UNIVERSITÉ DE SHERBROOKE

UNIVERSITÉ PARIS-EST

VÉRIFICATION ET VALIDATION DE POLITIQUES DE CONTRÔLE D'ACCÈS DANS LE DOMAINE MÉDICAL.

par

Nghi Huynh

Thèse en cotutelle présentée
en vue de l'obtention du grade de philosophiæ doctor (Ph.D.)

soutenue le 6 Décembre 2016

Le 6 Décembre 2016,

le jury a accepté la thèse de Monsieur Nghi HUYNH dans sa version finale.

Membres du jury

Professeur Marc FRAPPIER
Directeur de recherche
Département Informatique
Université de Sherbrooke, Canada

Amel MAMMAR
Directrice de recherche
Télécom Sud-Paris, France

Professeur Régine LALEAU
Directrice de recherche
Département Informatique, LACL
Université Paris-Est Créteil, France

Professeur Catalin DIMA
Membre interne
Département informatique, LACL
Université Paris-Est Créteil, France

Professeur André MAYERS
Membre interne
Département Informatique
Université de Sherbrooke, Canada

Professeur Yves LEDRU
Membre externe
Université Grenoble-Alpes, France

Professeur Kamel ADI
Rapporteur
Université du Québec en Ouataouais, France

Professeur Jean-Paul BODEVEIX
Rapporteur
Université Paul Sabatier (Toulouse), France

Sommaire

Dans le domaine médical, la numérisation des documents et l'utilisation des dossiers patient électroniques (*DPE*, ou en anglais *EHR* pour *Electronic Health Record*) offrent de nombreux avantages, tels que le gain de place ou encore la facilité de recherche et de transmission de ces données. Les systèmes informatiques doivent reprendre ainsi progressivement le rôle traditionnellement tenu par les archivistes, rôle qui comprenait notamment la gestion des accès à ces données sensibles. Ces derniers doivent en effet être rigoureusement contrôlés pour tenir compte des souhaits de confidentialité des patients, des règles des établissements et de la législation en vigueur.

SGAC, ou Solution de Gestion Automatisée du Consentement, a pour but de fournir une solution dans laquelle l'accès aux données du patient serait non seulement basé sur les règles mises en place par le patient lui-même mais aussi sur le règlement de l'établissement et sur la législation. Cependant, cette liberté octroyée au patient est source de divers problèmes : conflits, masquage des données nécessaires aux soins ou encore tout simplement erreurs de saisie. C'est pour cela que la vérification et la validation des règles d'accès sont cruciales : pour effectuer ces vérifications, les méthodes formelles fournissent des moyens fiables de vérification de propriétés tels que les preuves ou la vérification de modèles.

Cette thèse propose des méthodes de vérification adaptées à *SGAC* pour le patient : elle introduit le modèle formel de *SGAC*, des méthodes de vérifications de propriétés telles l'accessibilité aux données ou encore la détection de document inaccessibles. Afin de mener ces vérifications de manière automatisée, *SGAC* est modélisé en *B* et *Alloy* ; ces différentes modélisations donnent accès aux outils Alloy et ProB, et ainsi à la vérification automatisée de propriétés via la vérification de modèles ou

Sommaire

model checking.

Mots-clés: Contrôle d'accès, méthodes formelles, sécurité, protection des données, vérification, validation, politique de sécurité.

Remerciements

En tout premier lieu, j'aimerais remercier mes co-directeurs de thèse pour leur encadrement. Je tiens à remercier le professeur Marc Frappier qui m'a permis d'embarquer dans cette grande aventure. Sa bonne humeur et sa motivation m'ont été d'une aide précieuse dans les phases difficiles de la thèse. Je souhaite également remercier la professeure Régine Laleau pour m'avoir proposé cette thèse en cotutelle. Son dynamisme, ses encouragements et ses remarques constructives avec le recul nécessaire ont été des atouts notamment lors des phases de rédaction. Je remercie finalement ma dernière directrice de thèse, Amel Mammar pour son soutien continu qui s'est avéré inestimable, en particulier avec le début chaotique de ma vie de thésard. Sa disponibilité et sa réactivité ont été plus que bienvenues dans les périodes difficiles.

Ma thèse n'aurait pas été possible sans le soutien du Centre Hospitalier de l'Université de Sherbrooke qui a financé en partie ma thèse et fourni un sujet très intéressant avec une problématique concrète.

Je souhaite remercier tous les membres du GRIL au Québec, au sein duquel j'ai passé plusieurs années (maîtrise comprise), pour l'ambiance de travail idéale, en particulier Raphaël, pour toutes les riches discussions que nous avons eues sur des sujets divers et variés. Je n'oublie pas les membres du LACL en France qui m'ont accueilli chaleureusement et avec qui j'ai partagé de nombreux repas, cours, TDs, TPs, et j'ai une pensée particulière pour les membres de mon bureau, Yohan, Quentin, Thomas, Assal et Rodica.

Je n'oublie pas mes proches, mes amis, ma famille pour tout le soutien moral, les relectures et tant d'autres choses...

Abréviations

ANSI American National Standards Institute

CHUS Centre Hospitalier de l'Université Sherbrooke

CSP Communicating Sequential Processes

DAG Directed Acyclic Graph

DPE Dossier Patient Électronique

DSD Dynamic Separation of Duty

EHR Electronic Health Record

NIST National Institute of Standard and Technology

OCL Object Constraint Language

OrBAC Organisation Based Access Control

PDAG Parametric Directed Acyclic Graph

RBAC Role Based Access Control

SGAC Solution de Gestion Automatisée du Consentement

SOD Separation Of Duties

SSD Static Separation of Duty

UML Unified Modeling Language

XACML eXtensible Access Control Markup Language

XML eXtensible Markup Language

Table des matières

Introduction

Contexte et problématique

L'avènement de la dématérialisation des données, illustré par la prolifération des bases de données, facilite et accélère la circulation des informations. Dans cette optique, plusieurs pays ont décidé de mettre en place des structures de partage de données cliniques : les dossiers médicaux électroniques. Selon Inforoute Santé du Canada [24], organisme mandaté par le gouvernement canadien pour la mise en place des dossiers médicaux électroniques, ces derniers permettent aux équipes de soins d'avoir une image plus complète de la santé de leurs patients et d'améliorer la communication entre les équipes soignantes.

On retrouve parmi les avantages octroyés par l'usage des dossiers médicaux électroniques :

— une efficacité accrue dans les cabinets de médecins avec des démarches plus simples et moins longues pour l'accès aux données patient. Par conséquent, les médecins se concentrent davantage sur les soins à prodiguer aux patients ;

— une réduction des examens redondants, car les praticiens ont une vue précise des examens déjà effectués ;

— une sécurité du patient accrue, grâce à l'historique des réactions aux différents médicaments ;

— une communication plus efficace entre les différents acteurs de la santé ;

— une meilleure prise en charge des patients pour les soins préventifs et la gestion des maladies chroniques.

Cependant, les modalités d'accès au dossier diffèrent selon chaque pays, voire province. Certains incluent par défaut le consentement du patient, c'est-à-dire que ce dernier doit se manifester s'il ne veut pas que ses données soient partagées sans son accord préalable. C'est le cas par exemple du Québec [13], de l'Alberta [36], du Royaume-uni. D'autres pays (e.g. France [9], Suède [35], Washington [49]) n'incluent pas le consentement du patient par défaut, le patient devant se manifester s'il veut que ses données soient partagées entre les différents établissements.

Dans le domaine médical, le secret professionnel et l'éthique du personnel soignant garantissent la confidentialité des informations renseignées sur des supports papiers. Au Québec, l'accès aux dossiers médicaux papiers se fait sous la supervision des archivistes qui gèrent les accès allant de la simple consultation au cas d'urgence, en passant par les cas de transfert de patients entre deux établissements. Cependant, avec le passage progressif au format électronique, la question de la gestion de l'accès aux informations dématérialisées se pose.

Les solutions adoptées par certains pays permettent au personnel soignant d'accéder soit à l'intégralité du dossier, soit à un sous-ensemble prédéfini en fonction du rôle de l'intervenant. Ce choix, loin d'être satisfaisant, ne permet pas au patient d'exprimer son consentement d'une manière granulaire. En effet, accepter que ses données soient échangées équivaut à consentir aux accès de tous les intervenants : le patient ne peut permettre/interdire une personne en particulier d'accéder à une ressource en particulier.

Outre la nécessité de protéger le patient des fuites d'informations et des accès non autorisés, le contrôle d'accès doit prendre également en compte la sécurité physique du patient : en cas de restriction d'accès trop forte, le contrôle d'accès pourrait compromettre la santé du patient en empêchant le personnel médical de prodiguer les soins adéquats par manque d'information.

Afin de laisser le patient gérer de lui-même la divulgation de ses données tout en garantissant sa sécurité, le Québec a choisi comme solution le consentement encadré par la loi. En effet, le consentement du patient est requis pour accéder à son dossier, à l'exception de cadres spécifiques définis strictement par la loi. Par exemple, lors d'une situation de vie ou de mort, les accès nécessaires à la prise en charge du patient dans ce contexte sont autorisés sans le consentement de celui-ci.

Introduction

L'introduction du consentement permet au patient d'interdire des accès normalement autorisés par les règles de contrôle d'accès en vigueur, ou également d'autoriser des accès qui auraient été interdits : un patient peut autoriser une sage-femme extérieure qu'il connaît personnellement à accéder à ses données psychiatriques même si cela n'aurait pas été en temps normal autorisé, à cause du profil de l'intervenant.

Cependant, la possibilité donnée au patient de spécifier lui-même des règles d'accès via son consentement introduit les problèmes potentiels suivants.

1. Des **conflits** peuvent survenir entre les règles définies par le patient et celles provenant des deux autres sources que sont le règlement interne de l'hôpital et la législation en vigueur. Ce problème est traité au cas par cas par les archivistes. Cela est possible pour l'instant vu le faible nombre de patients exprimant leur consentement en dehors du consentement global. Dans l'optique d'une utilisation du consentement à plus grande échelle, une solution systématique de résolution de conflits semble nécessaire.
2. Le patient peut provoquer une **diminution drastique de la qualité des soins reçus** en interdisant au personnel soignant l'accès aux informations pertinentes. Ce problème peut survenir lorsque le patient tente de masquer des données importantes telles que des allergies médicamenteuses ou des antécédents. Afin d'assurer la sécurité du patient et le sensibiliser aux choix qu'il fait, une situation où le patient cache des données jugées importantes doit pouvoir être détectée et il doit pouvoir être averti des conséquences du masquage.
3. Le patient peut faire des **erreurs** en ajoutant ses règles : il peut par exemple penser qu'en cas de conflit la dernière règle ajoutée l'emporterait. Dans ces cas, il est nécessaire de pouvoir montrer au patient les répercussions des nouvelles règles ajoutées afin qu'il puisse comparer ce qu'il attendait de ces règles et le comportement effectif.
4. Le patient peut accumuler des règles au fur et à mesure et ne plus s'y retrouver. Ce problème intervient lorsque le patient n'entretient pas sa base de règles et, par exemple, modifie sans supprimer ses règles. Afin d'assurer un fonctionnement optimal et une lisibilité dans la base de règles du patient, il est nécessaire de pouvoir faire le tri dans les règles et différencier les règles qui sont **inutiles**,

redondantes et donc **supprimables**.

5. Un patient peut produire un nombre important de règles, et avoir beaucoup de documents dans son dossier. Le système de gestion de contrôle d'accès doit pouvoir supporter un **volume de données très important**, de l'ordre de plusieurs centaines de milliers de patients et données.

La vérification et la validation des règles d'accès et des propriétés les concernant deviennent donc primordiales afin d'assister au mieux le patient dans la mise en place de ses règles et que celles-ci n'induisent pas de risques pour sa vie en dégradant la qualité des soins prodigués par le personnel. De plus, la vérification doit pouvoir prendre en compte le volume important de données.

Objectifs

Durant nos travaux de maîtrise [21], une étude comparative des différents moyens de gestion du consentement dans les échanges de données médicales a été effectuée. Avec l'aide de différents scénarios fournis par le Centre Hospitalier de l'Université de Sherbrooke (CHUS), nous avons mis au point une méthode de gestion de consentement, SGAC (Solution de Gestion Automatisée du Consentement). Elle permet de prendre en compte le consentement du patient, les règles usuelles d'accès des intervenants des établissements de soins et ainsi que des règles issues de la législation en vigueur. La méthode de résolution de conflits qui surviennent entre les différentes règles, élaborée à la maîtrise, n'est pas parfaite, ni correctement définie et ne prend pas en compte tous les cas de figure. Cette thèse poursuit les travaux sur SGAC en le perfectionnant, notamment par l'ajout de méthodes de vérification appliquées aux politiques de contrôle d'accès aux données des patients. L'objectif, à terme, est d'élaborer un outil assurant au patient que le contrôle d'accès qu'il a mis en place est conforme à ce qu'il souhaite, et également cet outil doit être capable de l'avertir si les règles qu'il a définies risquent de dégrader la qualité des soins. Cet outil doit donc pouvoir simuler le comportement d'un ensemble de règles d'accès, résoudre les conflits entre celles-ci s'il y a lieu et détecter les cas d'informations inaccessibles.

Nous nous proposons donc dans cette thèse de :

— formaliser SGAC, décrire formellement le comportement de SGAC et notamment la méthode de résolution de conflits ;

— définir une méthode de détection de documents inaccessibles ;

— définir une méthode de détection de règles inefficaces ;

— développer un outil de vérification pour SGAC ;

— définir des méthodes d'optimisation afin d'améliorer le temps de traitement des requêtes par SGAC ainsi que le temps de la vérification.

Contribution et plan de thèse

Nous adoptons ici le format d'une thèse par articles, dans laquelle chacun des trois premiers chapitres correspond à un des articles rédigés au cours de la thèse.

1. Évaluation d'un modèle de contrôle d'accès normalisé

Après avoir identifié les besoins du Centre Hospitalier de l'Université de Sherbrooke (*CHUS*) pour la mise en place du contrôle d'accès pour le dossier patient électronique, nous nous sommes intéressés aux modèles de contrôle d'accès existants et leurs formalisations, afin de déterminer s'ils pouvaient être utilisés pour répondre à notre problématique. Nous avons commencé par étudier l'adéquation de RBAC (Role Based Access Control) [46], un modèle de contrôle d'accès normalisé et largement utilisé, pour déterminer si ce modèle pourrait permettre la mise en place du contrôle d'accès pour le dossier patient électronique.

Cette étude a requis de tout d'abord formaliser RBAC en B [1] en suivant la norme *ANSI*. La méthode B [1] est une méthode de spécification formelle qui permet de spécifier un système de son analyse jusqu'à son implémentation. Cette formalisation rigoureuse a révélé des problèmes au niveau de la norme allant des fautes typographiques aux erreurs de logique. Une liste détaillée des lacunes détectées et la méthode appliquée pour les vérifier, ainsi que des propositions de solutions de corrections font l'objet de l'article présenté dans le premier chapitre.

2. Proposition d'un modèle de contrôle d'accès

Au-delà des défauts identifiés dans la partie précédente, RBAC s'est par ailleurs avéré incompatible avec les besoins définis par le CHUS, du fait de l'impossibilité de gérer de manière satisfaisante les interdictions et les sources multiples de règles. En effet, les interdictions ne peuvent être exprimées directement, mais seulement en altérant les règles de permissions déjà en place, ces règles étant parfois issues de sources différentes. Il en découle une perte d'information et l'impossibilité de modifier une règle applicable à plusieurs patients.

Dans le cadre du même travail d'analyse de l'existant, nous avons aussi évalué d'autres modèles connus tels que OrBAC [27] (Organisation-Based Access Control)et XACML [43] (eXtensible Access Control Markup Language) pour vérifier s'ils pouvaient s'appliquer à notre problématique. OrBAC ne répond pas aux exigences du CHUS puisqu'il ne permet pas de résoudre les conflits automatiquement. Quant à XACML, il satisfait la plupart des critères du CHUS, mais il est difficile à mettre en œuvre car il nécessite d'ordonner les règles à la main. De plus, ses performances sont insuffisantes, en particulier quand le volume de règles est important.

Nous avons donc travaillé à l'élaboration d'un modèle formalisé de gestion du consentement : *SGAC* (Solution de Gestion Automatisée du Consentement). L'article présenté dans le chapitre 2 décrit le fonctionnement de ce modèle, sa formalisation mathématique, les méthodes de vérification de propriétés qui lui sont attachées ainsi qu'une comparaison de performance avec XACML. Cette dernière conclut à une nette supériorité de SGAC pour résoudre les conflits dans le cas du CHUS.

3. Comparaison de deux outils de vérification sur notre modèle

Le modèle mathématique de SGAC et les méthodes de vérification de propriétés associées décrites dans le second article ont été mis au point avec pour but de pouvoir être utilisés avec des outils de vérification automatisés. En effet, l'automatisation de la vérification permettra à terme de proposer au patient un outil capable de :

— l'aider à vérifier que les règles qu'il ajoute ont bien les effets escomptés, c'est-à-dire lui indiquer quelles sont les personnes qui auront accès à ses données et dans quels contextes spécifiques,

— simplifier la base des règles que le patient a conçu pour améliorer la lisibilité en détectant les règles qui n'affectent pas le comportement du système,

— le prévenir des dangers potentiels lors du masquage de données importantes, de la dégradation de la qualité des soins qu'il recevra.

L'article constituant le troisième chapitre présente une étude comparative de deux outils de vérification en logique du premier ordre : Alloy [25] et ProB [29], comme solutions d'implémentation de l'automatisation de la vérification des propriétés énoncées précédemment. La démarche appliquée a consisté à spécifier SGAC en Alloy et en B, qui sont les langages de spécification respectifs de Alloy et ProB, et à procéder ensuite à des tests de performances dans le but de déterminer comment évolue le temps de réponse des deux outils en fonction des divers paramètres tels que le nombre de règles, de patients, de documents et de contextes. Les résultats montrent que ProB possède de meilleures performances et qu'il peut analyser un plus grand nombre de règles, de patients et de documents. Les limites de ProB, à savoir des graphes avec 300 sommets, 160 règles et 100 contextes avec 200 requêtes en 15 minutes, permettent de valider l'utilisation de notre modèle dans le cadre réel : des techniques de réduction peuvent être utilisées pour réduire la taille des graphes et le nombre de règles. De surcroît, des optimisations peuvent être apportées au modèle en B, par exemple en programmant l'ordre dans laquelle sont résolues les contraintes du modèle.

4. Conclusion et perspectives

Le dernier chapitre clôt cette thèse en présentant notre conclusion ainsi que les perspectives.

État de l'art

L'état de l'art est réparti dans les chapitres 1, 2 et 3.

Dans le chapitre 1, nous nous intéressons tout d'abord au modèle *RBAC* et à sa formalisation. Pour cela, nous étudions la norme et des travaux concernant la formalisation de *RBAC*, et la détection d'erreurs dans les spécifications de la norme.

Introduction

Dans les chapitre 2 et 3, nous faisons une revue des différents modèles de contrôle d'accès qui pourraient convenir à la gestion du consentement : *RBAC*, *OrBAC*, *XACML*. Nous nous intéressons également à la vérification de propriétés faite avec la formalisation de ces modèles.

Chapitre 1

Validation de la norme RBAC ANSI 2012 à l'aide de B

Résumé

L'article présente une étude critique de la norme ANSI de *RBAC*, un modèle de contrôle d'accès basé sur les rôles. *RBAC* est un modèle très utilisé et a fait l'objet d'une norme *ANSI* qui a été revue en 2012. L'article relève des incohérences dans la tentative de formalisation de *RBAC* de la norme grâce à la méthode B, et suggère des solutions aux problèmes trouvés. Les problèmes trouvés vont des fautes de typographie aux défauts de cohérence en passant par des problèmes d'imprécision. Les incohérences de la normalisation ont été découverts grâce à des violations d'invariants.

Commentaires

Le travail a été fait dans le cadre d'une étude de l'adéquation de *RBAC* avec les besoins du *Centre Hospitalier de l'Université de Sherbrooke* (CHUS) : est-ce qu'un modèle de contrôle d'accès déjà existant pourrait satisfaire les besoins du CHUS ? Les modèles les plus utilisés ont donc été analysés, dont l'un des plus connus et des plus répandus, *RBAC*. RBAC ne répond pas à première vue aux exigences du CHUS, car il est par exemple impossible de spécifier des interdictions, mais a la chance d'être doté d'une norme introduisant une formalisation du

modèle. Cette norme présente toutefois des erreurs qui ont d'abord attiré notre attention, puis motivé une étude approfondie, étude qui a permis de mettre en évidence des erreurs beaucoup plus sévères et de graves lacunes dans la norme et sa pseudo-formalisation.

Cet article a été accepté à la conférence *ABZ 2014* qui a eu lieu à Toulouse, puis sélectionné et publié dans le journal *Science of Computer Programming*, pour l'édition spéciale consacrée à la conférence *ABZ*, en version longue qui est présentée ci-après. Ma contribution se résume comme suit :

— formalisation de *RBAC* en B en suivant à la lettre la description donnée dans la norme ;

— vérification du modèle à l'aide de preuves et d'outils tels qu'AtelierB et ProB ;

— détection des lacunes et proposition de correctifs.

A formal validation of the RBAC ANSI 2012 standard using B

Nghi Huynh
Université de Sherbrooke, Québec, Canada
Université Paris Est-Créteil, Val de Marne, France

Marc Frappier
Université de Sherbrooke, Québec, Canada

Amel Mammar
SAMOVAR, Télécom SudParis, CNRS, Université Paris-Saclay, EVRY, France

Régine Laleau
Université Paris Est-Créteil, Val de Marne, France

Jules Desharnais
Université Laval, Québec, Canada

Keywords: Role-Based Access Control; B method; invariant preservation

Abstract

We validate the RBAC ANSI 2012 standard using the B method. Numerous problems are identified: logical errors, inconsistencies, ambiguities, typing errors, missing preconditions, invariant violation, inappropriate specification notation. A clean version of the standard written in the B notation is proposed. We argue that the *ad hoc* mathematical notation used in the standard is inappropriate and we propose that a more methodological and tool-supported approach must definitely be used for writing standards, in order to avoid the issues identified in the paper. Human reviewing is insufficient to produce error-free international standards.

1.1 Introduction

RBAC is one of the most cited access-control models in the scientific literature (27 300 references in Google Scholar, 1 326 references in ACM digital library), and one of the most widely used models in industry [37]. It is an ANSI standard developed by INCITS (International Committee for Information Technology Standards) [2, 3, 46], with a first edition produced in 2004 and a recent revision published in 2012. It is recommended by numerous governmental agencies, like Canada's Health Infoway, for controlling access to sensitive information like electronic health records (EHR). In a recent project on access control and consent management, we decided to follow these recommendations and evaluate the adequacy of RBAC for managing access to EHR. We were surprised by the number of errors and inconsistencies found in the standard. Even more surprising, all errors can be found in both editions (2004 and 2012), and the 2012 edition has been reviewed/voted by more than 141 persons (as listed in the standard).

The standard is written using mathematical definitions in the style of Z, but without strictly following the Z syntax. The mathematical definitions have not been syntax-checked nor type-checked, thus several errors could have been easily avoided. Some mathematical notations are not drawn from Z and seem rather *ad hoc*, as they are not easily found in standard mathematical textbooks, leaving the reader to guess their meaning from the context. More importantly, not sticking to the Z syntax also leads to several ambiguities, since the mathematical text interpreted with the Z semantics does not always match the natural language description. In order to make sense of the mathematical definitions, the reader must assume declarations which have been omitted in the Z schemas, relying on the natural language text to make such inferences. This is contrary to good specification practice, where the mathematical text is the definitive description, since it offers more precision than natural language. The standard leaves out important concepts, which certainly do not help in reaching the objective stated in the introduction of the standard:

> Development [of] this standard was initiated [...] in recognition of a need among government and industry purchasers of information technology products for a consistent and uniform definition of role-based access con-

> trol (RBAC) features. [...] This lack of a widely accepted model resulted in uncertainty and confusion about RBAC's utility and *meaning*. This standard seeks to resolve this situation [...].

The idea of using mathematics to write the standard was certainly a good idea, as it significantly helped in describing abstract concepts, and allowed us to identify inconsistencies, ambiguities and missing elements. Finding errors in a natural language text is definitely more difficult, because too many interpretations are possible, and each reader picks one, according to his personal experience, knowledge and context. For comparison, we have also evaluated the XACML standard [43] where mathematics are not used at all. We found that it is far more difficult to grasp the subtle concepts of XACML and to be reasonably sure that we could comply to it. Thus, using mathematics is a great idea, but it is insufficient to achieve the highest level of confidence in the quality of a standard. In this paper, we hope to show that the use of a formal method, which has a formal syntax and a formal semantics, supported by tools like syntax checkers, type checkers, provers, model checkers and animators, can definitely help in producing a precise and unambiguous description of a standard. We have chosen to use the B method for its rich tool set. In addition, we believe that B has helped us in detecting errors that may not be easy to find with Z, mainly because B requires proving invariant preservation, whereas in Z, invariants are typically included in the state definition and in the definition of operations through the $\Delta State$ decoration, as it was implicitly done in the RBAC standard. Proving invariant preservation helps in finding missing preconditions in operations and in reviewing the behaviour of operations when proof obligations fail.

Li *et al* published a critique of the 2004 standard in [32]. They identified several technical problems and suggested improvements to the standard, which they formulated using plain mathematics [31]. The leading authors of the standard responded to this critique in [16], without really agreeing on any of the critique of [32] (even the typos and type errors identified by Li *et al* are still present in the 2012 version of the standard). The improvements suggested by Li *et al* in [31] do not simply correct the logical flaws, but also propose a different view of RBAC, where, among other things, the notion of session is not included in the core part of RBAC, and permissions are inherited when a role hierarchy is used. Noticing issues with the format of

the specification of [31], Power *et al* [41] provided a formal Z specification of its state space, leaving out the specification of administrative functions described in [3]. They also suggested normalisation functions for permission assignment, and formalised the three interpretations of role hierarchy suggested in [31].

In this paper, our objective is to show that formal methods can significantly help avoiding errors in the specification of RBAC. We take the RBAC standard as it is described in [3], and fix all errors that we have found, to the best of our understanding of the natural language description and the accompanying mathematical text found in [3]. We do not suggest any new behaviour or feature, contrary to [31, 41]. Another goal is to stress that formal methods should be used in a comprehensive manner when writing a standard. This includes specifying both the state invariant and the administrative functions since specifying only the state invariant is insufficient. Proving that administrative functions preserve the invariant provides a greater level of confidence in the standard. We have identified errors that neither [31] nor [41] identified. Using a specification animator is also crucial to validate a specification. It allows to uncover inappropriate behaviours which can not be detected by invariant preservation proofs.

The paper is structured as follows. Section 1.2 provides relevant excerpts of the RBAC standard [3] on data structures and administrative functions to update the value of RBAC data structures. Errors and omissions are identified and discussed. We describe and outline the structure of our B specification in Section 1.3. The complete specification is available in [12]. Section 1.4 provides an overview of the formal validation process we have used and discusses the advantages of using a formal method like B. Section 1.5 compares our findings with similar work on validating and specifying the RBAC standard. We conclude this paper with an appraisal of our work in Section 1.6.

1.2 Data structures of the ANSI RBAC standard

The RBAC standard [3] is decomposed in three components.

1. Core RBAC is the main component and is required in any RBAC system.
2. Hierarchical RBAC introduces a role hierarchy which defines role inheritance.

3. Constrained RBAC introduces separation of duties (SOD) constraints.

A compliant RBAC system is made of the Core RBAC component plus any combination of the other two.

1.2.1 Core RBAC

The main idea of RBAC is that permissions are assigned to roles and users are granted these permissions by being assigned to roles. The Core RBAC component includes the following sets: *USERS*, *ROLES*, *OPS*, *OBS* and *SESSIONS*, which respectively stand for the set of users, the set of roles, the set of operations, the set of objects on which are applied the operations and the set of sessions where a user can activate a role.

The following definitions are reproduced verbatim from [3]. As a convention, all verbatim excerpts from [3] are in blue, while problems are in red within the excerpts and numbered in superscript. Problems are explained in the text following the excerpts, numbered with Pi.

Core RBAC Reference Model

- $USERS, ROLES, OPS$ and OBS [1] (users, roles, operations and objects respectively).
- $UA \subseteq USERS \times ROLES$, a many-to-many mapping [2] user-to-role assignment relation.
- $assigned\_users : (r : ROLES) \rightarrow 2^{USERS}$ the mapping of role r onto a set of users.
  Formally: $assigned\_users(r) = \{u \in USERS | (u, r) \in UA\}$
- $PRMS = 2^{(OPS \times OBS)}$ [3], the set of permissions.
- $PA \subseteq PERMS$ [4] $\times ROLES$ a many-to-many mapping permission-to-role assignment relation.
- $assigned\_permissions(r : ROLES) \rightarrow 2^{PRMS}$, the mapping of role r onto a set of permissions.
  Formally: $assigned\_permissions(r) = \{p \in PRMS | (p, r) \in PA\}$

- $Op(p : PRMS) \rightarrow \{op \subseteq OPS\}$ [5], the permission to operation mapping, which gives the set of operations associated with permission p.
- $Ob(p : PRMS) \rightarrow \{ob \subseteq OBS\}$ [6], the permission to object mapping, which gives the set of objects associated with permission p.
- $SESSIONS$ = the set of sessions.
- $session\_users$ [7]$(s : SESSIONS) \rightarrow USERS$, the mapping of session s onto the corresponding user.
- $session\_roles(s : SESSIONS) \rightarrow 2^{ROLES}$, the mapping of session s onto a set of roles.
  Formally: $session\_roles(s_i) \subseteq \{r \in ROLES | (session\_users(s_i), r) \in UA\}$
- $avail\_session\_perm$ [8]$(s) \rightarrow 2^{PRMS}$, the permissions available to a user in a session $= \bigcup_{r \in session\_roles(s)} assigned\_permissions(r)$

Description of problems

P1 Typo: all functions of Section 7 (Functional Specification Overview) of [3] use set $OBJS$ instead of OBS. OBS is declared here and used everywhere in this section, but not in the rest of the standard.

P2 Improper terminology: in standard mathematics, a mapping is a function. The notion of a "many-to-many mapping" does not make sense strictly speaking. The term "relation" used further in the sentence suffices.

P3 Type error: functions of Section 7 (Functional Specification Overview) of [3] use this set as if it was defined as $OPS \times OBS$. Note that this set is never updated in any administrative functions of Section 7. This leads us to conclude that $PRMS$ is a type and that all operations on objects are possible, that is, the standard does not provide means for controlling which operations are valid on which objects. On the other hand, functions Op and Ob, declared afterwards, but undefined, hint at the usage of a subset of operations on objects; otherwise, they would be

useless. But these functions are not used in the rest of the standard.

P4 Typo: this symbol is not declared so far. One could presume that it is a typo for *PRMS* defined above and used everywhere in the rest of the data structure declarations, but *PRMS* is not used in Section 7; *PERMS* is used instead.

P5 Unused symbol: this function is not used in the rest of the specification. By its description, it is a derived function, but its definition is not provided; only its type. Moreover, the notation $\{op \subseteq OPS\}$ is not standard mathematics nor standard Z notation. One first guesses that its intended meaning is $\{op \mid op \subseteq OPS\}$, but since this sentence seems to be only providing a type for function *Op*, the set *OPS* would suffice.

P6 Unused symbol: same issues as for *Op*. This function is undefined and not used in the rest of the specification.

P7 Unused symbol: this function is not used in the rest of the standard. The function *user_sessions*, which maps users to sessions is used instead (and undeclared anywhere).

P8 Unused symbol: this function is not used in the rest of the standard. Administrative function **CheckAccess** provides the same information.

Appraisal of the definitions

Four symbols out of twelve are introduced upfront in the standard, but never used in the sequel. This generates unnecessary noise for the reader. Moreover, none of these definitions clearly emphasises under what conditions a user can use an operation on an object. This is quite surprising, because this is the core purpose of the standard. The definition of *avail_session_perm* describes the permissions *available* in a session, but it does not explicitly state that it determines if a user can execute an operation on an object. The reader has to wait until Section 7, page 17, where function **CheckAccess**,

buried among other administrative functions, nails it down in a decisive manner:

> This function returns a Boolean value meaning whether the subject of a given session is allowed or not to perform a given operation on a given object.

The standard introduces a number of symbols (sets, relations, functions), but does not state whether they are state variables, specification parameters, or sets used only for typing. For instance, no distinction is made on the nature of sets *USERS*, *ROLES*, *OPS* and *OBS*. The first two are state variables (since they are updated by some administrative functions of Section 7); the last two are never updated and can be considered as parameters of the specification used for typing only. These distinctions would be made if a formal specification like B, Z or ASM was used. The use of derived functions like *assigned_users*, *assigned_permissions* and *avail_session_perms* can create some confusion and inconsistencies when writing administrative functions. For instance, *UA* and *assigned_users*(r) are both updated and kept consistent in administrative functions updating them. Similarly for *PA* and *assigned_users*. On the other hand, function *avail_session_perm* is never maintained in the administrative functions. Following common practice in the B method, these derived functions would not be included as state variables, since they do not contain any new information. Their inclusion would only complicate the invariant preservation proof and the specification of operations. They would be included as DEFINITIONS, which are similar to LET constructs in programming languages. Li *et al.* [32] also suggested not to use derived functions.

Finally, ad hoc mathematical notations are used (*e.g.*, declaration of function *Op*), while in Section 7, the Z notation is said to be used for specifying operations. For the sake of uniformity, the Z notation could have also been used to define functions.

1.2.2 Hierarchical RBAC

This component introduces role hierarchies which define an inheritance relation among roles.

This relation has been described in terms of permissions: r_1 “inherits" role r_2 if[9] all privileges of r_2 are also privileges of r_1. [...]
This standard recognizes two types of role hierarchies—general role hierarchies and limited role hierarchies. General role hierarchies provide support for an arbitrary partial order to serve as the role hierarchy, to include the concept of multiple inheritances of permissions and user membership among roles. Limited role hierarchies impose restrictions resulting in a simpler tree structure (i.e., a role may have one or more immediate ascendants, but is restricted to a single immediate descendent).

General role hierarchy specification

- $RH \subseteq ROLES \times ROLES$ is a partial order on $ROLES$ called the inheritance relation written as $\succeq$, where $r_1 \succeq r_2$ only if[10] all permissions of r_2 are also permissions of r_1, and all users of r_1 are also users of r_2, i.e. , $r_1 \succeq r_2 \Rightarrow authorized\_permissions(r_2) \subseteq authorized\_permissions(r_1)$.
- $authorized\_users(r : Roles) \rightarrow 2^{USERS}$, the mapping of role r onto a set of users in the presence of a role hierarchy. Formally: $authorized\_users(r) = \{u \in USERS | r' \succeq r, (u, r') \in UA\}$ [11]
- *authorized_permissions*[12]$(r : ROLES) \rightarrow 2^{PRMS}$, the mapping of role r onto a set of permissions in the presence of a role hierarchy. Formally: $authorized\_permissions(r) = \{p \in PRMS | r' \succeq r$[13]$, (p, r') \in PA\}$

[...]
Roles in a limited role hierarchy are restricted to a single immediate descendent.
[...]
Node r_1 is represented as an immediate descendent[14] of r_2 by $r_1 \succ\succ r_2$, if $r_1 \succeq r_2$ but no role in the role hierarchy lies between r_1 and r_2. That is, there exists no role r_3 in the role hierarchy such that $r_1 \succeq r_3 \succeq r_2$, where $r_1 \neq r_2$ and $r_2 \neq r_3$[15].

Limited Role Hierarchy Specification

General Role Hierarchies[16] with the following limitation:

- $\forall r, r_1, r_2 \in ROLES, r \succeq r_1 \wedge r \succeq r_2$[17] $\Rightarrow r_1 = r_2$.

Description of problems

P9 Bad definition: this is the first sentence where the inheritance relation is described, and the standard uses a sufficient condition ("*all privileges of* r_2 *are also privileges of* r_1*"*) to describe it; the reader shall later understand that this is instead a necessary condition (*i.e.*, a consequence of stating $r_1 \succeq r_2$).

P10 Bad definition: this is the formal declaration of the inheritance relation, but it is provided in a necessary condition referring to two functions not declared yet (*authorized_users* and *authorized_permission*), leading the reader to question whether he has overlooked some definitions involving $\succeq$ in the previous sections. Moreover, part of the sentence is reformulated in mathematics *(authorized_permission)*, but the other part is not (*authorized_users*), so the reader is not sure if the mathematics covers one or both.

P11 Formal definition: it should be $\{u \in USERS | \exists r' \bullet r' \succeq r, (u, r') \in UA\}$ instead, otherwise r' would be a free variable. Same goes for the expression of *authorized_permissions*.

P12 Unused symbol: this function is never used in the rest of the standard. Moreover, it leads the reader to believe that the permissions of a role include the permissions inherited by the role, but this is not the case. The reader shall later learn, after reading the definition of **CheckAccess** page 17 and **CreateSession** and **AddActiveRole** page 21, that a user only gets the permissions of his active roles, and the inheritance hierarchy has no effect on the permissions of a role. The inheritance hierarchy only determines the users authorised to activate a role. For instance, following the definition of the two aforementioned administrative functions, if $r_1 \succeq r_2$ and $u \mapsto r_1 \in UA$, then user u is allowed to activate r_1 and r_2. By

activating r_1, user u only gets the permission granted to r_1 in PA; the permissions of r_2 can be exercised only if u also activates r_2. Li *et al.* [32] claim that inheritance as presented in the standard can be interpreted in three different ways, but we do not agree with them. If the reader sticks to the mathematical definitions of the standard, then there is only one plausible interpretation. Of course, the natural language text, the errors and superfluous definitions like *authorized_permissions* create confusion, diverting the reader from the mathematical text, which should prevail. This shows the importance of properly distinguishing between definitions and propositions.

P13 Error: it should be $r \succeq r'$, to match the necessary condition defined for $\succeq$ just above, *i.e.*,

$$r_1 \succeq r_2 \Rightarrow \mathit{authorized\_permissions}(r_2) \subseteq \mathit{authorized\_permissions}(r_1)$$

This error was also pointed out by Li *et al.* [32].

P14 Ambiguity: The sentence

> Roles in a limited role hierarchy are restricted to a single immediate descendent.

and its formal representation as the following assertion

$$\forall r, r_1, r_2 \in \mathit{ROLES}, r \succ\!\!\succ r_1 \wedge r \succ\!\!\succ r_2 \Rightarrow r_1 = r_2$$

(where we have corrected the error P17 on $\succeq$ explained below) entail that r_2 is the descendent in $r_1 \succ\!\!\succ r_2$. This usage is also consistent with the formal definition of operation **AddInheritance** provided on page 19 of [3].

> **AddInheritance**(r_asc, r_desc)
>
> This commands establishes a new immediate inheritance relationship $r\_asc \succ\succ r\_desc$ between existing roles r_asc, r_desc.

However, the following sentence defines r_1 as the descendent:

> Node r_1 is represented as an immediate descendent of r_2 by $r_1 \succ\succ r_2$, if $r_1 \succeq r_2$ but no role in the role hierarchy lies between r_1 and r_2.

Thus, there is confusion in the usage of the word "descendent". This confusion probably arises from two different meanings of the word "descendent". In the first case, it means "descending in power", whereas in the second case, it means "descendent" in an inheritance hierarchy.

P15 Error: the standard claims to define the covering relation of an ordered set, which they call immediate descendent, and which is typically used in Hasse diagrams. A third condition is missing to do so, namely $r_1 \neq r_3$. This error was also pointed out by Li *et al.* [32], but their suggested correction is incorrect: they suggest to replace $r_1 \neq r_2$ by $r_1 \neq r_3$, which is insufficient, because the intent of the authors is to define the covering relation of a partial order. All three inequalities are required.

P16 Version change: we have reproduced the 2004 version of the standard [2] here, because the 2012 version [3] uses Definition 2a instead, but there is no definition labelled with 2a in the standard.

P17 Error: the standard claims to define the notion of single immediate descendent in a partial order, *i.e.*, the partial order is a tree, as claimed in the following sentence:

> Limited role hierarchies impose restrictions resulting in a simpler tree structure (i.e., a role may have one or more immediate ascendants, but is restricted to a single immediate descendent).

To do so, the standard should use instead $r \succ\succ r_1 \wedge r \succ\succ r_2$. This error was also pointed out by Li *et al.* [32].

Appraisal of the definitions

Given all these problems, this section of the standard is quite hard to understand. The meaning of relation $\succeq$ is unclear until the specification of the administrative functions is provided in Section 7 of the standard. This is where the reader learns the *indirect* effect of $\succeq$ on the **CheckAccess** predicate, which describes if a user can perform an operation on an object in a given state of the RBAC system. Describing the connection between $\succeq$ and the active sessions would help clarify the meaning of $\succeq$. The following assertion, which is the body of function **CheckAccess**, would show that $\succeq$ does not directly impact the access a user has in a given state.

$$\begin{aligned} \textbf{CheckAccess}(s, op, ob) \quad \Leftrightarrow \quad & s \in SESSIONS \wedge op \in OPS \wedge ob \in OBJS \wedge \\ & \exists\, r \bullet r \in ROLES \wedge r \in session\_roles(s) \wedge \\ & (op \mapsto ob) \mapsto r \in PA \end{aligned}$$

This assertion shows that what is accessible is determined by the roles activated by a user in a session. One then has to find out how variable *session_roles* is updated, by looking at the administrative functions updating it. This is where $\succeq$ comes into play. Function **AddActiveRole**(u, s, r) says that user u can activate role r in session s if $u \in authorized\_users(r)$.

1.2.3 Constrained RBAC

Constrained RBAC adds Separation of Duty relations to the core RBAC model. **Static Separation of Duty** is specified by a role set rs and an integer n such that $2 \leq n \leq card(rs)$. That type of constraint specifies that a user can be assigned to at

most $(n-1)$ roles of rs. Formally, let SSD be the set of the static separation of duty constraints :

- $SSD \subseteq 2^{ROLES} \times \mathbb{N}$
- $\forall (rs, n) \in SSD, \forall t \subseteq rs : |t| \geq n \Rightarrow \bigcap_{r \in t} assigned\_users(r) = \emptyset$
- In presence of role hierarchy
  $\forall (rs, n) \in SSD, \forall t \subseteq rs : |t| \geq n \Rightarrow \bigcap_{r \in t} authorized\_users(r) = \emptyset$

Dynamic Separation of Duty is specified by a role set rs and an integer n such that $2 \leq n \leq card(rs)$. That type of constraint specifies that a user can simultaneously hold at most $(n-1)$ roles of rs, *during one session*. Formally, let DSD be the set of dynamic separation of duty constraints :

- $DSD \subseteq 2^{ROLES} \times \mathbb{N}$
- $\forall rs \in 2^{ROLES}, n \in \mathbb{N}, (rs, n) \in DSD \Rightarrow n \geq 2, |rs| \geq n$ and
  $\forall s \in SESSIONS, \forall rs \in 2^{ROLES}, \forall role2subset \in 2^{ROLES},$
  $\forall n \in \mathbb{N}, (rs, n) \in DSD, role2subset \subseteq rs,$
  $role2subset \subseteq session\_roles(s) \Rightarrow |role2subset| < n.$

We did not find any problem with this part of the specification. However, these constraints could be expressed in a simpler manner, which we have done in our B specification [12].

1.2.4 Administrative functions

Administrative functions describe how the RBAC system state evolves. The standard claims to use the Z notation for specifying administrative functions.

> The notation used in the formal specification of the RBAC functions is a subset of the Z notation. The only change is the representation of a schema as follows:
>
> Schema-Name (Declaration) $\lhd$ Predicate; ...; Predicate $\rhd$
>
> Most abstract data types and functions used in the formal specification are defined in Section 3, RBAC Reference Model. New abstract data types and functions are introduced as needed.

Some examples of such specifications are provided below to illustrate problems with the adapted Z notation used in the standard. They are provided in Figure 1.1, 1.2, 1.3 and 1.4. The specification uses the following B operators.

- $\mathsf{dom}(r) = \{x \mid \exists y \cdot x \mapsto y \in r\}$ is the domain of relation r;
- $\mathsf{ran}(r) = \{y \mid \exists x \cdot x \mapsto y \in r\}$ is the range of relation r;
- $\mathsf{id}(s) = \{x \mapsto x \mid x \in s\}$ is the identity relation on set s;
- $s \lhd r \triangleq \{x \mapsto y \mid x \mapsto y \in r \wedge x \in s\}$ is the domain restriction of relation r by set s;
- $r \rhd s \triangleq \{x \mapsto y \mid x \mapsto y \in r \wedge y \in s\}$ is the range restriction of relation r by set s;
- $s \lhd\!\!\!\cdot\, r \triangleq \{x \mapsto y \mid x \mapsto y \in r \wedge x \notin s\}$ is the domain antirestriction of relation r by set s;
- $r \rhd\!\!\!\cdot\, s \triangleq \{x \mapsto y \mid x \mapsto y \in r \wedge y \notin s\}$ is the range antirestriction of relation r by set s;
- $r[s] \triangleq \mathsf{ran}(s \lhd r)$ is the image set of set s by relation r;
- $\mathsf{closure1}(r) \triangleq r^+$ is the transitive closure of relation r;
- $\mathsf{closure}(r) \triangleq r^* = r^+ \cup \mathsf{id}(s)$ is the reflexive-transitive closure of relation r defined on set s;

AddUser
This command creates a new RBAC user. [...]

AddUser (user : NAME) [18]
◁
$user \notin USERS$
$USERS' = USERS \cup \{user\}$
$user\_sessions'$ [19] $= user\_sessions \cup \{user \mapsto \emptyset\}$
▷

AddUser $(user) =$
PRE
$user \in USERS \wedge user \notin Users$
THEN
$Users := Users \cup \{user\}$
END;

Figure 1.1 – **AddUser** administrative function specification and translation

— **op**$(\vec{x})$ = **PRE** C **THEN** $S_1 \parallel \ldots \parallel S_n$ **END** is the declaration of an operation **op** with parameters $\vec{x}$, precondition C and assignment statements $S_1, \ldots, S_n$ which are simultaneously executed ("$\parallel$").

Description of problems

P18 Notation: the notation used in the standard omits important elements of a Z operation schema. First, it does not identify the state space of the operation. A typical Z operation schema will include a $\Delta State$ declaration, introducing unprimed and primed variables, to denote the before and after states, and their associated invariant. The predicate part should describe the relationship between unprimed and primed variables. Primed variables which are not subject to any condition are allowed to take any value. Obviously, this convention has not been followed in the standard, because we do not expect operation **AddUser** to let all other state variables take any value after execution. Thus, we must assume that the standard uses the convention that primed variables x' which are not occurring in the operation specification are preserved with the equality $x' = x$. However, this

DeleteUser
This command deletes an existing user from the RBAC database. [...]

DeleteUser(user : NAME)
◁
$user \in USERS$
$[\forall s \in SESSIONS \bullet s \in user\_sessions(user) \Rightarrow DeleteSession(s)$ [20]$]$
$UA' = UA - \{r : Roles \bullet user \mapsto r\}$
$assigned\_users' = \{r : Roles \bullet r \mapsto (assigned\_users(r) - \{user\})\}$
$USERS' = USERS - \{user\}$
▷

DeleteUser $(user) =$
PRE
$user \in USERS \wedge$
$user \in Users$
THEN
$Sessions := Sessions - User\_sessions[\{user\}]$
||
$User\_sessions := \{user\} \lhd User\_sessions$
||
$Session\_roles := User\_sessions[\{user\}] \lhd Session\_roles$
||
$UA := \{user\} \lhd UA$
||
$Users := Users - \{user\}$
END;

Figure 1.2 – **DeleteUser** administrative function specification and translation

DeleteSession(*user*,*session*)
This function deletes a given session with a given owner user. [...]

DeleteSession (user,session : NAME) [21] =
◁
user ∈ *USERS*; *session* ∈ *SESSIONS*; *session* ∈ *user_sessions*(*user*)
user_sessions′ = *user_sessions* − {*user* ↦ *user_sessions*(*user*)} ∪
{*user* ↦ *user_sessions*(*user*) − {*session*}}
session_roles′ = *session_roles* − {*session* ↦ *session_roles*(*session*)}
SESSIONS′ = *SESSIONS* − {*session*}
▷

DeleteSession (*user*,*sess*) =
PRE
user ∈ *USERS* ∧ *user* ∈ *Users* ∧ *sess* ∈ *SESSIONS* ∧ (*user* ↦ *sess*) ∈ *User_sessions*
THEN
User_sessions := *User_sessions* − {*user* ↦ *sess*}
||
Sessions := *Sessions* − {*sess*}
||
Session_roles := {*sess*} ⩤ *Session_roles*
END;

Figure 1.3 – **DeleteSession** administrative function specification and translation

DeleteRole
This command deletes an existing role from the RBAC database. [...]

$$
\begin{array}{l}
\textbf{DeleteRole (role : NAME)}\ ^{22} = \\
\lhd \\
\quad role \in ROLES \\
\quad [\,\forall s \in SESSIONS \bullet role \in session\_roles(s) \Rightarrow DeleteSession(s)] \\
\quad UA' = UA - \{u : USERS \bullet u \mapsto role\} \\
\quad assigned\_users' = assigned\_users - \{role \mapsto assigned\_users(role)\} \\
\quad PA' = PA - \{op : OPS, obj : OBJ \bullet (op, obj) \mapsto role\} \\
\quad assigned\_permissions' = assigned\_permissions - \\
\qquad\qquad \{role \mapsto assigned\_permissions(role)\} \\
\quad ROLES' = ROLES - \{role\} \\
\rhd
\end{array}
$$

DeleteRole $(role) =$
PRE
$role \in ROLES \wedge role \in Roles$
THEN
$User\_sessions := User\_sessions \rhd \mathsf{dom}(Session\_roles \rhd \{\ role\ \})$
||
$Session\_roles := \mathsf{dom}(Session\_roles \rhd \{\ role\ \}) \lhd Session\_roles$
||
$Sessions := Sessions - \mathsf{dom}(Session\_roles \rhd \{\ role\ \})$
||
$UA := UA \rhd \{role\}$
||
$PA := PA \rhd \{role\}$
||
$Roles := Roles - \{role\}$
END;

Figure 1.4 – **DeleteRole** administrative function specification and translation (1/2)

DeleteRoleRH(*role*) =
PRE
$role \in ROLES \land role \in Roles$
THEN
DeleteRole(*role*)
||
$RH := \{role\} \lhd RH \rhd \{role\}$
END

DeleteRoleHC (*role*) =
PRE
$role \in ROLES \land role \in Roles \land$
$\forall\ ssd.(ssd \in Ssd \implies role \notin ssd) \land$
$\forall\ dsd.(dsd \in Dsd \implies role \notin dsd)$
THEN
DeleteRoleRH(*role*)
END;

Figure 1.5 – **DeleteRole** administrative function specification and translation (2/2)

convention has not been followed everywhere. For instance, symbol $\succ\succ$ is used in operation **AddInheritance** where $\succeq$ is updated, but $\succ\succ$ is not. However since $\succ\succ$ is supposed to be the covering relation of $\succeq$, we can't assume the equality $\succ\succ'=\succ\succ$, because it would break the invariant linking $\succeq$ and $\succ\succ$. This may suggest that the standard assumes that derived functions need not to be explicitly updated since their definition acts like a state invariant which is assumed to be maintained by operations, as it is the case in Z when $\Delta State$ is used. But the standard doesn't follow this convention either. For instance, in operations maintaining variable *UA*, which maps users to roles, variable *assigned_users* is also maintained, which is not needed, since *assigned_users* is derived from *UA*.

P19 Undeclared symbol: variable *user_sessions* has not been declared in the data structures in the previous section. Variable *session_users*, which has been declared in the data structure section, is not updated by this operation. So the assumption we made in P18 to make sense of the notation used is broken here, be-

cause it makes *session_users* inconsistent with *user_sessions*. Luckily, *session_users* does not seem to be used at all in the specification of administrative functions, so we deduce that its declaration is superfluous in the data structure section of the standard, which solves the inconsistency problem.

P20 Notation: this is one example of operation call which does not follow the Z syntax and that is logically unsatisfiable. The reader must suppose that a more "imperative programming language" view is used here. There are other cases in the standard (*e.g.*, **AddAscendant**, **AddDescendant**, where the two calls are represented implicitly as a conjunction, but sequential composition should have been used, to make sense out of it).

P21 Signature inconsistency: **DeleteSession** is declared with parameters **(user, session: NAME)**, but called as **DeleteSession(session)** in **DeleteRole** and **DeleteUser**. Since a session is related to a single user, as provided by the unused function *session_users*, there is no need for parameter **user**. Note also that updating function *session_users* is simpler than updating its functional inverse *user_sessions*, *i.e.*,

$$session\_users' = \{session\} \lhd session\_users \ .$$

The Z domain subtraction is not used in the standard, and that makes the specification harder to read.

P22 Operation **DeleteRole** does not update relation "$\succeq$" and separation of duty constraints SSD and DSD. The last two ones raise more serious issues to deal with. We see two options:

— remove the deleted role from all the constraint role sets where it appears;

— restrict the operation to a role which is not used in SSD/DSD constraints.

The first option raises the issue of updating the cardinality. Recall that an SSD/DSD constraint (RS, n) states that at most $n-1$ roles of RS can be assigned to/activated by a user. It is subject to the invariant $n \geq 2 \wedge |RS| \geq n$. If $|RS| < n$,

then the constraint can never be violated and it is useless. After deleting a role, we have the following cases:

— $n > 2 \wedge |RS'| = n-1$: n must be decremented by 1, in order for the constraint to satisfy the state invariant;

— if $n = 2 \wedge |RS'| = 1$, the constraint is deleted because it does not satisfy the state invariant $|RS'| \geq n$ and n cannot be fixed by decrementing n, since $n \geq 2$ is required by the state invariant;

— $|RS'| \geq n$: n could be decremented by 1 or left unchanged; it depends on the particular access-control requirements of the application.

In any case, the constraint could be deleted if it does not make sense in the security requirements of the application. Furthermore, removing a role in a constraint role set may introduce constraint redundancy: if two constraints have the same role set, the one with the bigger cardinality is redundant. Then, **DeleteRole** should in addition remove the redundant constraint. Given these cases, it seems safer to let the RBAC manager manually adjust SSD/DSD affected by a role deletion before deleting a role. Hence, we have added a precondition in our specification of **DeleteRoleHC** in Figure 1.4 to check that a role is not used in any SSD/DSD constraint.

We have discovered this issue by proving that operations preserve state invariants and it hasn't been raised in [31, 41].

Appraisal of the definitions

There are two main issues in this section. The first one is the inappropriate usage of the Z notation, which leads to incorrect specifications of several operations that are logically unsatisfiable, but the intent of the specifiers is reasonably understandable. The second one raises a more serious problem; there are missing preconditions in operation **DeleteRole** which cause an invariant violation. We have proposed a new version of this operation in order to have a coherent set of SSD/DSD constraints when a role is deleted.

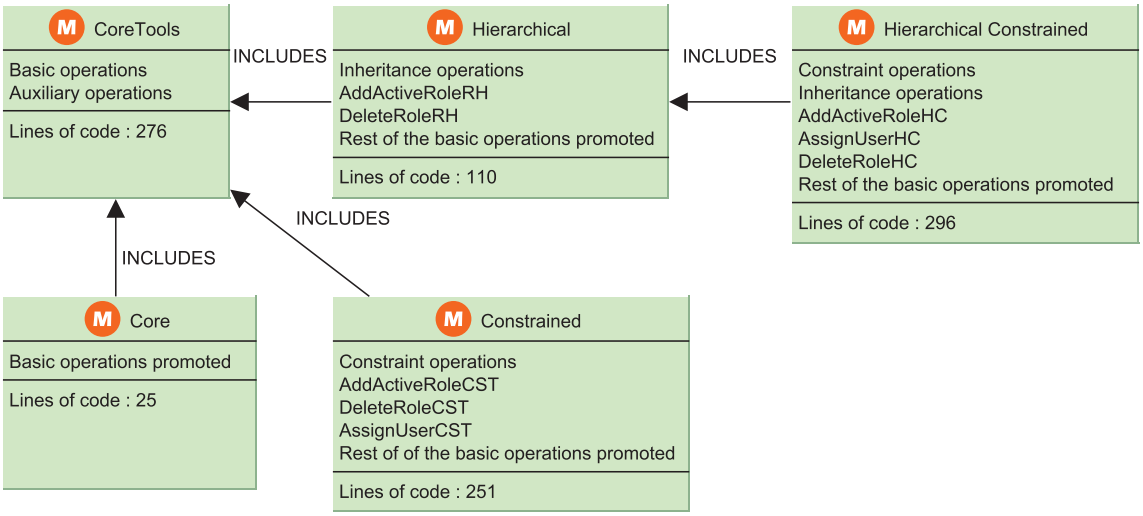


Figure 1.6 – Architecture of our B specification of the RBAC standard

1.3 The B specification of the RBAC standard

Due to space limitation, our B specification is partly omitted and fully provided in [12]. Our specification is structured as follows. Each RBAC component has its own machine, and the Core RBAC machine is included in the other two components. We have a total of five machines; their relationship is illustrated in Figure 1.6.

— CoreTools.mch: This is an auxiliary machine which contains the declaration of elements which are needed in each of the machines representing an RBAC component. Thus, it contains the common features needed by each RBAC component: abstract sets, state variables and operations representing the core behaviour of administrative functions, with weakened preconditions to be reused and strengthened in the other machines according to their needs.

— Core.mch: This machine includes CoreTools.mch and represents Core RBAC.

— Hierarchical.mch: This machine includes CoreTools.mch and introduces the inheritance relation among roles. It represents Hierarchical RBAC.

— Constrained.mch: This machine includes CoreTools.mch and introduces the static and dynamic constraints for separation of duty, by adding them to the invariants. It represents Constrained RBAC.

Component	LOC	Proof Obligations	W.D. PO	Automatic	Interactive
CoreTools.mch	275	51	0	42	9
Core.mch	25	0	0	0	0
Hierarchical.mch	110	16	0	6	10
Constrained.mch	250	105	22	69	58
HierarchyConst.mch	295	112	25	69	68

Table 1.1 – Statistics on our RBAC model in B

— HierarchyConst.mch: This machine includes Hierarchical.mch and adds constraints to represent separation of duty. It represents the combination of Hierarchical RBAC and Constrained RBAC.

Table 1.1 provides statistics on model size and proof obligations, including well-definedness proof obligations. Automatic proofs were automatically discharged by the prover; interactive proofs required human intervention to guide the prover in finding a proof. For information, it took around a hundred hours to fully read, understand, model the standard: most of the time spent have been used into proofs, debugging the specification and animation.

1.3.1 CoreTools.mch

Figure 1.7a presents the static part of CoreTools.mch, which contains the declaration of the core sets, the core variables and the invariants. Choices have been made to simplify the model by removing the relation *assigned_user* since it is derived from *UA*. The same goes for *assigned_permission* and *PA*. CoreTools.mch has all the features needed for Core RBAC except the fact that it does more than Core RBAC: it has auxiliary operations which are called by machines including CoreTools.mch. In B, a machine D which includes a machine C has read-only access to the variables and sets of C. To modify the variables of C, machine D must invoke C's operations. Operation promotion lets the including machine claim the promoted operation from the included machine as its own. Proofs are also inherited: including a machine already proven reduces the number of proofs obligation to discharge in the including ma-

chine. Some administrative functions like **AddActiveRole** have different behaviours depending on the component. CoreTools.mch factors out the common behaviour of the operations in the different components. For example, in Core RBAC, the function **AddActiveRole**(*user*,*session*,*role*) activates the *role* in the *session* of the user *user* only if the role can be activated, *i.e. role* is assigned to *user*. This is not true in presence of a role hierarchy since the user can activate any role he has been assigned directly or by role inheritance. Finally, we have simplified the signature of operation **CreateSession** by omitting parameter *ars*, which is a set of roles to activate when creating the session; this can be achieved by calling operation **AddActiveRole** after creating the session.

1.3.2 Core.mch

Since CoreTools.mch already provides all the operations, sets and variables needed to run a Core RBAC system, Core.mch is very simple. It only includes CoreTools.mch and promotes most of the operations of CoreTools.mch except those operations which are needed only in the other two RBAC components.

1.3.3 Hierarchical.mch

This machine introduces the role hierarchy and includes CoreTools as shown in figure 1.7b. The only modification we have made with respect to the standard is to use a directed acyclic graph RH such that $\succeq = RH^*$, as suggested in [31]. This greatly simplifies the maintenance of the role hierarchy, while preserving the intent of the RBAC standard. Indeed, this allows operation **DeleteInheritance** to cancel the changes made with **AddInheritance**. This cannot always be done with the model used in the standard. Since the standard updates only $\succeq$ and since $\succ\!\!\succ$ is the smallest relation whose closure equals $\succeq$, this leads to the behaviour presented in Figure 1.8 as pointed out by Li *et. al.* Adding pair $r_2 \mapsto r_3$ also removes pair $r_1 \mapsto r_3$ from $\succ\!\!\succ$, since this pair can be obtained by the transitive closure of $\{r_1 \mapsto r_2, r_2 \mapsto r_3\}$. The invariant $\succeq = (\succ\!\!\succ)^*$ and the condition that $\succ\!\!\succ$ represents the immediate successor relation forces $\succ\!\!\succ$ to be the transitive reduction of $\succeq$. Adding a new pair to $\succeq$ also leads to add this pair to $\succ\!\!\succ$ since it was not in $\succeq$. Thus, some of the pairs of $\succ\!\!\succ$ in the before

MACHINE
CoreTools
SETS
USERS ;
ROLES ;
SESSIONS ;
ACTIONS ;
RESOURCES
VARIABLES
Users,
Roles,
Sessions,
Actions,
Resources,
Permissions,
PA,
UA,
User_sessions,
Session_roles,
INVARIANT
$Users \subseteq USERS \wedge$
$Roles \subseteq ROLES \wedge$
$Sessions \subseteq SESSIONS \wedge$
$Actions \subseteq ACTIONS \wedge$
$Resources \subseteq RESOURCES \wedge$
$Permissions \subseteq Actions \times Resources \wedge$
$User\_sessions \subseteq Users \times Sessions \wedge$
$PA \subseteq Permissions \times Roles \wedge$
$UA \subseteq Users \times Roles \wedge$
$Session\_roles \subseteq Sessions \times Roles$

(a) Static part of CoreTools.mch

MACHINE
Hierarchical

INCLUDES
CoreTools

PROMOTES
AddUser,
DeleteUser,
AddRole,
CreateSession,
DeleteSession,
AssignUser,
DeassignUser,
GrantPermission,
RevokePermission,
DropActiveRole,
CheckAccess

VARIABLES
RH

INVARIANT
$RH \subseteq Roles \times Roles$

INITIALISATION
$RH := \emptyset$

(b) Static part of Hierarchical.mch

Figure 1.7 – Static part of CoreTools.mch and Hierarchical.mch

state might be removed in the new state when they no longer represent the immediate successor and they can be retrieved in the transitive closure by a combination with the added pair. Cancelling the addition does not return these deleted pairs.

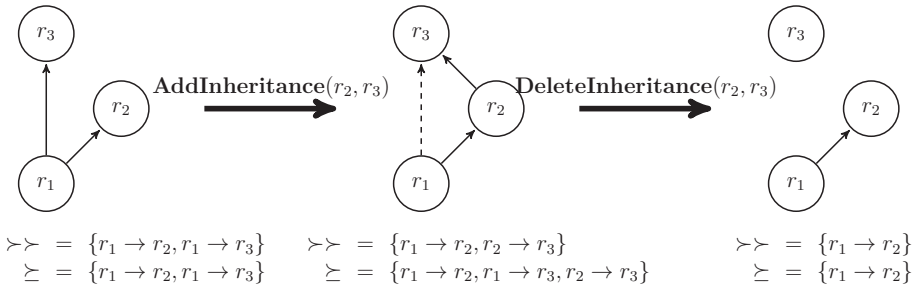


Figure 1.8 – Adding and deleting the same role following the standard

The administrative function **AddActiveRole** does not have the same precondition in Core RBAC and in Hierarchical RBAC. Thus, we have defined an auxiliary operation **AuxAddActiveRoleRH** in CoreTools.mch which has a weaker precondition than **AddActiveRole**. This auxiliary operation is called by operation **AddActiveRoleRH** in Hierarchical.mch with the additional preconditions to deal with the inheritance hierarchy *RH* instead of looking solely at *UA*. Operation **AddActiveRoleHC** adds the precondition to check dynamic separation of duty. These operations are illustrated in Figures 1.9 and 1.10. Note that in B, when an operation calls an operation of an included machine, one must prove that the precondition of the called operation is satisfied, which forces the specifier to essentially repeat the precondition of the called operation into the calling operation. This proof obligation guarantees that an operation is never called outside of its precondition.

1.3.4 Constrained.mch

This machine introduces the static and dynamic constraints. It includes CoreTools.mch and adds invariants to check compliance to each constraints. A simplification of the constraints has been made. In the standard, each constraint has an identifier which permits to retrieve the role set and the cardinality associated with the constraint. In our model, the role set is used as an identifier: if there are two

AddActiveRole(*user*,*sess*,*role*) =
PRE
user ∈ *USERS* ∧ *user* ∈ *Users* ∧
sess ∈ *SESSIONS* ∧ *sess* ∈ *Sessions* ∧
role ∈ *ROLES* ∧ *role* ∈ *Roles* ∧
user ↦ *sess* ∈ *User_sessions* ∧
sess ↦ *role* ∉ *Session_roles* ∧
user ↦ *role* ∈ *UA*
THEN
Session_roles := *Session_roles* ∪ {*sess* ↦ *role*}
END;

AuxAddActiveRole(*user*,*sess*,*role*) =
PRE
. . . *parameter typing conditions* . . .
user ↦ *sess* ∈ *User_sessions* ∧
sess ↦ *role* ∉ *Session_roles* ∧
THEN
Session_roles := *Session_roles* ∪ {*sess* ↦ *role*}
END;

Figure 1.9 – Administrative function **AddActiveRole** specified in CoreTools.mch, Hierarchical.mch and HierarchyConst.mch (1/2)

AddActiveRoleRH (*user*,*sess*,*role*) =
PRE
... parameter typing conditions ...
$user \mapsto sess \in User\_sessions \wedge$
$sess \mapsto role \notin Session\_roles \wedge$
$\exists\, role2.(role2 \in Roles \wedge user \mapsto role2 \in UA \wedge (role2 \mapsto role \in \mathsf{closure1}(RH) \vee role2 = role))$
THEN
AuxAddActiveRole(*user*,*sess*,*role*)
END;

AddActiveRoleHC(*user*,*sess*,*role*) =
PRE
... parameter typing conditions ...
$user \mapsto sess \in User\_sessions \wedge$
$sess \mapsto role \notin Session\_roles \wedge$
$\exists\, role2.(role2 \in Roles \wedge user \mapsto role2 \in UA \wedge (role2 \mapsto role \in \mathsf{closure1}(RH) \vee role2 = role))$
$\forall\, dsd.(dsd \in Dsd \implies \mathbf{card}((Session\_roles[\{sess\}] \cup \{role\}) \cap dsd) < dsd\_card(dsd))$
THEN
AddActiveRoleRH(*user*,*sess*,*role*)
END;

Figure 1.10 – Administrative function **AddActiveRole** specified in CoreTools.mch, Hierarchical.mch and HierarchyConst.mch (2/2)

constraints with the same role set, one of them is redundant. In fact, the constraint with the higher cardinality is redundant since, when this constraint is violated, the constraint with the lower cardinality also is violated. Thus, we chose to identify each constraint by its role set which permits the removal of redundant constraints. The redundancy check must be also done when adding or deleting a role member from a constraint. The invariants stating that *ssd_card* and *dsd_card* are total functions (see Figure 1.11) ensure that there is no redundancy among the constraints.

1.3.5 HierarchyConst.mch

At first glance, one could think that the combination of Hierarchical RBAC and Constrained RBAC is a simple inclusion of both machines, but this is inappropriate since the preconditions of operations of one machine are insufficient to preserve the invariant of the other machine, or are too strong for the other machine. Moreover, the invariants on separation of duty constraints cannot be reused from Constrained.mch, because they involve the role hierarchy when combined with hierarchical RBAC. Thus, there is no point in defining a machine for SSD variables, including it into Constrained RBAC and then into the combination of Hierarchical RBAC and Constrained RBAC. The easiest solution was to include Hierarchical.mch in HierarchyConst.mch, to promote operations which required no changes, and to define new invariants and operations with appropriate preconditions for the remaining operations.

1.3.6 Proving acyclicity of the role hierarchy

We tried to prove an invariant which was not in the standard, the acyclicity of the role hierarchy, expressed as $RH^+ \cap \mathsf{id}(Roles) = \emptyset$. This turns out to be surprisingly non trivial. Since Atelier B has no rule about the transitive closure, it was impossible to prove it without adding new rules in the prover. We had to add the following rules, which can be manually proved using laws found in standard relational algebra textbooks like [47]; we have also checked them using Alloy [26]. Let S be some set,

MACHINE
Constrained

INCLUDES
CoreTools

PROMOTES
AddUser,
DeleteUser,
AddRole,
CreateSession,
DeleteSession,
DeassignUser,
GrantPermission,
RevokePermission,
DropActiveRole,
CheckAccess

VARIABLES
ssd_card,
dsd_card,
Ssd,
Dsd

INVARIANT

$Ssd \subseteq \mathcal{P}\ (Roles) \wedge$
$Dsd \subseteq \mathcal{P}\ (Roles) \wedge$
$ssd\_card \in Ssd \rightarrow \mathbf{NAT} \wedge$
$dsd\_card \in Dsd \rightarrow \mathbf{NAT} \wedge$
$\forall\ ssd.(ssd \in Ssd \implies ssd\_card(ssd) \geq 2 \wedge ssd\_card(ssd) \leq \mathbf{card}(ssd)) \wedge$
$\forall\ dsd.(dsd \in Dsd \implies dsd\_card(dsd) \geq 2 \wedge dsd\_card(dsd) \leq \mathbf{card}(dsd)) \wedge$
$\forall\ user.(user \in Users \implies \forall\ ssd.(ssd \in Ssd \implies \mathbf{card}(UA[\{user\}] \cap ssd) < ssd\_card(ssd))) \wedge$
$\forall\ sess.(sess \in Sessions \implies \forall\ dsd.(dsd \in Dsd \implies \mathbf{card}(Session\_roles[\{sess\}] \cap dsd) < dsd\_card(dsd)))$

Figure 1.11 – Static part of Constrained.mch

$X, X_1, X_2, X_3, X_4 \subseteq S$, let $x, y, z \in S$ and let $r, s \subseteq S \times S$.

$$(r \cup s)^+ = r^+ \cup ((r^*; s)^+; r^*) \tag{1.1}$$

$$X_1 \cap X_4 = X_2 \cap X_3 = \emptyset \;\Rightarrow\; (X_1 \times X_2 \cup X_3 \times X_4)^+ = (X_1 \times X_2)^+ \cup (X_3 \times X_4)^+ \tag{1.2}$$

$$(X \times \{x\})^+ = X \times \{x\} \tag{1.3}$$

$$r \subseteq s \;\Rightarrow\; r^+ \subseteq s^+ \tag{1.4}$$

$$\mathsf{dom}(r^+) \subseteq \mathsf{dom}(r) \tag{1.5}$$

$$\mathsf{ran}(r^+) \subseteq \mathsf{ran}(r) \tag{1.6}$$

$$\{x \mapsto y, y \mapsto z\} \subseteq r^+ \Rightarrow \{x \mapsto z\} \in r^+ \tag{1.7}$$

Since these rules can't be proved using Atelier B, we also decided to use relation algebra [47] and Kleene algebra [28], of which binary relations are models, to formally prove preservation of acyclicity when adding a new pair in an acyclic relation. The proof is provided below. In addition, we have carried the same task using the automated theorem prover Prover9 [34]. This can be found along the B specification [12].

For the sake of concision, we adopt some of the conventions of abstract relation algebra. For instance, we write PQ instead of $P;Q$ for relational composition. Let $L = \mathit{Roles} \times \mathit{Roles}$ denote the universal relation, $\overline{P} = L - P$, $I = \mathsf{id}(\mathit{Roles})$, $A \subseteq \mathit{Roles} \times \mathit{Roles}$ denote the role hierarchy RH, $B = \{x \mapsto y\}$ where $x \neq y$ and $\{x, y\} \subseteq \mathit{Roles}$, be a new pair to add to A.

Theorem 1 *Assuming*

$$A^+ \cap I = \emptyset \quad (1.8) \qquad B^{-1} \cap A^+ = \emptyset \quad (1.9)$$

$$BB = \emptyset \quad (1.10) \qquad BLB \subseteq B \quad (1.11)$$

then

$$(A \cup B)^+ \cap I = \emptyset.$$

Condition (1.8) states that the role hierarchy is acyclic. Condition (1.11) states that B is a relation of the form $X \times Y$ for some (possibly empty) subsets X and Y of *Roles*. Because of condition (1.10), X and Y are disjoint. Conditions (1.10) and (1.11) are obviously satisfied when $X = \{x\}$, $Y = \{y\}$ and $x \neq y$. Finally, condition (1.9)

states that there is no path in the role hierarchy from the codomain Y to the domain X of B, in order to avoid the creation of a cycle. The conclusion of the theorem is that adding a new pair to RH returns an acyclic role hierarchy.

To prove this theorem, we use the following laws of [28, 47], which include the laws of Boolean algebra, since a relation algebra is a Boolean algebra.

$$PQ \subseteq R \;\Leftrightarrow\; P^{-1}\overline{R} \subseteq \overline{Q} \;\Leftrightarrow\; \overline{R}Q^{-1} \subseteq \overline{P} \tag{1.12}$$

$$P \cup RQ \subseteq R \;\Rightarrow\; PQ^* \subseteq R \tag{1.13}$$

We also need the following two lemmas, which follow from (1.8) to (1.11).

$$A^*BA^* \cap I = \emptyset \tag{1.14}$$

$$(A \cup B)^+ \subseteq A^+ \cup A^*BA^* \tag{1.15}$$

Proof of (1.14)

$A^*BA^* \cap I = \emptyset$

$\Leftrightarrow$ $\langle\ P \cap Q = \emptyset \Leftrightarrow P \subseteq \overline{Q}\ \rangle$

$A^*BA^* \subseteq \overline{I}$

$\Leftrightarrow$ $\langle\ (1.12),\ (PQ)^{-1} = Q^{-1}P^{-1}\ \rangle$

$B^{-1}A^{*-1}I \subseteq \overline{A^*}$

$\Leftrightarrow$ $\langle\ PI = P,\ (1.12),\ \overline{\overline{P}} = P\ \rangle$

$A^*A^* \subseteq \overline{B^{-1}}$

$\Leftrightarrow$ $\langle\ P^*P^* = P^*,\ P \cap Q = \emptyset \Leftrightarrow P \subseteq \overline{Q}\ \rangle$

$A^* \cap B^{-1} = \emptyset$

$\Leftrightarrow$ $\langle\ P^* = I \cup P^+\ \rangle$

$(I \cup A^+) \cap B^{-1} = \emptyset$

$\Leftrightarrow$ $\langle$ Distributivity, $I \cap B^{-1} = I \cap B = (I \cap B)(I \cap B) \subseteq BB = \emptyset$ by (1.10), and (1.9) $\rangle$

true

Proof of (1.15)

$$
\begin{array}{ll}
& (A \cup B)^+ \subseteq A^+ \cup A^*BA^* \\
\Leftrightarrow & (A \cup B)(A \cup B)^* \subseteq A^+ \cup A^*BA^* \\
\Leftarrow & \quad \langle\ (1.13)\ \rangle \\
& A \cup B \cup (A^+ \cup A^*BA^*)(A \cup B) \subseteq A^+ \cup A^*BA^* \\
\Leftrightarrow & \quad \langle\ \text{Distributivity, } A \subseteq A^+,\ B \subseteq A^*BA^*,\ A^+A \subseteq A^+, \\
& \quad\ A^+B \subseteq A^*BA^*, \\
& \quad\ A^*BA^*A \subseteq A^*BA^*\ \rangle \\
& A^*BA^*B \subseteq A^+ \cup A^*BA^* \\
\Leftarrow & BA^*B \subseteq \emptyset \\
\Leftrightarrow & \quad \langle\ BA^*B = BB \cup BA^+B\ \rangle \\
& BB \subseteq \emptyset\ \wedge\ BA^+B \subseteq \emptyset \\
\Leftarrow & \quad \langle\ (1.10) \text{ and since } A^+ \subseteq \overline{B^{-1}} \text{ by } (1.9)\ \rangle \\
& B\overline{B^{-1}}B \subseteq \emptyset \\
\Leftrightarrow & \quad \langle\ (1.12),\ (PQ)^{-1} = Q^{-1}P^{-1},\ \overline{P}^{-1} = \overline{P^{-1}},\ P^{-1-1} = P\ \rangle \\
& LB^{-1}\overline{B} \subseteq \overline{B} \\
\Leftrightarrow & \quad \langle\ (1.12),\ (PQ)^{-1} = Q^{-1}P^{-1},\ L^{-1} = L\ \rangle \\
& BLB \subseteq B \\
\Leftarrow & \quad \langle\ (1.11)\ \rangle \\
& \textbf{true}
\end{array}
$$

Proof of Theorem 1

$$
\begin{array}{ll}
 & (A \cup B)^+ \cap I = \emptyset \\
\Leftarrow & \qquad \langle\ (1.15)\ \rangle \\
 & (A^+ \cup A^* B A^*) \cap I = \emptyset \\
\Leftrightarrow & \qquad \langle\ \text{Distributivity}\ \rangle \\
 & A^+ \cap I = \emptyset\ \wedge\ A^* B A^* \cap I = \emptyset \\
\Leftrightarrow & \qquad \langle\ (1.8), (1.14)\ \rangle \\
 & \textbf{true}
\end{array}
$$

1.4 Overview of the formal validation approach

Table 1.2 summarises the types of errors we have found in the RBAC standard. The process we have followed to discover these errors was the following. We carefully reviewed the text of the standard, in order to develop a good understanding of the specification. In parallel, we incrementally built the specification, piece by piece, animating it and model checking it using ProB [11, 29]. Several iterations were necessary in order to get the structure and the behaviour right. When no more errors were found using ProB, we used Atelier B [5] to discharge the proof obligations of the machines.

Ambiguities, typos, unused symbols, undeclared symbols, bad definitions were discovered both by reading the specification and trying to translate it into B. It is surprising how the objective of formalizing a text changes the perspective of a reader. In such a context, the reader pays careful attention to every possible detail, especially when he has no *a priori* knowledge of what the document should say. Simple errors which seem innocuous to the expert when noticed are very disturbing for the reader with no *a priori* knowledge. Such a reader does not know which elements are typos and assumes first that he has missed something somewhere, or that a subtle distinction he has not grasped yet must exist.

Constructing a formal specification in a language like B forces the reader to question each element of the standard and find out where it fits in the formal specification. The reader questions any redundant concept, because as a specifier, he knows that any

redundancy will induce additional work when proving the correctness of the B specification. Moreover, B imposes a clear structure (machine parameters, sets, constants, variables, invariants, operations, preconditions, state modifications) with various validation mechanisms like syntax checking (and all the consistency checking that is included with syntax checking), type checking, invariant preservation and inappropriate behaviour found by specification animation. An error quickly breaks one of these validation steps. These validation steps do not exist when one simply writes a natural language text and unexecutable mathematics. We believe this explains in part why so many errors were left in by so many of the readers of the standard. And it also shows the benefits of using a comprehensive method like B with numerous safeguards.

Error type	**Nbr**	**Description**
Ambiguity	1	A term is used with two different meanings
Logical error	3	Incorrect mathematical expressions to specify a concept
Improper terminology	1	Standard mathematical meaning is improperly used
Bad definition	2	A definition is ill-structured
Invalid precondition	1	An operation's precondition is insufficient to preserve an invariant
Formal notation	4	The formal notation is incorrectly used and has no meaning
Type error	1	An operator is used with expressions of incompatible types
Typo	3	
Undeclared symbol	1	
Unused symbol	5	
Total	22	

Table 1.2 – Summary of error types found in the standard

1.5 Related work

Our objective in this paper is to analyse the text and the mathematical description of the ANSI RBAC standard, to validate its consistency and clarity. We do not address

the modelling choices made by the authors of the standard. We rather want to argue about how the standard was written and how standards should be written in order to avoid ambiguities and inconsistencies. The reader is referred to [17] for a concise appraisal of the concepts of RBAC and its industrial adoption issues. Numerous other papers have proposed critiques and extensions of the RBAC model itself. This review of this literature is outside the scope of this paper.

Despite the fact that RBAC is widely spread and well known and that the ANSI standard contains several errors and inconsistencies, there are relatively few papers addressing its consistency and clarity. Li *et al.* [32] pointed out 7 errors out of the 22 we have found (*i.e.*, P1, P4, P12, P13, P15, P17, P21) and suggested other features for RBAC. For instance, they disagree with the importance of having sessions with multiple role activation; their proposed model differs from RBAC on that aspect. They also pointed out the need of maintaining explicitly added dominance relationships in the role hierarchy because it has "significant weakness when one considers updates to [it]", which we have adopted. They discuss three interpretations for the role hierarchy, arguing that the standard leaves room for interpretation. We take a different viewpoint by sticking to the mathematical text of the standard, which leads to a single interpretation, in our opinion. This single interpretation becomes clear when considering the Core RBAC administrative functions that Hierarchical RBAC modifies: only **CreateSession** and **AddActiveRole** are redefined to take the role hierarchy into account. These operations must check in their precondition that the role to activate is either a role assigned to the user or a role inherited by one of the roles assigned to the user. Thus, role inheritance only affects role activation, and even if r_1 inherits from r_2, they must still be activated separately. None of the three interpretations proposed by Li *et al.* is coherent with the mathematical specification provided in the standard. Among others, Li *et al.* missed important problems with the **DeleteRole** administrative function as shown in P22 where the role hierarchy and separation of duty constraints are not updated during a role deletion. These errors are discovered by checking the preservation of invariants. Power *et al.* based their models[41, 40] on [32], thus they did not find them either, but proposed an approach to normalise and compare RBAC systems. Hu *et al.* [19] managed to find problems with the **DeleteRole** function by modelling the standard in Alloy [26] and then specifying

functional properties to test: functional properties are generated from a UML model of RBAC with OCL constraints which are then translated and given to Alloy. Their test has been made only on a portion of the RBAC model. Our approach includes the complete verification and proof of the entire RBAC model. We also enhanced it by providing necessary preconditions on **DeleteRole** and slightly modifying the way the constraints are specified in order to guarantee that each constraint is unique and non-redundant. Issues like constraint management upon role deletion (see P22) have not been discussed by [41, 32, 40]. All of these stem from checking invariants in our model. In addition, animation of our models with tools like ProB allows us to test policies: for instance, given an initial situation and policies, it is possible to check whether a user can get all the available permissions or if a user can get redundant permissions.

1.6 Conclusion

RBAC is a widely adopted access-control model and is also widely used in commercial products, such as database management systems or enterprise management systems. The RBAC model has been published as the NIST RBAC model [46] and adopted as an ANSI/INCITS standard in 2004, which has been revised in 2012. In this paper, we have pointed out a number of technical errors identified using formal methods, by modelling in a B the RBAC specification, then animating it and proving it. Using mathematics without a methodological framework and a supporting tool set is bound to open the door to errors. The B method seems to be particularly appropriate for specifying standards of dynamic systems like RBAC. The fact that B makes a clear distinction between the specification of operations and the state properties that these operations must satisfy (*i.e.*, invariant preservation) proved to be very useful in validating the RBAC standard. The example of role deletion (problem P22) is a nice illustration of this. This case study also shows that human-based reviews are insufficient to detect errors in a standard. Mechanical verification is essential; syntax checking, type checking, animation, model checking and theorem proving are complementary in finding errors in a specification. This exercise of specifying RBAC in B shows that B has all the necessary features to specify and validate a system

like RBAC. However, it also shows that specifying different versions of a system, like the various components of RBAC and their combinations, and factoring common behaviours is not as straightforward as it seems to be. This problem is similar to engineering product line architectures [38]. Formal methods like B may benefit from the results obtained in this field to streamline specification engineering, although formal correctness surely imposes strong constraints on reuse and sharing mechanisms.

Acknowledgements The authors would like to thank the anonymous referees for their valuable comments which helped improve this paper. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by the Agence de la santé et des services sociaux de l'Estrie in Québec.

Chapitre 2

SGAC : une méthode de contrôle d'accès centrée sur le patient

Résumé

Cet article présente la méthode *SGAC* : une méthode de contrôle d'accès qui permet au patient d'exprimer son consentement quant aux accès à son dossier médical.

Cette méthode contient le modèle conçu avec l'aide du *CHUS*, ébauché durant ma maîtrise, sa formalisation et ainsi que son implémentation sous forme d'outil. L'article décrit donc les besoins auxquels *SGAC* répond, son fonctionnement avec divers exemples illustrant la méthode de résolution de conflits de *SGAC*. L'article présente également la formalisation complète de *SGAC* qui permet de vérifier diverses propriétés telles que la détection de données inaccessibles, la vérification d'un droit d'accès ou encore la détermination des contextes qui font que la requête soit acceptée. Enfin, les performances de l'implémentation de *SGAC* sont comparées à celles de *XACML*.

Commentaires

Ce travail s'inscrit dans la continuité de ma thèse : après avoir formalisé *RBAC*, nous avons formalisé *SGAC* afin de voir quelles propriétés pouvaient être vérifiées.

Il est possible avec notre formalisation de SGAC de vérifier si une personne a accès ou non à une ressource, quelles sont les données cachées ou encore détecter des règles inefficaces, voire redondantes. Cet article a été accepté à la conférence RCIS 2016 ayant eu lieu à Grenoble, et a été récompensé du Best Paper Award. Ma contribution sur cet article est la suivante :

- formalisation de *SGAC* ;
- définition de méthodes de vérification des propriétés
  - d'accès ;
  - de détection de données cachées ;
  - de contextes validant une requête ;
  - de détection de règles redondantes.
- comparaison de *SGAC* avec les autres modèles de contrôle d'accès.

SGAC: A Patient-Centered Access Control Method

Nghi Huynh
Université de Sherbrooke, Québec, Canada
Université Paris Est-Créteil, Val de Marne, France

Marc Frappier
Université de Sherbrooke, Québec, Canada

Herman Pooda
Université de Sherbrooke, Québec, Canada

Amel Mammar
SAMOVAR, Télécom SudParis, CNRS, Université Paris-Saclay, EVRY, France

Régine Laleau
Université Paris Est-Créteil, Val de Marne, France

Keywords: healthcare, access control method, consent management, formal model, verification.

Abstract

This paper presents SGAC(*Solution de Gestion Automatisée du Consentement, automatised 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). The development of this model has been achieved in the scope of a project with the Sherbrooke University Hospital, and thus has been adapted to take into account laws and regulations applicable in Québec and Canada, as they set bounds to patient wishes: under strictly defined contexts, patient consent can be overridden to protect his/her life. Moreover, since patient wishes and laws can be in conflict, SGAC provides a mechanism to

address this problem. Besides, laws do not cover all cases where consent should be overridden to ensure patient safety. To this end, we define a formal model of SGAC which allows for property verification, making it possible to detect these cases. A performance comparison with XACML (WSO2/Balana) is presented and demonstrates the superior performances of SGAC.

2.1 Introduction

Before being electronic, patient data were stored physically in each health centre. In Québec, access to health records is managed by specially trained staff, the archivists, who are responsible for applying the laws and regulations to access request. Laws set a frame within which patients can manage access to their health record, as long as they are not endangering themselves. Access control in healthcare knows two major contentious concerns: patient data confidentiality and patient safety. The former is about non-disclosure of data their owner would judge confidential; the latter is about the rules not being too restrictive and a burden for the health worker when requesting all necessary data to provide suitable care to the patient. Having patients specifying access rules to their records (thus expressing their consent) is a way to address the first concern. To address the second concern, laws and regulations set a frame that allows accesses to patient data without consent under strictly defined contexts. The problems this approach rises are multiple: laws generally set frame only for exceptional cases and not for everyday care, thus it does not always allow to override patient consent in order to give him/her suitable care, for instance when the patient is hiding important data like medicinal allergies. Furthermore, conflicts may arise between hospital rules, which define health workers regular access, patient rules, and break-the-glass rules which must provide full access to the physicians 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 rules from different sources. This method allows formal verification in order to detect cases where suitable care cannot be given.

The rest of this paper is structured as follows. Section 2.2 provides requirements for access control and consent management used at the Sherbrooke University Hospital (CHUS) and scenarios illustrating expected behaviours. Section 2.3 introduces SGAC. We illustrate how SGAC behaves in Section 2.4. Section 2.5 provides a complete formal model of SGAC. Section 2.6 compares our findings with similar work on access control in healthcare. A performance comparison with XACML [43] is given in

section 2.7. We conclude this paper with an appraisal of our work in Section 2.8.

2.2 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 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 at 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 at 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 to restrict access to his EHR when he is hospitalised; when he is not hospitalised, Sam wants to deny access to his EHR to all hospital staff.
Requirements: Req. 9.

2.3 SGAC Data Structures, Rules and Requests

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

2.3.1 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 $A \times B$.

— $\mathsf{id}(A) = \{x \mapsto x \mid x \in A\}$ denotes the identity relation on A, i.e. the relation that associates each element of A to itself.

— $A \leftrightarrow B = \mathcal{P}(A \times B)$ denotes the set of relations between A and B.

— $\mathsf{dom}(R) = \{x \in A \mid \exists y \in B \bullet x \mapsto y \in R\}$ 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 partial function such that $\mathsf{dom}(f) = A$.

— $R_1 \fatsemi R_2 = \{x \mapsto z \mid \exists y \in B \bullet x \mapsto y \in R_1 \wedge y \mapsto z \in R_2\}$ 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* V 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 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 $sink(G) = G.V - \mathsf{dom}(G.E)$.

2.3.2 Using graphs

In SGAC, two directed acyclic graphs 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.

Fig. 2.1 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 Fig. 2.1, if a permission is given to the *General Practice* department then this permission is inherited by *GP Physicist* and *GP Nurse*, *Bob* and *Alice*.

Fig. 3.1b 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. Documents are instances of sinks of the resource type graph. A document has attributes which can be given as parameters to non-sink vertices. For instance, a certain AIDS screening test can have many attributes such has: the patient it is related to, the visit when it was ordered, the ID of the screening test etc... There is a functional dependency between the document type identifier and the other attributes, making the key *document identifier* sufficient to retrieve a document, and all its attributes.

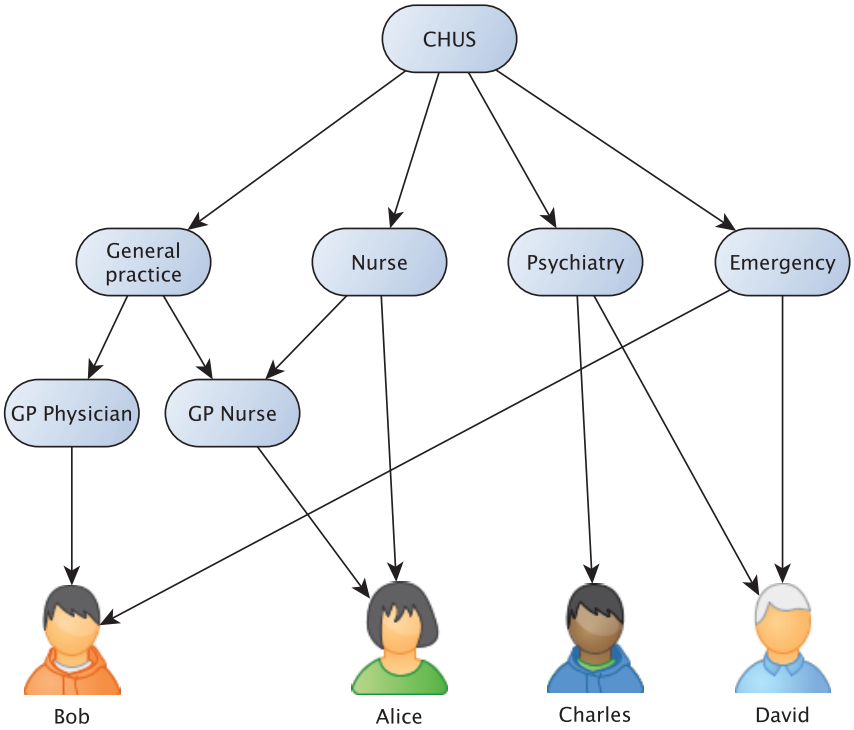


Figure 2.1 – Subject graph example

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

Fig. 2.2b illustrates the resource type graph being instantiated for the document *Blood 123* of the patient *Simon* during his visit no. *2*.

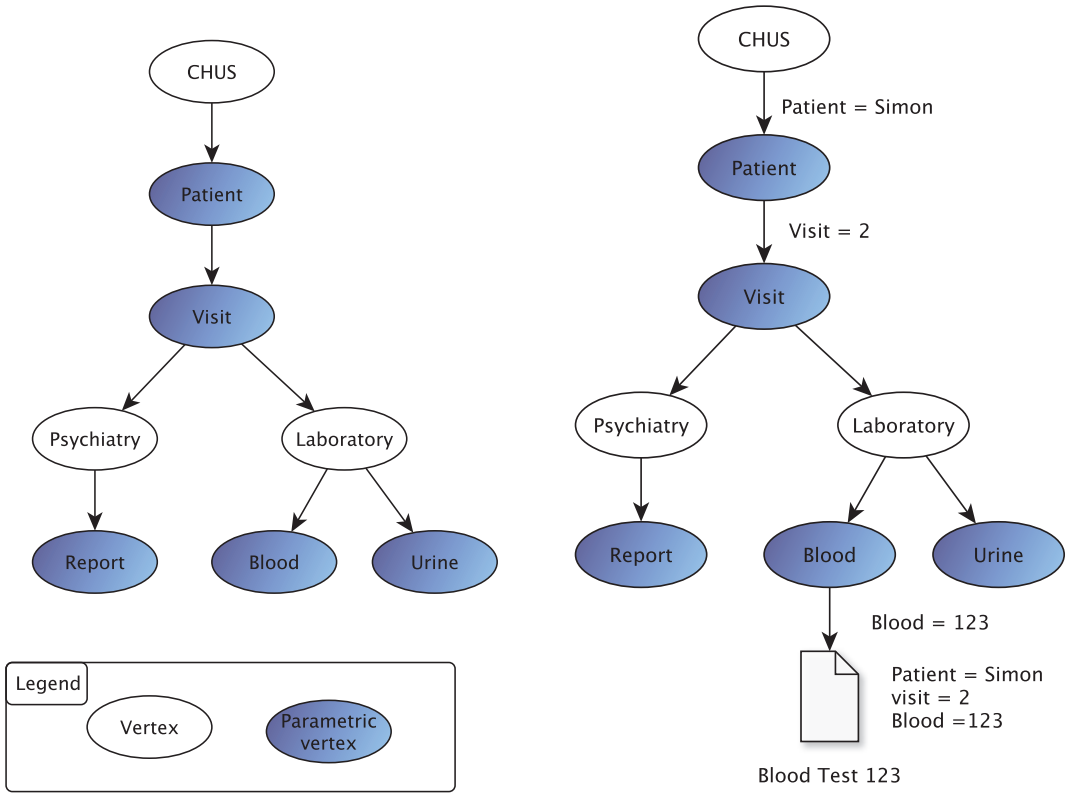
These two graphs define the basis on which rules and requests are built.

2.3.3 Rule and request specification

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

— a *subject*: a person or a group of people to control;

— a *resource*: the data to be protected;



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

Figure 2.2 – Resource graph example

— an *action*: the operation the *subject* wants to do on the *resource*;

— 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 rest of the paper, we describe rule conditions in natural language.

A request is the demand the *subject* issues in order to execute an *action* on a *resource*. It has then 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*.

2.3.4 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 prohibitions have precedence over permissions.

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 2.5 shall clarify the third clause in some subtle cases, to avoid any ambiguity in its interpretation.

Rule	Resource	Subject	Pri.	Mod.	Cond.
r_1	Psychiatry (patient = Anna)	CHUS	2	−	*TRUE*

Table 2.1 – Scenario 1 rule (for the action *read*)

This conflict resolution method is absolutely autonomous and does not require the intervention of an external actor. Section 2.4 illustrates the conflict resolution technique with three examples.

2.4 Examples

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

2.4.1 Example 1: basics

Let's model scenario 1. The resource type graph must be instantiated with the parameters defining Anna's data. Modelling Anna's rule consists in prohibiting access to the documents descending from the vertex *Psychiatry* in the resource graph. The vertex *Patient* gets the unique identifier of the patient Anna. The vertex *CHUS* in the subject graph (Fig. 2.1) 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 2.1.

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 action matches, the condition is verified, and the parameter fits. 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.

2.4.2 Example 2: let's get started

In this example, the rule base is as follow:

— the laws and regulations allow emergency physicians to access (read and write) the data of any patient who is in a life-threatening situation;

— the hospital allows general physicians to read and write data for any patient under their care;

— the hospital allows nurses to read 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 2.2 – Example 2 rules (for the action *read*)

This can be represented by the rule base presented in Table 2.2. 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 translates the fact that any physician in the *Emergency* department can access a record if its owner's life is threatened: an authorisation given to the vertex *Emergency* to read 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 translates the fact that a physician is allowed to read 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 read 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 translates the fact that a nurse is allowed to read 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 Fig. 2.3.

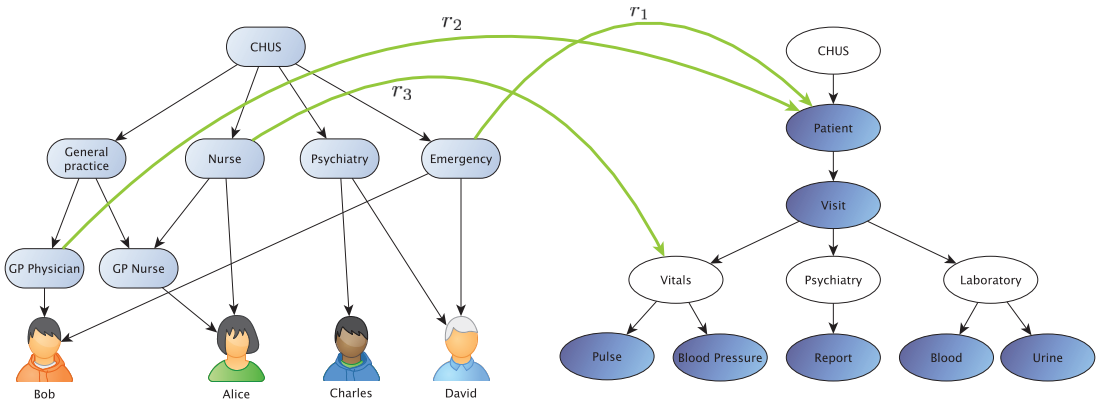


Figure 2.3 – Example 2 graphs with rules

Now let's say that patient Anna is treated for some light mental disorder by Charles, a psychiatrist. 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 2.3.

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

Table 2.3 – Example 2: Access of the CHUS personnel to Anna's Record, wrt Fig. 2.3

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 2.4.

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

Table 2.4 – Example 2: Access of the CHUS personnel to Sam's Record, wrt Fig. 2.3

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.

2.4.3 Example 3: adding consent

In this example, we take the same initial rule base (Table 2.2), 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 affected by r_5 can access Anna's vitals. The accesses at this point are presented in Table 2.5.

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 can't access Anna's

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

Table 2.5 – 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	Bob	2	+	*TRUE*

Table 2.6 – Example 3 rules (for the action *read*)

data, unless there in an emergency context where Anna's life is threatened. The final rule base of this example is presented in Table 2.6 and with the graphs in Fig. 2.4.

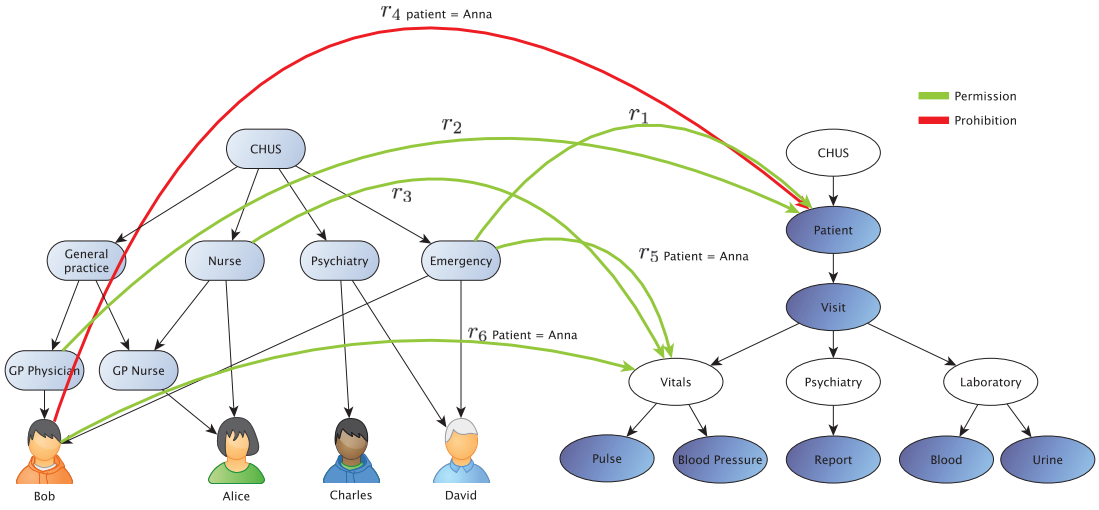


Figure 2.4 – Example 3 graphs with rules

2.5 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.

2.5.1 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 $S = (V, E, Z)$ with:

— (V, E) is a DAG;

— $V \subseteq SUBJECT$ is the set of the subjects;

— $Z = V \cap PERSON$ represents the persons, and elements of $V - Z$ are entities which represent groups of persons;

— $Z \subseteq sink(S)$ since a person is a sink of S.

$S.E$ represents the inheritance relation: recall that a rule on a subject s is inherited by all the successors of s in $S.E^+$ 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.

2.5.2 Resource Graph

As mentioned in Section 2.3.2, 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) as follows: $R = (G, K)$ where G is a DAG and $H = (T, D, U, W)$ denotes constraints linking the DAG G to the documents. More precisely, $G = (V, E, P)$ where V is the set of the vertices, E the edges and P the set of the parametric vertices. We denote by $DOCUMENT$ the set of all documents.

— $G.P$ denotes parametric vertices that are called *parametric groups* and the elements of $G.V - G.P$ are called *groups*. Parametric groups introduce exactly

one parameter, like *patient*, *visit* etc...

$$G.P \subseteq G.V$$

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

$$sink(R) = sink(G.V, G.E) \subseteq G.P;$$

— D denotes the set of all the documents, and U the type of a document. Each document has exactly one type, so

$$U \in D \rightarrow sink(R);$$

$$D \subseteq DOCUMENT$$

— W denotes a valuation of parameters of the documents. The parameter valuation W 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

$$W \in D \rightarrow (P \nrightarrow T);$$

— W is defined for each parameter inherited by a document,

$$\forall\, d \in D \bullet \mathsf{dom}(W(d)) = G.E^{-1*}[U(d)] \cap G.P$$

2.5.3 Rule

A rule l is a septuplet which contains:

— a modality *mod*;

— a resource *res* with the valuation *val* of its parameters;

— an action act;

— a subject sub;

— a priority pri;

— a condition con.

We denote by $ACTION$ and $RULE$ the sets of all actions, and of all rules. Since a rule depends on a subject graph and a resource type graph, we introduce the object $Policy$, composed of a subject graph, a resource type graph, and a set of rules. Each rule of the policy targets elements of the graphs of the policy. Formally, we denote by $P = (S, R, L)$ a policy and we have:

— $S = (V, E, P)$ a subject DAG;

— $R = (G, H)$ a resource type PDAG;

— $L \subseteq RULE$ the set of rules of the policy.

We have to link the rules of a policy to the graphs by the following constraints:

— the subject is a person of S:

$$\forall\, l \in L \bullet l.sub \in sink(S)$$

— the action belongs to $ACTION$ and the priority is a positive real:

$$\forall\, l \in L \bullet l.act \in ACTION$$

$$\forall\, l \in L \bullet l.pri \in \mathbb{R}^{+}$$

— $l.res$ is a vertex of $R.G.V$ and $l.val$ is a valuation of parametric groups of $R.G.P$ with adequate values:

$$\forall\, l \in L \bullet l.res \in R.G.V \wedge l.val \subseteq R.G.P \times R.H.T$$

— the only possible modalities are permission, and prohibition;

$$\forall\, l \in L \bullet l.mod \in \{permission, prohibition\}$$

We also introduce the function *documents*:

$$documents(R, v, K) = \{d | d \in R.H.D \wedge R.H.U(d) \in sink(R, v) \wedge K \subseteq R.H.W(d)\}.$$

The function $documents(R, v, K)$ returns all documents reachable from vertex v in PDAG R with document parameter valuation K.

For example, $documents(R, Visit, \{patient \mapsto Simon\})$ denotes the set of all documents issued during any visit of patient *Simon*. The blood test of the example of Fig. 2.2b denoted by *bt* has the following associated parameters:

$$\begin{aligned} & R.H.W(bt) \\ = \; & \{patient \mapsto Simon, visit \mapsto 2, id \mapsto 123\} \\ \supseteq \; & \{patient \mapsto Simon\} \\ = \; & K, \end{aligned}$$

thus $bt \in documents(R.H, Visit, \{patient \mapsto Simon\})$.

2.5.4 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 only targets one document at a time.

2.5.5 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.

Rule extraction

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

$$Rules(P, q) = \{l \mid l \in P.L \wedge sub \wedge act\_con \wedge doc\}$$

where:

— $sub := q.sub \in (P.S).E^*[\{l.sub\}]$;

— $act\_con := (l.act = q.act) \wedge eval_f(l.con)$;

— $doc :=$

$$\begin{array}{ll} & (P.R.U)(q.doc) \in P.S.E^*[l.res] \\ \wedge & l.val \subseteq P.R.W(q.doc) \end{array}$$

— the function $eval_f(f)$ evaluates the formula f, taking into account values of variables occurring in f.

Then for a policy P and a request q, $Rules(P, q)$ designates all rules of $P.L$ of which:

— action corresponds to $q.act$;

— subject is $q.sub$ or an ancestor of $q.sub$;

— condition is evaluated to $TRUE$;

— reachable documents contains $q.doc$.

Rule ordering

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

$$\begin{array}{l} \forall\, x, y \in RULE \bullet \\ \quad x \prec y \\ \Leftrightarrow \\ \quad \begin{array}{ll} & y.pri < x.pri \\ \vee & (x.pri = y.pri \wedge y.sub \in S.E^+[\{x.sub\}]) \end{array} \end{array}$$

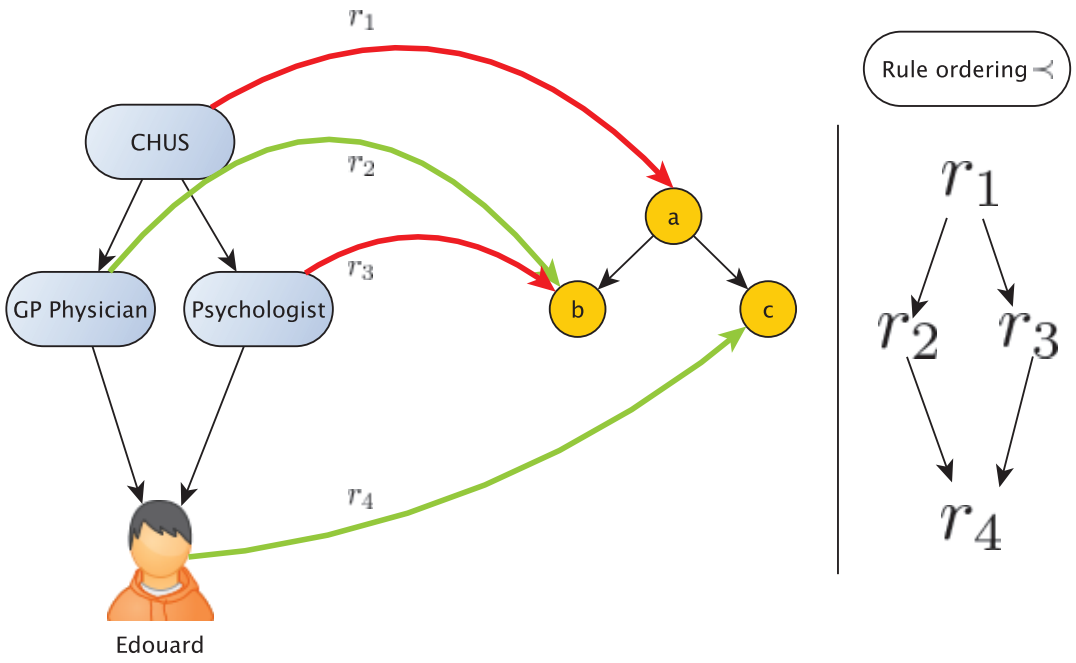


Figure 2.5 – Illustration of the partial order relation $\prec$

This relation $\prec$ captures the fact that precedence is given to the rule with a lower priority 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 Fig. 2.5, 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 can't be compared with $\prec$.

If r_1, r_2, r_3 and r_4 are the only applicable rules then precedence over the other rule 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 $max_{\prec}$. Formally,

$$max_{\prec} = \{x \in Rules(P,q) \mid \nexists y \in Rules(P,q) \bullet x \prec y\}$$

We define $<$ on rules as follows:

$$\begin{array}{l} \forall\, x, y \in Rg.V \bullet \\ \quad x < y \\ \Leftrightarrow \\ \qquad x \prec y \\ \quad \vee \quad (\ x, y \in max_{\prec} \\ \qquad \wedge\, y.mod = prohibition \\ \qquad \wedge\, x.mod \neq y.mod \\ \qquad) \end{array}$$

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. $Rg_{P,q}$ is defined by:

- $Rg_{P,q}.V = Rules(P,q)$;
- $Rg_{P,q}.E$ 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.

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 other. We denote by the function $eval(P,q)$ the evaluation of the request q in the policy P; $eval(P,q)$ returns $TRUE$ if q is approved. In order to determine $eval$:

1. we determine first all applicable rules by calculating $Rules(P,q)$;
2. we create the DAG $Rg_{P,q}$;

3. we verify the following property:

$$Prop(P,q) := sink(Rg_{P,q}) \neq \emptyset \wedge \forall\, l \in sink(Rg_{P,q}) \bullet l.mod = permission.$$

We define *eval(P,q)* as follows:

$$eval(P,q) := eval_f(Prop(P,q))$$

If all sinks of Rg are permissions, then a permission is returned and *eval* returns *TRUE*. If $Rg.V$ is empty (*i.e.*, no rules are applicable), a *prohibition* is returned and *eval* returns *FALSE*.

As noted before, $<$ ensures that all sinks of Rg have the same modality. To see this, let $r_1, r_2 \in sink(Rg)$ with $r_1.mod \neq r_2.mod$. There are two cases:

- $r_1.pri = r_2.pri$: according to the definition of $<$, we have $r_1 < r_2$ or $r_2 < r_1$, which is absurd since $r_1, r_2 \in sink(Rg)$.
- $r_1.pri \neq r_2.pri$ then we have $r_1 < r_2$ or $r_2 < r_1$, which is absurd.

Thus we have the following properties for *eval*:

$$eval(P,q) \Leftrightarrow\ \wedge \exists\, l \in sink(Rg_{P,q}) \bullet l.mod = permission$$

and

$$eval(P,q) \Leftrightarrow Rules(P,q) \neq \emptyset \wedge \nexists x \in max_{\prec} \bullet x.mod = prohibition$$

The first says that if a sink of $Rg_{P,q}$ is a permission, then access is granted. The second says that if there is at least one applicable rule and if there is no prohibition in the maximal elements of $Rules(P,q)$ wrt $\prec$, then access is granted.

2.5.6 Example

Let's say that the patient Anna is to be hospitalised in the CHUS. She did work there when she was 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

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 2.7 – Rule Base Example, for the action *read*

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 Tab. 2.7 as $P_1.L$. We use Fig. 2.1 as the subject graph $P_1.S$ and Fig. 2.2a as the resource 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 $P_1.R.H$, which we simply denote by H in the sequel:

— $H.D = \{bt_1, bt_2, pr_1\}$;

— $H.U = \{bt_1 \mapsto Blood, bt_2 \mapsto Blood, pr_1 \mapsto Report\}$;

— $H.W =$
$\{bt_1 \mapsto \{Patient \mapsto Anna, Visit \mapsto 1, Blood \mapsto 1\},$
$bt_2 \mapsto \{Patient \mapsto Anna, Visit \mapsto 2, Blood \mapsto 2\},$
$pt_1 \mapsto \{Patient \mapsto Anna, Visit \mapsto 2, Report \mapsto 1\}\}.$

We then have:

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

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

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

We then calculate $Rg_{P_1,q_1}.E = \{r_1 \mapsto r_2\}$ since Alice belongs to the subject *Nurse*. We have $sink(Rg_{P_1,q_1}) = \{r_2\}$. Thus: $eval(P_1, q_1) = false$. Moreover, in all possible contexts, Alice can't access Anna's data, since $RulesRg_{P_1,q_1}$ will not vary with contexts.

Now Bob wants to access bt_2, $q_2 = (Bob, read, bt_2)$. We suppose that Anna is fine, and that Bob is her attending physician.

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

Since, Bob belongs to *GP Physician* and *Emergency*, he is affected by any rules targeting one of the two entities. The calculus of $Rg_{P_1,q_2}.E$ 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. Then we take the transitive reduction of $<$, thus $Rg_{P_1,q_2}.E = \{r_4 \mapsto r_3, r_3 \mapsto r_5\}$. We have then $sink(Rg) = \{r_5\}$. Bob's request is thus denied.

But in an emergency context, where Anna's life would be threatened, Bob would have access to this data, more precisely, to all Anna's data, since $sink(Rg_{P_1,q_2}) = \{r_6\}$ in this context for any data requested by Bob.

2.5.7 Potential danger detection

We are working on the formalisation of SGAC in B [1] and in Alloy [25]. This allows for the detection of potential dangerous situations, for instance when the patient hides important data from the medical staff. In that case, the following property must hold

for the patient p within the policy P:

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

Model checking this property allows for counter-example exhibition, thus identify a patient who has concealed all his/her data, and warn him/her about a potential danger. This verification can be done for all patient:

$$\forall\, p \in \mathsf{ran}(P.R.H.W)[Patient],\\ \forall\, d \in documents(P.R, Patient, \{Patient \mapsto p\}),\\ \exists\, i \in (P.S).Z \bullet eval(P, (i, read, d))$$

This property can be simplified into:

$$\forall\, d \in documents(P.R, Patient, \{\}),\ \exists\, i \in (P.S).Z \bullet eval(P, (i, read, d))$$

Finding a patient who has all of his data hidden is the same as finding a document which is completely hidden. Moreover, our formalisation of SGAC allows for access verification.

— *Determination of necessary 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.

2.6 Related Work

RBAC (*Role Based Access control*) [15, 45], 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 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 [41] and in B [22]. Verified properties 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 [27] (*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 parameterise 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 [6]. 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 [44] 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 [43] (*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;

— 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 [4] formalises 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 2.8 summarises the difference between the different models for which a formal model exists.

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

Table 2.8 – Comparison between access control models having a formal model

2.7 Performance comparison

Since XACML is an industrial standard and that it 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 policies, one for each level of priority (law, patient and hospital). We use first-applicable as the policy combination algorithm. Within a 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 expression matching any path that contains the rule subject.

To compare the performance of XACML with SGAC, we have used Balana [50], 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 $(b^h - 1)/(b - 1)$. Rules are randomly generated. The size of the trees ranged from 1093 to 21845 vertices. Fig. 2.6 shows the average request processing time versus the number of rules given in thousands. Here are some of 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 ($\leq$ 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.

2.8 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 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

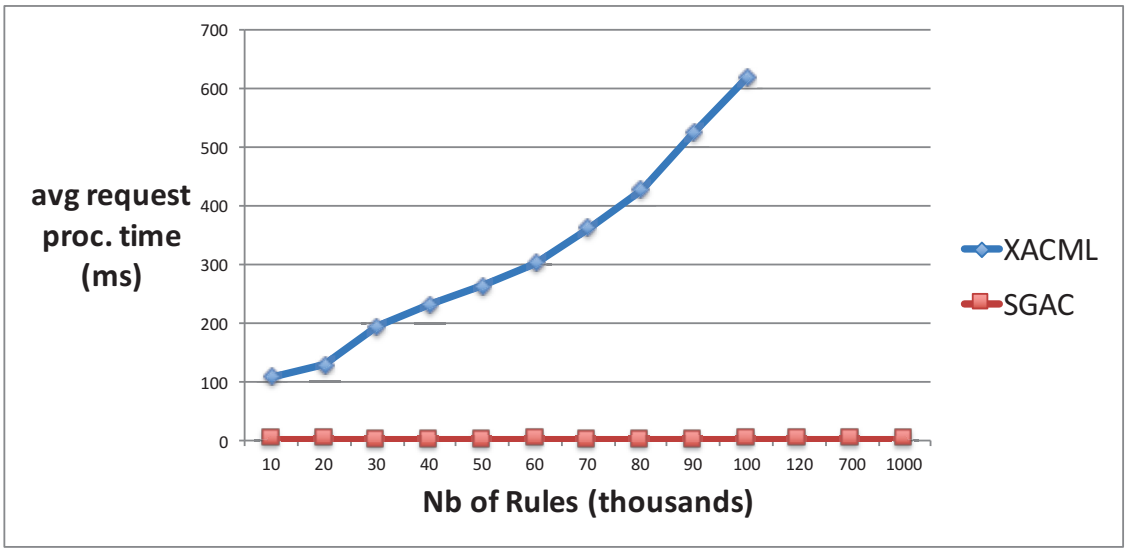


Figure 2.6 – Performances summary

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 ensure patient safety, we have proposed a formal model of SGAC policies to enable automated analysis of policy properties. In future work, we plan to explore tools like Alloy [25], ProB [29] and Yices [10], to automatically analyse SGAC policies.

Chapitre 3

Utilisation d'Alloy et de ProB pour l'analyse de politiques de sécurité SGAC.

Résumé

Cet article compare deux outils Alloy et ProB sur la vérification de propriétés pour SGAC. Pour cela, une modélisation de SGAC est faite dans les langages Alloy et B, ainsi qu'une analyse des performances du temps de vérifications de différentes propriétés avec les deux outils. Les différentes propriétés vérifiées sont :

— des propriétés d'accès : on vérifie si un utilisateur a accès à un document dans un contexte précis ;

— détection de documents cachés : on vérifie s'il existe des documents qui sont inaccessibles dans un contexte précis, et on détermine ces documents ;

— détection des contextes acceptants : étant donné un ensemble de règle et une requête, on détermine les contextes qui font que l'ensemble de règles accepte la requête dans ces contextes trouvés ;

— détection des règles inefficaces : on recherche dans la base de règles, les règles qui ne jouent dans aucun contexte un rôle déterminant dans la décision rendue par le système.

L'analyse des performances révèle que :

— le temps de traitement est indépendant du nombre de contexte ;

— le temps de traitement est plus fortement assujetti à la taille de la base de règle, *i.e.*le temps augmente beaucoup plus rapidement lorsqu'on augmente la taille de la base de règle qu'un autre paramètre ;

— le temps de traitement est linéaire en la taille du graphe ;

— ProB a de loin de meilleurs temps de traitement, grâce notamment à sa capacité de résolution de contraintes "programmable".

Les résultats de ProB sont assez prometteurs pour envisager son utilisation pour la vérification des règles d'accès de SGAC. Cet article a été soumis à la conférence *HASE2017* et il est présentement en cours d'évaluation.

Commentaires

Ce travail s'inscrit dans la continuité de ma thèse : après avoir formalisé SGAC, comment peut on vérifier de manière automatisée des propriétés sur les politiques d'accès de SGAC ? La formalisation en Alloy et en B a nécessité plusieurs optimisations à cause des différentes limitations de chaque outils : Alloy ne prend en charge qu'un nombre limité d'éléments, ce qui empêche la déclaration de l'ensemble des requêtes et ProB doit être "guidé" dans la résolution de contraintes afin de contrôler l'explosion combinatoire dans la résolution de contraintes. Ma contribution à cet article se compose de la formalisation de SGAC dans les deux langages, ainsi que des différentes optimisations. J'ai également mis en place un protocole de test pour comparer les deux outils : on compare les performances des deux outils pour tous les paramètres identifiés fixés, sauf un que l'on fait varier. Pour cela, j'ai développé une application qui génère le code Alloy et B à partir de graphes orientés acycliques (DAG) générés aléatoirement, de règles générées aléatoirement également et des propriétés à vérifier. Comme les graphes et les règles sont aléatoires, les tests ont été effectués sur de multiples graphes ainsi que sur des multiples requêtes afin d'obtenir un échantillonnage moyen.

Verification of SGAC Access Control Policies using Alloy and ProB

Nghi Huynh
Université de Sherbrooke, Québec, Canada
Université Paris Est-Créteil, Val de Marne, France

Marc Frappier
Université de Sherbrooke, Québec, Canada

Amel Mammar
SAMOVAR, Télécom SudParis, CNRS, Université Paris-Saclay, EVRY, France

Régine Laleau
Université Paris Est-Créteil, Val de Marne, France

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

Abstract

This paper investigates the verification of access control policies for SGAC, a new healthcare access-control model, using Alloy and ProB, two first-order logic model checkers based on distinct technologies.

SGAC supports permission and prohibition, rule inheritance among subjects and resources ordered by acyclic graphs; conflicts are autonomously managed using rule precedence based on priority, specificity and modality.

In order to protect patient privacy while ensuring effective caregiving in safety-critical situations, we check four types of properties: accessibility, availability, contextuality and rule effectivity. Our performance results show that ProB performs two orders of magnitude better than Alloy, thanks to its programmable approach to constraint solving. Results are promising enough to consider ProB for verifying patient policies in SGAC.

3.1 Introduction

With medical data being stored electronically, access control over these sensitive data has become crucial and compulsory. But control over medical data is not an easy task. Access control

— must ensure the patient's privacy;

— must not hinder health worker's work and endanger the patient's life.

SGAC (*Solution de Gestion Automatisée du Consentement — Automated Consent Management Solution*) [23] is an access control method which has been developed for the Sherbrooke University Hospital. It allows patients and the hospital to specify fine-grained access control rules over the medical data. In order to ensure patient safety and privacy, properties must be checked on the access control policies. For instance, the hospital would like to ensure that crucial patient data is available when the patient's life is in danger. Patients want to ensure that sensitive data that could damage their reputation, employability or social relationships are only disclosed to the appropriate persons in the right context.

SGAC is a sophisticated access control system which supports permission and prohibition, rule inheritance, priority definition and complex rule conditions. It can support large graphs of subjects and resources (>200K vertices) and large number of rules (>1M), while providing an excellent response time of around 3ms per access request. It is implemented in Node.js for scalability (*i.e.*, the controller can be easily parallelised using Node.js server facilities).

The flexibility of SGAC's policy language makes it mandatory to use automated verification techniques to check properties of SGAC access control policies. SGAC is based on relational structures and first-order logic. Building a custom property verification tool for such a language is an expensive task. Reusing existing model-checking tools is more cost effective and less risky in terms of long-term maintenance, while allowing for leveraging of future improvements.

There are three main classes of model checkers for first-order logic: SAT-based approaches like Alloy [26], constraint-based approaches like ProB [11], and SMT-based approaches like CVC4 [8], Yices [10] and Z3 [7]. 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 the ones used in SGAC [18, 14, 30, 48].

In this paper, 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 rules that are ineffective, *i.e.*, rules that are always overridden by other rules, and thus have no effect on the access granting decisions, and hence may denote misrepresented safety/privacy requirements.

Applicability of a model checking tool in our industrial context means: i) the ease of modelling SGAC policies; ii) the capability to deal with the access control policy of a patient, which includes graphs and rules. Ease of modelling is important, from a safety perspective. The easier it is to model a policy and the properties to check, the less likely there will be errors in the modelling. SGAC policies can be huge. They may contain millions of rules and hundreds of millions of documents. We don't expect any model checker to be able to handle such large numbers. 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.

The rest of this paper is structured as follows. Section 3.2 introduces SGAC, and its conflict resolution mechanism for ordering rules applicable to a given request. Section 3.3 presents Alloy and ProB, and a brief overview of the formalisation of SGAC in these respectives languages. Section 3.4 provides examples of how the formalisation is used in order to verify properties. Verification performance results are given in Section 3.5. Section 3.6 compares our findings with similar work on access control.

We conclude this paper with an appraisal of our work in Section 3.7.

3.2 SGAC: presentation

SGAC has been developed in collaboration with the Sherbrooke Area University Hospital Network (CIUSSS-Estrie), which includes 17 000 employees and more than 1 000 physicians. It has been designed to be generic enough to be reused in other domains. The SGAC access control engine has nothing specific about the medical domain. We present here all elements of SGAC needed to understand property verification. A more detailed discussion about our model and illustration of it can be found in [23].

SGAC handles requests made by users to access documents and returns a permission or a prohibition depending on the rules specified by the patients, the hospitals or required by laws and regulations.

3.2.1 Rule and request specification

A rule controls access requests of subjects to resources. It is defined by:

— a *subject*: a person or a group of people to control;

— a *resource*: the data to be protected;

— an *action*: the operation the *subject* wants to do on the *resource*;

— a *priority*: a number which defines the priority of the rule;

— a *modality*: a permission or a prohibition which defines the effect of the rule;

— a *condition*: a formula which determines the applicability of the rule. This formula represent a specific context, in which the rule is applicable.

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*.

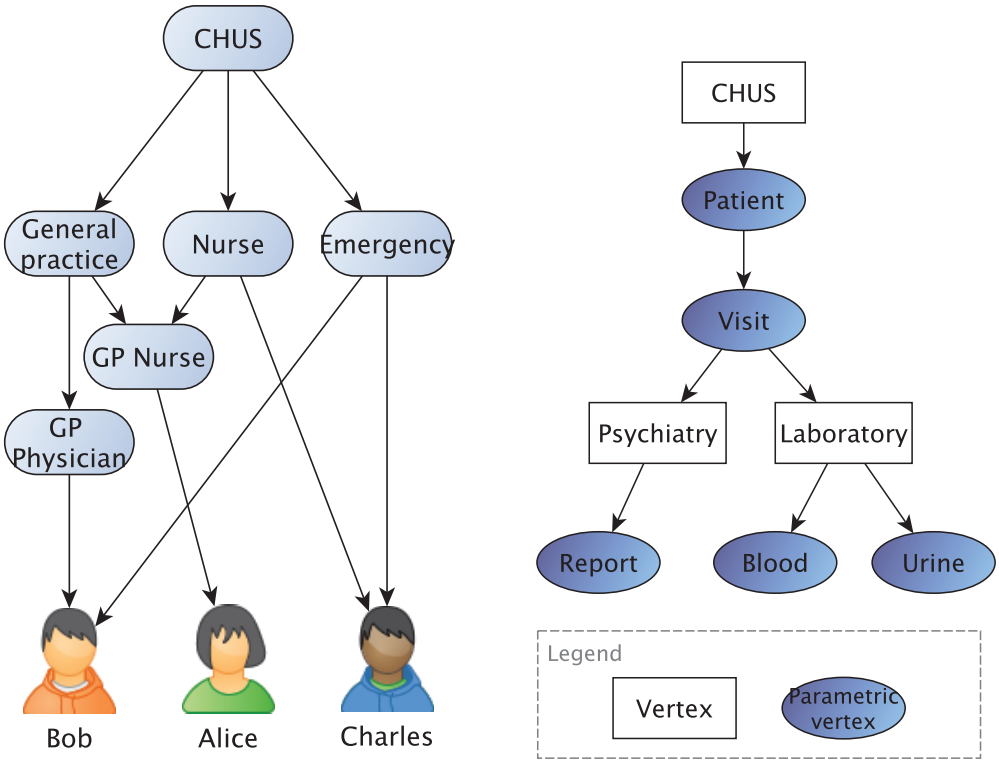
3.2.2 Subject graph and resource graph

In order to specify rules and requests, two directed acyclic graphs are needed:

— 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 medical documents and their organisation in the healthcare facilities.

A resource denotes a group of documents. A document is a particular kind of resource; subject can submit access request on documents only.



(a) Subject graph example (b) Resource type graph

Figure 3.1 – Subject and resource graphs example

Fig. 3.1a illustrates a subject graph. 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 vertices reachable from s. For instance in Fig. 3.1a, if a permission is given to the General Practice department then this permission is inherited by GP Physician and GP Nurse, Bob and Alice. Persons are sinks of this graph.

In order to minimize the size of the resource graph, which could include several hundred millions of records in a hospital like CHUS, vertices are parameterized, as shown in Fig. 3.1b. Thus the resource Patient denotes all records of all patients. In a rule, a resource vertex can be instantiated: for instance, the resource Patient denotes all patient records, whereas the resource Patient = Anna denotes all records of the specific patient Anna. The actual medical records of a patient, called documents, are sinks of this graph. In the rest of this paper, we suppose that all parameterized documents nodes are instantiated with their instance. To conduct tractable analysis, we analyse a subgraph of the resource graph (*e.g.*, only Anna's records).

3.2.3 Behaviour and conflict resolution

A rule applies to a request when i) there is a path from the rule's subject to the request's subject, ii) there is a path from the rule's resource to the request's document, and the rule's condition hold. 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 prohibitions have precedence over permissions.

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 in this ordering.

The conflict resolution method relies on the fact that, generally, a rule which targets a smaller group (inclusion-wise) than other rules should have precedence over these. For instance, if a patient has a rule that permits nurses to access his/her data

and another rule that prohibits a specific nurse from doing that, then the more specific rule should have precedence when this specific nurse issues a request. This conflict resolution method is absolutely autonomous and does not require the intervention of an external actor.

3.2.4 Example

We illustrate the presented behaviour with the following example: let Bill be an anaesthetist and a surgeon. Since he has two profiles, he inherits access privileges from both of them. In Fig. 3.2, rules which apply to a request of Bill to read the document D are: r_1, r_2, r_3 and r_4. We suppose they share the same priority. In the case where all these four rules are active under the same context (*i.e.*, their condition holds): surgeons are not able to access the document D while anaesthetists can, making Bill unable to access the document. We have r_1 that is less specific than r_2, and the same goes for r_3 and r_4. Since r_2 and r_4's subjects are incomparable, precedence is given to the prohibition, resulting in a prohibition for Bill's request. If there is a context where only r_1, r_3 and r_4 are active, Bill's request would be granted in this context since anaesthetists would be allowed to access the document.

3.3 Formalisation of SGAC

In this section, we present an overview of the formalisation of SGAC in Alloy and B.

3.3.1 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

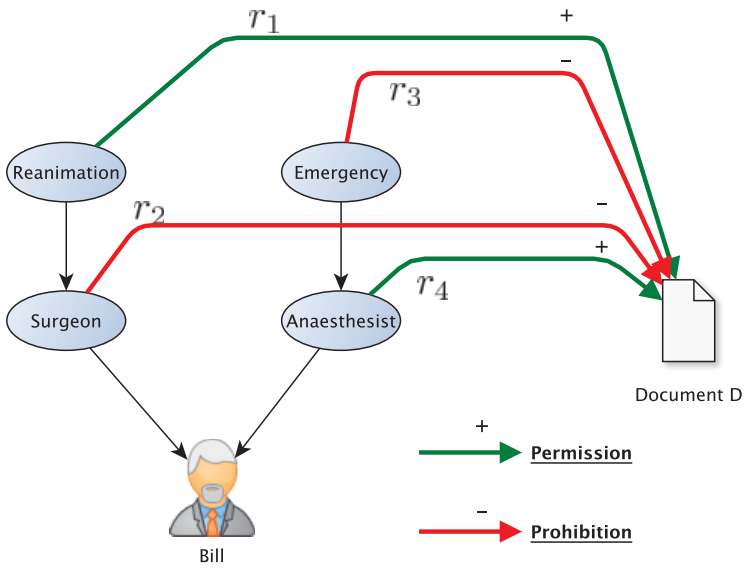


Figure 3.2 – Example of rule precedence

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.

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 Fig. 3.3 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. 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`. 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.

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, we have two choices:

— 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 Fig. 3.3.

— Declare a constraint as a **`fact`**: this constraint is declared outside a signature. This is the case for the constraint on line 9 of Fig. 3.3.

Operator `#`e returns the number of elements of the set e. For instance, in Fig. 3.3, 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 Fig. 3.3, 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 Fig. 3.3.

```
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 3.3 – Example of Alloy declarations of signatures and constraints

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.

Generating specification instances

Once all signatures and constraints have been 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:

— **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.

```
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 3.4 – Using Alloy commands

— **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* does not hold.

Fig. 3.4 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 Fig. 3.3 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.

3.3.2 SGAC in Alloy

To model SGAC in Alloy, we define the following basic types (Alloy signatures):

— the *subjects*, that represent the users, with their graph;

— the *resources*, that represent the data, with their graph;

— the *contexts*, which represent different conditions where a rule can apply;

— the *modalities*, prohibition or permission;

— the *rules*, that represents rules.

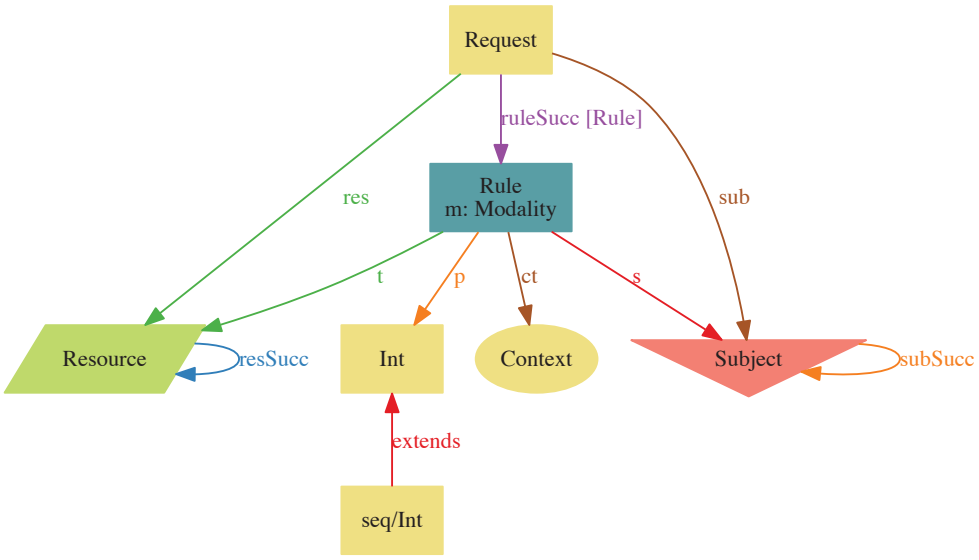


Figure 3.5 – Basic structure of SGAC

From these basic types, we can define the edges of the subject and resource graphs: *subSucc*, *resSucc*. We do so by adding a field in the *subject* and *resource* signatures, mapping a *subject* or a *resource* to a set of same type objects (lines 2 & 6). The type *rule* is mapped to one *subject*, one *resource*, one *modality*, one integer (*priority*) and a set of *context*.

Figure 3.5 illustrates the relationships between these signatures.

In order to capture the fact that conditions may have dependencies between them, our first attempt consisted in using booleans and truth tables to model the dependencies. Since it was very resource consuming, we use instead sets of abstract contexts which are by far more efficient. For instance the condition c_1 which states "*in September*" must imply the condition c_2 which states "*between August and November*". We use two contexts con_1 and con_2 such that con_1 denotes the time window [August, November] minus September, and con_2 the whole month of September. This way $c_1 = \{con_2\}$ and $c_2 = \{con_1, con_2\}$, and "$c_1 \implies c_2$" is equivalent to "$context_1 \subseteq context_2$".

```
1  sig Subject {
2    subSucc: set Subject
3  }
4
5  sig Resource {
6    resSucc: set Resource
7  }
8
9  fact{
10   acyclic[subSucc,Subject]
11   acyclic[resSucc,Resource]
12 }
13
14 abstract sig Modality {}
15 one sig prohibition, permission extends  Modality {}
16
17 sig Context {}
18
19 sig Rule {
20   p : one Int,
21   s : one Subject,
22   t : one Resource,
23   m : one Modality,
24   ct : set Context
25 }{
26   p >= 0
27 }
28
29 pred evalRuleCond[r:Rule,c:Context]{
30 c in r.ct
31 }
```

Figure 3.6 – Alloy declaration of the Subject and Resource graphs and Rules signature

We then add the *request* type that basically is the Cartesian product of the sinks of the subject and resource graphs. To order the rules and determine which rule will have precedence over the others, we have to create a rule graph.

In [23], the approach we proposed was to build a rule graph from the applicable rules which have their conditions verified. Thus for each couple (*req*,*con*), we had to create a graph to determine the result of the request *req* in the context *con*. The sinks of the rule graph determine the result of the request: if all sinks are permissions, the request is granted, if not, it is denied. The edges of the graph are represented by *ruleSucc*. In this case *ruleSucc* is a quaternary relation: this leads to a heavy computational burden. We came up with an alternative to avoid computing a rule graph for each couple and reuse the graph made without regards to contexts. We now build a rule graph for each request only. To evaluate a request *req* given a context *con* instead of looking for sinks of the rule graph, we look for the deepest rules that apply for the context *con*, *i.e.*vertices that have no successor which contains *con* in its condition. Those vertices are called *pseudo-sinks*.

The resulting Alloy code for the request signature declaration is presented in Fig. 3.7: 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.

The next step is to specify how the rules are ordered in the 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. The rule r_1 is *lessSpecific* that a rule r_2 means that either r_1 has a higher priority value or r_1's subject is a predecessor (strict) of r_2's. Note that all rules cannot be compared with that relation.

We then define *isPrecededBy*, based on *lessSpecific*, which compare two rules as *lessSpecific* does, and in addition, in the case 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 represen-

```
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 3.7 – Alloy declaration of the request

tations, maximal elements appear at the bottom of the graphs.

The requirement that only maximal elements of *lessSpecific* can be compared by their modality is crucial: for instance, if we take the example of Fig. 3.8, we have `isLessSpecific[`r_1`,`r_2`]` and `isLessSpecific[`r_3`,`r_4`]` but r_2 and r_4 cannot be compared with this predicate. If we remove the maximal element condition from the predicate `isPrecededBy` (denoted by '$<$') 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.

Finally, for readability we want a rule graph with as few edges as possible denoting rule precedence, with the same coverage than *isPrecededBy*, so we compute the transitive kernel of relation *isPrecededBy*.

3.3.3 The B-Method and the ProB tool

The B language [1] relies on first order logic, arithmetic and set theory. The B language is used to specify systems, by describing state variables and operations that modify these state variables. A system in B is described in a *machine*. In B, a machine contains:

— a clause **SETS** containing the declaration of basic sets and types;

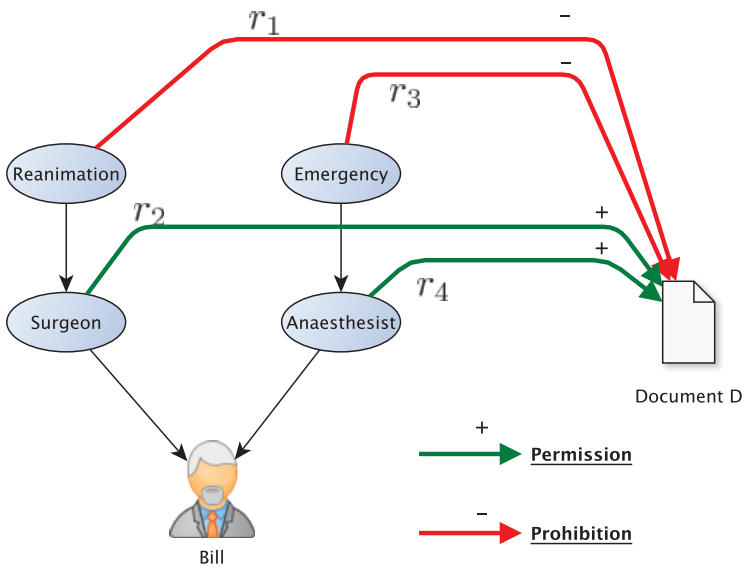


Figure 3.8 – Rule ordering example

— a clause **CONSTANTS** containing the declaration of all constants;

— a clause **PROPERTIES** containing all properties binding constants, such as their type, or their valuation;

— a clause **VARIABLES** containing the declaration of all variables used;

— 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, with regards to **INVARIANT**;

— a clause **OPERATIONS** containing the operations that can be done on the variables. In B, an operation is defined by preconditions and postconditions on variables and can return a value.

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. In our case, we use the constraint solver to verify the properties

```
1 pred lessSpecific[r1,r2: Rule]{
2      (r2.p < r1.p)
3   or (     r2.p = r1.p
4         and r2.s in (r1.s).^subSucc)
5 }
6
7 pred maximal[r:Rule]{
8 no r1 : Rule | lessSpecific[r,r1]
9 }
10
11 pred isPrecededBy[r1,r2:Rule]{
12   (
13         lessSpecific[r1,r2]
14     or
15     (
16           not lessSpecific[r2,r1]
17       and maximal[r1]
18       and maximal[r2]
19       and r2.m = prohibition
20       and r1.m != r2.m
21     )
22   )
23 }
24
25 fact {
26 all rq: Request | all r1,r2: appRules[rq] |
27     r1 in req.ruleSucc.r2
28   <=>
29     (   isPrecededBy[r1,r2]
30     and not some r3 : appRules[rq] |
31               isPrecededBy[r1,r3]
32           and isPrecededBy[r3,r2]
33     )
34 }
```

Figure 3.9 – Alloy declaration of the predicates and function needed to sort the rules

such as for, instance, "the rules allow the health worker X to access to the resource Y".

Let A, B, C be sets.

— $\mathcal{P}(A) = \{X \mid X \subseteq A\}$, 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 $A \times B$.

— $\mathsf{id}(A) = \{x \mapsto x \mid x \in A\}$ denotes the identity relation on A, i.e. the relation that associates each element of A to itself.

— $A \leftrightarrow B = \mathcal{P}(A \times B)$ denotes the set of relations between A and B.

— $\mathsf{dom}(R) = \{x \in A \mid \exists\, y \in B \bullet x \mapsto y \in R\}$ 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 partial function such that $\mathsf{dom}(f) = A$.

— $R_1 \fatsemi R_2 = \{x \mapsto z \mid \exists\, y \in B \bullet x \mapsto y \in R_1 \wedge y \mapsto z \in R_2\}$ 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 $R^0 = \mathsf{id}(A)$.

— $R^+ = \bigcup_{n \geq 1} R^n = \mathit{closure}(R)$ denotes the transitive closure of R, *i.e.*, the smallest transitive relation which contains R.

— Let $R \in A \leftrightarrow A$. $R^* = R^+ \cup \mathsf{id}(A) = \mathit{closure1}(R)$ 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 relation projection of $A \times B$ in A. Let $prj2(A, B)$ denotes the second relation projection 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 relation that maps the element x of A to the expression $e(x)$ with regards to the predicate P.

SETS

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

Figure 3.10 – **SETS** clause of the B specification of SGAC

— Let $S = struct(Id_1 : T_1, \ldots, Id_n : T_n)$ denotes the record, or set formed by the ordered collection of n types T_i called field of the record. Each field has a unique identifier Id_i called label. Then $S'Id_1$ denotes the field of type T_1 with the label Id_1 of the record S.

3.3.4 Formalisation 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 clause of a B machine, in a way pretty similar to the Alloy specification. This approach failed because ProB was unable to solve the constraints in an efficient manner.

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, and 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 Fig. 3.10. 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 the Fig. 3.11.
3. Declaration of the types of the constants in **PROPERTIES**, and initialisation of the two graphs and the rule base, presented in Fig. 3.11:
   - *e_sub, e_res* denote the edges of the subject and resource graphs;
   - $REQUEST\_T$ denotes the type of request, being the Cartesian product of the sinks of the graphs;
   - *rule* 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 $cl\_e\_sub \cap \mathsf{id}(V\_SUB) = \emptyset$ for instance;
   - we set the values of the constants:
     - we set the constant *lessSpecific* (same semantic as the relation in Alloy) by comparing each rule with their priority and their subject;
     - we then set the graphs and the rules.
4. Declaration of the rule ordering related variables in **VARIABLES**, plus some variables that avoid recomputing some heavy calculus 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 mapping 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);

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

Figure 3.11 – **PROPERTIES** clause of the B specification of SGAC

VARIABLES
applicable,
conRule,
isPrecededBy,
ruleSucc,
cl1_ruleSucc,
pseudoSink
INVARIANT
$applicable \in \quad REQUEST\_T \rightarrow \mathcal{P}\ (RULE\_T) \wedge$
$conRule \in CONTEXT \rightarrow \mathcal{P}\ (RULE\_T) \wedge$
$isPrecededBy \in REQUEST\_T \rightarrow (RULE\_T \leftrightarrow RULE\_T) \wedge$
$ruleSucc \in REQUEST\_T \rightarrow (RULE\_T \leftrightarrow RULE\_T) \wedge$
$cl1\_ruleSucc \in REQUEST\_T \rightarrow (RULE\_T \leftrightarrow RULE\_T) \wedge$
$pseudoSink : (REQUEST\_T \times \quad CONTEXT) \rightarrow \mathcal{P}\ (RULE\_T)$

Figure 3.12 – **VARIABLES** and **INVARIANT** clauses of the B specification of SGAC

— *cl1_ruleSucc* denotes the transitive closure of *ruleSucc*;

— *pseudoSink* denotes a function that returns given a request and a context, the pseudosinks of the rule graph of that request, with that context.

5. Declaration of the types of the variable, the constraints binding the variables in the **INVARIANT**, presented in Fig. 3.12;
6. Initialisation of the variables with a sequence, with regards to the dependency in Fig 3.13 and Fig 3.14: 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 we use variables and constant, to force ProB to set them in the proper order.
7. Declaration of the operations which represent the different properties we want to verify.

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

Figure 3.13 – **INITIALISATION** clause of the B specification of SGAC (1)

$$
\begin{array}{l}
pseudoSink := \lambda\ (req,con).(req \in REQUEST\_T \wedge con \in \mathbf{dom}(conRule) \\
\quad | \{\ ru \mid ru \in applicable(req) \wedge \\
\qquad ru \in conRule(con) \wedge \\
\qquad \forall\ subr.(\\
\qquad\quad (subr \in cl1\_ruleSucc(req)[\{ru\}]) \implies \neg\ (\ subr \in conRule(con)) \\
\qquad)\quad \}\ )
\end{array}
$$

Figure 3.14 – **INITIALISATION** clause of the B specification of SGAC (2)

In B, we can use macros that we define in the **DEFINITIONS** clause: the macros are replaced in the same manner as *#define* in C. The macros we use are presented in Fig. 3.15: we define here the definitions for *applicable*, for *isPrecededBy*, with *applicable_def* and *maxElem*, and the definition of *access_def*.

3.4 Properties verification

We check the following properties.

3.4.1 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 are found, the request is denied. This is illustrated for the B specification in Fig. 3.17 and in Fig. 3.16 for Alloy.

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 29) 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 are found, the request is denied. For efficiency, we try to keep the number of explicitly

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 3.15 – **DEFINITIONS** clause of the B specification of SGAC

declared signatures to the minimum for each run. Thus for each request evaluation, each request is explicitly declared in separate Alloy file.

The formalisation of it in B is provided in Fig. 3.17 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.

3.4.2 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 is presented in Fig. 3.18. We define the predicate `HiddenDocument`: 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): Alloy tries to find a counterexample with a granted request that targets the document.

We do the same in B: the operation *HiddenDocument* presented in Fig. 3.19 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 Fig. 3.20. 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 context. The operation *GrantinContexts* presented in Fig. 3.21 returns all contexts that make the given request granted.

```
fun pseudoSinkRule[req: Request,c:Context]
    : set Rule{
  {r : appRules[req] |evalRuleCond[r,c] and

      all ru : r.^(req.ruleSucc) |
            not evalRuleCond[ru,c]
  }
}

pred accessCondition
  [req:Request,c:Context]{
  (
    no  r:pseudoSinkRule[req,c] |
    r.m=prohibition


  ) and
  some r:appRules[req] | evalRuleCond[r,c]

}

one sig req0 extends Request{}{
  sub=s1
  res=r0
}

run accessReq0_c1{
  accessCondition[req0,c1]
} for 4
```

Figure 3.16 – Checking Access with Alloy

$access \leftarrow$ **CheckAccess**(req,con) =
PRE
$req \in REQUEST\_T \wedge$
$con \in CONTEXT$
THEN
$access :=$ **bool**($access\_def(req,con)$)
END;

Figure 3.17 – 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 3.18 – Detection of hidden documents with Alloy

$hidden \leftarrow$ **HiddenDocument**$(con) =$

PRE

$con \in CONTEXT$

THEN

$hidden :=$ **bool**$(\exists\ (hdoc).($

$hdoc \in V\_RES$ - **dom**$(e\_res) \wedge$

$\forall\ req\ .(req \in REQUEST\_T \wedge$ **prj2**$(V\_SUB, V\_RES)(req)=hdoc$

$\Rightarrow$

$\neg\ access\_def(req,con))))$

END;

Figure 3.19 – Detection of hidden documents with B

```
1  one sig GrantingContext{
2    acc: set Context
3  }{}
4
5  pred grantingContextDet[req:Request]{
6      all c: Context | accessCondition[req,c]
7    <=> c in GrantingContext.acc
8  }
9
10 run grantingContextDetermination{
11   grantingContextDet[req1]
12 } for 5
```

Figure 3.20 – Determination of the contexts that make a request accepted with Alloy Analyzer

$granting \leftarrow$ **GrantingContexts**(req) =
PRE
  $req \in REQUEST\_T$
THEN
  $granting :=$
    $\{\ con \mid con \in CONTEXT \wedge$
      $access\_def(req, con)$
    $\}$
END;

Figure 3.21 – Determination of the granting contexts in B

3.4.3 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 priority, 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 not to be tagged as ineffective 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.*then the permission is the only pseudo-sink.

We introduce the predicate `ineffectiveRule` in Fig. 3.22 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 than the given rule;

```
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 3.22 – Determination of the ineffective rules in Alloy

— 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). Since we specify a property over all the requests, we are compelled to use the command **`check`** because all the requests are not explicitly declared for efficiency. Thus, we cannot use a signature wrapping up all ineffective rules as we did with `GrantingContext`.

In B, all the ineffective rules are returned by executing the operation *IneffectiveRuleSet* presented Fig. 3.23. 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 applies. 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.

ineffectiveSet ← **IneffectiveRuleSet** =
PRE
TRUE = TRUE
THEN
ineffectiveSet := {
ru | *ru* ∈ *RULE_T* ∧
¬ (
∃ (*req*,*con*).(
req ∈ *REQUEST_T* ∧
ru ∈ *conRule*(*con*) ∧
ru ∈ *pseudoSink*(*req*,*con*) ∧
(
pseudoSink(*req*,*con*) - {*ru*} = ∅ ∨
(
(*rules*(*ru*))'*mo* = *pro* ∧
∀ *ru2*.(*ru2*:(*pseudoSink*(*req*,*con*)-{*ru*}) ⟹
(*rules*(*ru2*))'*mo* = *per*)
)
)
)
)
}
END

Figure 3.23 – Detection of ineffective rules in B

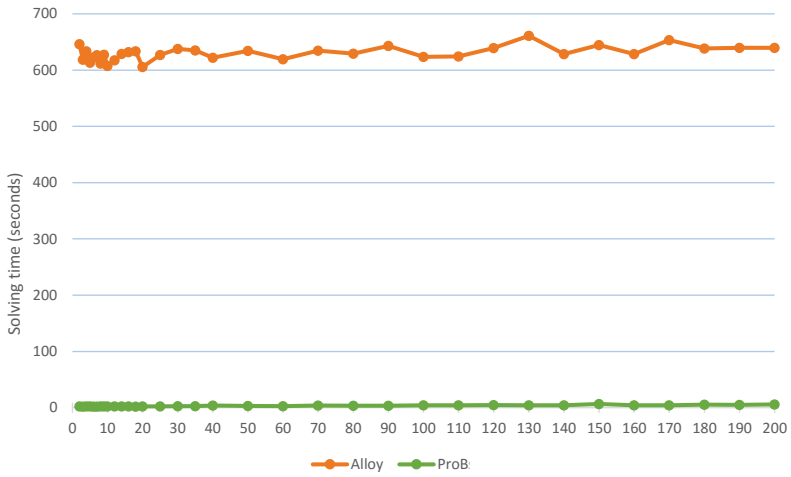


Figure 3.24 – Varying the number of contexts versus the solving time (30 vertices, 12 rules)

3.5 Performance tests

To test our models, we randomly generate graphs and requests. We control 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 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 four properties. Tests were performed on a virtual server (Intel Xeon 3.10GHz) using Java 1.7 with 12GB of RAM.

3.5.1 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 Fig. 3.24 show that the solving time is quite constant and seems to be independent of the number of contexts.

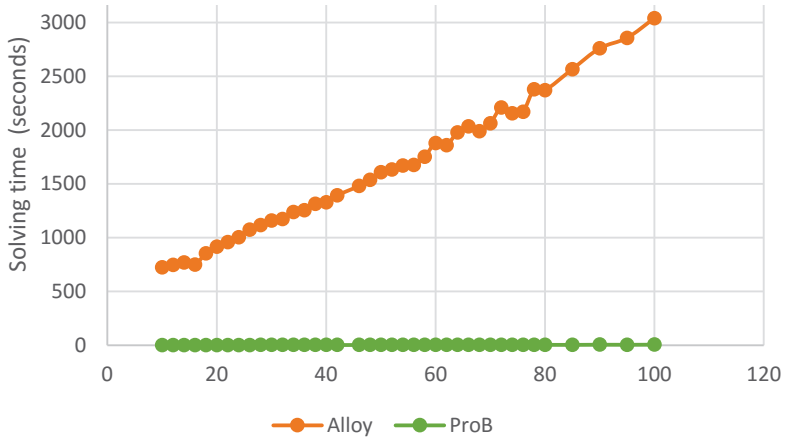


Figure 3.25 – Varying the number of vertices versus the solving time (13 rules, 10 contexts)

3.5.2 Varying the number of vertices

For this second test in Fig. 3.25, 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

3.5.3 Varying the number of rules

For this test in Fig. 3.26, the number of vertices in each graph is set to 100 and the number of context 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.

3.5.4 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

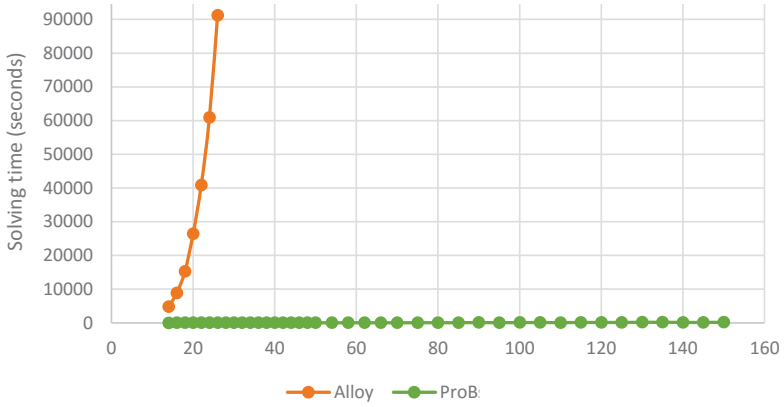


Figure 3.26 – Varying the number of rules versus the solving time (100 vertices, 30 contexts)

of possible requests. In Fig. 3.27, 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 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).

3.5.5 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.

3.6 Related work

RBAC [45], a standardized and well-known access-control model based on roles, allows for specification of permissions associated to a role, to execute actions on resources. The lack of prohibition and condition in this model makes the verification of an access property easier, since there are no conflicts. However, it has been shown in [39] that RBAC is inappropriate for fine-grained access control as found in healthcare requirements like CIUSSS-Estrie's. RBAC has been formalised in Z [41, 42] and

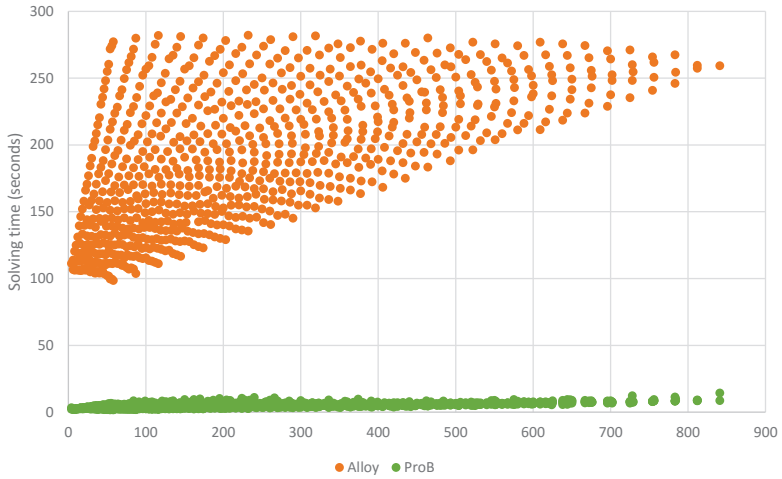


Figure 3.27 – Varying the number of requests versus the solving time (30 vertices, 20 rules and 10 contexts)

B [15]. Traditional RBAC properties are checked (role activation, role hierarchy and separation of duty).

OrBAC [27], a logic-based access-control model, introduces the notion of organisation and includes among other things explicit prohibitions and contexts. A rule can be defined to be only applicable in specific contexts. Conflicts are detectable by static checking with the Prolog-based tool MotOrBAC [6]. OrBAC does not fit our needs since it do not have an automatised conflict resolution, and once a static check reveals a conflict, a human intervention is required to solve it.

XACML [43] is an attribute-based access-control language that features prohibitions and conditions, and allows to determine how conflicts are managed by rule combination algorithms. As shown in [39], XACML does not natively support rule inheritance, since it does not include a graph of subjects or resources; it can be simulated using paths in resource name, but this complexifies the maintenance of a rule base, while providing poor performance for very large rule bases. Formalisations of XACML has been made using process algebra [4], a logic-based language that can be used with a SAT solver [20], and Alloy [33]. These formalisations allow for access property verification. Furthermore, it cannot be reused easily with our rule ordering.

3.7 Conclusion

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 B. B 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 [30], and for university time tables [18]. 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.

Conclusion

Les objectifs de cette thèse sont d'étendre le modèle de contrôle d'accès issu de nos travaux de maîtrise, SGAC (*Solution de Gestion Automatisée du Consentement*) afin de permettre au patient d'exprimer son consentement, et de lui fournir des outils lui permettant de créer une politique d'accès qui lui convient. Pour cela, nous avons choisi de doter SGAC d'une formalisation, de formaliser sa méthode de résolution de conflit et de définir des méthodes de vérification des propriétés suivantes :

— un utilisateur peut-il accéder à un document dans un certain contexte ?

— un patient possède-t-il une donnée qui est inaccessible de tous dans un certain contexte ?

— quelles sont les contextes qui autorisent un certain utilisateur d'accéder à un certain document ?

— quelles sont les règles qui n'interviennent jamais dans le processus de décision de contrôle d'accès ?

Le Centre Hospitalier de l'Université de Sherbrooke (CHUS) fournit un cadre d'application réel : SGAC, et les méthodes de vérifications de propriétés doivent pouvoir prendre en charge un grand nombre de patients, de données et de règles.

Pour ce faire, nous avons dans un premier temps conduit une étude comparative de plusieurs modèles de contrôle d'accès existants, et nous avons notamment choisi de modéliser un des modèles les plus connus et utilisé : RBAC (*Role-Based Access Control*). La littérature sur ce modèle standardisé est abondante, et des formalisations de celui-ci existent dans divers langages. Pourtant, subsistent dans le standard des incohérences, des oublis voire des erreurs détectées grâce à la formalisation en B proposée dans le chapitre 2. RBAC, du fait de plusieurs lacunes, entre autres l'impos-

sibilité de spécifier des interdictions, ne peut convenir à la gestion du consentement. Ce premier pan de la thèse a permis une immersion dans la vérification appliquée à des modèles de contrôle d'accès et a donné un aperçu des différentes propriétés vérifiées sur ce type de modèles.

Dans un second temps, nous avons défini une formalisation de SGAC, modèle conçu pour répondre à la problématique de la gestion du consentement. Issu de nos travaux de maîtrise, SGAC a été complété et affiné grâce à cette formalisation qui permet de vérifier diverses propriétés, comme par exemple la détection de cas dangereux, où le patient empêcherait les personnes aptes à le soigner d'avoir accès aux informations nécessaires. Nous définissons à partir de cette formalisation des méthodes pour

— vérifier l'accès d'un utilisateur à une ressource sous un contexte donné,

— vérifier qu'un document soit inaccessible sous un contexte donné.

La formalisation de SGAC a constitué la majeure partie du travail, notamment au niveau de l'identification de tous les concepts présents et leurs relations. Les défis auxquels nous avons été confronté sont le couplage des ressources avec les paramètres, et la relation de comparaison des règles. Cette dernière s'est révélée plus délicate que prévu et doit être décomposée en deux étapes majeures :

1. comparaison sur la priorité et la spécificité des sujets qu'on notera $\prec$ (des règles peuvent être incomparables par $\prec$) ;
2. extension de $\prec$ pour traiter les cas où deux sont incomparables, en utilisant la modalité.

Enfin, dans l'optique d'une utilisation de SGAC et de sa composante vérification dans un cadre réel où il faut prendre en compte un nombre important de données, le choix du vérificateur de modèles pour mener les vérifications automatisées des propriétés étudiées précédemment est crucial. Nous avons donc effectué une étude comparative de deux outils, Alloy et ProB. Notre étude montre la supériorité manifeste de ProB dans la vérification de propriétés sur SGAC : meilleur temps de vérification et facilité d'utilisation. Cela vient principalement du fait qu'il est possible avec ProB de spécifier une propriété sur un ensemble, ce qui n'est pas toujours le cas avec Alloy. En effet, Alloy ne peut répondre à une propriété que par "satisfiable"

ou “non satisfiable”, alors que ProB permet de retourner un ensemble qui satisfait une propriété : il faut donc autant d'exécutions que de documents pour détecter les documents inaccessibles en Alloy, contre une seule avec ProB. Ce troisième volet de la thèse, à la fois technique et expérimental, inclut le développement d'une application permettant de convertir les graphes de sujets, de ressources et les règles ainsi que les propriétés dans les deux langages puis de comparer les performances des deux outils. Le modèle présenté dans [23] ne pouvant pas être utilisé comme tel, un travail d'adaptation du modèle pour chaque outil s'est imposé afin de contourner leurs limitations respectives. Par exemple pour pouvoir effectuer des vérifications en Alloy, au lieu de déclarer l'ensemble des requêtes dans sa totalité, seule la requête à évaluer est explicitement déclarée : on utilise la commande *check* qui cherche un contre-exemple lorsqu'on doit vérifier l'existence d'une requête satisfaisant une certaine propriété, ou une propriété que toutes les requêtes doivent satisfaire.

Perspectives

Cette thèse ouvre plusieurs perspectives de recherche :

— réduction du temps nécessaire à la vérification,

— adaptation de SGAC à d'autres contextes,

— ajouts de nouvelles fonctionnalités telle que la délégation.

L'exploration d'autres outils tels que les vérificateurs de modèles basés sur des *SMT solvers* peut conduire à une amélioration du temps nécessaire à la vérification. Ce gain de temps peut également être obtenu grâce à diverses optimisations. On distingue deux types d'optimisations : les optimisations faites sur le modèle traduit pour un outil spécifique, et les optimisations faites sur le modèle avant traduction. Dans le chapitre 3, des optimisations ont été appliquées sur le modèle traduit, mais il doit être possible d'en trouver davantage : par exemple en omettant le calcul de la réduction transitive ou encore en factorisant l'évaluation des requêtes. Les optimisations effectuées sur l'ensemble des règles, des graphes de sujets et/ou ressources ne dépendent pas d'un outil donné. Ces optimisations peuvent être faites localement :

— Au niveau des règles : le patient peut vouloir réduire le nombre de ses règles

tout en conservant le comportement de son ensemble de règles. Ainsi, les règles redondantes ou inutiles doivent être enlevées, certaines règles peuvent être regroupées ou fusionnées.

— Au niveau des sujets : l'organisation ou la hiérarchie des sujets peut potentiellement être simplifiée en réduisant le nombre de sommets (groupe de sujets) tout en conservant les accès de chaque intervenant, par exemple en fusionnant les groupes d'utilisateurs dont les utilisateurs ont les mêmes privilèges. Cette optimisation doit se faire en s'assurant qu'elle n'engendre pas de règles supplémentaires : il est simple d'imaginer un graphe de sujets réduit à l'ensemble des utilisateurs, sans aucun groupe, avec leur accès conservés par le report des règles de groupe sur chaque utilisateur. Cela multiplierait le nombre de règles et rendrait leur maintenance difficile.

Une autre perspective est de transposer SGAC à un autre domaine d'application, par exemple celui des infrastructures réseau. Les pare-feux disposent de règles d'adressage des paquets. Ces règles qui définissent le comportement du pare-feu peuvent être adaptées au format des règles SGAC.

Les règles du pare-feu sont ordonnées, et la première applicable trouvée est celle qui sera appliquée : l'entretien de la base de règles du pare-feu est ardu et très difficile, chaque modification de la base devant être faite à la main avec des risques importants d'erreur. L'utilisation de SGAC pourrait rendre la modification des politiques d'accès réseaux beaucoup plus simple et également permettre la vérification de propriétés telles que la vérification de blocage effectif de certains types paquets, ou la détection de règles inefficaces.

La délégation, qui permet à un utilisateur de confier ses droits d'accès à un autre utilisateur, constitue un autre axe de développement ajoutant des degrés de complexité supplémentaire. En plus de la délégation simple des droits d'accès, il est possible de déléguer un droit de délégation, il devient donc possible de créer une chaîne de délégations. Cela peut devenir problématique lorsqu'il y a révocation des délégations : une révocation entraîne également la révocation des sous-délégations, mais si une personne avait reçu la délégation d'un autre personne, cette personne peut ne pas être affectée par la révocation. Cette notion ajoute par ailleurs de potentiels conflits entre les règles appliquées à un individu et celles qu'on lui a déléguées : quel est le

comportement à adopter lorsqu'un utilisateur, autorisé à accéder à un document, hérite d'une interdiction par la délégation, et vice-versa ? Étendre SGAC avec la notion de délégation est un défi intéressant et qui possède une application concrète : un médecin peut se faire remplacer, et son remplaçant doit être, en principe, en mesure de continuer le travail de son prédécesseur.

Bibliographie

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

[2] ANSI.
« Role Based Access Control, INCITS 359-2004 », 2004.

[3] ANSI.
« Role Based Access Control, INCITS 359-2012 », 2012.

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

[5] Clearsy.
« Atelier B ».
http ://www.atelierb.eu/.

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

[7] Leonardo Mendonça de Moura et Nikolaj Bjørner.
« Z3 : An Efficient SMT Solver ».
Dans C. R. Ramakrishnan et Jakob Rehof, éditeurs, *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 de *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

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

[9] « Dossier Médical Partagé ».
http://www.dmp.gouv.fr/.

[10] Bruno Dutertre.
« Yices 2.2 ».
Dans Armin Biere et Roderick Bloem, éditeurs, *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 de *Lecture Notes in Computer Science*, pages 737–744. Springer, 2014.

[11] Michael Leuschel *et al.*
« ProB ».
http://www.stups.uni-duesseldorf.de/ProB.

[12] Nghi Huynh *et al.*
« B Specification of the RBAC 2012 Standard », 2014.
http://info.usherbrooke.ca/mfrappier/RBAC-in-B.

[13] Santé et Services sociaux Québec.
« Dossier Santé Québec ».
http://www.dossierdesante.gouv.qc.ca/.

[14] Jerome Falampin, Hung Le-Dang, Michael Leuschel, Mikael Mokrani et Daniel Plagge.
« Improving Railway Data Validation with ProB ».
Dans *Industrial Deployment of System Engineering Methods*, pages 27–43. Springer, 2013.

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

[16] David F. Ferraiolo, Richard Kuhn et Ravi S. Sandhu.
« RBAC Standard Rationale : Comments on "A Critique of the ANSI Standard on Role-Based Access Control" ».
Security Privacy, IEEE, 5(6):51–53, 2007.

[17] Virginia N. L. Franqueira et Roel Wieringa.
« Role-Based Access Control in Retrospect ».
IEEE Computer, 45(6):81–88, 2012.

[18] Dominik Hansen, David Schneider et Michael Leuschel.
« Using B and ProB for Data Validation Projects ».
Dans *Proceedings ABZ 2016*, volume 9675 de *LNCS*, pages 167–182. Springer, 2016.

[19] Hongxin Hu et Gail-Joon Ahn.
« Enabling verification and conformance testing for access control model ».
Dans *SACMAT 2008, 13th ACM Symposium on Access Control Models and Technologies, Estes Park, CO, USA, June 11-13, 2008, Proceedings*, pages 195–204, 2008.

[20] Graham Hughes et Tevfik Bultan.
« Automated verification of access control policies using a SAT solver ».
STTT, 10(6):503–520, 2008.

[21] Nghi Huynh.
« Résolution de conflits dans la gestion du consentement des dossiers médicaux informatisés ».
Mémoire de maîtrise, Département Informatique, Université de Sherbrooke, 2011.

[22] Nghi Huynh, Marc Frappier, Amel Mammar, Régine Laleau et Jules Desharnais.
« Validating the RBAC ANSI 2012 Standard Using B ».
Dans Yamine Ait Ameur et Klaus-Dieter Schewe, éditeurs, *Abstract State Machines, Alloy, B, TLA, VDM, and Z : 4th International Conference, ABZ*

2014, Toulouse, France, June 2-6, 2014. Proceedings, pages 255–270. Springer Berlin Heidelberg, 2014.

[23] Nghi Huynh, Marc Frappier, Herman Pooda, Amel Mammar et Régine Laleau.
« SGAC : A patient-centered access control method ».
Dans *2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS)*, pages 1–12, June 2016.

[24] « Inforoute Santé du Canada ».
https://www.infoway-inforoute.ca/fr/.

[25] Daniel Jackson.
« Alloy : A Lightweight Object Modelling Notation ».
ACM Trans. Softw. Eng. Methodol., 11(2):256–290, avril 2002.

[26] Daniel Jackson.
Software Abstractions, Logic, Language, and Analysis.
MIT Press, 2012.

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

[28] Dexter Kozen.
« A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events ».
Information and Computation, 110:366–390, 1994.

[29] Michael Leuschel et Michael J. Butler.
« ProB : an automated analysis toolset for the B method ».
STTT, 10(2):185–203, 2008.

[30] Michael Leuschel, Jérôme Falampin, Fabian Fritz et Daniel Plagge.
« Automated Property Verification for Large Scale B Models with ProB ».
Formal Aspects of Computing, 23(6):683–709, 2011.

[31] Ninghui Li, Ji-Won Byun et Elisa Bertino.
« A critique of the ANSI Standard on Role-Based Access Control ».
Technical Report TR 2005-29, Purdue University, 2005.

[32] Ninghui Li, Ji-Won Byun et Elisa Bertino.
« A Critique of the ANSI Standard on Role-Based Access Control ».
Security Privacy, IEEE, 5(6):41–49, Nov 2007.

[33] Mahdi Mankai et Luigi Logrippo.
« Access control policies : Modeling and validation ».
Dans *5th NOTERE Conference (Nouvelles Technologies de la Répartition)*, pages 85–91, 2005.

[34] W.W. McCune.
« Prover9 and Mace4 ».
http://www.cs.unm.edu/~mccune/prover9.

[35] « Nationell patientöversikt (National Patient Overview) ».
http://www.inera.se/.

[36] « Netcare ».
http://www.albertanetcare.ca/.

[37] Alan C. O'Connor et Ross J. Loomis.
« Economic Analysis of Role-Based Access Control », 2010.
RTI International.

[38] Klaus Pohl, Günter Böckle et Frank van der Linden.
Software Product Line Engineering.
Springer, 2005.

[39] Herman Pooda.
« Évaluation et comparaison des modèles de contrôle d'accès ».
Mémoire de maîtrise, Université de Sherbrooke, 2015.

[40] David J. Power, Mark Slaymaker et Andrew c. Simpson.
« Automatic Conformance Checking of Role-Based Access Control Policies via Alloy ».
Dans *Engineering Secure Software and Systems - Third International Sympo-*

sium, ESSoS 2011, Madrid, Spain, February 9-10, 2011. Proceedings, pages 15–28, 2011.

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

[42] Nafees Qamar, Yves Ledru et Akram Idani.
« Validation of security-design models using Z ».
Dans *Proceedings of the 13th international conference on Formal methods and software engineering*, ICFEM'11, pages 259–274. Springer, 2011.

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

[44] Giovanni Russello, Changyu Dong et Naranker Dulay.
« Authorisation and Conflict Resolution for Hierarchical Domains ».
Dans *POLICY*, pages 201–210. IEEE Computer Society, 2007.

[45] Ravi S. Sandhu, Edward J. Coynek, Hal L. Feinsteink et Charles E. Youmank.
« Role-Based Access Control Model ».
IEEE Computer, 29(2):38–47, 1996.

[46] Ravi S. Sandhu, David F. Ferraiolo et D. Richard Kuhn.
« The NIST Model for Role-Based Access Control : Towards a Unified Standard ».
Dans 5^{th} *ACM Workshop on Role-Based Access Control*, RBAC '00, pages 47–63. ACM, 2000.

[47] Gunther Schmidt et Thomas Ströhlein.
Relations and Graphs : Discrete Mathematics for Computer Scientists.
Springer, EATCS Monographs on Theoretical Computer Science, 1993.

[48] Junaid Haroon Siddiqui et Sarfraz Khurshid.
« Symbolic Execution of Alloy Models ».
Dans Shengchao Qin et Zongyan Qiu, éditeurs, *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods,*

ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings, volume 6991 de *Lecture Notes in Computer Science*, pages 340–355. Springer, 2011.

[49] « Washington Public Health Meaningful Use ».
http://www.doh.wa.gov/ForPublicHealthandHealthcareProviders/HealthcareProfessionsandFacilities/PublicHealthMeaningfulUse.

[50] WSO2.
« Balana ».
https://github.com/wso2/balana.