérôme Falampin, Fabian Fritz, and Daniel Plagge. Automated property verification for large scale B models with ProB. *Formal Aspects of Computing*, 23(6):683–709, 2011.

[15] David J. Power, Mark Slaymaker, and Andrew C. Simpson. On Formalizing and Normalizing Role-Based Access Control Systems. *The Computer Journal*, 52(3):305–325, 2009.

[16] Erik Rissanen. *eXtensible Access Control Markup Language (XACML) Version 3.0.* OASIS, 2010.

[17] Giovanni Russello, Changyu Dong, and Naranker Dulay. Authorisation and conflict resolution for hierarchical domains. In *POLICY*, pages 201–210. IEEE Computer Society, 2007.

[18] Ravi S. Sandhu, Edward J. Coynek, Hal L. Feinsteink, and Charles E. Youmank. Role-based access control model. *IEEE Computer*, 29(2):38–47, 1996.

[19] Junaid Haroon Siddiqui and Sarfraz Khurshid. Symbolic execution of alloy models. In Shengchao Qin and Zongyan Qiu, editors, *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*, volume 6991 of *Lecture Notes in Computer Science*, pages 340–355. Springer, 2011.

[20] WSO2. Balana. `https://github.com/wso2/balana`.