ANR programme ARPEGE 2008

Systèmes Embarqués et Grandes Infrastructures

Projet SELKIS : Une méthode de développement de systèmes d'information médicaux sécurisés : de l'analyse des besoins à l'implémentation.

ANR-08-SEGI-018

Février 2009 - Décembre 2011

Principles of the coupling between UML and formal notations

Livrable numero 3.1

Akram Idani, Yves Ledru, Jean-Luc Richier, Mohamed-Amine Labiadh
Laboratoire d'Informatique de Grenoble

Frédéric Gervais, Régine Laleau, Jérémy Milhau, Marc Frappier
Université Paris-Est, LACL, IUT Sénart Fontainebleau

Juillet 2010

Table des matières

1 Introduction **3**

2 Synthèse des travaux de couplage d'UML et de notations formelles **4**
2.1 Introduction 4
2.2 Couplage d'UML et B 5

3 Taking into account functional models in the V&V of security design models **7**
3.1 Introduction 7
3.2 Tools for V&V of role-based authorisation constraints 8
3.2.1 USE for the validation of security policies 8
3.2.2 SecureMova 9
3.3 Motivating example 9
3.4 Using testing and verification techniques 10
3.4.1 Some solutions to explore 10
3.4.2 Support of history-based constraints 11
3.5 Conclusion 12

4 Formalisation du contrôle d'accès statique en B **13**
4.1 Principes de la traduction 13
4.2 Formalisation en B du modèle fonctionnel 14
4.2.1 Intégration des opérations de base dans le diagramme de classes 15
4.2.2 Traduction des classes 16
4.2.3 Traduction des attributs de classes 17
4.2.4 Traduction des associations 20
4.2.5 Amélioration du modèle fonctionnel 22
4.3 Transformation du modèle de sécurité 23
4.3.1 Approche proposée 23
4.3.2 Affectation d'utilisateurs aux rôles (relation User_Assignement) 24
4.3.3 Affectation de permissions aux rôles (relation Permission_Assignement) . 26

5 Formalising dynamic access control rules **32**
5.1 Integrating ASTD into the security metamodel 32
5.2 Systematic translation rules from ASTD to Event-B 33
5.3 Event-B Background 34

5.4 ASTD Background 34
5.4.1 ASTD Operators 35
5.4.2 An ASTD Case Study 36
5.4.3 Motivations 37
5.5 Translation 38
5.5.1 Automata 38
5.5.2 Sequence 39
5.5.3 Choice 40
5.5.4 Kleene Closure 41
5.5.5 Synchronization Over a Set of Action Labels 41
5.5.6 Quantified Interleaving 42
5.5.7 Quantified Choice 43
5.5.8 Guard 43
5.5.9 Process call 43
5.6 Animation and Model Checking of the Case Study 43
5.7 Conclusion and Future Work 44

A Spécifications B issues d'un modèle fonctionnel **46**

B Spécifications B issues du modèle de sécurité **50**

Chapitre 1

Introduction

Ce livrable s'inscrit dans le cadre des travaux entrepris par le LIG autour du couplage de modèles graphiques et formels ainsi que ceux proposés par le LACL autour des ASTD (Algebraic State Transition Diagrams). Ces travaux cherchent à tirer profit des outils de preuve et d'animation assistant les méthodes formelles telle que la méthode B en vue d'effectuer des vérifications automatisées de modèles graphiques.

Le travail présenté dans ce livrable se base sur le méta-modèle de sécurité résultant du WP2 et dont l'intention est d'élaborer les liens entre les aspects fonctionnels et sécuritaires à un niveau conceptuel. Les modèles explorés sont exprimés au moyen de notations graphiques UML. L'intérêt de ces notions est qu'elles sont faciles à comprendre et expriment de manière intuitive les besoins fonctionnels et de contrôle d'accès. En revanche, le manque d'outils de vérification assistant ces modèles constitue un frein devant leur application dans un contexte sécuritaire. Ce livrable aborde donc cette problématique en donnant les principes de la formalisation du modèle fonctionnel et de sécurité en B. L'objectif est d'une part, de donner une sémantique formelle précise aux modèles graphiques engendrés par le méta-modèle de sécurité, et d'autre part de vérifier leur correction au moyen d'outils de preuve et d'animation. Pour ce faire, nous proposons de formaliser en B (i) le modèle fonctionnel, (ii) le modèle de contrôle d'accès statique, et (iii) le modèle de contrôle d'accès dynamique.

Dans le chapitre 2 nous présentons un état de l'art des travaux de couplage de notations formelles en B et semi-formelles en UML. Dans le chapitre 3 nous discutons l'intérêt de la prise en compte du modèle fonctionnel lors des activités de vérification et de validation. Ensuite, dans le chapitre 4 nous donnons les principales règles de génération de spécifications B à partir du modèle fonctionnel et du modèle contrôle d'accès statique. Finalement, le chapitre 6 présente les fondements de la formalisation du contrôle d'accès dynamique.

Chapitre 2

Synthèse des travaux de couplage d'UML et de notations formelles

2.1 Introduction

L'idée de coupler les méthodes semi-formelles et formelles n'est pas nouvelle. Elle a été introduite dans les années 90 sous la dénomination des ≪ *approches mixtes* ≫ [FKV91, And95] et a permis de mettre au point un certain nombre de pratiques de modélisation. En effet, nous soulignons, dans ce contexte, l'existence de plusieurs stratégies de couplage des méthodes formelles et semi-formelles ; en voici un bref aperçu [Ida06] :

Stratégie transitionnelle. Cette stratégie se base sur une intégration par dérivation (ou traduction) impliquant une phase de transition de spécifications semi-formelles vers des spécifications formelles équivalentes. Dans ce cadre, le modèle formel résultant peut alors être enrichi, raffiné, etc, et l'application des techniques de vérification et de preuves ainsi que l'utilisation d'outils développés autour des méthodes formelles deviennent alors possibles.

Stratégie évolutive. Il s'agit d'étendre le langage semi-formel en y introduisant des notations formelles. Ce mécanisme permet d'exprimer formellement certaines propriétés du modèle semi-formel. Le système résultant évolue alors d'une simple description semi-formelle à un système conjugué plus précis.

Stratégie d'enrichissement. (appelée aussi *intégration conjointe* dans [And95] et *intégration par extension* dans [Mey01]) : le paradigme objet est perçu comme un mécanisme de structuration. Il s'agit de définir un langage de spécification formelle "à objets"[1] en étendant un langage de spécification formelle existant. Nous citons, à titre de référence, Z++ [Lan92], Object-Z [Smi95] et VDM++ [DvK92].

Stratégie de visualisation. L'objectif de cette stratégie est de construire une vue complémentaire graphique, bien que partielle, sur des spécifications formelles. Cette technique nécessite une phase de correspondance établissant les liens structurels et sémantiques entre un sous-ensemble de constructions du langage formel et ceux du langage semi-formel cible. Les modèles obtenus servent, principalement, de documentation graphique du modèle formel.

Une instance de ces stratégies de couplage est l'intégration des notations UML et formelles en B et en Z qui peut être synthétisée en trois approches principales : (*i*) la dérivation de UML vers une méthode formelles, (*ii*) la dérivation d'une méthode formelle vers UML, et (*iii*) l'intégration conjointe. Dans ce rapport nous discuterons principalement le premier sens du couplage entre UML et notations formelles. En effet, dans ce travail nous nous intéressons à une intégration de

[1] Connus aussi sous l'appellation "langages formels Orientés Objets" [Lan95].

méthodes selon la stratégie transitionnelle. Nous proposons donc des règles de transformation permettant de traduire le modèle fonctionnel (initialement exprimé en UML) ainsi que le modèle de sécurité en une spécification formelle en B et en Z.

2.2 Couplage d'UML et B

UML et B sont deux techniques de spécifications reconnues en génie logiciel ; leur couplage est motivé par le souhait de pouvoir les utiliser ensemble dans un processus de développement de logiciels intégrant à la fois structuration et précision. Dans ce rapport, nous présentons un aperçu de travaux de couplage de ces deux formalismes tout en mettant l'accent sur l'importance de combiner la lisibilité d'une méthode graphique telle que UML et la rigueur d'une méthode formelle telle que B.

Les travaux de dérivation de UML vers B ont commencé avec la thèse de Nguyen [Ngu98] où principalement une stratégie transitionnelle a été adoptée. En effet, les fondements de ce travail ont porté sur un catalogue de règles traduisant des diagrammes de OMT [RBL+90] en B. La motivation principale de cette dérivation était de conserver les acquis des méthodes semi-formelles déjà très répandus et de les renforcer d'un point de vue formel sans que cela ne nécessite une reconstruction du système.

Depuis, plusieurs équipes ont travaillé sur le lien entre UML et la méthode B. Nous pouvons citer notamment en France les travaux du CEDRIC/CNAM à Evry [Mam02, Lal02], du LORIA à Nancy [Mey01, Led02], ou de l'Université de Versailles [Mar02], et à l'étranger des travaux menés au Royaume Uni à l'Université de Southampton [SB06b] ou à l'Imperial College de Londres [LCA04]. D'autres, comme par exemple [BBM03], ont présenté un cadre applicatif de ce couplage (*i.e.* le ferroviaire). Somme toute, l'objectif de tels travaux est de préciser la sémantique d'UML, de compléter ce langage pour augmenter son pouvoir d'expression, et de traduire les spécifications semi-formelles en B pour profiter des outils de la méthode B. Ci-dessous, nous présentons une synthèse des visées de chaque travail avec quelques précisions sur les diagrammes UML concernés.

(a) Travaux du CEDRIC/CNAM à Evry [Mam02, Lal02]

Objectif : Définition d'un environnement formel pour le développement d'applications bases de données qualifiées de sûres. L'apport de ce travail ne porte pas sur la proposition de règles de traduction, mais plutôt l'exploitation et l'adaptation de règles déjà mises au point dans [Ngu98]. Le point de départ est donc une spécification B obtenue par traduction d'un ensemble de diagrammes UML dédiés à la description des applications bases de données. Des tactiques de raffinement sont définies en vue de générer (petit à petit) à partir des spécifications B les différentes tables relationnelles ainsi que les requêtes SQL comme l'insertion d'un nouveau tuple et la suppression d'un ensemble de tuples existants, etc.

Diagrammes UML concernés : le diagramme de classes, les diagrammes états/transitions et les diagrammes de collaborations.

(b) Travaux de l'Université de Versailles [Mar02]

Objectif : Mise en œuvre d'un cadre transformationnel ciblant la prise en compte de contraintes exprimées en OCL. Dans ses travaux, R. Marcano reprend l'essentiel des approches de dérivation de UML vers B en y ajoutant des règles de traduction d'annotations OCL. Une autre contribution intéressante est l'intégration de ces règles au sein de la plate-forme BRILLANT [CPP+05].

Diagrammes UML concernés : Diagrammes de classes et diagramme d'états/transitions augmentés par des contraintes OCL.

(c) Travaux du LORIA à Nancy [Mey01, Led02]

Objectif : Définition d'un cadre théorique et pratique mettant en œuvre des transformations de UML vers B. Les règles de traduction de base appliquées sont inspirées des travaux de K. Lano et de H.-P. Nguyen sur la formalisation en B des concepts de OMT. Toutefois, E. Meyer a apporté, d'une part, des solutions pour la traduction de l'héritage en UML, et d'autre part, une formalisation nouvelle des diagrammes d'états/transitions en distinguant les concepts d'événement et de transition. Le point de départ est une spécification UML (estimée finie) décrivant aussi bien les aspects structurels que comportementaux d'un système. Sur cette base, H. Ledang associe à chaque règle de traduction un schéma de dérivation montrant techniquement comment l'ensemble des constructions du modèle UML (composé d'une palette de diagrammes) sont projetées vers le modèle B. D'autres apports des travaux de H. Ledang sont à signaler comme par exemple : la prise en compte du diagramme des cas d'utilisation et des transitions ayant plusieurs actions en séquences. Par ailleurs, H. Ledang a proposé quelques règles de traduction de certaines constructions OCL telles que les invariants, les pre et post-conditions d'opérations, etc.

Diagrammes UML concernés : Diagrammes de classes, Diagrammes d'états/transitions, diagrammes de collaborations et diagramme de cas d'utilisation.

(d) Travaux de l'Université de Southampton [SB06b, SB04]

Objectif : Ces travaux ont ciblé les projets MATISSE (www.matisse.qinetiq.com) et PUSSEE (www.keesda.com/pussee) et ont eu pour vocation l'obtention d'une spécification B (dite "naturelle") exemptée de constructions liées au mécanisme de traduction et pouvant compliquer les preuves formelles. Par exemple, au lieu de dériver une machine B pour chaque classe, les auteurs génèrent une machine B par paquetage et traduisent toutes les classes du paquetage par des constructions B encapsulées au sein de cette même machine. Les contraintes d'intégrité sont exprimées dans un nouveau formalisme (μB) et sont fondées sur la notation B. Par ailleurs, C. Snook et ses co-auteurs dans [SB06b] apportent un brin d'originalité à cette technique en spécialisant les concept d'UML avec des stéréotypes particuliers indiquant leurs contre-parties B.

Diagrammes UML concernés : Diagramme d'état-transitions et diagramme de classes stéréotypées.

(e) Travaux de l'Imperial College de Londres [LCA04]

Objectif : Ces travaux ont principalement porté sur un ensemble de constructions UML adaptées aux systèmes réactifs. Il s'agit de modèles UML-RSDS (Reactive System Development Support) dont la sémantique se veut précise [Lan98]. L'objectif visé est de vérifier la correction du modèle. Pour ce faire, un mélange de diagrammes UML et de notations formelles (B, SMV) est utilisé.

Diagrammes UML concernés : diagrammes UML-RSDS (diagrammes de classes, diagrammes d'états/transitions, diagrammes de cas d'utilisation).

Ainsi, le sens de dérivation de UML vers B est largement étudié aujourd'hui, se concrétisant par l'existence de plusieurs outils de traduction : *e.g.* `UML2B` [HLMK04], `UML2SQL` [LM00, ML04], `U2B` [SB04], `ArgoUML+B` [LSC03]. Cependant, bien que les fondements théoriques de la dérivation de UML vers B ont fait l'objet de nombreuses études, les outils qui en surgissent n'ont été exploités qu'à un niveau académique.

Chapitre 3

Taking into account functional models in the V&V of security design models

Designing a security policy for an information system is a non-trivial task. Variants of the RBAC model can be used to express such policies as access-control rules associated to constraints. In this chapter, we advocate that currently available tools do not take sufficiently into account the functional description of the application and its impact on authorisation constraints and dynamic aspects of security. We suggest to translate both security and functional models into a formal language, such as B or Z, whose analysis and animation tools will help verify and validate a larger set of security scenarios.

3.1 Introduction

The design of today's information systems must not only take into account the expected functionalities of the system, but also various kinds of non-functional requirements such as performance, usability or security. Security policies are designed to fulfill non-functional requirements such as confidentiality, integrity and availability. They are usually expressed as abstract rules, independently of target technologies. In the past, various access control models have been proposed to define security policies. In this chapter, we focus on role-based access control models (RBAC) [SCFY96, DDR03], including evolutions such as SecureUML [BDL06]. An important feature of such models is the notion of role : permissions are granted to roles which represent a set of users. Moreover users may play several roles with respect to the secure system.

Constraints can be associated to these access control models. They allow to express Separation of Duty properties [CW87], and other properties on roles (e.g. precedence, see Sect. 3.2). Constraints may also link permissions to contextual information, such as the current state of the information system. This is one of the interesting features of SecureUML which groups UML diagrams of the application with security information describing the access control rules. In the remainder, we will refer to the UML diagrams of the application as the *functional model*. The term *security model* will refer to the access control model.

Constraints give flexibility to describe security policies, but result in complex descriptions which need tool support for their verification and validation. Verification checks that the description is consistent, in particular, it must check that constraints are not contradictory, which would result in unsatisfiable policies. Validation checks that the policy corresponds to the user's requirements.

With such complex models, $V\&V$ analyses can become a difficult task. The separation between the functional model and the security model is an interesting solution, based on separation of concerns. However, existing works are mainly interested by the security part. They propose techniques to verify the consistency of an access control policy without taking into account the impact of the functional part. Although it is definitely useful to analyse both models in isolation, interactions between these models must also be taken into account. Such interactions result from the fact that constraints expressed in the security model also refer to information of the functional model. Hence, evolutions of the functional state will influence the security behaviour. Conversely, security constraints can impact the functional behaviour. For example, it is important to consider both security and functional models in order to check liveness properties on the information system. Indeed, it can be the case that security constraints are too strong and block the system.

Only a few tools have been proposed to support these V&V activities on role-based authorisation constraints. In Sect. 3.2, we will review the features of two such tools, USE and SecureMova, which are representative of the current state of the art. In Sect. 3.3, we present a security rule whose validation requires to take into account dynamic aspects of the functional model. Such aspects cannot be investigated with the currently available tools. Finally in Sect. 3.4, we propose solutions based on formal methods such as B or Z, to address these issues.

3.2 Tools for V&V of role-based authorisation constraints

In this section, we briefly review the features of two tools which support the validation of role-based security policies with constraints. In both cases, the constraints are written in OCL [WK98], a language based on first order logic predicates over the constructs of an UML class diagram.

3.2.1 USE for the validation of security policies

The USE tool (UML-based Specification Environment) [GBR07] allows to evaluate OCL constraints on a given object diagram. These constraints are usually invariants associated to the classes of the diagram, but can also stand for pre- or post-conditions if the object diagram represents the initial or final state of some operation. The tool also allows to program a random generator for object diagrams, and to program sequences of object diagrams.

Sohr et al [SDAG08] have adapted this tool for the analysis of security policies. Their work focuses on the security model, i.e. users, roles, sessions and permissions, constrained by OCL assertions. This allows to express properties such as :

- Cardinality : a given role has at most n users.
- Precedence : u may be assigned to role r_2 only if u is already member of r_1.
- Separation of Duty : roles r_3 and r_4 are conflicting.
- Separation of duty for colluding users, e.g. two members of the same family may not take conflicting roles.
- Context-dependent permissions, e.g. only doctors of a patient's current hospital may have access to his/her medical record.

The last two properties cannot be expressed on a pure security model. The model must be augmented with functional information, e.g. some attribute $currentHospital$ should be added to the users. Another possibility is to explicitly include this information in the constraints, e.g. in [SDAG08] all sets of colluding users are listed in the OCL rules. Both cases correspond to extensions of the RBAC+constraint model which do not really scale up. Such information definitely belongs to the functional model.

Sohr et al [SDAG08] report on two kinds of validation activities. An object diagram can be given to the tool, and the tool will check which constraints are violated. The object diagram can be user-defined, randomly generated, or member of a programmed sequence. This allows to detect unsatisfiable constraints, i.e. constraints which are always false. They have also developed a tool named authorisation editor, which implements the administrative, system and review functions of the RBAC standard. The tool is connected to the API of USE so that the constraints of the security policy are checked after each operation. This allows to detect erroneous dynamic behaviour of the security policy. For example, if two roles are constrained both by a precedence and a conflict relations, it will always be impossible to find a sequence of RBAC administrative and system operations which leads to create the second role.

3.2.2 SecureMova

In [BCDE09], Basin et al report on SecureMova, a tool which supports SecureUML+ComponentUML. The tool allows to create a functional diagram, i.e. a class diagram, and to relate it to permission rules. Constraints can be attached to permissions and these constraints may refer to the elements of the functional diagram.

For example, this would allow to express the property that doctors may access a medical record if and only if they are employed by the hospital of the patient. A class $HOSPITAL$ may be defined in the functional model and relations drawn between $HOSPITAL$ and $PATIENT$, and between $HOSPITAL$ and $DOCTOR$. The OCL constraint associated to the permission to access a medical record would navigate through the functional model to retrieve the $PATIENT$ associated to the medical record, and his/her current $HOSPITAL$. It would also retrieve the $DOCTOR$ corresponding to the user asking to access the medical record and retrieve his/her associated $HOSPITAL$. Finally, the constraint will compare these two hospitals.

With SecureMOVA it is possible to ask questions about a current state, i.e. a given object diagram. Such queries return the actions authorized for a given role, or to a given user. They also allow to investigate on overlapping permissions, i.e. permissions which have a common set of associated actions. The tool provides an extensive set of queries over a given model, possibly associated with a given initial state. All reported examples [BCDE09] are of static nature, i.e. they don't allow to sequence actions (either administrative or functional) and check that a given sequence is permitted by the combination of the security and functional models.

3.3 Motivating example

Our motivating example is based on the constraint stated above : "If a doctor wants to access the medical record of a given patient, he must belong to the same hospital as the patient". Let us now consider a malicious doctor, who wants to access the information of a patient in another hospital. Since the patient and the doctor belong to different hospitals, the doctor will not be permitted to access this information. In order to validate the rules of the security policy, one may try several typical situations and query about the permitted/forbidden actions. Using a tool such as SecureMova, one would provide an object diagram od_1 with one doctor and one patient linked to two different hospitals, and query if the doctor may perform action $readMedicalRecord$ on the patient's medical record. The tool would answer that the doctor is not authorized to perform this action.

Further validation of this security policy should explore dynamic aspects of the policy. For example, is it possible for this malicious doctor to access the patient's information ? Using only static tools, one can check that, given an object diagram od_2 where the malicious doctor belongs to the same hospital as the patient, he will be granted this access. The next question to investigate is : does there exist a sequence of actions which leads a malicious doctor to belong

to the same hospital as the patient ? This requires to animate a sequence of actions which leads from od_1 to od_2. Such a sequence will presumably call an intermediate operation $joinHospital$ which will link the malicious doctor to the hospital of the patient. Here the dynamic analysis will allow to identify these intermediate actions and check which role has permission to perform these actions.

Another way to group the malicious doctor and the patient in the same hospital is to transfer the patient in the hospital of the doctor. In this second sequence, one should investigate who has the permission to perform such a transfer.

This simple example shows that the validation of a security policy may require dynamic analyses to identify sequences of actions leading to an unwanted state. Moreover, these sequences of actions are not restricted to the standard RBAC functions and may refer to operations defined in the functional model. This is actually the case when constraints referring to the functional model are expressed on permissions. Current tools, such as the ones presented in Sect. 3.2, which focus on static queries or on the dynamic execution of the sole RBAC functions are not sufficient to perform such dynamic investigations. In [SDAG08], Sohr also proposes the use of Linear Temporal Logic (LTL) as a way to express a larger set of constraints, and then uses a theorem prover to detect erroneous dynamic behaviours. Still, this work does not consider the functional model and needs to be extended to address the case of our malicious doctor.

3.4 Using testing and verification techniques

The example of Sect. 3.3 shows that there is a need for dynamic analyses when designing a security policy. Moreover, when the security policy refers to the functional model through the use of constraints, the dynamic analysis should not only cover the RBAC standard functions, but also take into account the behaviour of the functional model.

Dynamic analyses can take two forms : tests and verifications. Tests correspond to the execution of a sequence of actions on the security and functional models, or on their implementations. The test sequence is either defined by the security policy designer, possibly on the basis of use cases, or it may be the output of a test generation tool on the basis of some coverage of the models. Test can contribute to both validation and verification activities. Tests based on use cases correspond to the validation activity because they contribute to show that the security policy meets the users/customers needs. Tests based on model coverage contribute to verification. They can check that all covered behaviours of the model will respect some global properties of the security policy like separation of duties. Tests also contribute to detect unsatisfiable constraints because such constraints may forbid any state different from the empty state.

Tests can only check a limited number of behaviours. When absolute guarantees are needed, verification techniques should be considered. Verification techniques include model-checking and symbolic proof techniques. Both techniques are of interest in the verification of a security policy. Proof techniques can show the existence of some state, and hence prove that constraints are satisfiable, or establish that some property, like separation of duty, is an invariant of the model. Model-checking is based on model exploration, and can be used to find a sequence of actions leading to a given state or property. In our motivating example, model-checking tools should be experimented to find a path between od_1 and od_2.

3.4.1 Some solutions to explore

Testing techniques require the availability of executable models or implementations. Security policies based on RBAC can easily be made executable, as demonstrated by Sohr in his authorisation editor [SDAG08]. Executability of the functional model can be achieved in two ways :

either by providing an implementation of the model which can interface with the contextual constraints of the security model, or by providing an executable model. Providing an implementation makes sense in a context where the functional system is designed first, without considering security aspects, and where a security policy must be designed later for this application. It also makes sense during a maintenance phase where a given implemented security policy must evolve. Some prototypes of RBAC can be coupled with an existing implementation. For example, the MotOrBAC tool provides an API between its security engine and the application [ACCBCB08].

The other way is to get an executable functional model. In the case of USE or SecureMova, the model is expressed as a class diagram combined with OCL predicates. In order to turn UML methods into executable ones, one need to provide an implementation of the methods. Actually, USE allows to define a body for each method using an imperative language based on OCL. It seems that this feature was not explored in [SDAG08] and might be interesting to investigate. Another way is to animate the methods based on their pre- and post-conditions. We don't know of tools which support this approach for OCL, but they exist for formal languages such as Z[ISO02] or B[Abr96b]. For example, the Jaza tool [Utt05] can be used to animate the operations of a Z specification.

The B language actually appears as an interesting option. Several tools have been defined to translate UML models into B specifications ; they show at least the feasibility of such translations [ML06, SB06c]. Regarding the security model, Sohr[SDAG08] has already shown that it can be specified in UML+OCL. Since the B language is based on the same principles as OCL (first order predicate logic and set theory), it is possible to propose a similar translation of the security model into B specifications. The B specifications produced from both security and functional models can then be analysed using either animation tools such as ProB [LB08] or proof tools such as Atelier-B [1]. ProB also includes model-checking facilities which can be of interest to search for malicious sequences of operations.

3.4.2 Support of history-based constraints

The constraints expressed in OCL only refer to a given instant of time. When expressing history-based constraints, it is necessary to refer to several distincts instants of time. For example, let us consider the following rule : "If a patient has left the hospital, all doctors belonging to the hospital during the patient's stay will keep read access to his medical record.". If we want to express this rule as a read permission associated to an OCL constraint, we need to extend the functional model with information about past states, and this information is for security's sake only. This goes against separation of concerns.

In [SDAG08], Sohr suggests the use of TOCL, an extension of OCL with temporal operators. Although this provides a way to express the history based constraints, it appears that no tool is currently available to support the use of this formalism. Sohr suggests to translate TOCL into LTL as future work.

Another approach in order to model such history-based constraints and dynamic aspects of a security policy is process algebra. Process algebra can be used to model workflows of actions and all ordering and security constraints related to a dynamic security policy. Based on process algebra and Statecharts, the ASTD notation (Algebraic State Transition Diagrams [FGL+08]) is a formal, graphical and state-based specification language. A security rule can be described using a hierarchical automata notation. Rules, expressed as processes, can then be combined using operators such as sequence, choice, synchronization or interleaving. Guards can also be used in order to model security constraints. ASTD allows processes to be quantified. Quantifications reduce model complexity by defining, for instance, the behavior of all entities of a class using a single quantified ASTD. ASTD models are also executable using *i*ASTD [SM+10], an interpreter

[1] http://www.atelierb.eu

for ASTD. *i*ASTD efficiently determines if actions can be executed by the model and computes the ASTD state after the execution. Combined with graphical representation of states it provides a visual animation, helping to validate the model. Systematic translation of ASTD specification to B or Event-B [MFGL10] can also be used to perform proofs and model checking and further verify the specification of dynamic rules of the security policy.

3.5 Conclusion

This chapter has addressed the verification and validation (V&V) of security policies. These are essential activities when designing or modifying a security policy. We have focussed on policies based on access-control models with constraints. These constraints may concern the elements of the security model, e.g. to express incompatible roles, but may also refer to elements of the functional model controlled by the security policy. In many cases, the state of the functional model is used as a context to grant access rights.

Separation of concerns suggests to treat the functional and security models in isolation. Unfortunately, when constraints establish a link between these models, V&V activities must consider both the security model and the functional model. In Sect. 3.2, we have stated that current V&V tools cover static aspects of the functional model, as well as static and dynamic aspects of the security model. A motivating example, presented in Sect. 3.3 has illustrated the need for dynamic analyses which take into account both models. Such analyses can be conducted as testing or verification (model-checking or proof). Both kinds of analyses require to describe the behaviours of both models.

Based on these considerations, it appears that formal languages can provide an interesting framework to support these activities. Such languages are often supported by mature proof, model-checking and animation tools. Our current work goes into that direction. This approach is currently investigated in the Selkis project, funded by the French national research agency. The project includes the translation of security models, based on RBAC with constraints, into the Z and B formal languages, in order to allow the use of their associated tools for static and dynamic analyses.

Chapitre 4

Formalisation du contrôle d'accès statique en B

Comme évoqué dans le chapitre 1, plusieurs travaux ont cherché à définir des règles de transformation d'UML en B. Somme toute, ces travaux proposent des règles complémentaires qui permettent de couvrir une grande partie des construction UML utilisées dans les diagrammes de classes et d'états/transitions. Dans [IL10] nous avons proposé une plateforme IDM permettant de combiner des règles alternatives provenant de ces approches. L'objectif en est de disposer d'un cadre outillé supportant une grande variété de règles de transformation et de pouvoir, par conséquent, sélectionner les règlses les mieux adaptées aux besoins de la transformation. Dans la suite nous allons commencer par présenter un ensemble de règles adaptées à la traduction du modèle fonctionnel. Nous présenterons ensuite la technique que nous proposons pour formaliser une politique de contrôle d'accès associée à ce modèle fonctionnel.

4.1 Principes de la traduction

Les termes "*shallow embedding*" et "*deep embedding*" [BGG+93, WN04] sont souvent utilisés pour désigner un changement ou une intégration de formalismes. Le premier terme désigne une traduction directe d'un modèle source vers un modèle cible, alors que le second terme désigne une traduction de formalismes aboutissant à des constructions qui représentent des types de données. En suivant une logique semblable, P. Facon et R. Laleau, dans [FL95] distinguent deux approches de dérivation de spécifications formelles à partir de spécifications semi-formelles : l'approche *interprétée* et l'approche *compilée*. Notons qu'une illustration détaillée et complète de l'application de ces deux approches à une dérivation de UML vers B est présentée dans [Lal02].

L'approche compilée (ou par traduction). Cette approche consiste à donner des règles permettant de traduire un modèle semi-formel directement dans un langage formel. Nous citons en guise d'exemple la règle suivante de traduction du concept d'héritage extraite de [FL95] :

Règle de traduction : faire une machine abstraite B par classe de la hiérarchie avec un lien *uses* de chaque spécialisation vers la généralisation, plus une machine abstraite globale qui inclut (*includes*) toutes les autres machines abstraites et qui en fait représente l'interface de la hiérarchie.

La plupart des travaux de dérivation de UML vers B adoptent une approche compilée [Mam02, Lal02, Mey01, SB06b] fondée sur un ensemble de règles de traduction. L'intérêt majeur d'une

telle technique est qu'elle est réalisée par une traduction directe d'un modèle semi-formel à une spécification formelle. Cependant, son principal inconvénient est que la sémantique des correspondances entre B et UML n'est pas explicitement formalisée étant donné qu'elle est cachée au niveau des règles de traduction.

L'approche interprétée (ou par méta-modélisation). Cette approche est fondée sur un mélange des spécifications formelles issues aussi bien des concepts du méta-modèle que des éléments du modèle semi-formel sujet de la traduction. Il s'agit précisément de proposer, une fois pour toutes, une formalisation du méta-modèle du modèle semi-formel ; et d'effectuer, ensuite, la traduction de la partie propre à chaque application et l'injecter au niveau de la formalisation du méta-modèle. Par exemple, pour traiter l'héritage, une classe n'est plus comme précédemment immédiatement traduite par une machine abstraite. Nous avons à la place une seule machine abstraite générique qui comprend :

– L'ensemble de tous les objets du système,
– Une fonction associant à chaque nom de classe, les ensembles d'objets instances de la classe,
– Une fonction associant à chaque nom d'association la relation correspondante entre objets, et
– Une fonction associant à chaque objet et attribut la valeur correspondante.

Notre point de vue. Contrairement à l'approche *interprétée*, l'approche *compilée* se veut plus directe car les spécifications formelles qui en découlent reflètent de manière naturelle et explicite les éléments du modèle (ou de l'application) en question. Cependant, pour des besoins propres à des systèmes particuliers, l'approche interprétée permet de préciser et de clarifier les éléments des méta-modèles qui leurs sont dédiés.

Dans notre travail, nous dissocions la formalisation en B du modèle fonctionnel de celle du modèle de contrôle d'accès. Notre processus de dérivation de spécifications formelles produit ainsi deux modèles B :

(*i*) Un premier modèle B issu du modèle fonctionnel via une approche compilée. Les spécifications formelles qui en résultent peuvent être enrichies par la prise en compte de contraintes fonctionnelles et servent pour vérifier la correction du modèle fonctionnel.

(*ii*) Un deuxième modèle B qui représente la politique de sécurité en vue de contrôler l'accès aux diverses entités fonctionnelles. Ce modèle est généré en suivant une approche interprétée. En effet, nous traduisons une fois pour toute, notre méta-modèle de sécurité en B. Nous traduisons ensuite la partie propre à la politique de contrôle d'accès et nous l'injectons dans le modèle B issu du méta-modèle de sécurité.

Les liens entre ces deux modèles B sont explicités par les opérations. En effet, les opérations issues du modèle fonctionnel permettent d'effectuer des opérations incontrôlées, alors que celles issues du modèle de sécurité ont pour objectif d'effectuer ce contrôle.

4.2 Formalisation en B du modèle fonctionnel

Dans le but d'illustrer les diverses règles de traduction d'un diagramme de classes UML en B, nous allons nous baser sur un extrait du diagramme de classes de l'étude de cas IFREMMONT (figure 4.1). Dans ce diagramme nous considérons uniquement les classes **Patient** et **ManagementAct** avec une relation de composition indiquant les actes de soins associés à chaque patient.

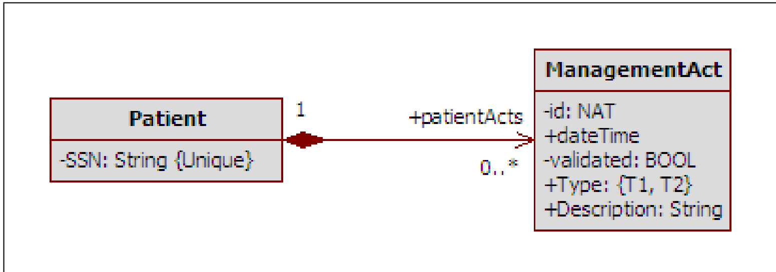


Fig. 4.1 – Extrait du diagramme de classes de l'étude de cas IFREMMONT

4.2.1 Intégration des opérations de base dans le diagramme de classes

Les spécifications B que nous allons produire à partir du diagramme de classes ont vocation à être opérationnelles. Cela nous permettra de les animer dans le but de voir l'évolution de l'état du système et d'observer l'effet qu'un scénario d'exécution pourrait avoir sur les propriétés invariantes. Le point d'entrée à ces spécifications B est donc les opérations qu'elles fournissent. Elles peuvent être des constructeurs/destructeurs d'instances, des constructeurs/destructeurs de liens, des getters/setters d'attributs, ou des getters de liens. Nous faisons donc évoluer, d'emblée, le diagramme de classes en y intégrant toutes ces opérations. La figure 4.2 présente l'évolution d'un diagramme de classes composé de deux classes (A et B), d'une association (R) entre ces classes et d'un attribut[1].

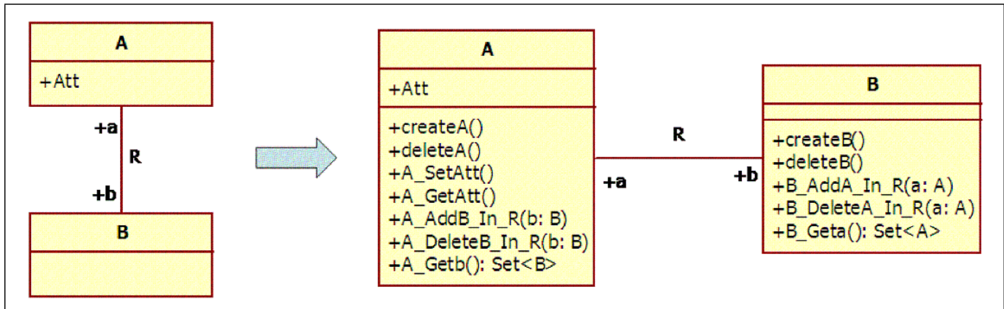


Fig. 4.2 – Intégration des opérations de base dans le diagramme de classes

- Constructeurs d'instances : createA, createB
- Destructeurs d'instances : deleteA, deleteB
- Constructeurs de liens : A_AddB_In_R, B_AddA_In_R
- Destructeurs de liens : A_DeleteB_In_R, B_DeleteA_In_R
- Getter d'attributs : A_GetAtt
- Setter d'attributs : A_SetAtt
- Getters de liens : A_Getb, B_Geta

Le corps de chacune de ces opérations de base sera spécifié directement en B et sera discuté au fur et à mesure que nous présentons la traduction des éléments structurels auxquels ces opérations sont rattachées.

Notons que dans la suite nous proposons de générer une machine B unique, que nous appelons **Functional_Model**, en vue de traduire tous les concepts d'un diagramme de classes.

[1] La visibilité des attributs n'est pas importante à ce niveau.

4.2.2 Traduction des classes

La notion d'ensemble abstrait en B (abstract sets) représente une abstraction d'un ensemble d'objets du monde réel. Cette définition est proche de la notion de classe en UML et est d'ailleurs utilisée par toutes les approches de transformations d'UML en B pour formaliser des classes UML. Néanmoins, pour affiner cette traduction, [Ngu98, Mey01] considèrent qu'un ensemble abstrait B pourrait représenter des objets instances possibles d'une classe, alors que les objets effectifs (ou instances existantes) devraient être formalisés autrement. L'objectif principal de cette vision est de permettre l'introduction de constructeurs et destructeurs d'instances effectives. Par exemple, [Lal02] traduit une classe **Client** par :

- (i) L'ensemble abstrait[2] $CLIENT$ désignant l'ensemble des clients possibles.
- (ii) La variable[3] $Client$ désignant l'ensemble des clients effectifs. Et,
- (iii) L'invariant[4] : $Client \subseteq CLIENT$.

Des variétés de cette traduction existent. Par exemple, C. Snook *et al.*, dans [SB04] considèrent l'invariant "$Client \in \mathbb{P}(CLIENT)$" au lieu de "$Client \subseteq CLIENT$". Hung Ledang [Led02] rajoute une couche pour distinguer l'ensemble de tous les objets possibles $OBJECTS$ et définit l'ensemble des instances possibles par une constante[5] incluse dans l'ensemble abstrait $OBJECTS$.

Nous choisissons de mettre en oeuvre la traduction proposée par [Ngu98, Mey01, Lal02]. Celle-ci introduit, dans la machine **Functional_Model**, les structures associées à une classe **A** de la manière suivante :

MACHINE
  Functional_Model
SETS $\mathcal{P}_A$ `/*Ensemble des instances possibles*/`
VARIABLES $\mathcal{E}_A$ `/*Ensemble des instances effectives*/`
INVARIANT $\mathcal{E}_A \subseteq \mathcal{P}_A$
INITIALISATION $\mathcal{E}_A := \emptyset$

Nous proposons ensuite de générer, dans la clause **OPERATIONS** de cette machine, les opérations de base **createA** and **deleteA** permettant la création et la suppression d'instances effectives.

createA(obj) $\widehat{=}$
  PRE obj $\in \mathcal{P}_A \wedge$ obj $\notin \mathcal{E}_A$ **THEN**
    $\mathcal{E}_A := \mathcal{E}_A \cup \{obj\}$
    `/* Initialisation des attributs obligatoires */`
  END

deleteA(obj) $\widehat{=}$
  PRE obj $\in \mathcal{P}_A \wedge$ obj $\in \mathcal{E}_A$ **THEN**
    $\mathcal{E}_A := \mathcal{E}_A - \{obj\}$
    `/* Suppression des valeurs des attributs ; et`
    `Mise à jour des liens entrepris avec d'autres`
    `instances de classes */`
  END

[2] clause SETS.
[3] clause VARIABLES.
[4] clause INVARIANT.
[5] Clause CONSTANTS.

Dans le but d'illustrer cette traduction nous présentons, ci-dessous, les structures associées à la classe **Patient** (figure 4.1) dans la machine **Functional_Model** :

MACHINE
  Functional_Model
SETS
  PATIENTS
VARIABLES
  Patients
INVARIANT
  Patients $\subseteq$ PATIENTS
INITIALISATION
  Patients := $\emptyset$

Quant aux opérations de base **createPatient** and **deletePatient** permettant la création et la suppression d'instances effectives de la classe Patient, elles sont données dans la figure 4.3.

createPatient(obj) $\hat{=}$
  PRE obj $\in$ PATIENTS $\wedge$ obj $\notin$ Patients **THEN**
    Patients := Patients $\cup$ {obj}
    `/* Initialisation des attributs obligatoires */`
  END

deletePatient(obj) $\hat{=}$
  PRE obj $\in$ PATIENTS $\wedge$ obj $\in$ Patients **THEN**
    Patients := Patients $-$ {obj}
    `/* Suppression des valeurs des attributs ; et`
    `Mise à jour des liens entrepris avec d'autres`
    `instances de classes */`
  END

FIG. 4.3 – Constructeur et destructeur de la classe Patient

Notons que le constructeur et le destructeur doivent mettre en œuvre des traitements particuliers, notamment l'initialisation d'un attribut obligatoires pour chaque instance créée, ou la mise-à-jour des liens entrepris avec d'autres instances de classes en cas de suppression, etc. Les lignes commentées dans le corps du constructeur et du destructeur seront discutées lors de la traduction des attributs et des associations.

4.2.3 Traduction des attributs de classes

A. Types des attributs

Les types des attributs, autres que les types de base B (*i.e.* NAT, BOOL, etc) et les types classe, seront traduits par des ensembles abstraits dans la clause SETS. Par exemple, le type `String` de l'attribut `SSN` est traduit par l'ensemble abstrait `LesSSNs`. Ceci permet de représenter l'ensemble des chaînes de caractères possibles pour `SSN`.

Les types énumérés seront traduits par des ensembles énumérés dans la clause SETS. Par exemple, le type de l'attribut `Type` de la classe ManagementAct donnera lieu à l'ensemble énuméré LesTypes $= \{T1, T2\}$.

B. Attributs mono-valués

Un attribut dans une classe est généralement traduit par une relation fonctionnelle $\mathcal{R}$ associant l'ensemble des instances effectives et le type de l'attribut. Par exemple, l'attribut Att de la classe A de la figure 4.2 donne lieu à une variable Att typée ainsi au niveau de l'invariant de typage :

$$\text{A_Att} \in \mathcal{E}_A \ \mathcal{R} \ \mathcal{T}_{\text{Att}}$$

Où $\mathcal{E}_A$ correspond à l'ensemble des instances effectives de la classe A et $\mathcal{T}_{\text{Att}}$ au type de l'attribut Att. Les spécialisations de la relation $\mathcal{R}$ dépendent de la nature de l'attribut : obligatoire ou optionnel, unique ou non. Le tableau 4.1 donne les différentes valeurs de $\mathcal{R}$.

	Optionnel	Obligatoire
Unique	$\rightarrowtail\!\!\!\!+$	$\rightarrowtail$
Non unique	$\nrightarrow$	$\rightarrow$

Tab. 4.1 – Relations fonctionnelles issues des attributs de classes

Par exemple, l'attribut `SSN` de la classe Patient (figure 4.1) est un attribut mono-valué, optionnel et unique. Il sera donc traduit par une injection partielle comme suit :

$$\text{Patient_SSN} \in \mathit{Patients} \rightarrowtail\!\!\!\!+ \text{LesSSNs}$$

L'attribut `Validated` de la classe ManagementAct est un attribut mono-valué, obligatoire et non-unique ; il sera donc traduit par une fonction totale :

$$\text{ManagementAct_Validated} \in \mathit{ManagementActs} \rightarrow \mathbf{BOOL}$$

B. Opérations de base

Les getters et les setters

Des getters et des setters sont produits pour chaque attribut de classe. Ils permettent de lire et de mettre à jour les attributs qu'ils soient publics ou privés. Lors de la traduction du modèle fonctionnel nous ne faisons aucune distinction entre un attribut privé ou un attribut public. En effet, c'est le modèle de sécurité qui définit la politique d'accès à ces attributs. Soit, par exemple, l'attribut Att (figure 4.2) défini par : $\text{A_Att} \in \mathcal{E}_A \ \mathcal{R} \ \mathcal{T}_{\text{Att}}$; alors les opérations de lecture (A_GetAtt) et d'écriture (A_SetAtt) sont spécifiées en B comme suit :

```
A_SetAtt(obj) =
   PRE obj ∈ P_A ∧ obj ∈ E_A THEN
      ANY att WHERE
         att ∈ T_Att [∧ att ∉ ran(R)]
       THEN
         R(obj) := att
       END
   END ;
```

```
att ← A_GetAtt(obj) =
   PRE obj ∈ P_A ∧ obj ∈ E_A THEN
        att := R(obj)
   END ;
```

L'ensemble $\mathcal{P}_A$ correspond à l'ensemble abstrait représentant les instances possibles de A. Le prédicat $[\wedge\ att \notin \mathbf{ran}(\mathcal{R})]$ correspond à une condition d'unicité et est défini en particulier pour les attributs avec une contrainte {Unique} comme l'attribut SSN de la classe Patient. En guise d'exemple nous présentons ci-dessous les getters et les setters de l'attribut SSN.

```
ssn ← patient_GetSSN(obj) =
    PRE obj ∈ PATIENTS ∧ obj ∈ Patients THEN
        ssn := patient_SSN(obj)
    END ;
```

```
patient_SetSSN(obj) =
    PRE obj ∈ PATIENTS ∧ obj ∈ Patients THEN
        ANY ssn WHERE
            ssn ∈ LesSSNs ∧ ssn ∉ ran(patient_SSN)
         THEN
            patient_SSN(obj) := ssn
         END
    END ;
```

Prise en compte des attributs obligatoires dans les constructeurs

Outre les getters et les setters générés automatiquement pour tous les attributs, les constructeurs d'instances effectives doivent intégrer la valeur par défaut d'un attribut obligatoire. On suppose donc que dans le diagramme de classes UML, un attribut avec une valeur par défaut est un attribut obligatoire. Par exemple, la valeur par défaut de l'attribut `Validated` de ManagementAct est `false`. Le constructeur `createManagementAct` sera donc comme suit :

```
createManagementAct(obj) ≙
    PRE obj ∈ MANAGEMENTACTS ∧ obj ∉ ManagementActs THEN
        ManagementActs := ManagementActs ∪ {obj}
        ManagementAct_Validated(obj) := FALSE
    END
```

FIG. 4.4 – Constructeur de ManagementAct

Mise-à-jour des attributs dans les destructeurs

La suppression d'une instance effective *obj* d'une classe A de l'ensemble des instances effectives $\mathcal{E}_A$ doit être suivie de la suppression de la valeur de chaque attribut A_Att associé à *obj*. Ceci est considéré dans le destructeur deleteA comme suit :

```
deleteA(obj) ≙
    PRE obj ∈ 𝒫_A ∧ obj ∈ ℰ_A THEN
        ℰ_A := ℰ_A − {obj} ||
        A_Att := {obj} ⩤ A_Att
    END
```

Par exemple, la figure 4.5 présente le destructeur de la classe Patient en prenant en compte la mise-à-jour de la relation patient_SSN issue de l'attribut SSN.

```
deletePatient(obj) ≙
   PRE obj ∈ PATIENTS ∧ obj ∈ Patients THEN
      Patients := Patients − {obj} ||
      patient_SSN := {obj} ⩤ patient_SSN
   END
```

Fig. 4.5 – Destructeur de la classe Patient prenant en compte l'attribut SSN

4.2.4 Traduction des associations

Dans le cas général, une association **assos** liant deux classes A et B est traduite par une relation fonctionnelle entre les ensembles des instances effectives issus de ces classes :

$$assos \in \quad \mathcal{E}_A \;\; \mathcal{R} \;\; \mathcal{E}_B$$

Les spécialisations de la relation $\mathcal{R}$ dépendent des multiplicités des deux côtés de l'association *assos*. Par exemple, des multiplicités $*$ et 1 respectivement du côté de A et de B, donnent lieu à une fonction totale $\rightarrow$ de $\mathcal{A}$ vers $\mathcal{B}$. Le tableau 4.2 extrait de [Ida06] liste les diverses relations fonctionnelles en B en leur associant des multiplicités UML.

		du côté de A		Du côté de B	
Spécialisations de $\mathcal{R}$	expression	min	max	min	max
Relation	$\leftrightarrow$	0	*	0	*
Fonction partielle	$\nrightarrow$	0	*	0	1
Fonction totale	$\rightarrow$	0	*	1	1
Injection partielle	$\rightarrowtail\!\!\!\!\!+$	0	1	0	1
Injection totale	$\rightarrowtail$	0	1	1	1
Surjection partielle	$\twoheadrightarrow\!\!\!\!\!+$	1	*	0	1
Surjection totale	$\twoheadrightarrow$	1	*	1	1
Bijection partielle	$\rightarrowtail\!\!\!\!\twoheadrightarrow\!\!\!\!\!+$	1	1	0	1
Bijection totale	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	1	1	1	1

Tab. 4.2 – Table des multiplicités associée aux spécialisations de relations fonctionnelles en B

Cependant, ce tableau ne couvre pas tous les cas de figure comme par exemple des multiplicités fixes ou des multiplicités $1..*$ des deux extrémités de *assos*. Il devient donc opportun de traduire chaque côté de l'association *assos* de manière spécifique. Pour ce faire, nous présentons le tableau 4.3 extrait de [Oss06]. Dans cette approche, l'invariant spécifiant *assos* par une relation entre $\mathcal{E}_A$ et $\mathcal{E}_B$ est remplacé par un nouveau prédicat $Cmult1(assos) \;\wedge\; Cmult2(assos)$ (2eme et 3eme colonne du tableau 4.3). Ainsi, on obtient l'invariant suivant pour des multiplicités $1..*$ des deux extrémités de *assos* :

$$assos \in \mathcal{E}_A \leftrightarrow \mathcal{E}_B \wedge \mathrm{dom}(assos) = \mathcal{E}_A \wedge \mathrm{ran}(assos) = \mathcal{E}_B$$

Ainsi, la traduction de l'association *patientActs* liant la classe *Patient* à la classe *ManagementAct* (figure 4.1) peut être effectuée de deux manières. La première suit le tableau 4.2 et produit simplement une fonction totale de l'ensemble *ManagementActs* vers l'ensemble *Patients* :

$$patientActs \in ManagementActs \rightarrow Patients$$

La deuxième traduction suit le tableau 4.3 et produit l'invariant suivant :

$$patientActs \in Patients \leftrightarrow ManagementActs \;\wedge$$
$$patientActs^{-1} \in ManagementActs \rightarrow Patients$$

L'avantage de cette deuxième traduction est qu'elle respecte le sens de navigabilité de l'association (*i.e.* de la classe *Patient* vers la classe *ManagementAct*).

Multiplicité	Cmult2(assos) (du côté de B)	Cmult1(assos) (du côté de A)
* ou 0..*	$assos \in \mathcal{E}_A \leftrightarrow \mathcal{E}_B$	$assos^{-1} \in \mathcal{E}_B \leftrightarrow \mathcal{E}_A$
0..1	$assos \in \mathcal{E}_A \nrightarrow \mathcal{E}_B$	$assos^{-1} \in \mathcal{E}_B \nrightarrow \mathcal{E}_A$
1	$assos \in \mathcal{E}_A \rightarrow \mathcal{E}_B$	$assos^{-1} \in \mathcal{E}_B \rightarrow \mathcal{E}_A$
1..*	$assos \in \mathcal{E}_A \leftrightarrow \mathcal{E}_B \wedge$ $\mathrm{dom}(assos) = \mathcal{E}_A$	$assos^{-1} \in \mathcal{E}_B \leftrightarrow \mathcal{E}_A \wedge$ $\mathrm{ran}(assos) = \mathcal{E}_B$
n	$assos \in \mathcal{E}_A \leftrightarrow \mathcal{E}_B \wedge$ $\forall x.(x \in \mathcal{E}_A \Rightarrow \mathrm{card}(assos[\{x\}]) = n)$	$assos^{-1} \in \mathcal{E}_B \leftrightarrow \mathcal{E}_A \wedge$ $\forall x.(x \in \mathcal{E}_B \Rightarrow \mathrm{card}(assos^{-1}[\{x\}]) = n)$
0..n	$assos \in \mathcal{E}_A \leftrightarrow \mathcal{E}_B \wedge$ $\forall x.(x \in \mathcal{E}_A \Rightarrow \mathrm{card}(assos[\{x\}]) \leq n)$	$assos^{-1} \in \mathcal{E}_B \leftrightarrow \mathcal{E}_A \wedge$ $\forall x.(x \in \mathcal{E}_B \Rightarrow \mathrm{card}(assos^{-1}[\{x\}]) \leq n)$
1..n	$assos \in \mathcal{E}_A \leftrightarrow \mathcal{E}_B \wedge$ $\mathrm{dom}(assos) = \mathcal{E}_A \wedge$ $\forall x.(x \in \mathcal{E}_A \Rightarrow \mathrm{card}(assos[\{x\}]) \leq n)$	$assos^{-1} \in \mathcal{E}_B \leftrightarrow \mathcal{E}_A \wedge$ $\mathrm{ran}(assos) = \mathcal{E}_B \wedge$ $\forall x.(x \in \mathcal{E}_B \Rightarrow \mathrm{card}(assos^{-1}[\{x\}]) \leq n)$
n..*	$assos \in \mathcal{E}_A \leftrightarrow \mathcal{E}_B \wedge$ $\forall x.(x \in \mathcal{E}_A \Rightarrow \mathrm{card}(assos[\{x\}]) \geq n)$	$assos^{-1} \in \mathcal{E}_B \leftrightarrow \mathcal{E}_A \wedge$ $\forall x.(x \in \mathcal{E}_B \Rightarrow \mathrm{card}(assos^{-1}[\{x\}]) \geq n)$
n..m	$assos \in \mathcal{E}_A \leftrightarrow \mathcal{E}_B \wedge$ $\forall x.(x \in \mathcal{E}_A \Rightarrow n \leq \mathrm{card}(assos[\{x\}]) \leq m)$	$assos^{-1} \in \mathcal{E}_B \leftrightarrow \mathcal{E}_A \wedge$ $\forall x.(x \in \mathcal{E}_B \Rightarrow n \leq \mathrm{card}(assos^{-1}[\{x\}]) \leq m)$

où n et m sont des entiers positifs tels que $n \geq 2 \wedge n \leq m$.

TAB. 4.3 – Prise en compte des multiplicités pour la dérivation d'une association

Génération des opérations de base

Dans la section 4.2.1 nous avons présenté les opération de base qui permettent d'enrichir le diagramme de classes avant sa traduction en B. Les opérations relatives à la manipulation d'une association R entre deux classes A et B sont :
- Constructeurs de liens : A_AddB_In_R, B_AddA_In_R
- Destructeurs de liens : A_DeleteB_From_R, B_DeleteA_From_R
- Getters de liens : A_Getb, B_Geta

Notons que la génération automatique de ces opérations est conditionnée par divers paramètres : la navigabilité, la composition ou l'existence d'une multiplicité minimale égale à 1.

(*i*) Prise en compte de la navigabilité : une association R navigable uniquement de A vers B indique qu'il est possible, à partir d'une instance x de A, de retrouver les instances de B liées à x via R. De ce fait, la spécification B intègre uniquement les opérations : A_AddB_In_R, A_DeleteB_In_R et A_Getb[6]. Ces opérations permettent d'agir sur R en ayant une instance donnée de A. Si R est navigable dans les deux sens alors on aura dans la spécification B toutes les opérations qui permettent d'agir sur R à partir d'instances de A ou de B.

(*ii*) Prise en compte de la composition ou de l'existence d'une multiplicité minimale égale à 1 : Si R est une association de composition indiquant que des objets de type A sont composés d'objets de type B, alors on considère que l'ajout et la suppressions d'instances de B est contrôlé par A. De ce fait, la spécification B n'intégrera pas les opérations B_AddA_In_R et B_DeleteA_In_R. Cette même règle s'applique si le diagramme de classes indique que chaque instance de B est nécessairement rattachée à au moins une instance de A via R. Dans ces deux cas on ne génère pas le constructeur de B (Create_B).

Dans la figure 4.1, l'association patientActs est une composition navigable uniquement de Patient vers ManagementAct. Par conséquent, on ne produit pas dans la spécification B le constructeur

[6] Si le nom de rôle est explicité dans le diagramme de classes alors il apparaît dans l'étiquette du getter, sinon le getter serait A_GetR.

CreateManagementAct. Cela permettra d'interdire la création d'instances isolées de *ManagementAct*. Les opérations de manipulation de patientActs seront donc : patient_AddpatientActs, patient_DeletepatientActs et patient_GetpatientActs.

L'opération patient_AddpatientActs joue donc le rôle de constructeur et devra par conséquent initialiser les attributs obligatoires de ManagementAct conformément à la section 4.2.2.

```
patient_AddpatientActs(obj) =
  PRE obj ∈ PATIENTS ∧ obj ∈ Patients THEN
    ANY ma WHERE
      ma ∈ MANAGEMENTACTS ∧ ma ∉ ManagementActs ∧
      ma ∉ dom(patientActs)
    THEN
      ManagementActs := ManagementActs ∪ {ma} ||
      patientActs := patientActs ∪ {(ma ↦ obj)} ||
      managementact_validated(ma) := FALSE
    END
  END ;
```

De même, l'opération patient_DeletepatientActs joue le rôle de destructeur et devra faire les actions de mise à jours nécessaires :

```
patient_DeletepatientActs(obj) =
  PRE obj ∈ PATIENTS ∧ obj ∈ Patients THEN
    ANY ma WHERE
      ma ∈ MANAGEMENTACTS ∧ ma ∈ ManagementActs ∧
      ma ∈ patientActs ⁻¹ [{obj}] ∧
      managementact_validated(ma) = FALSE
    THEN
      ManagementActs := ManagementActs - {ma} ||
      patientActs := {ma} ⩤ patientActs ||
      managementact_validated := {ma} ⩤ managementact_validated ||
      managementact_dateTime := {ma} ⩤ managementact_dateTime ||
      managementact_Type := {ma} ⩤ managementact_Type
    END
  END ;
```

Le prédicat *managementact_validated(ma)* = **FALSE** de l'opération patient_DeletepatientActs est introduit manuellement et correspond à une contrainte fonctionnelle. Celle-ci indique qu'un ManagementAct validé ne peut être supprimé.

Quant au getter de lien permettant de retrouver les actes de soin associés à un patient, il est comme suit :

```
managementacts ← patient_GetpatientActs (obj) =
  PRE obj ∈ PATIENTS ∧ obj ∈ Patients THEN
    managementacts := patientActs ⁻¹ [{obj}]
  END ;
```

4.2.5 Amélioration du modèle fonctionnel

La spécification B issue du diagramme de classes fonctionnel nécessite certains traitement manuels, notamment lors de l'introduction des contraintes ou de nouvelles opérations. Parmi les

propriétés fonctionnelles que nous pouvons citer sur la base de l'exemple de la figure 4.1 nous avons :

(i) L'opération de validation d'un acte de soin.

(ii) Une fois validé un acte de soin ne peut plus être modifié

(iii) Un acte de soin validé dispose nécessairement d'un type et d'une date

La propriété (i) se traduit par l'intégration de l'opération *managementAct_Validate* à la spécification B. Cette opération est pré-conditionnée par le fait que l'acte de soin en cours de validation n'a pas déjà été validé et qu'il dispose d'un type et d'une date.

$$
\begin{array}{l}
\textbf{managementact_Validate}(obj) = \\
\quad \textbf{PRE} \\
\quad\quad obj \in MANAGEMENTACTS \wedge \\
\quad\quad obj \in ManagementActs \wedge \\
\quad\quad managementact_validated(obj) = \textbf{FALSE} \wedge \\
\quad\quad obj \in \textbf{dom}(managementact_Type) \wedge \\
\quad\quad obj \in \textbf{dom}(managementact_dateTime) \\
\quad \textbf{THEN} \\
\quad\quad \textbf{managementact_validated}(obj) := \textbf{TRUE} \\
\quad \textbf{END}
\end{array}
$$

La prise en compte de la propriété (ii) est effectuée par :
- L'ajout de la pré-condition “managementact_validated(obj) = FALSE” à toutes les opérations de modification d'un acte de soin (*e.g.* setters d'attributs)
- L'ajout de la pré-condition précédente à l'opération *patient_DeleteManagementAct*
- L'interdiction de la suppression d'une instance de Patient ayant au moins un acte de soin validé.
- La suppression du setter *managementact_SetValidated*.

Quant à la propriété (iii), elle est prise en compte par l'introduction de l'invariant suivant :

$$
\begin{array}{l}
\textbf{dom}(managementact_validated \rhd \{\textbf{TRUE}\}) \subseteq \textbf{dom}(managementact_Type) \wedge \\
\textbf{dom}(managementact_validated \rhd \{\textbf{TRUE}\}) \subseteq \textbf{dom}(managementact_dateTime)
\end{array}
$$

4.3 Transformation du modèle de sécurité

4.3.1 Approche proposée

La figure 4.7 illustre les principes de la traduction que nous proposons en vue de traduire un modèle fonctionnel augmenté par une politique de contrôle d'accès. Celle-ci n'est autre qu'une instance du méta-modèle de sécurité de la figure 4.6. La machine B nommée “Security Model” joue le rôle de filtre et permet de contrôler l'usage des opérations encapsulées dans la machine B “Functional Model”. En effet, l'utilisateur courant interagit avec la spécification “Security Model” qui lui donne accès uniquement aux opérations auxquelles il a droit dans la politique sécurité (Policy instance).

Rappelons que dans le but de traduire le modèle de sécurité nous adoptons une approche interprétée ; et ce, en suivant les étapes suivantes :
- Proposer une formalisation « stable » du méta-modèle (“Security Model”)
- Traduire l'instance du méta-modèle et l'injecter dans la formalisation du méta-modèle (“Policy instance”)

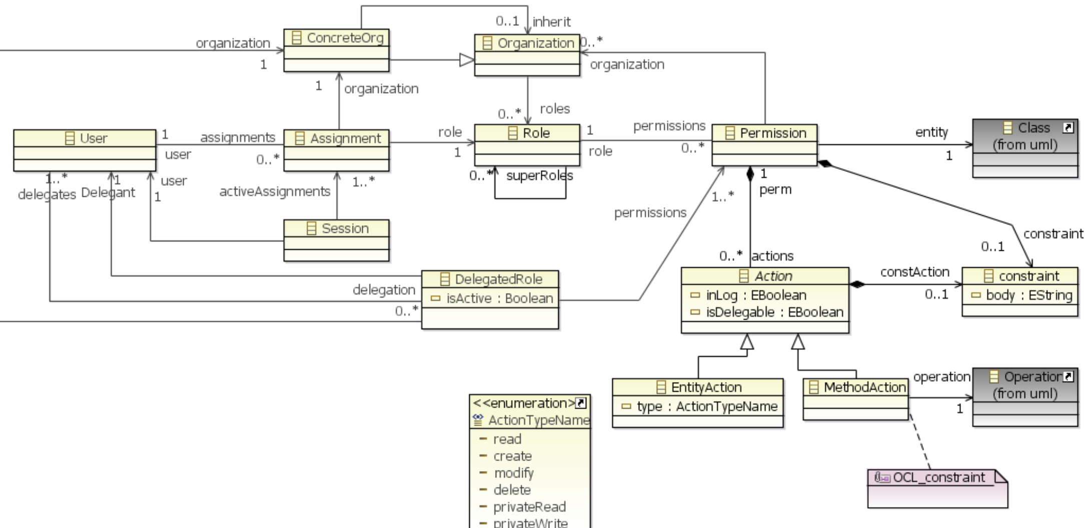


FIG. 4.6 – Méta-modèle de sécurité

Ensuite, à chaque opération de “Functional Model” on associe une opération sécurisée dans “Security Model”. L’opération sécurisée se charge donc de vérifier que l’utilisateur courant dispose d’une permission lui permettant d’appeler l’opération associée dans le modèle fonctionnel. Si c’est le cas, alors l’utilisation d’un animateur tel que ProB [LB03a] permettra d’animer l’opération sécurisée et de faire évoluer, par conséquent, l’état du modèle fonctionnel.

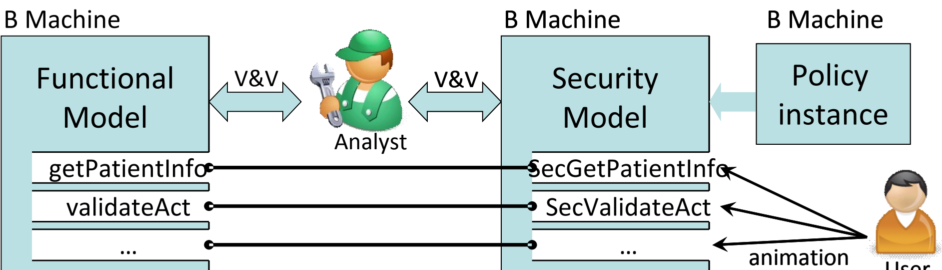


FIG. 4.7 – Principe de la traduction

4.3.2 Affectation d’utilisateurs aux rôles (relation User_Assignement)

La formalisation de la relation User_Assignement est réalisée au moyen d’une machine B distincte appelée “UserAssignements.mch”. Celle-ci contient la formalisation de la portion du méta-modèle de sécurité mettant en jeu les méta-classes USER et ROLE (figure 4.8).

La formalisation en B de cette patrie du méta-modèle de sécurité est présentée ci-dessous.

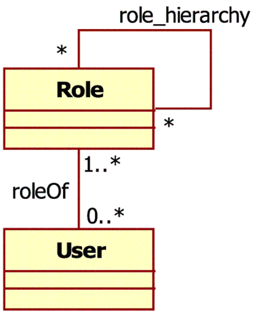


FIG. 4.8 – Portion du méta-modèle de sécurité proposé pour core-RBAC

MACHINE
  UserAssignements
SETS
  ROLES; *USERS*
VARIABLES
  roleOf,
  Roles_Hierarchy,
  currentUser
INVARIANT
  $Roles_Hierarchy \in ROLES \leftrightarrow ROLES \wedge$
  $roleOf \in USERS \rightarrow \mathbb{P}\,(ROLES) \wedge$
  $\mathbf{closure1}(Roles_Hierarchy) \cap \mathbf{id}(ROLES) = \emptyset \wedge$
  $currentUser \in USERS$

Dans ce modèle nous considérons que la relation $roleOf$ associe pour chaque utilisateur un ensemble de rôles. L'invariant "$\mathbf{closure1}(Roles_Hierarchy) \cap \mathbf{id}(ROLES) = \emptyset$" indique que la hiérarchie de rôles ne doit pas contenir de cycle. La variable $currentUser$ sert uniquement pour l'animation du modèle et permet d'identifier l'utilisateur courant. Nous introduisons ainsi l'opération $changeUser$ qui permet de changer d'utilisateur courant.

OPERATIONS
  changeUser(*user*) =
  PRE
    $user \in USERS$
  THEN
    $currentUser := user$
  END
END

La deuxième étape de l'approche interprétée, consistant à formaliser une instance du méta-modèle et l'injecter dans la spécification B du méta-modèle, se traduit par l'introduction de valuations dans les ensembles USERS et ROLES et par l'initialisation adéquate du modèle. Prenons, à titre d'exemple, l'instance du méta-modèle illustrée dans la figure 4.9.

La formalisation de cette instance sera injectée dans la spécification B du méta-modèle de la façon suivante :

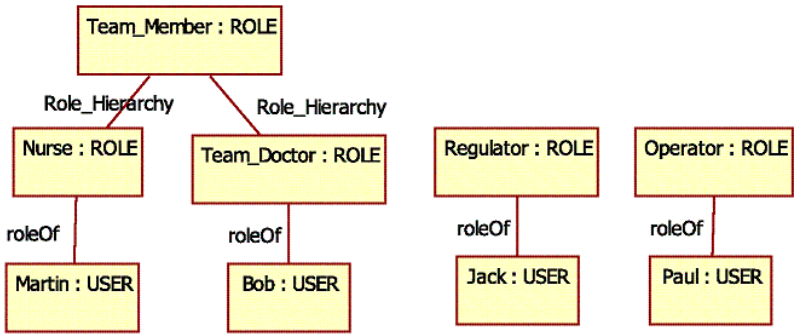


FIG. 4.9 – Instance du méta-modèle de la figure 4.8

SETS

$ROLES = \{Team_Doctor,\ Nurse,\ Operator,\ Team_Member,\ Regulator\}$;
$USERS = \{Bob,\ Paul,\ Martin,\ Jack,\ none\}$
...

INITIALISATION

$$
\begin{aligned}
&roleOf := \{(Bob \mapsto \{Team_Doctor\}),\\
&\quad (Paul \mapsto \{Operator\}),\\
&\quad (Martin \mapsto \{Nurse\}),\\
&\quad (Jack \mapsto \{Regulator\}),\\
&\quad (none \mapsto \emptyset\ )\} \ ||\\
&Roles_Hierarchy := \{(Team_Doctor \mapsto Team_Member),\\
&\quad (Nurse \mapsto Team_Member)\} \ ||\\
¤tUser := none
\end{aligned}
$$
...

Notons que l'utilisateur *none* est un utilisateur fictif indiquant, quand il est affecté à $currentUser$, qu'aucun utilisateur réel n'est connecté.

4.3.3 Affectation de permissions aux rôles (relation Permission_Assignement)

Comme pour la relation *User_Assignement* la formalisation de Permission_Assignement suit une logique semblable. Nous commençons tout d'abord par traduire en B les entités du méta-modèle associées aux concepts de permission, d'action, etc. Cette formalisation est présentée ci-dessous :

MACHINE
  RBAC_Model
INCLUDES
  Functional_Model,
  UserAssignements
SETS
  ENTITIES ;
  Attributes ;
  Operations ;
  $KindsOfAtt = \{public, private\}$;
  PERMISSIONS ;
  $ActionsType = \{read, create, modify, delete, privateRead, privateModify\}$;
  $Stereotypes = \{readOp, modifyOp\}$
VARIABLES
  AttributeKind, *AttributeOf*, *OperationOf*,
  constructorOf, *destructorOf*, *setterOf*, *getterOf*,
  PermissionAssignement, *EntityActions*,
  MethodActions, *StereotypeOps*,
  isPermitted
INVARIANT
  $AttributeKind \in Attributes \rightarrow KindsOfAtt \wedge$
  $AttributeOf \in Attributes \rightarrow ENTITIES \wedge$
  $OperationOf \in Operations \rightarrow ENTITIES \wedge$
  $constructorOf \in Operations \rightarrowtail\!\!\!\!\!\!+ ENTITIES \wedge$
  $destructorOf \in Operations \rightarrowtail\!\!\!\!\!\!+ ENTITIES \wedge$
  $setterOf \in Operations \rightarrowtail\!\!\!\!\!\!+ Attributes \wedge$
  $getterOf \in Operations \rightarrowtail\!\!\!\!\!\!+ Attributes \wedge$
  $StereotypeOps \in Stereotypes \leftrightarrow Operations \wedge$
  $setterOf \cap getterOf = \emptyset \wedge$
  $PermissionAssignement \in PERMISSIONS \rightarrow (ROLES \times ENTITIES) \wedge$
  $EntityActions \in PERMISSIONS \mapsto\!\!\!\!\!\!+ \mathbb{P}\ (ActionsType) \wedge$
  $MethodActions \in PERMISSIONS \mapsto\!\!\!\!\!\!+ \mathbb{P}\ (Operations) \wedge$
  $isPermitted \in ROLES \leftrightarrow Operations$

Les ensembles ENTITIES, Attributes, Operations et KindsOfAtt contiennent les éléments du modèle fonctionnel nécessaires à l'expression des permissions selon notre méta-modèle. Les relations entre ces ensembles qui permettent de reconstruire la partie fonctionnelle sont :
– AttributeKind : indique pour chaque attribut de classe s'il est privé ou public
– AttributeOf : indique l'entité fonctionnelle dans laquelle un attribut est encapsulé
– OperationOf, constructorOf et destructorOf : indiquent l'entité fonctionnelle dans laquelle une opération est encapsulée
– setterOf, getterOf : rattachent les setters et les getters à leurs attributs

La distinction entre les différents types d'opération (OperationOf, constructorOf, destructorOf, setterOf et getterOf) est indispensable au calcul des permissions en fonction de l'utilisateur courant. En effet, une permission, associée à un role R et une entité E, et contenant une action de type ≪**EntityAction**≫**Create**, indique que si l'utilisateur courant est affecté au rôle R alors il a accès à l'opération issue de $contructorOf^{-1}(E)$.

Dans notre méta-modèle de sécurité on peut exprimer des permission de type *Read* ou *Modify* donnant ainsi accès aux opérations de lecture ou de modification. Outre les getters (qui effectuent des lectures sur les attributs de classes) et les setters (qui effectuent des modifications sur les attributs de classes) nous donnons la possibilité à l'analyste de sétéréotyper les opérations de

manière spécifique en indiquant s'il s'agit d'opérations de lecture ou de modification. Cela est représenté par l'ensemble énuméré *Stereotypes* et la relation *StereotypeOps*.

Les autres constructions de cette spécification B sont nécessaires à l'expression des permissions. Chaque permission de l'ensemble *PERMISSIONS* est associée à un couple (*role*, *entity*) où $role \in ROLES$ et $entity \in ENTITIES$. La relation *EntityActions* représente les permissions exprimé sur une entité de manière globale (lecture, écriture, création, etc). Quant à la relation *MethodActions*, elle définit les permissions spécifiques à certaines opérations de l'entité sécurisée.

Exemple d'illustration

La figure 4.10 exprime une politique de contrôle d'accès associée au modèle fonctionnel de la figure 4.1. Dans ce diagramme, nous distinguons les permissions associées à l'entité *Patient* de celles associées à *ManagementAct*.

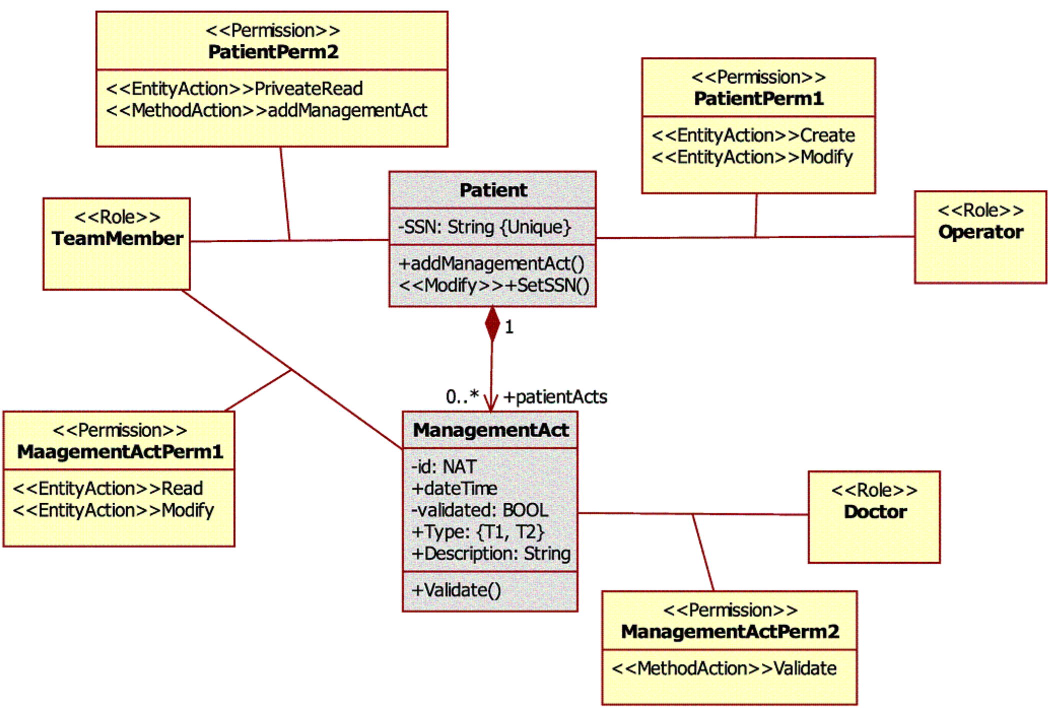


Fig. 4.10 – Expression d'une politique de sécurité sur l'exemple de la figure 4.1

- PatientPerm1 : indique qu'un opérateur peut créer une instance de Patient (≪EntityAction≫Create), et exécuter l'opération de modification SetSSN.
- PatientPerm2 : définit les permissions accordées aux membres de l'équipe médicale. Ces derniers peuvent consulter l'attribut privé SSN ((≪EntityAction≫PrivateRead) et peuvent associer de nouveaux actes de soin à un patient (≪MethodAction≫addManagementAct)
- ManagementActPerm1 : permet aux membres de l'équipe médicale de lire et de modifier uniquement les attributs publics d'un acte de soin.
- ManagementActPerm2 : indique qu'un docteur peut valider un acte de soin en exécutant la méthode Validate de ManagementAct.

Introduction des données fonctionnelles dans la formalisation du métamodèle

SETS

$ENTITIES = \{Patient,\ ManagementAct\}$;
$Attributes = \{SSN,\ Validated,\ dateTime,\ Type\}$;
$Operations =$
$\{CreatePatient,\ DeletePatient,\ Patient_AddManagementAct,\ Patient_SetSSN,\ Patient_GetSSN,\ Patient_DeleteManagementAct,\ Patient_GetManagementActs,\ Managementact_SetdateTime,\ Managementact_GetdateTime,\ Managementact_SetValidated,\ Managementact_GetValidated,\ Managementact_SetType,\ Managementact_GetType,\ Managementact_Validate\ \}$;

INITIALISATION

$AttributeKind := \{(SSN \mapsto private),$
$(Validated \mapsto private),$
$(dateTime \mapsto public),$
$(Type \mapsto public)\}$
$||$
$AttributeOf := \{(SSN \mapsto Patient),$
$(Validated \mapsto ManagementAct),$
$(dateTime \mapsto ManagementAct),$
$(Type \mapsto ManagementAct)\}$
$||$
$OperationOf := \{(CreatePatient \mapsto Patient),$
$(DeletePatient \mapsto Patient),$
$(Patient_AddManagementAct \mapsto ManagementAct),$
$(Patient_SetSSN \mapsto Patient),$
$(Patient_GetSSN \mapsto Patient),$
$(Patient_DeleteManagementAct \mapsto Patient),$
$(Patient_GetManagementActs \mapsto Patient),$
$(Managementact_SetdateTime \mapsto ManagementAct),$
$(Managementact_GetdateTime \mapsto ManagementAct),$
$(Managementact_SetValidated \mapsto ManagementAct),$
$(Managementact_GetValidated \mapsto ManagementAct),$
$(Managementact_SetType \mapsto ManagementAct),$
$(Managementact_GetType \mapsto ManagementAct),$
$(Managementact_Validate \mapsto ManagementAct)\}$
$||$
$constructorOf := \{(CreatePatient \mapsto Patient)\}$
$||$
$destructorOf := \{(DeletePatient \mapsto Patient)\}$
$||$
$StereotypeOps := \{(modifyOp \mapsto Patient_SetSSN),$
$(readOp \mapsto Managementact_GetValidated)\}$
$||$
$setterOf := \{(Patient_SetSSN \mapsto SSN),$
$(Managementact_SetdateTime \mapsto dateTime),$
$(Managementact_SetValidated \mapsto Validated),$
$(Managementact_SetType \mapsto Type)\}$
$||$
$getterOf := \{(Patient_GetSSN \mapsto SSN),$
$(Managementact_GetdateTime \mapsto dateTime),$
$(Managementact_GetValidated \mapsto Validated),$
$(Managementact_GetType \mapsto Type)\}$

Etant donnée que la manipulation des entités fonctionnelles ne s'effectue que par le biais des opérations de base produites automatiquement ainsi que les opérations complémentaires ajoutées

manuellement à la machine "Functional Model", alors l'intégration des données fonctionnelles dans la formalisation du méta-modèle de sécurité implique la prise en compte des étiquettes de toutes ces opérations.

Introduction des données de sécurité dans la formalisation du méta-modèle

```
SETS
...
    PERMISSIONS =
        {PatientPerm1, PatientPerm2, ManagementActPerm1, ManagementActPerm2} ;
```

```
INITIALISATION
...
    PermissionAssignement := {(PatientPerm1 ↦ (Operator ↦ Patient)),
                              (PatientPerm2 ↦ (Team_Member ↦ Patient)),
                              (ManagementActPerm1 ↦ (Team_Member ↦ ManagementAct)),
                              (ManagementActPerm2 ↦ (Team_Doctor ↦ ManagementAct))}
    ||
    EntityActions := {(PatientPerm1 ↦ {create, modify}),
                      (PatientPerm2 ↦ {privateRead}),
                      (ManagementActPerm1 ↦ {read, modify})}
    ||
    MethodActions := {(PatientPerm2 ↦ {Patient_AddManagementAct}),
                      (ManagementActPerm2 ↦ {Managementact_Validate})}
    ||
    isPermitted := ∅
```

Introduction des opérations sécurisées

La relation **isPermitted**, initialisée à l'ensemble vide et déduite de la formalisation de la politique de sécurité, sert à retrouver toutes les opérations permises pour un rôle donné. Par exemple, l'interprétation de PatientPerm2 permet d'introduire dans la relation isPermitted les couples $(Team_Member \mapsto Patient_GetSSN)$ et $(Team_Member \mapsto Patient_AddManagementAct)$. Ceci est réalisé au moyen de la clause DEFINITION en B, et d'une opération de calcul de ces permission :

setPermissions = **PRE** *isPermitted* = ∅ **THEN** *isPermitted* := *permissions* **END** ;

Cette relation permet de sécuriser les opérations issues du modèle fonctionnel en leur associant une garde de la forme : $operation \in isPermitted[currentRole]$. Voici par exemple, l'opération de création sécurisée associée à la classe Patient.

```
OPERATIONS
...
secure_createPatient(obj) =
    PRE obj ∈ PATIENTS ∧ obj ∉ Patients THEN
        SELECT
            CreatePatient ∈ isPermitted[currentRole]
        THEN
            createPatient(obj)
        END
    END ;
...
```

Si l'utilisateur courant dispose d'un rôle ayant un droit de création sur l'entité Patient alors il pourra animer l'opération secure_createPatient. Cette opération fait appel à l'opération de base du modèle B fonctionnel createPatient et peut par conséquent faire évoluer l'état du système.

Chapitre 5

Formalising dynamic access control rules

5.1 Integrating ASTD into the security metamodel

The metamodel shown in Fig. 4.6 can be instantiated in order to define a static access control policy. The policy is based on authorizations or prohibitions given to users to execute actions on the IS. However, dynamic access controls may be useful in order to express obligation or separation of duty (SoD) rules. We propose to use the ASTD notation (Algebraic State Transition Diagrams) [FGL+08] to specify such dynamic rules. This aspect of access control is named dynamic because authorizations can be granted depending on previous executed events. Patterns were introduced [KFL10] in order to express these rules.

A metamodel combining static and dynamic aspects of an access control policy is proposed in Fig. 5.1. This metamodel is based on the metamodel described previously and on the ASTD metamodel shown in Fig. 5.2 in order to link common concepts. New classes are also introduced in order to describe elements from obligation or SoD concept.

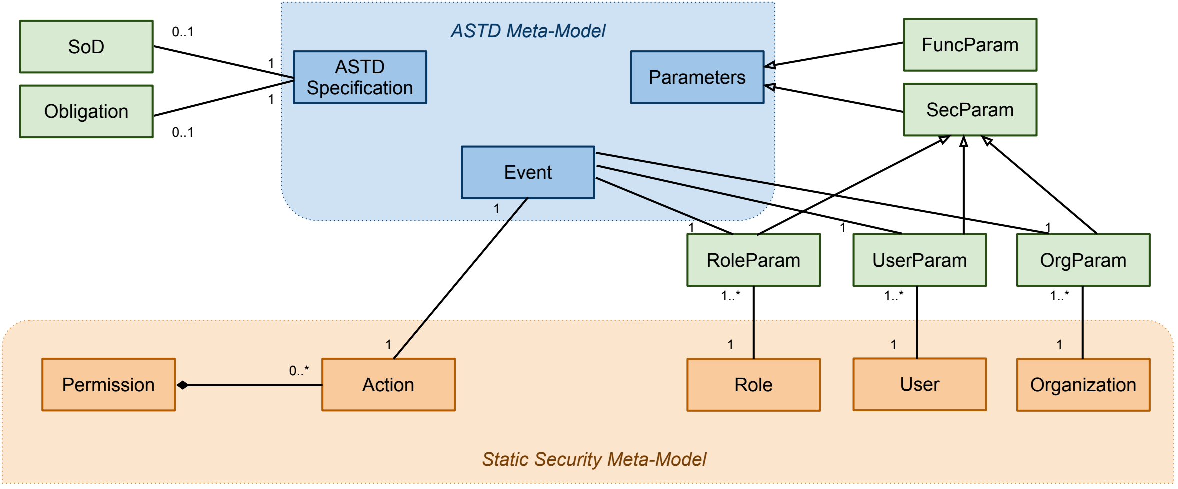


FIG. 5.1 – Méta-modèle de sécurité.

Similarly to the translation from UML diagrams to formal specifications written using the B language, translation rules from ASTD to Event-B were developed and are presented in the following sections. We chose Event-B to benefit from its tool support. However translation into B is quite similar.

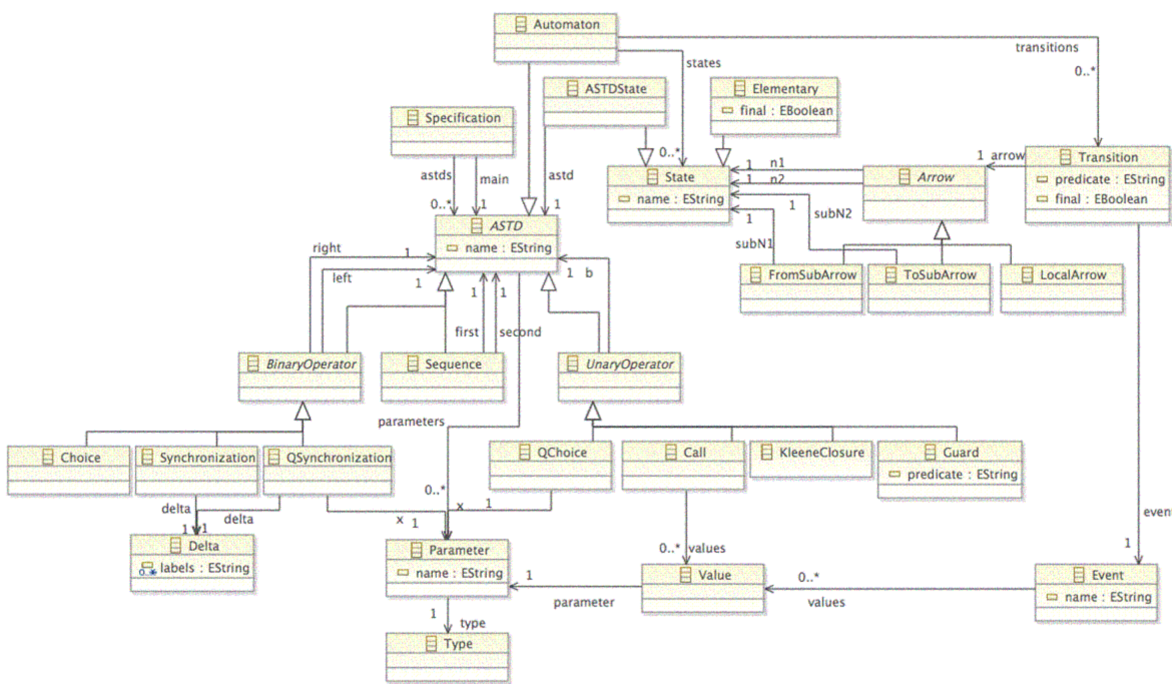


FIG. 5.2 – Méta-modèle des ASTD.

The final architecture of the secure IS, including static and dynamic aspects, is based on successive filtering of actions. First, the IS receives the user request as an action, with its parameters and a security context including the user id, the role and the organization he/she belongs to. These informations are given to machine M_{wf} that takes care of dynamic rules of the access control policy. Then the action and its context are transmitted to M_{sec} to be checked against static rules. Finally, if static rules allow the execution, the functional machine M_{fun} receives the action but can still reject it if it does not conform to functional constraints. In case one of the machines rejects the action of the user, the system is not updated and an error message is returned.

5.2 Systematic translation rules from ASTD to Event-B[1]

Information Systems (IS) are taking an increasingly important place in today's organizations. As computer programs connected to databases and other systems, they induce increasing costs for their development. Indeed, with the importance of the Internet and their high computer market penetration, IS have become the de-facto standard for managing most of the aspects of a company strategy. In the context of IS, formal methods can help improving the reliability, security and coherence of the system and its specification. The APIS (Automated Production of Information system) project [F+07] offers a way to specify and generate code, graphical user interfaces, databases, transactions and error messages of such systems, by using a process algebra-based specification language. However, process algebra, despite their formal aspect, are not as easily understandable as semi-formal graphical notations, such as UML [RJB96]. In order to address this issue, a formal notation combining graphical elements and process algebra was introduced : Algebraic State Transition Diagrams (ASTD) [FGL+08]. Using ASTD, one can specify the behavior of an IS. The interpreter *i*ASTD [SM+10] can efficiently execute ASTD specifications. However, there is no tool allowing proof of invariants or property check over

[1]The main results described in the following sections are presented in [MFGL10]

an ASTD specification. This chapter aims to define systematic translation rules from an ASTD specification to Event-B [Abr10] in order to model check or prove properties using tools of the RODIN platform [ABHV06]. Moreover, translation results will allow to bridge other process algebras (like EB^3 [FF09] or CSP [Hoa85]) with Event-B as they share a similar semantics with ASTD. Event-B is first introduced to the reader in Section 5.3. An overview of ASTD and a case study will be then presented. This case study will help readers unfamiliar with ASTD to discover the formalism in Section 5.4. The Event-B machine resulting from translation rules applied to this case study will be described as well as rules and relevant steps of translation in Section 5.5. Finally, future work and evolution perspectives will be presented.

5.3 Event-B Background

Event-B [Abr10] is an evolution of the B method [Abr96a] allowing to model discrete systems using a formal mathematical notation. The modeling process usually follows several refinement steps, starting from an abstract model to a more concrete one in the next step. Event-B specifications are built using two elements : *context* and *machine*. A *context* describes the static part of an Event-B specification. It consists of declarations of *constants* and *sets*. *Axioms*, which describe types and properties of constants and sets, are also included in the context. A *machine* is the dynamic part of an Event-B specification. It has a state consisting of several *variables* that are first initialized. Then *events* can be executed to modify the state. An event can be executed if it is enabled, *i.e.* all the conditions prior to its execution hold. Theses conditions are named *guards*. Among all enabled events, only one is executed. In this case, substitutions, called *actions*, are applied over variables. All actions are applied simultaneously, meaning that an event is atomic. The state resulting from the execution of the event is the new state of the machine, enabling and disabling events. Alongside the execution of events, *invariants* must hold. An invariant is a property of the system written using a first-order predicate on the state variables. In order to ensure that invariants hold, *proofs* are performed over the specification.

5.4 ASTD Background

ASTD is a graphical notation linked to a formal semantics allowing to specify systems such as IS. An ASTD defines a set of traces of actions accepted by the system. ASTD actions correspond to events in Event-B. Event-B actions and substitutions, as they modify the state of an Event-B machine, can be binded to the change of state in ASTD. The ASTD notation is based on operators from the EB^3 [FSD03] method and was introduced as an extension of Harel's Statecharts [Har87]. An ASTD is built from transitions, denoting action labels and parameters, and states that can be elementary (as in automata) or ASTD themselves. Each ASTD has a type associated to a formal semantics. This type can be automata, sequence, choice, Kleene closure, synchronization over a set of action labels, choice or interleaving quantification, guard and ASTD call. One of ASTD most important features is to allow parametrized instances and quantifications, aspects missing from original Statecharts. An ASTD can also refer to attributes, which are defined as recursive functions on traces accepted by the ASTD, as in the EB^3 method. Such a recursive function compares the last action of the trace and maps each possible action to a value of the attribute it is defining. Computing this value may imply to call the function again on the remaining of the trace.

5.4.1 ASTD Operators

Several operators, or ASTD types, are used to specify an IS. We detail them in the following paragraphs. Operators will be further illustrated in Section 5.4.2 with the introduction of a case study.

Automata In an ASTD specification, one can describe a system using hierarchical states automata with guarded transitions. Each automata state is either elementary or another ASTD of whichever type. Transitions can be on states of the same depth, or go up or down of one level of depth. A transition decorated by a bullet ($\bullet$) is called a final transition. A final transition is enabled when the source state is final. As in Statecharts, an history state allows the current state of an automata ASTD to be saved before leaving it in order to reuse it later.

Sequence A sequence is applied to two ASTD. It implies that the left hand side ASTD will be executed and will reach a final state before the right hand side ASTD can start. There is no immediate equivalent of this operator in Harel's Statecharts, but its behavior can be reproduced with guards and final transitions. A sequence ASTD is noted with a double arrow $\Rightarrow$.

Choice A choice, noted $|$ allows the execution of only one of its operands, like a choice in regular expressions or in process algebras. The choice of the ASTD to execute is made on the first action executed. After the execution of the first action, the chosen ASTD is kept until it terminates its execution. If both operands of a choice ASTD can execute the first action, then a nondeterministic choice is made between the two ASTD. The behavior of a choice ASTD can be modeled in Statecharts using internal transition from an initial state, in a similar way to automata theory with ϵ transitions.

Kleene Closure As in regular expressions, a Kleene closure ASTD noted $*$ allows its operand to be executed zero, one or several times. When the state of its operand is final, a new iteration can start. There is no similar operator in Statecharts, but the same behavior can be reproduced with guards and transitions.

Synchronization Over a Set of Action Labels As the name suggests, this operator allows the definition of a set of actions that both operands must execute at the same time. It is similar to Roscoe's CSP parallel operator $\underset{X}{\|}$. There are some similarities with AND states of Statecharts and synchronization ASTD. A synchronization over the set of actions Δ is noted $|[\Delta]|$. We derive two often used operators from synchronization : interleaving, noted $|||$, is the synchronization over an empty set ; parallel, noted $\|$, synchronizes ASTD over the set of common actions of its operands, like Hoare's CSP $\|$.

Quantified Interleaving A quantified interleaving models the behavior of a set of concurrent ASTD. It sets up a quantification set that will define the number of instances that can be executed and a variable that can take a value inside the quantification set. Each instance of the quantification is linked to a single value, two different instances have two different values. This feature lacks in Statecharts, as we have to express distinctly each instance behavior, but was proposed as an extension and named "parametrized-and" state by Harel. A quantified interleaving of variable x over the set T is noted $||| \; x : \mathrm{T}$.

Quantified Choice A quantified choice, noted $| \; x : \mathrm{T}$, lets model that only one instance inside a set will be executed. Once the choice is made, no more instances can be executed. As in

quantified synchronization, the instance is linked to one value of a variable in the quantification set. An extension of Statecharts named a similar feature “parametrized-or” state.

Guard Usually, guards are applied to transitions. With the guard ASTD, one can forbid the execution of an entire ASTD until a condition holds. The predicate of a guard can use variables from quantifications and attributes. A predicate $P(x)$ guarding an ASTD is noted $\implies P(x)$

ASTD **call** An ASTD call simply links to other parts of the specification using the name of another ASTD. The same ASTD can be called several times, in different locations of the specification. It allows the designer to reuse ASTD in the same specification and helps synchronize processes. An ASTD call is made by writing the name of the ASTD called and its parameters (if any).

5.4.2 An ASTD Case Study

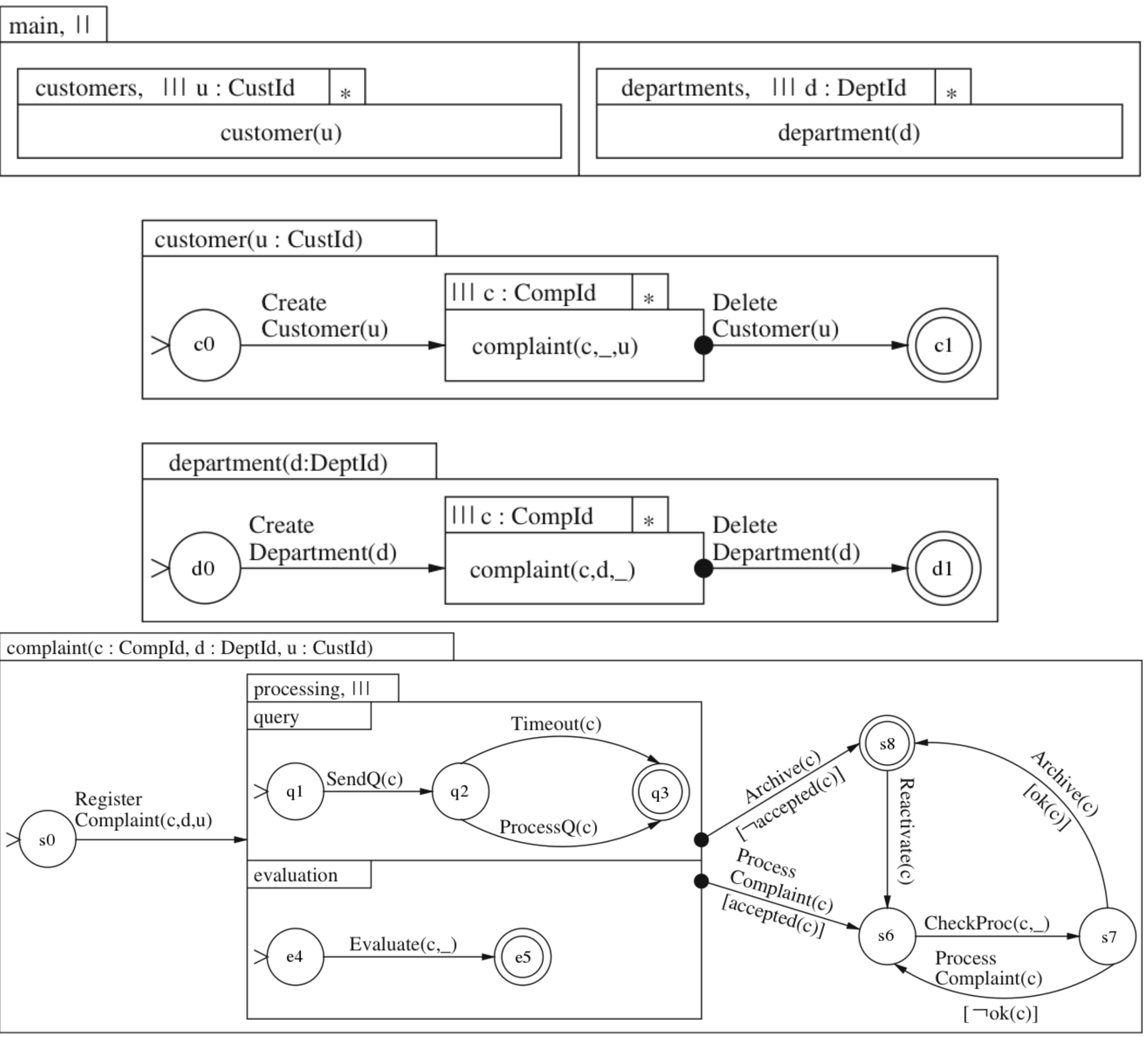


FIG. 5.3 – An ASTD specification describing a complaint management system.

In order to present features and expressiveness of the ASTD notation to the reader, Fig. 5.3 introduces the case study that will be used throughout this chapter. This ASTD models an in-

formation system designed to manage complaints of customers in a company. In this system, each complaint is issued from a customer relatively to a department. This example is inspired from [Van98]. The **main** ASTD, whose type is a synchronization over common actions, describes the system as a parallel execution of interleaved customers and departments processes. The IS lifecycle of a given customer is described by the parametrized ASTD **customer** (u), the same applies for the description of the company departments in the ASTD **department** (d). In the initial state, a customer or a department must be created. Then complaints regarding these entities can be issued. This is described using an ASTD call. The final transition means that the event can be executed if the source state is final. In our case, in order to delete a customer or a department, any related complaints must be closed. Finally, the ASTD describing the checking and processing of a complaint c issued by customer c about department d is given by **complaint** (c, d, u). After registering a complaint in the system, it must be evaluated by the company and a questionnaire is sent to the customer in order to detail his/her complaint. The specification takes into account the possibility that the customer does not answer the questionnaire with the $\mathsf{Timeout}(c)$ event. Then, if the complaint is accepted and the questionnaire received or timed out, a check is performed. In case of refusal of the complaint, it is archived, but it can be reactivated later. The only final state is state $s8$, meaning that the complaint was archived (solved or not). In this specification, no attribute modification is performed. An ASTD only describes traces and has no consequences on updates to be performed against IS data, such as attributes that are stored in databases. However, an ASTD can access attribute values to use them in guards, as shown in both $\mathsf{Archive}(c)$ actions.

5.4.3 Motivations

ASTD are not the only way to specify IS behavior. The UML-B [SB06a] method introduces a behavior specification in the form of a Statecharts. Using Statecharts, it is easy to describe an ordered sequence of actions whereas using B, it is easier to model interleaving events. A systematic translation of Statecharts into B machines is proposed by [SZ02]. Compared to Statecharts, ASTD offer additional operators to combine ASTD in sequence, iteration, choice and synchronisation. When a UML-B specification models a system, it can only describe the life-cycle of a single instance of a class whereas ASTD specification models the behavior of all instances of all classes of the system. A new version of UML-B [SBS09] introduces the possibility to refine class and Statecharts as part of the modeling process, and can translate it into Event-B. The UML-B approach can describe the evolution of entity attributes using B substitutions, a feature that ASTD lacks. csp2B [But00] provides better proofs (on the B machine) and model checking (on the CSP side) tools than Statecharts but lacks the visual representation of the specification given by UML Statecharts. It is also limited to a subset of CSP specifications, where the quantified interleaving operator must not be nested. ASTD aims to be a compromise in both visual and synchronization aspects. On the other hand, ASTD lacks proofs and model checking allowed by the B side of UML-B and csp2B approaches. In order to answer this issue, a systematic translation of ASTD specifications into Event-B is proposed.

The choice between classical B and Event-B was made at an early stage by comparing tools and momentum of both methods. It appears that community efforts and tool development are currently focused on Event-B. Despite the fact that classical B offers some convenient notation such as IF / THEN statements or operation calls, Event-B appeared as a good compromise for our efforts. Classical B translation rules inspired by Event-B rules might be written.

TAB. 5.1 – Event-B representation of ASTD states

ASTD state	State domain	Initial State
choice	$State \in \{$ `none`, `first`, `second` $\}$	`none`
sequence	$State \in \{$ `left`, `right` $\}$	`left`
Kleene closure	$State \in \{$ `neverExecuted` , `started` $\}$	`neverExecuted`
synchronization	-	-
quantified choice	$State \in \{$ `notMade`, `made` $\} \nrightarrow$ QUANTIFICATIONSET	`notMade` $\mapsto 0$
quantified synch	$State \in$ QUANTIFICATIONSET $\rightarrow$ STATESET	initial for all
guard	$State \in \{$`checked`, `notChecked` $\}$	`notChecked`
ASTD call	-	-

5.5 Translation

Translation from ASTD to Event-B is achieved in several steps. Fig. 5.4 presents the architecture of the translation process. A context derived from ASTD operators introduces constants and sets needed to code their semantics. This context is the same in all translations and is described by Table 5.1. It codes elements from the semantics of all types of ASTD except automata, and is inspired of mathematical definition of ASTD semantics. Constants, sets and axioms defined in this context may be re-used in other part of the Event-B translation, hence this context is extended by a translation specific context. Automata states are translated into such a specific context since automata states depend on the ASTD specification to translate. For each ASTD, a variable and an invariant corresponding to its type are created. The invariant associates the variable to the set of values it can take, as defined in both contexts. In the following sections, we provide translation rules for each ASTD type, generating appropriate contexts and machines.

5.5.1 Automata

The first part of automata translation concerns the static part, the context. Several elements are introduced in the context : states, initial states, final states and transition functions.

States States from automata ASTD are represented as constants and grouped into state sets in order to facilitate later use. Even hierarchical states are represented by a constant.

Initial States Since an ASTD can be reset by the execution of a Kleene closure, initial states are defined as separate constants. They are also useful in the initialisation event of the machine generated in next step of our translation.

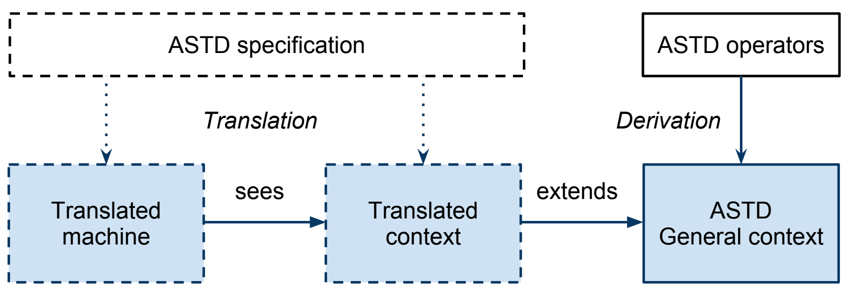


FIG. 5.4 – The architecture resulting from the translation process.

Final Predicates A final predicate is a function taking a state as argument and returning TRUE or FALSE depending if the state is final or not. The number of arguments depends on the type of the ASTD. This predicate is useful in the case of final transitions, sequences or Kleene closures, when transitions are activated if, and only if, a state is final. Hence, a final predicate is written for each ASTD type in the context common to all translations.

Transition Functions A transition function for each action label is generated. It takes as argument the current state of an automata ASTD and returns the resulting state. Transition functions are deterministic and partial.

The generated context for our case study defines 40 constants, 5 sets and 29 axioms. It is not presented here for the sake of conciseness. Then, for the dynamic part, for each distinct action label in the translated automata ASTD, a single event will be produced. If the action has a guard, a WHEN clause *i.e.* a guard, is generated. If the ASTD action has arguments (in the case of quantified variable for instance), an ANY clause is built accordingly and a guard specifying a type for the variable is added. Then a guard testing that the execution of the action is allowed *i.e.* the current state is in the domain of the transition function of the event. The modification of the state is applied by generating a THEN substitution.

Translation rules for automata ASTD are presented in Table 5.2. When a transition, an initial state or a final state is found, the first rule applies. In the case of a final transition, the second rule then applies. In the second pattern translation, the guard numbered `g1` of Table 5.2 is added to event e that was generated by applying first rule. In our case study, the second rule is applied for the ProcessComplaint(c) action. The guard added in this case is described by guard `grdAutomata`.

`grdAutomata` : $isFinalProcessing(isFinalQuery(StateQuery(c)) \mapsto isFinalEvaluate(StateEval(c))) = \text{TRUE}$

Constants $isFinalX$ and $StateX$ refer to ASTD **X** in Fig. 5.3. An interleaved state is final if, and only if, both of its operand states are final. For this reason, guard `grdAutomata` checks if both states of **Query** and **Evaluation** ASTD are final. A pair (x, y) is noted $x \mapsto y$ in Event-B.

The action CreateCustomer(u) is translated into the event described below. $grd1$ describes the set in which the parameter u can take its value. $grd2$ verifies that a customer is in a state of the domain of transition function $TransCreateCustomer$. $act1$ describes the state update for action CreateCustomer(u) : it only modifies the state of customer u according to the transition function $TransCreateCustomer$.

Event $CreateCustomer \mathrel{\widehat=}$
any
u
where
`grd1` : $u \in USERSET$
`grd2` : $StateCustomer(u) \in dom(TransCreateCustomer)$
then
`act1` : $StateCustomer(u) := TransCreateCustomer(StateCustomer(u))$
end

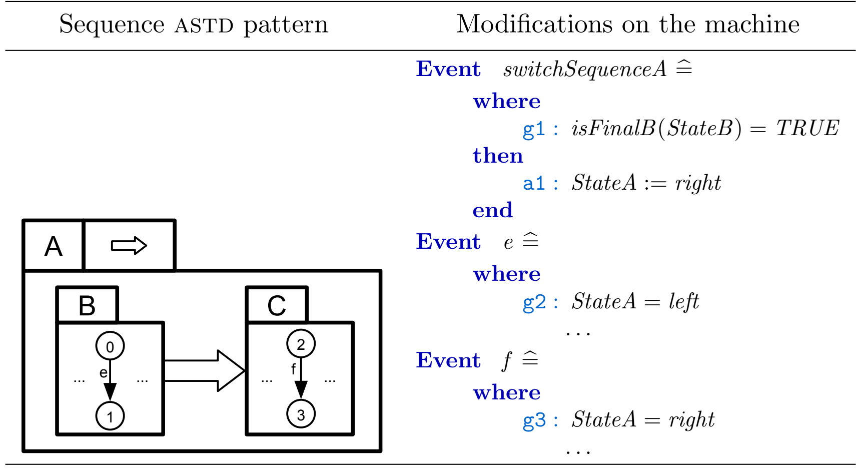
5.5.2 Sequence

Because of the number of possibilities to determine whether or not a sequence can switch from `left` state to `right` state, an extra event is introduced. This event is similar to an internal event of the IS and will verify that all the conditions for the switch from left to right side to happen

Tab. 5.2 – Automata ASTD to Event-B translation rules

Automata ASTD pattern	Added to the context	Modifications on the machine
A aut 0 e(x) [p(y)] 1	**SETS** StatesA **CONSTANTS** s0, s1 initA, isFinalA, TransE **AXIOMS** ax1 : $partition($ $StatesA, \{s0\}, \{s1\})$ ax2 : $initA = s0$ ax3 : $isFinalA = \{s0 \mapsto FALSE,$ $s1 \mapsto TRUE\}$ ax4 : $TransE = \{s0 \mapsto s1\}$	**Event** $e \widehat{=}$ **any** x **where** g1 : $x \in XSET$ g2 : $P(y)$ g3 : $StateA \in$ $dom(TransE)$ **then** a1 : $StateA :=$ $TransE(StateA)$ **end**
A aut B e 0	**CONSTANTS** isFinalB **AXIOMS** ax1 : $isFinalB = \ldots$ // Depends on B type	**Event** $e \widehat{=}$ **where** g1 : $isFinalB(StateB)$ $= TRUE$ $\ldots$

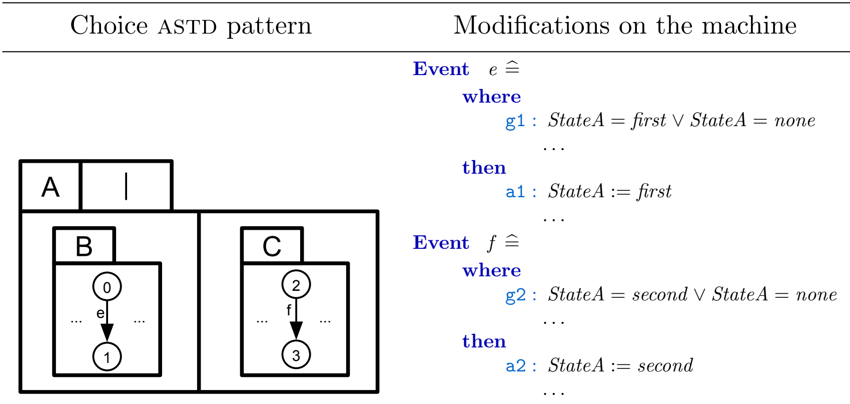
holds and then change the state of the sequence. For example, if an ASTD named **A** is a sequence of ASTD **B** and **C**, the generated event will be called *switchSequenceA*. Then, in order to ensure that the current state allows the execution of every events of ASTD **B** and **C**, a guard is added to each event of **B** and **C** to check if the state of ASTD **A** is $left$ or $right$ respectively. As for automata, a final predicate must be generated in the context for ASTD **B** state. Translation rule is described in the following table.

Sequence ASTD pattern	Modifications on the machine
	**Event** $switchSequenceA \widehat{=}$ **where** g1 : $isFinalB(StateB) = TRUE$ **then** a1 : $StateA := right$ **end** **Event** $e \widehat{=}$ **where** g2 : $StateA = left$ $\ldots$ **Event** $f \widehat{=}$ **where** g3 : $StateA = right$ $\ldots$

5.5.3 Choice

A choice ASTD can be in three states as described by the general ASTD context : `none` when the choice is not made yet, `first` or `second` depending of the side chosen. The translation rule for a choice ASTD is presented in the following table. If an ASTD named **A** is a choice between ASTD **B** and **C**, then a guard and an action are added to each event. Events from **B** will receive guard

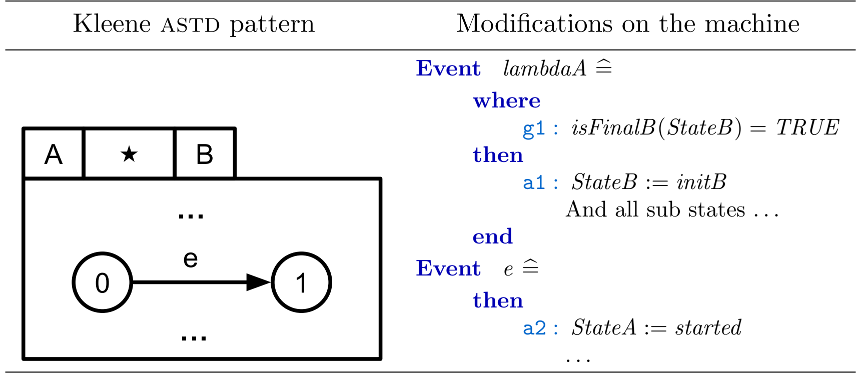
g1 and action a1. A similar transformation of events from ASTD **C** is also needed with guard g2 and action a2.



Choice ASTD pattern	Modifications on the machine
A | B C 0 e 1 2 f 3	**Event** $e \;\widehat{=}$ **where** g1 : $StateA = first \vee StateA = none$ ... **then** a1 : $StateA := first$ ... **Event** $f \;\widehat{=}$ **where** g2 : $StateA = second \vee StateA = none$ ... **then** a2 : $StateA := second$ ...

5.5.4 Kleene Closure

When an iteration of a Kleene closure ASTD is completed, its operand must be reset to initial state. For this reason, an additional event is generated. In the IS, this event is internal and hidden, in the ASTD specification, the semantics off Kleene operator handles the process, but in Event-B the reset must be described. This event will be activated when the its operand is final, and will reinitialize all sub-states in the hierarchy. The following table details the resulting Event-B machine.



Kleene ASTD pattern	Modifications on the machine
A ★ B ... e 0 1 ...	**Event** $lambdaA \;\widehat{=}$ **where** g1 : $isFinalB(StateB) = TRUE$ **then** a1 : $StateB := initB$ And all sub states ... **end** **Event** $e \;\widehat{=}$ **then** a2 : $StateA := started$ ...

As presented for automata and sequence, a final predicate must be generated in the context for ASTD under the Kleene closure operator.

5.5.5 Synchronization Over a Set of Action Labels

For actions that are not synchronized, nothing is introduced or modified by the translation of synchronization ASTD. This is the case for interleaving ASTD and action labels not common to both operands of the parallel operator. In the case of a synchronized action, guards from both operand must be put in conjunction, and substitutions applied conjointly.

Synchronization ASTD pattern	Modifications on the machine
	Event $e \mathrel{\widehat{=}}$ **where** gB : $guardsfromB$ ASTD gC : $guardsfromC$ ASTD $\ldots$ **then** a1 : $StateB := \ldots$ a2 : $StateC := \ldots$ $\ldots$

In our case study, the only synchronization ASTD is **main**. Common actions of both sides are only actions appearing in the ASTD **complaint** (c, d, u). For each one of the generated events of **complaint** (c, d, u), the guards `readyInCustomer` and `readyInDepartment` must hold. cc and dc states correspond to states where the customer and the department respectively are in the complaint quantified interleaving ASTD.

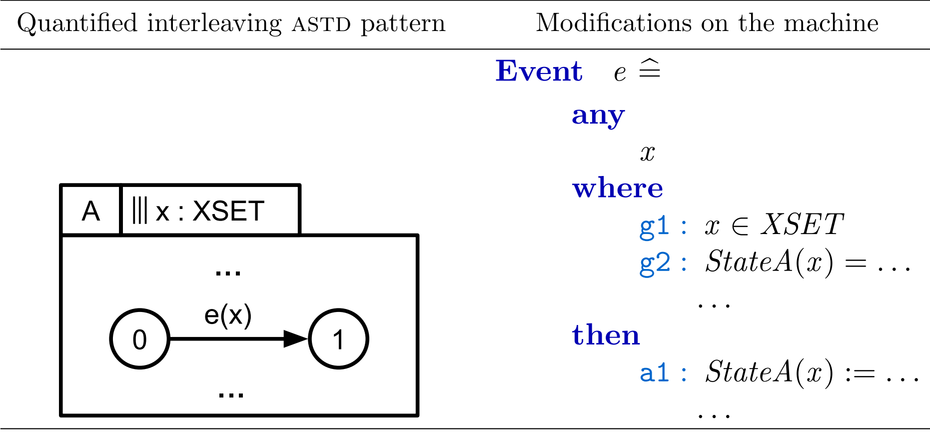
`readyInCustomer` : $StateCustomer(AssociationCustomer(c)) = cc$

`readyInDepartment` : $StateDepartment(AssociationDepartment(c)) = dc$

Theses guards check that the customer associated to the complaint c is in the state allowing him to complain *i.e.* created and not deleted, and if the department associated to the complaint c exists in the IS.

5.5.6 Quantified Interleaving

The quantified interleaving does not introduce additional constraints to events. The following table shows how variables induced by quantified interleaving are handled in events.

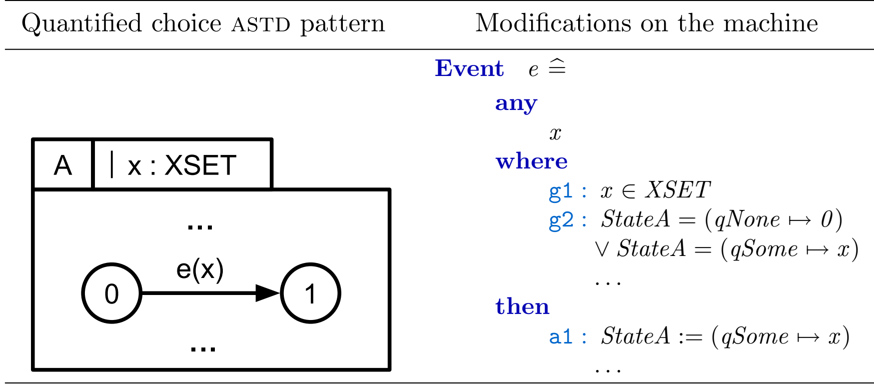


Quantified interleaving ASTD pattern	Modifications on the machine
	Event $e \mathrel{\widehat{=}}$ **any** x **where** g1 : $x \in XSET$ g2 : $StateA(x) = \ldots$ $\ldots$ **then** a1 : $StateA(x) := \ldots$ $\ldots$

Entities and associations patterns are common in EB^3 and ASTD as mentionned in [FSD03]. Such pattern are expressed using interleaving quantifications. In order to code in Event-B the association between several entities, a table variable must register their link. In our case study, we can see that a 1-n association between a customer and a complaint is created. When a complaint is created, an unique customer u is linked to the complaint c. The same applies to the department associated to the complaint. An Event-B variable is created in order to save the link between a complaint and a customer (respectively a department) and is updated whenever a complaint is registered in the system.

5.5.7 Quantified Choice

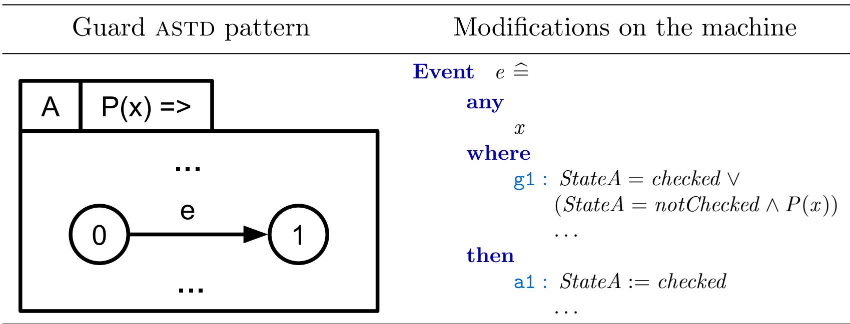
Similarly to the choice operator, the quantified choice implies that for all events using it, a check is performed about whether the choice was made or not. In the case of an action labeled `e` and taking x as a parameter, where x is the variable of a quantified choice ASTD named **A** then the guard `g2` described in the following table must hold. The substitution `a1` must also be executed in case this is the first call of an action with this quantified variable. All the events of ASTD **A** will be modified to include this guard and substitution.



Quantified choice ASTD pattern	Modifications on the machine
	Event $e \widehat{=}$ **any** x **where** g1 : $x \in XSET$ g2 : $StateA = (qNone \mapsto 0)$ $\vee\ StateA = (qSome \mapsto x)$ ... **then** a1 : $StateA := (qSome \mapsto x)$ ...

5.5.8 Guard

There are two cases for guard state : the guard was checked and held when we executed an event ; the guard did not hold, and no event was executed. These cases are handled with guard `g1` and substitution `a1` for a guard ASTD named **A** guarded with predicate $P(x)$. All the events of ASTD **A** will be modified to include this guard and substitution.



Guard ASTD pattern	Modifications on the machine
	Event $e \widehat{=}$ **any** x **where** g1 : $StateA = checked \vee$ $(StateA = notChecked \wedge P(x))$ ... **then** a1 : $StateA := checked$ ...

5.5.9 Process call

An ASTD that calls other ASTD does not need any constraint over its actions in Event-B. The translation will be achieved as if the entire called ASTD was substituted for the ASTD call. We do not deal with recursive ASTD calls yet.

When the translation process is completed, we can now access all the tools offered by Rodin to animate, model check and prove elements of the translated ASTD specification.

5.6 Animation and Model Checking of the Case Study

The final generated system, a context and a machine, translated from our case study represents 270 lines of Event-B, including 40 constants, 5 sets and 29 axioms for the static part and 7 variants, 7 invariants, 17 events (one for initialization, 13 representing ASTD actions and 3

internal events for Kleene Closure induced resets) representing 57 guards and 33 actions for the dynamic part. During the construction of translation rules, animation helped to correct rules, to improve the quality of translation rules and to factor contexts in order to separate static elements from machine. It was chosen to limit the size of quantification sets to three elements each. Only three departments, customers and complaints can be registered inside the system at any time. The screen capture was taken after the execution of 150 events and shows the state of variables of the machine. In order to informally verify the consistency of the Event-B machine with the initial ASTD specification, we generated a set of traces of events executed via the ProB animator. Then, for each trace, we removed the internal events introduced by the translation process such as `lambdaComplaint(c)`. Then we interpreted the initial ASTD specification with *i*ASTD and executed the traces. We could not find a trace of events that could not be interpreted by *i*ASTD. A more formal proof of the consistency of the translation must be performed, but first results are encouraging. Formal proof of translation rules is work in progress, and will be based on simulation.

Regarding the Event-B machine, 86 proof obligations were generated and 62 were automatically proved. The 24 remaining are proved manually and involved functional and set operators that are known for not being proved automatically. The manual proofs raised no specific difficulty. This Event-B specification was model checked for deadlocks and invariant violations using the consistency checking feature of ProB. More than 111 500 nodes were visited and 226 000 transitions activated. No deadlock nor invariant violation were found. More invariant properties might be written in order to be proved. Since ASTD only focuses on event control and not on event effects on the IS, when an event is executed, there is no way to know only by looking at the ASTD specification how IS state will evolve. Hence, no invariant can be generated during the translation. But it could be interresting to express invariants on ASTD as it was done with Statecharts [Sek08]. For instance we could add an invariant to ASTD **Department** saying that whenever transition DeleteDepartment(d) is active, no complaint about this department must be registred in the system.

5.7 Conclusion and Future Work

We have presented a set of translation rules allowing generation of Event-B contexts and machines from ASTD specifications. The animation of the resulting machine using ProB [LB03b] animator helped to find errors and to tweak translation rules. Kleene closure and sequence operators were the most tricky to translate since these operators defines the ordering of events and because they introduce additional events in order to code semantics of ASTD in Event-B. A formal proof of the translation rules will be performed in order to entrust the translation process.

Refinement is one of the most important features of Event-B modelling process. In our approach, this aspect is missing. Indeed, we are translating an ASTD specification modelling a concrete system. Because of that, there is no need to refine the Event-B machine resulting from the translation process. It would have been relevant to introduce refinement in the translation process if a similar notion existed in ASTD, but it is currently not the case. Proof is an important aspect of Event-B that our approach would like to take advantage of. Alongside with formal IS specification, we advocate writing security or functional properties during the modeling process. This way, properties can be checked against the system as soon as it is modeled. Expressing these properties as Event-B invariants and proving invariant preservation in the translated machine is an important step of IS specification validation. Another feature of Event-B we do not use is composition. This may be very useful for the translation of some ASTD operators such as synchronization. It could lead to a more modular approach of translation, in a way similar to ASTD.

It would be interesting to compare the machine resulting of the translation process with a hand-written Event-B specification for the same system. Indeed, we would like to know if the automatic prover can do the same job with the hand-written and the translated machine. This study is work in progress and may result in an evolution of translation rules. Another step that we currently work on is to implement an ASTD modeler as a Rodin plugin. Using benefits from The Eclipse Graphical Modeling Framework (GMF) [Ecl], a graphical editor could be used to build complete IS specifications. One could interpret them using the *i*ASTD [SM+10] interpreter and then translate them to Event-B on the fly in order to perform model checking or proofs. This integrated tool would allow a great flexibility and would combine advantages of process algebra's power of expression, graphical representation's ease of understanding and Event-B's tools for proving, checking and animating.

Annexe A

Spécifications B issues d'un modèle fonctionnel

Nous considérons uniquement les classes **Patient** et **ManagementAct** avec une relation de composition indiquant les actes de soins associés à chaque patient.

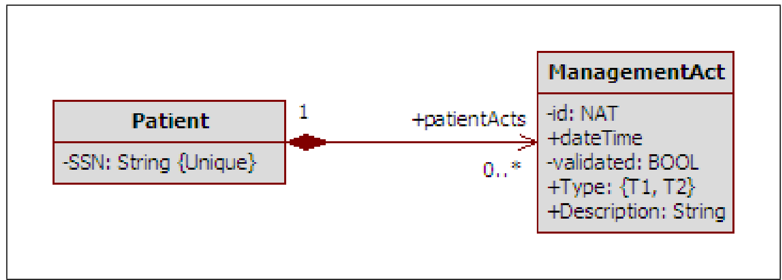


FIG. A.1 – Extrait du diagramme de classes de l'étude de cas IFREMMONT

MACHINE

$Functional_Model$

SETS

$PATIENTS$;

$MANAGEMENTACTS$;

$SSNs$;

$Types$;

$dateTimes$

VARIABLES

$Patients$, $patient_SSN$,

$ManagementActs$,

$patientActs$,

$managementact_validated$,

$managementact_Type$,

$managementact_dateTime$

INVARIANT

$Patients \subseteq PATIENTS \wedge$

$ManagementActs \subseteq MANAGEMENTACTS \wedge$

$patientActs \in ManagementActs \rightarrow Patients \wedge$
$patient_SSN \in Patients \rightarrowtail\!\!\!\!\rightarrow\!\!\!\!\!+ SSNs \wedge$
$managementact_validated \in ManagementActs \rightarrow \mathbf{BOOL} \wedge$
$managementact_Type \in ManagementActs \nrightarrow Types \wedge$
$managementact_dateTime \in ManagementActs \nrightarrow dateTimes \wedge$
$\mathbf{dom}(managementact_validated \rhd \{\mathbf{TRUE}\}) \subseteq \mathbf{dom}(managementact_Type) \wedge$
$\mathbf{dom}(managementact_validated \rhd \{\mathbf{TRUE}\}) \subseteq \mathbf{dom}(managementact_dateTime)$

INITIALISATION

$Patients := \emptyset \;||$
$ManagementActs := \emptyset \;||$
$patientActs := \emptyset \;||$
$patient_SSN := \emptyset \;||$
$managementact_validated := \emptyset \;||$
$managementact_Type := \emptyset \;||$
$managementact_dateTime := \emptyset$

OPERATIONS

createPatient$(obj) =$
PRE
$obj \in PATIENTS \wedge$
$obj \notin Patients$
THEN
$Patients := Patients \cup \{obj\}$
END;

deletePatient$(obj) =$
PRE
$obj \in PATIENTS \wedge$
$obj \in Patients \wedge$
$\mathbf{TRUE} \notin managementact_validated[patientActs^{-1}[\{obj\}]]$
THEN
$Patients := Patients - \{obj\} \;||$
$ManagementActs := ManagementActs - patientActs^{-1}[\{obj\}] \;||$
$managementact_validated := patientActs^{-1}[\{obj\}] \lhd\!\!\!- managementact_validated \;||$
$managementact_Type := patientActs^{-1}[\{obj\}] \lhd\!\!\!- managementact_Type \;||$
$managementact_dateTime := patientActs^{-1}[\{obj\}] \lhd\!\!\!- managementact_dateTime \;||$
$patientActs := patientActs \rhd\!\!\!- \{obj\} \;||$
$patient_SSN := \{obj\} \lhd\!\!\!- patient_SSN$
END;

patient_SetSSN$(obj) =$
PRE $obj \in PATIENTS \wedge obj \in Patients$ **THEN**
ANY ssn **WHERE**
$ssn \in SSNs \wedge ssn \notin \mathbf{ran}(patient_SSN)$
THEN

patient_SSN$(obj) := ssn$
END
END ;
$ssn \leftarrow$ **patient_GetSSN**$(obj) =$
PRE $obj \in PATIENTS \wedge obj \in Patients$ **THEN**
$ssn := patient_SSN(obj)$
END ;

patient_AddManagementAct$(obj) =$
PRE $obj \in PATIENTS \wedge obj \in Patients$ **THEN**
ANY ma **WHERE**
$ma \in MANAGEMENTACTS \wedge ma \notin ManagementActs \wedge$
$ma \notin$ **dom**$(patientActs)$
THEN
$ManagementActs := ManagementActs \cup \{ma\} \,||$
$patientActs := patientActs \cup \{(ma \mapsto obj)\} \,||$
managementact_validated$(ma) :=$ **FALSE**
END
END ;
patient_DeleteManagementAct$(obj) =$
PRE $obj \in PATIENTS \wedge obj \in Patients$ **THEN**
ANY ma **WHERE**
$ma \in MANAGEMENTACTS \wedge ma \in ManagementActs \wedge$
$ma \in patientActs^{-1} [\{obj\}] \wedge$
$managementact_validated(ma) =$ **FALSE**
THEN
$ManagementActs := ManagementActs$ - $\{ma\} \,||$
$patientActs := \{ma\} \lhd patientActs \,||$
$managementact_validated := \{ma\} \lhd managementact_validated \,||$
$managementact_dateTime := \{ma\} \lhd managementact_dateTime \,||$
$managementact_Type := \{ma\} \lhd managementact_Type$
END
END ;
$managementacts \leftarrow$ **patient_GetManagementActs** $(obj) =$
PRE $obj \in PATIENTS \wedge obj \in Patients$ **THEN**
$managementacts := patientActs^{-1} [\{obj\}]$
END ;
$validated \leftarrow$ **managementact_Getvalidated**$(obj) =$
PRE $obj \in MANAGEMENTACTS \wedge obj \in ManagementActs$ **THEN**
$validated := managementact_validated(obj)$
END ;
managementact_SetType$(obj) =$

```
    PRE
        obj ∈ MANAGEMENTACTS ∧
        obj ∈ ManagementActs ∧
        managementact_validated(obj) = FALSE
    THEN
      ANY type WHERE
        type ∈ Types
      THEN
        managementact_Type(obj) := type
      END
    END ;
type ← managementact_GetType(obj) =
    PRE obj ∈ MANAGEMENTACTS ∧ obj ∈ ManagementActs THEN
        type := managementact_Type(obj)
    END ;
managementact_SetdateTime(obj) =
    PRE
        obj ∈ MANAGEMENTACTS ∧
        obj ∈ ManagementActs ∧
        managementact_validated(obj) = FALSE
    THEN
      ANY datetime WHERE
        datetime ∈ dateTimes
      THEN
        managementact_dateTime(obj) := datetime
      END
    END ;
datetime ← managementact_GetdateTime(obj) =
    PRE obj ∈ MANAGEMENTACTS ∧ obj ∈ ManagementActs THEN
        datetime := managementact_dateTime(obj)
    END ;
managementact_Validate(obj) =
    PRE
      obj ∈ MANAGEMENTACTS ∧
      obj ∈ ManagementActs ∧
      managementact_validated(obj) = FALSE ∧
      obj ∈ dom(managementact_Type) ∧
      obj ∈ dom(managementact_dateTime)
    THEN
      managementact_validated(obj) := TRUE
    END
END
```

Annexe B

Spécifications B issues du modèle de sécurité

MACHINE
$\quad UserAssignements$
SETS
$\quad ROLES = \{Team_Doctor, Nurse, Operator, Team_Member, Regulator\};$
$\quad USERS = \{Bob, Paul, Martin, Jack, none\}$
VARIABLES
$\quad roleOf,$
$\quad Roles_Hierarchy,$
$\quad currentUser$
INVARIANT
$\quad Roles_Hierarchy \in ROLES \leftrightarrow ROLES \wedge$
$\quad roleOf \in USERS \rightarrow \mathbb{P}\ (ROLES) \wedge$
$\quad \mathbf{closure1}(Roles_Hierarchy) \cap \mathbf{id}(ROLES) = \emptyset \wedge$
$\quad currentUser \in USERS$
INITIALISATION
$\quad roleOf := \{(Bob \mapsto \{Team_Doctor\}),$
$\qquad\qquad (Paul \mapsto \{Operator\}),$
$\qquad\qquad (Martin \mapsto \{Nurse\}),$
$\qquad\qquad (Jack \mapsto \{Regulator\}),$
$\qquad\qquad (none \mapsto \emptyset\ )\}\ ||$
$\quad Roles_Hierarchy := \{(Team_Doctor \mapsto Team_Member),$
$\qquad\qquad\qquad (Nurse \mapsto Team_Member)\}\ ||$
$\quad currentUser := none$
OPERATIONS
$\quad$ **changeUser**$(user) =$
$\quad$ **PRE**
$\qquad user \in USERS$
$\quad$ **THEN**
$\qquad currentUser := user$
$\quad$ **END**
END

MACHINE

RBAC_Model

INCLUDES

Functional_Model,

UserAssignements

SETS

$ENTITIES = \{Patient, ManagementAct\}$;

$Attributes = \{SSN, Validated, dateTime, Type\}$;

Operations = {*CreatePatient*, *DeletePatient*, *Patient_AddManagementAct*, *Patient_SetSSN*, *Patient_GetSSN*,
Patient_DeleteManagementAct, *Patient_GetManagementActs*, *Managementact_SetdateTime*, *Managementact_GetdateTime*,
Managementact_SetValidated, *Managementact_GetValidated*, *Managementact_SetType*, *Managementact_GetType*,
Managementact_Validate
} ;

$KindsOfAtt = \{public, private\}$;

$PERMISSIONS = \{PatientPerm1, PatientPerm2, ManagementActPerm1, ManagementActPerm2\}$;

$ActionsType = \{read, create, modify, delete, privateRead, privateModify\}$;

$Stereotypes = \{readOp, modifyOp\}$

VARIABLES

AttributeKind, *AttributeOf*, *OperationOf*,

constructorOf, *destructorOf*, *setterOf*, *getterOf*,

PermissionAssignement, *EntityActions*,

MethodActions, *StereotypeOps*,

isPermitted

INVARIANT

$AttributeKind \in Attributes \rightarrow KindsOfAtt \land$

$AttributeOf \in Attributes \rightarrow ENTITIES \land$

$OperationOf \in Operations \rightarrow ENTITIES \land$

$constructorOf \in Operations \rightarrowtail\!\!\!\!\!+ \; ENTITIES \land$

$destructorOf \in Operations \rightarrowtail\!\!\!\!\!+ \; ENTITIES \land$

$setterOf \in Operations \rightarrowtail\!\!\!\!\!+ \; Attributes \land$

$getterOf \in Operations \rightarrowtail\!\!\!\!\!+ \; Attributes \land$

$StereotypeOps \in Stereotypes \leftrightarrow Operations \land$

$setterOf \cap getterOf = \emptyset \land$

$PermissionAssignement \in PERMISSIONS \rightarrow (ROLES \times ENTITIES) \land$

$EntityActions \in PERMISSIONS \nrightarrow \mathbb{P}\,(ActionsType) \land$

$MethodActions \in PERMISSIONS \nrightarrow \mathbb{P}\,(Operations) \land$

$isPermitted \in ROLES \leftrightarrow Operations$

DEFINITIONS

$allEntityActions == \{pp, at \mid pp \in PERMISSIONS \land at \in ActionsType$
$\land\; pp \in \mathbf{dom}(EntityActions) \land at \in EntityActions(pp)\}$;

$PermEntitiesCreation == \mathbf{ran}(\{create\} \lhd (allEntityActions^{-1} ; PermissionAssignement))$;

$PermOpCreation == (PermEntitiesCreation ; constructorOf^{-1})$;

$PermEntitiesDestruction == \mathbf{ran}(\{delete\} \lhd (allEntityActions^{-1} ; PermissionAssignement))$;

$PermOpDestruction == (PermEntitiesDestruction ; destructorOf^{-1})$;

$PermEntitiesPRead == \mathbf{ran}(\{privateRead\} \lhd (allEntityActions^{-1} ; PermissionAssignement))$;

$PermOpPRead == (PermEntitiesPRead ; (getterOf ; AttributeOf)^{-1})$;

$publicGetters == getterOf \rhd \mathbf{dom}(AttributeKind \rhd \{public\})$;

$PermEntitiesRead == \mathbf{ran}(\{read\} \lhd (allEntityActions^{-1} ; PermissionAssignement))$;

$PermOpRead == (PermEntitiesRead ; (publicGetters ; AttributeOf)^{-1})$;

$PermEntitiesPModify == \mathbf{ran}(\{privateModify\} \lhd (allEntityActions^{-1} ; PermissionAssignement))$;
$PermOpPModify == (PermEntitiesPModify ; (setterOf ; AttributeOf)^{-1})$;

$publicSetters == setterOf \rhd \mathbf{dom}(AttributeKind \rhd \{public\})$;
$PermEntitiesModify == \mathbf{ran}(\{modify\} \lhd (allEntityActions^{-1} ; PermissionAssignement))$;
$PermOpModify == (PermEntitiesModify ; (publicSetters ; AttributeOf)^{-1})$;

$PermEntitiesAbsoluteRead == \mathbf{ran}(\{privateRead, read\} \lhd (allEntityActions^{-1} ; PermissionAssignement))$;
$PermEntitiesAbsoluteModify == \mathbf{ran}(\{privateModify, modify\} \lhd (allEntityActions^{-1} ; PermissionAssignement))$;
$PermOpReadOps == (PermEntitiesAbsoluteRead ; (StereotypeOps[\{readOp\}] \lhd OperationOf)^{-1})$;
$PermOpModifyOps == (PermEntitiesAbsoluteModify ; (StereotypeOps[\{modifyOp\}] \lhd OperationOf)^{-1})$;

$PermOpMethodAction == \{ro, op \mid ro \in ROLES \land op \subseteq Operations \land$
$ro \in \mathbf{dom}((MethodActions^{-1} ; PermissionAssignement)[\{op\}])\}$;
$PermOpMethodActions == \{ro, op \mid ro \in ROLES \land op \in Operations \land op \in \mathbf{union}(PermOpMethodAction[\{ro\}])\}$;

$currentRole == (roleOf(currentUser) \cup \mathbf{ran}(roleOf(currentUser) \lhd \mathbf{closure1}(Roles_Hierarchy)))$;
$permissions == PermOpCreation \cup$
$PermOpDestruction \cup$
$PermOpPRead \cup$
$PermOpReadOps \cup$
$PermOpRead \cup$
$PermOpPModify \cup$
$PermOpModifyOps \cup$
$PermOpModify \cup$
$PermOpMethodActions$

INITIALISATION

$AttributeKind := \{(SSN \mapsto private),$
$(Validated \mapsto private),$
$(dateTime \mapsto public),$

$$
\begin{aligned}
&(Type \mapsto public)\} \\
&|| \\
&AttributeOf := \{(SSN \mapsto Patient), \\
&\quad (Validated \mapsto ManagementAct), \\
&\quad (dateTime \mapsto ManagementAct), \\
&\quad (Type \mapsto ManagementAct)\} \\
&|| \\
&OperationOf := \{(CreatePatient \mapsto Patient), \\
&\quad (DeletePatient \mapsto Patient), \\
&\quad (Patient_AddManagementAct \mapsto ManagementAct), \\
&\quad (Patient_SetSSN \mapsto Patient), \\
&\quad (Patient_GetSSN \mapsto Patient), \\
&\quad (Patient_DeleteManagementAct \mapsto Patient), \\
&\quad (Patient_GetManagementActs \mapsto Patient), \\
&\quad (Managementact_SetdateTime \mapsto ManagementAct), \\
&\quad (Managementact_GetdateTime \mapsto ManagementAct), \\
&\quad (Managementact_SetValidated \mapsto ManagementAct), \\
&\quad (Managementact_GetValidated \mapsto ManagementAct), \\
&\quad (Managementact_SetType \mapsto ManagementAct), \\
&\quad (Managementact_GetType \mapsto ManagementAct), \\
&\quad (Managementact_Validate \mapsto ManagementAct)\} \\
&|| \\
&constructorOf := \{(CreatePatient \mapsto Patient)\} \\
&|| \\
&destructorOf := \{(DeletePatient \mapsto Patient)\} \\
&|| \\
&StereotypeOps := \{(modifyOp \mapsto Patient_SetSSN), \\
&\quad (readOp \mapsto Managementact_GetValidated)\} \\
&|| \\
&setterOf := \{(Patient_SetSSN \mapsto SSN), \\
&\quad (Managementact_SetdateTime \mapsto dateTime), \\
&\quad (Managementact_SetValidated \mapsto Validated), \\
&\quad (Managementact_SetType \mapsto Type)\} \\
&|| \\
&getterOf := \{(Patient_GetSSN \mapsto SSN), \\
&\quad (Managementact_GetdateTime \mapsto dateTime), \\
&\quad (Managementact_GetValidated \mapsto Validated), \\
&\quad (Managementact_GetType \mapsto Type)\} \\
&|| \\
&PermissionAssignement := \{(PatientPerm1 \mapsto (Operator \mapsto Patient)), \\
&\quad (PatientPerm2 \mapsto (Team_Member \mapsto Patient)), \\
&\quad (ManagementActPerm1 \mapsto (Team_Member \mapsto ManagementAct)), \\
&\quad (ManagementActPerm2 \mapsto (Team_Doctor \mapsto ManagementAct))\} \\
&|| \\
&EntityActions := \{(PatientPerm1 \mapsto \{create, modify\}), \\
&\quad (PatientPerm2 \mapsto \{privateRead\}), \\
&\quad (ManagementActPerm1 \mapsto \{read, modify\})\} \\
&|| \\
&MethodActions := \{(PatientPerm2 \mapsto \{Patient_AddManagementAct\}), \\
&\quad (ManagementActPerm2 \mapsto \{Managementact_Validate\})\}
\end{aligned}
$$

```
  ||
    isPermitted := ∅
OPERATIONS
setPermissions = PRE isPermitted = ∅ THEN isPermitted := permissions END ;
secure_createPatient(obj) =
    PRE obj ∈ PATIENTS ∧ obj ∉ Patients THEN
        SELECT
            CreatePatient ∈ isPermitted[currentRole]
        THEN
            createPatient(obj)
        END
    END ;
secure_deletePatient(obj) =
    PRE obj ∈ PATIENTS ∧ obj ∈ Patients THEN
        SELECT
            DeletePatient ∈ isPermitted[currentRole]
        THEN
            deletePatient(obj)
        END
    END ;
secure_patient_SetSSN(obj) =
    PRE obj ∈ PATIENTS ∧ obj ∈ Patients THEN
        SELECT
            Patient_SetSSN ∈ isPermitted[currentRole]
        THEN
            patient_SetSSN(obj)
        END
    END ;
ssn ← secure_patient_GetSSN(obj) =
    PRE obj ∈ PATIENTS ∧ obj ∈ Patients THEN
        SELECT
            Patient_GetSSN ∈ isPermitted[currentRole]
        THEN
            ssn ← patient_GetSSN(obj)
        END
    END ;
secure_patient_AddManagementAct(obj) =
    PRE obj ∈ PATIENTS ∧ obj ∈ Patients THEN
        SELECT
            Patient_AddManagementAct ∈ isPermitted[currentRole]
        THEN
            patient_AddManagementAct(obj)
        END
    END ;
secure_patient_DeleteManagementAct(obj) =
    PRE obj ∈ PATIENTS ∧ obj ∈ Patients THEN
        SELECT
            Patient_DeleteManagementAct ∈ isPermitted[currentRole]
        THEN
            patient_DeleteManagementAct(obj)
```

```
        END
    END ;
managementacts ← secure_patient_GetManagementActs(obj) =
    PRE obj ∈ PATIENTS ∧ obj ∈ Patients THEN
        SELECT
            Patient_GetManagementActs ∈ isPermitted[currentRole]
        THEN
            managementacts ← patient_GetManagementActs(obj)
        END
    END ;

validated ← secure_managementact_Getvalidated(obj) =
    PRE obj ∈ MANAGEMENTACTS ∧ obj ∈ ManagementActs THEN
        SELECT
            Managementact_GetValidated ∈ isPermitted[currentRole]
        THEN
            validated ← managementact_Getvalidated(obj)
        END
    END ;

secure_managementact_SetType(obj) =
    PRE obj ∈ MANAGEMENTACTS ∧ obj ∈ ManagementActs THEN
        SELECT
            Managementact_SetType ∈ isPermitted[currentRole]
        THEN
            managementact_SetType(obj)
        END
    END ;
type ← secure_managementact_GetType(obj) =
    PRE obj ∈ MANAGEMENTACTS ∧ obj ∈ ManagementActs THEN
        SELECT
            Managementact_GetType ∈ isPermitted[currentRole]
        THEN
            type ← managementact_GetType(obj)
        END
    END ;
secure_managementact_SetdateTime(obj) =
    PRE obj ∈ MANAGEMENTACTS ∧ obj ∈ ManagementActs THEN
        SELECT
            Managementact_SetdateTime ∈ isPermitted[currentRole]
        THEN
            managementact_SetdateTime(obj)
        END
    END ;
datetime ← secure_managementact_GetdateTime(obj) =
    PRE obj ∈ MANAGEMENTACTS ∧ obj ∈ ManagementActs THEN
        SELECT
            Managementact_GetdateTime ∈ isPermitted[currentRole]
        THEN
            datetime ← managementact_GetdateTime(obj)
```

```
        END
    END ;
secure_managementact_Validate(obj) =
    PRE obj ∈ MANAGEMENTACTS ∧ obj ∈ ManagementActs THEN
        SELECT
            Managementact_Validate ∈ isPermitted[currentRole]
        THEN
            managementact_Validate(obj)
        END
    END ;

changeCurrentUser(user) =
    PRE user ∈ USERS ∧ isPermitted ≠ ∅ THEN
        changeUser(user)
    END
END
```

Bibliographie

[ABHV06] Jean Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. *Lecture Notes in Computer Science*, 4260 :588, 2006.

[Abr96a] Jean Raymond Abrial. *The B-Book : Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Abr96b] J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

[ACCBCB08] Fabien Autrel, Frédéric Cuppens, Nora Cuppens-Boulahia, and Céline Coma-Brebel. MotOrBAC 2 : a security policy tool. In *SARSSI'08 : 3e conf. sur la Sécurité des Architectures Réseaux et des Systèmes d'Information*. (Télécom Bretagne), 2008.

[And95] Pascal André. *Méthodes formelles à Objets pour le développement du logiciel : études et propositions*. PhD thesis, Université de Rennes 1, Juillet 1995.

[BBM03] Philippe Bon, Jean-Louis Boulanger, and Georges Mariano. Semi formal modelling and formal specification : UML & B in simple railway application. In *ICSSEA'03*, 2003.

[BCDE09] David A. Basin, Manuel Clavel, Jürgen Doser, and Marina Egea. Automated analysis of security-design models. *Information & Software Technology*, 51(5) :815–831, 2009.

[BDL06] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security : From uml models to access control infrastructures. *ACM TOSEM*, 15(1) :39–91, 2006.

[BGG+93] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In Victoria Stavridou, Thomas F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design : Theory, Practice and Experience*, volume A-10 of *IFIP Transactions A : Computer Science and Technology*, pages 129–156, Nijmegen, The Netherlands, 1993. North-Holland.

[But00] Michael Butler. csp2b : A practical approach to combining CSP and b. *Formal Aspects of Computing*, 12(3) :182–198, November 2000.

[CPP+05] Samuel Colin, Dorian Petit, Vincent Poirriez, Jérôme Rocheteau, Rafael Marcano, and Georges Mariano. BRILLANT : An Open Source and XML-based platform for Rigourous Software Development. In Bernhard K. Aichernig and Bernhard Beckert, editors, *SEFM*, pages 373–382. IEEE Computer Society, 2005.

[CW87] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–195, 1987.

[DDR03] D.F.Ferraiolo, D.R.Kuhn, and R.Chandramouli. *Role-Based Access Control*. Computer Security Series. Artech House, 2003.

[DvK92] Eugène Dürr and Jan van Katwijk. VDM++ : a formal specification language for object-oriented designs. In *Proceedings of the seventh international conference on Technology of object-oriented languages and systems*, pages 63–77, Hertfordshire, UK, 1992. Prentice Hall International (UK) Ltd.

[Ecl] Eclipse Consortium. Eclipse graphical modeling framework (gmf).

[F+07] Benoît Fraikin et al. Synthesizing information systems : the APIS project. In Colette Rolland, Oscar Pastor, and Jean-Louis Cavarero, editors, *First International Conference on Research Challenges in Information Science (RCIS)*, page 12, Ouarzazate, Morocco, April 2007.

[FF09] B. Fraikin and M. Frappier. Efficient symbolic computation of process expressions. *Science of Computer Programming*, 2009.

[FGL+08] M. Frappier, F. Gervais, R. Laleau, B. Fraikin, and R. St-Denis. Extending statecharts with process algebra operators. *Innovations in Systems and Software Engineering*, 4(3) :285–292, 2008.

[FKV91] Martin D. Fraser, Kuldeep Kumar, and Vijay K. Vaishnavi. Informal and formal requirements specification laguages : Bridging the gap. *IEEE Transactions on Software Engineering*, 17(5) :454–465, 1991.

[FL95] P. Facon and Régine Laleau. Des spécifications informelles aux spécifications formelles : compilation ou interprétation ? In *Actes du 13ème congrès INFORSID*, 1995.

[FSD03] Marc Frappier and Richard St-Denis. EB^3 : an entity-based black-box specification method for information systems. *Software and System Modeling*, 2(2) :134–149, 2003.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. USE : A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, 69(1-3) :27–34, 2007.

[Har87] D. Harel. Statecharts : A visual formalism for complex systems. *Science of computer programming*, 8(3) :231–274, 1987.

[HLMK04] Lotfi Hazem, Nicole Levy, and Rafael Marcano-Kamenoff. UML2B : un outil pour la génération de modèles formels. In Jacques Julliand, editor, *AFADL'2004 - Session Outils*, 2004.

[Hoa85] Charles Antony Richard Hoare. CSP–Communicating Sequential Processes. *Prenctice Hall*, 1985.

[Ida06] Akram Idani. *B/UML : Mise en relation de spécifications B et de descriptions UML pour l'aide à la validation externe de développements formels en B*. PhD thesis, Université de Grenoble 1 – France, Novembre 2006.

[IL10] Akram Idani and Mohamed-Amine Labiadhand Yves Ledru. Infrastructure dirigée par les modèles pour une intégration adaptable et évolutive de UML et B. *RSTI - Ingénierie des Systèmes d'Information (ISI)*, 15(3), 2010.

[ISO02] ISO. *Information technology – Z formal specification notation – Syntax, type system and semantics*, 2002.

[KFL10] Pierre Konopacki, Marc Frappier, and Regine Laleau. Expressing access control policies with an event-based approach. Technical Report TR-LACL-2010-6, LACL (Laboratory of Algorithms, Complexity and Logic), University of Paris-Est (Paris 12), 2010.

[Lal02] Régine Laleau. Conception et développement formels d'applications bases de données. Hdr, Université d'Evry, 2002.

[Lan92] Kevin Lano. Z^{++}. In Susan Stepney, Rosalind Barden, and David Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 105–112. Springer, 1992.

[Lan95] Kevin Lano. *Formal Object-Oriented Development*. Springer-Verlag New York, Inc., USA, 1995.

[Lan98] Kevin Lano. Logical specification of reactive and real-time systems. *Journal of Logic and Computation*, 8(5) :679–711, 1998.

[LB03a] M. Leuschel and M. Butler. ProB : A Model Checker for B. In *FME 2003 : Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer-Verlag, 2003.

[LB03b] Michael Leuschel and Michael Butler. ProB : A model checker for b. In *FME 2003 : Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer Berlin / Heidelberg, 2003.

[LB08] Michael Leuschel and Michael J. Butler. ProB : an automated analysis toolset for the B method. *STTT*, 10(2) :185–203, 2008.

[LCA04] Kevin Lano, David Clark, and Kelly Androutsopoulos. UML to B : Formal Verification of Object-Oriented Models. In *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 187–206. Springer, 2004.

[Led02] Hung Ledang. *Traduction systématique de spécifications UML en B*. PhD thesis, Université de Nancy 2, 2002.

[LM00] Régine Laleau and Amel Mammar. An Overview of a Method and Its Support Tool for Generating B Specifications from UML Notations. In *15th IEEE International Conference on Automated Software Engineering*, pages 269–272, 2000. IEEE Computer Society Press.

[LSC03] Hung Ledang, Jeanine Souquières, and Sébastien Charles. Argo/UML+B : un outil de transformation systématique de spécification UML en B. In *AFADL'2003*, pages 3–18, January 2003.

[Mam02] Amel Mammar. *Un environnement formel pour le développement d'applications bases de données*. PhD thesis, CNAM-Paris, Novembre 2002.

[Mar02] Rafael Marcano. *Spécification formelle à objets en UML/OCL et B : une approche transformationnelle*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2002.

[Mey01] Eric Meyer. *Développements formels par objets : utilisation conjointe de B et d'UML*. PhD thesis, Université de Nancy 2, Mars 2001.

[MFGL10] Jérémy Milhau, Marc Frappier, Frédéric Gervais, and Régine Laleau. Systematic translation rules from ASTD to event-B. In *8th Int. Conf. on Integrated Formal Methods*, LNCS, 2010.

[ML04] Amal Mammar and Régine Laleau. UML2SQL : Un environment intégré pour le développement d'implémentations relationnelles à partir de diagrammes UML. In Jacques Julliand, editor, *AFADL'2004 - Session Outils*, 2004.

[ML06] Amel Mammar and Régine Laleau. A formal approach based on UML and B for the specification and development of database applications. *Automated Software Engineering*, 13(4) :497–528, 2006.

[Ngu98] Hong Phuong Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. PhD thesis, CNAM-Paris, Décembre 1998.

[Oss06] D. Okalas Ossami. *Aide à la construction de spécifications multi-vues UML et B.* PhD thesis, Université de Nancy 2, Octobre 2006.

[RBL+90] J. Rumbaugh, M. Blaha, W. Lorensen, F. Eddy, and W. Premerlani. *Object-Oriented Modeling and Design.* Prentice Hall, 1990.

[RJB96] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The unified modeling language.* University Video Communications, 1996.

[SB04] Colin Snook and Michael Butler. U2B-A tool for translating UML-B models into B. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*, 2004.

[SB06a] C. Snook and M. Butler. UML-B : Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1) :122, 2006.

[SB06b] Colin Snook and Michael Butler. UML-B : Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1) :92–122, 2006.

[SB06c] Colin Snook and Michael Butler. UML-B : Formal modeling and design aided by UML. *ACM Transactions on Software Engineering Methodology*, 15(1) :92–122, 2006.

[SBS09] Mar Yah Said, Michael Butler, and Colin Snook. Language and tool support for class and state machine refinement in UML-B. In *FM 2009 : Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 579–595. Springer Berlin / Heidelberg, 2009.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2) :38–47, 1996.

[SDAG08] Karsten Sohr, Michael Drouineaud, Gail-Joon Ahn, and Martin Gogolla. Analyzing and managing role-based access control policies. *IEEE Trans. Knowl. Data Eng.*, 20(7) :924–939, 2008.

[Sek08] E. Sekerinski. Verifying Statecharts with State Invariants. In *13th IEEE International Conference on Engineering of Complex Computer Systems*, pages 7–14. IEEE, 2008.

[SM+10] K. Salabert, J. Milhau, et al. iASTD : un interpréteur pour les ASTD. In *Atelier Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 2010)*, pages 3–6, Poitiers, France, 9-11 June 2010. Actes AFADL.

[Smi95] Graeme Smith. *The Object-Z Specification Language.* Advances in Formal Methods Series. Kluwer Academic Publishers, 1995.

[SZ02] Emil Sekerinski and Rafik Zurob. Translating statecharts to b. In *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 128–144. Springer Berlin / Heidelberg, 2002.

[Utt05] Mark Utting. *Jaza User Manual and Tutorial*, 2005. http ://www.cs.waikato.ac.nz/˜marku/jaza/.

[Van98] W. M. P. Van Der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1) :21–66, 1998.

[WK98] Jos B. Warmer and Anneke G. Kleppe. *The Object Constraint Language : Precise Modeling With UML.* Addison-Wesley, October 1998.

[WN04] Martin Wildmoser and Tobias Nipkow. Certifying Machine Code Safety : Shallow versus Deep Embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223, pages 305–320, 2004.