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 3.2

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

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

Février 2011

Table des matières

Chapitre 1

Introduction

Le présent livrable est une mise à jour du livrable 3.1. Il constitue d'une part un approfondissement des divers concepts que nous avons développés précédemment, et il présente d'autre part, les nouvelles pistes que nous avons exploitées.

Rappelons que les livrables 3.1 et 3.2 s'inscrivent 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 document 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 et aussi en Z. 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 (i) le modèle fonctionnel, (ii) le modèle de contrôle d'accès statique, et (iii) le modèle de contrôle d'accès dynamique.

Ce document est structuré comme suit :

- Chapitre 2 : dresse un état de l'art des travaux de couplage de notations formelles en B et semi-formelles en UML en mettant l'accent sur leur intérêt et leurs usages.
- Chapitre 3 : discute l'intérêt de la prise en compte du modèle fonctionnel lors des activités de vérification et de validation et montre l'impact de ce modèle sur l'étude des aspects sécuritaires d'un système.
- Chapitre 4 : donne les principes de la formalisation en Z du modèle fonctionnel et du modèle de sécurité ainsi que l'intérêt d'une telle démarche pour l'animation de scénarios
- Chapitre 5 : montre par la pratique comment notre noyau de sécurité - exprimé en Z - peut être exploité pour expliciter (au moyen de requêtes) certaines propriétés d'une politique de sécurité.
- Chapitre 6 : donne les principales règles de génération de spécifications B à partir du modèle fonctionnel et du modèle de contrôle d'accès statique. Ces règles ont été énoncées en vue d'être automatisées et outillées dans notre outil IDM de transformation.

- Chapitre 7 : présente les fondements de la formalisation du contrôle d'accès dynamique. Dans ce travail, le formalisme des ASTD est utilisé pour exprimer graphiquement les aspects dynamiques et B en vue raisonner formellement sur ces modèles.
- Chapitre 8 : présente un outil d'interprétation des ASTD dit iASTD et donne vie aux ASTD.
- Chapitre 9 : démontre rigoureusement les règles de transformation des ASTD en B et met l'accent sur la justesse des choix réalisés.

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, Pas95] 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 [Pas95] 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++ [VDM92].

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, Reg02], 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, Reg02]

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, UML04], `U2B` [SB04], `ArgoUML+B` [Arg03]. 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) [GBR07b] 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 [BCDE09b], 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 [BCDE09b] 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 [Utt05b] 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 [?] 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

Specification security-design models using Z

This chapter is aimed at formally specifying and validating security-design models of an information system. It combines semi-formal and formal methods, integrating specification languages such as UML and an extension, SecureUML, with the Z language. The modeled system addresses both functional and security requirements of a given application. The functional specification is built automatically from the UML diagram, using our RoZ tool. The secure part of the model uses a generic security-kernel written in Z, free from applications specificity, which implements the concepts of RBAC (Role-Based Access Control). The final modeling step is to create a link between the functional model and the security kernel. The integrated model is then animated using the Jaza tool, which contributes to its validation. Our approach is demonstrated on a case-study from the health care sector where confidentiality and integrity appear as core challenges to protect medical records.

4.1 Introduction

It is often stated that most of the faults of software systems are traced back to deficiencies in specifications [Mar03]. In secure information systems, specifications include functional aspects, describing how information is processed, and security aspects, which describe who may access the functionalities of the information system. Separation of concerns tends to separate functional and security models. But since the security model refers to elements of the functional one, it is necessary to integrate them to fully address security concerns and perform more complete security analyses. One of the key considerations in secure systems development is to evolve an integrated model of functional and non-functional aspects right from the beginning. This gives rise to the concept of security-design models (e.g.,[Jür04], [BDT06]). Such models can adequately support the study of malicious attacks and threats that are otherwise confronted in the installed systems. Latest advancements [Jür04] in security critical systems have stated that security properties cannot be retrofitted.

This chapter tries to combine formal and semi-formal methods, in order to incorporate the precision of formal languages into intuitive graphical models. As for SecureUML [BDT06], we propose to express functional and security models as UML diagrams. These diagrams are then translated into a single formal specification, expressed in the Z language [Spi92]. We then use Jaza [Utt05a] to animate the model and contribute to its validation. Validation is performed by asking queries about the access control rules, as done in the SecureMova tool [BCDE09a]. It is also achieved by playing scenarios, which lead the system through several state changes and

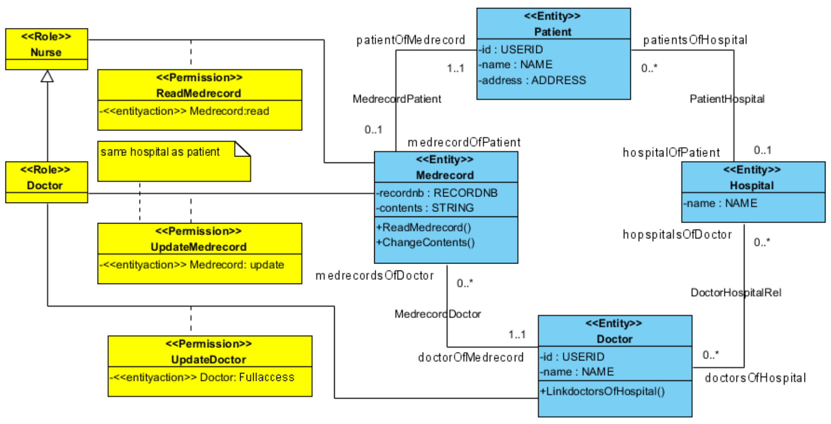


FIG. 4.1 – Security policy model using SecureUML

involve both the security and the functional model. Such dynamic scenarios can exhibit security flaws, which cannot be detected by static queries.

In the past, we developed RoZ, a tool which transforms a UML class diagram, annotated with Z assertions, into a Z specification [DLCP00]. The resulting Z specifications can be animated using Jaza [Led06]. Our mid-term goal is to upgrade RoZ in order to address security concerns in our UML models.

Our security model is based on RBAC (Role-Based Access Control) [FSG$^+$01] and SecureUML [BCDE09a]. Several attempts have been made to specify RBAC in Z [FSG$^+$01],[AK06],[PSS08]. Most of them are specifications of the RBAC meta-model. As far as we know, none of these has been used in conjunction with an animator in order to validate a given security policy. Currently, several tools exploit OCL in order to validate RBAC or SecureUML specifications. Sohr et al [SDAG08] have adapted the USE OCL tool [GBR07a] for the analysis of security policies. SecureMova [BCDE09a] is a tool dedicated to SecureUML, an extension of RBAC. It allows to query the security policy, and can also evaluate which actions are permitted for a given role in a given context, depicted as an object diagram. Still, both tools only address the functional model statically, i.e. they don't animate the operations of the functional model.

This chapter presents our translation of functional and security models into Z and how these can be validated, using the Jaza animator. Our approach has the following goals : 1) modeling a secure system and stating its properties in terms of Z annotations, 2) comprehensible and easily expressible specifications due to the fact that graphical models are employed, 3) validation by queries over specified access-control policy of a system in question, and 4) validation through animation of the integrated model.

The chapter is structured as follows. Sect. 4.2 introduces an illustrative example. Sect. 4.3 recalls the principles of the translation of the functional model, while Section 4.4 features the specification of the security kernel. The integration of both models is described in Sect. 4.5. Section 4.6 features the validation activities, based on animation. Finally, Sect. 4.7 presents related work and conclusion along with future directions appear in Sect. 5.7.

4.2 Illustrative example : medical information system

Fig. 4.1 models a simple medical information system. The figure has two sides where functional features on the right, are decoupled from security features on the left. The functional part describes four classes : patients, doctors, hospitals and medical records. Each medical record corresponds to exactly one patient. Its field *contents* stores confidential information whose integrity

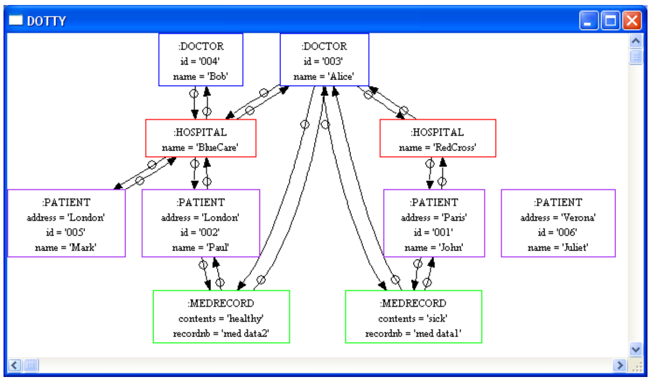


Fig. 4.2 – Object diagram for the functional model produced from the output of Jaza

must be preserved. The functional part also records the current hospital hosting the patient, the doctors working in this hospital, and the one responsible for the patient's medical record. Fig. 5.6 gives an object diagram corresponding to this functional model. It features 4 patients, 2 doctors (Alice and Bob), 2 medical records, and 2 hospitals. Alice is linked to both hospitals, while Bob only works for one of them.

The left part of Fig. 4.1 describes the access control rules of the information system. In SecureUML and RBAC, users of the system are abstracted into roles, and permissions are granted to roles. Fig. 4.1 features two roles : *Nurse* and *Doctor*. An inheritance relation links *Doctor* to *Nurse*, expressing that doctors inherit all permissions of nurses. Confidentiality and integrity must be ensured for medical records. Two permissions rule the access to class Medrecord. Permission *ReadMedRecord* is granted to nurses (and inherited by doctors). It expresses that nurses and doctors have read access to medical records. It refers to *entity action read* which designates operations accessing the class without modifying it. Permission *UpdateMedrecord* grants additional rights to doctors, who may update medical records. A constraint *Same hospital as patient* restricts this permission to the doctors linked to the same hospital as the patient. In Fig. 5.6, it means that only Alice has the right to modify the medical record of John, numbered "med data1", because she is the only doctor linked to RedCross hospital. In SecureUML, such constraints are expressed in OCL ; here, they will be expressed in the Z language. In Fig. 4.1, a third permission grants to all doctors full access, i.e. read and update access, to objects of class Doctor.

We will study the following attack scenario : Bob, a malicious doctor, wants to corrupt the integrity of John's medical record. Since Bob is not working for RedCross hospital, the access control rules should deny him this modification.

4.3 Translating the functional model into Z

The RoZ tool automatically translates the functional model, corresponding to the right part of Fig. 4.1, into a Z specification. RoZ [DLCP00] transforms a UML class diagram, annotated with Z assertions, into a single Z specification. The tool also generates basic operations such as setters and getters for the attributes and the associations.

Here are some elements of the formal specification generated from the functional diagram of the medical record information system. First, the types of the class attributes are introduced as given types.

$[NAME, USERID, STRING, RECORDNB]$

Every class is translated into two Z schemas. The first one, a schema type, describes the type of the elements of the class. This schema corresponds to the class intent and lists the class attributes. Schemas *MEDRECORD* and *DOCTOR* describe the intent of the corresponding classes. A second schema describes the extension of the class, i.e. the set of objects belonging to the class. Schemas *MedrecordExt* and *DoctorExt* correspond to these extensions ; each of these includes a finite set of objects corresponding to the type of the class.

MEDRECORD

$recordnb : RECORDNB$

$contents : STRING$

DOCTOR

$id : USERID$

$name : NAME$

MedrecordExt

$Medrecord : \mathbb{F}\, MEDRECORD$

DoctorExt

$Doctor : \mathbb{F}\, DOCTOR$

During a Jaza animation, each object is represented as a list of pairs *attribute* == *value*. The list is enclosed between $\langle\!|\ \ldots\ |\!\rangle$. Here is the Jaza representation of sets *Doctor* and *Medrecord* corresponding to the state of Fig. 5.6.

$$
\begin{aligned}
&Doctor' == \{\langle\!|\ id == \text{"003"}, name == \text{"}Alice\text{"}\ |\!\rangle, \langle\!|\ id == \text{"004"}, name == \text{"}Bob\text{"}\ |\!\rangle\},\\
&Medrecord' == \{\langle\!|\ contents == \text{"}healthy\text{"}, recordnb == \text{"}meddata2\text{"}\ |\!\rangle,\\
&\qquad\qquad\quad \langle\!|\ contents == \text{"}sick\text{"}, recordnb == \text{"}meddata1\text{"}\ |\!\rangle\},
\end{aligned}
$$

UML associations are translated by RoZ as a pair of functions corresponding to both roles of the association. For example, functions *hospitalsOfDoctor* and *doctorsOfHospital* describe the association between doctors and hospitals. Their domain and range are constrained by predicates of the schema. Additional predicates express that the inverse role can be constructed from the direct one.[1]

DoctorHospitalRel

$HospitalExt\,; DoctorExt$

$hospitalsOfDoctor : DOCTOR \pfun \mathbb{F}\, HOSPITAL$

$doctorsOfHospital : HOSPITAL \pfun \mathbb{F}\, DOCTOR$

$\operatorname{dom} hospitalsOfDoctor \subseteq Doctor \land \bigcup(\operatorname{ran} hospitalsOfDoctor) \subseteq Hospital$

$\ldots$

Here is how Jaza represents role *doctorsOfHospital* corresponding to Fig. 5.6.

$$
\begin{aligned}
doctorsOfHospital' == \quad &\{(\langle\!|\ name == \text{"}BlueCare\text{"}\ |\!\rangle, \quad \{\langle\!|\ id == \text{"003"}, name == \text{"}Alice\text{"}\ |\!\rangle,\\
&\qquad\qquad\qquad\qquad\qquad\quad \langle\!|\ id == \text{"004"}, name == \text{"}Bob\text{"}\ |\!\rangle\}),\\
&(\langle\!|\ name == \text{"}RedCross\text{"}\ |\!\rangle, \quad \{\langle\!|\ id == \text{"003"}, name == \text{"}Alice\text{"}\ |\!\rangle\})\},
\end{aligned}
$$

Finally, here are the specifications of several operations. First, *MRChangeContents* is a setter for field *contents*. This operation, which works on the type of medical records, must be promoted to impact the actual contents of the class extension, and to modify the related associations. The promoted operation is named *MRChangeContentsPr*, and takes an additional input $x?$ designating the object to modify.

MRChangeContents

$\Delta MEDRECORD$

$newcontents? : STRING$

$contents' = newcontents? \land recordnb' = recordnb$

[1] These predicates are omitted for space reasons.

$MRChangeContentsPr == MRChangeContents \wedge \ldots$

Operation *DRLinkDoctors* creates a link between a doctor and an hospital. Its predicates distinguish between the case where a first doctor is linked to the hospital, and the case where doctors were already linked to this hospital.

DRLinkDoctors

$\Xi HospitalExt ; \Xi DoctorExt ; \Delta DoctorHospitalRel$
$hospital? : HOSPITAL ; doctor? : DOCTOR$

$$\begin{aligned}
&(hospital? \notin \operatorname{dom} doctorsOfHospital) \implies \\
&\quad (doctorsOfHospital' = doctorsOfHospital \oplus \{hospital? \mapsto \{doctor?\}\}) \\
&(hospital? \in \operatorname{dom} doctorsOfHospital) \implies \\
&\quad (doctorsOfHospital' = doctorsOfHospital \oplus \{hospital? \mapsto \\
&\qquad\qquad (doctorsOfHospital(hospital?) \cup \{doctor?\})\})
\end{aligned}$$

These operations are sufficiently detailed to animate the model with Jaza. After several steps, one may end up with a state corresponding to Fig. 5.6. Nevertheless, these operations don't take into account the access control rules. In particular, they are not aware of which user is executing them. This will be the responsibility of the security kernel described in the next section.

4.4 The security kernel

4.4.1 Permissions

A *permission assignment* links a *role* to an *operation* on a given *class*, also called the protected *resource*. These four types are introduced in Z as given types or as enumerated types. When considering enumerated types, the values of the type must be extracted from the UML diagram in order to instantiate the security kernel. Here are the type declarations corresponding to Fig. 4.1. *PERMISSION* stores the names of the permissions. Schema *Sets* includes sets of values corresponding to each of these types.

$[PERMISSION]$
$ROLE ::= Doctor \mid Nurse$
$RESOURCE ::= Medrecords \mid Patients \mid$
$\quad Doctors \mid Hospitals$
$ABS\_ACTION ::= EntityRead \mid$
$\quad EntityUpdate \mid EntityFullAccess$

Sets

$role : \mathbb{F}\, ROLE$
$resource : \mathbb{F}\, RESOURCE$
$permission : \mathbb{F}\, PERMISSION$
$abs\_action : \mathbb{F}\, ABS\_ACTION$

Here are the values of these variables during the Jaza animation.

$$\begin{aligned}
&abs\_action' == && \{EntityFullAccess, EntityRead, EntityUpdate\}, \\
&permission' == && \{\text{"}ReadMedrecord\text{"}, \text{"}UpdateDoctor\text{"}, \text{"}UpdateMedrecord\text{"}\}, \\
&resource' == && \{Doctors, Hospitals, Medrecords, Patients\}, \\
&role' == && \{Doctor, Nurse\},
\end{aligned}$$

Schema *ActionAssignment* links roles to a tuple made of the name of the permission, the abstract action allowed by the permission and the kind of resource associated to this permission.

ActionAssignment

$action\_Asmt : ROLE \leftrightarrow (PERMISSION \times ABS\_ACTION \times RESOURCE)$

The permissions of Fig. 4.1 are stored during a Jaza session as :

$$\begin{aligned} \mathit{action\_Asmt}' == \ & \{(\mathit{Doctor}, (\text{"}\mathit{UpdateDoctor}\text{"}, \mathit{EntityFullAccess}, \mathit{Doctors})), \\ & (\mathit{Doctor}, (\text{"}\mathit{UpdateMedrecord}\text{"}, \mathit{EntityUpdate}, \mathit{Medrecords})), \\ & (\mathit{Nurse}, (\text{"}\mathit{ReadMedrecord}\text{"}, \mathit{EntityRead}, \mathit{Medrecords}))\} \end{aligned}$$

4.4.2 Role hierarchy

RBAC allows to define hierarchical relations between roles. This is captured by schema *RoleInherits*. The predicates forbid circularity in the role hierarchy, and forbid the use of roles not declared in set *role*.

RoleInherits

Sets

$\mathit{role\_Inherits} : \mathit{ROLE} \leftrightarrow \mathit{ROLE}$

$\mathit{role\_Inherits}^{+} \cap \mathrm{id}\, \mathit{role} = \varnothing$

$\mathrm{dom}\, \mathit{role\_Inherits} \subseteq \mathit{role} \wedge \mathrm{ran}\, \mathit{role\_Inherits} \subseteq \mathit{role}$

Fig. 4.1 features a simple role hierarchy, expressed in Jaza as :

$\mathit{role\_Inherits}' == \{(\mathit{Nurse}, \mathit{Doctor})\},$

Schema *InheritAssignment* computes *comp_Asmt* which is *action_Asmt* combined with the inherited permissions.

InheritAssignment

$\mathit{RoleInherits}$

$\mathit{ActionAssignment}$

$\mathit{comp\_Asmt} : \mathit{ROLE} \leftrightarrow (\mathit{PERMISSION} \times \mathit{ABS\_ACTION} \times \mathit{RESOURCE})$

$$\begin{aligned} & \mathit{comp\_Asmt} = \{r : \mathrm{dom}\, \mathit{action\_Asmt}; x : \mathit{role}; a : \mathrm{ran}\, \mathit{action\_Asmt} \\ & \quad \mid ((r \mapsto x) \in ((\mathit{role\_Inherits}^{+}) \cup (\mathrm{id}\, \mathit{role}))) \wedge ((r \mapsto a) \in \mathit{action\_Asmt}) \bullet (x \mapsto a)\} \end{aligned}$$

In our example, permission *ReadMedrecord* is inherited by doctors from nurses.

$$\begin{aligned} \mathit{comp\_Asmt}' == \ & \{(\mathit{Doctor}, (\text{"}\mathit{ReadMedrecord}\text{"}, \mathit{EntityRead}, \mathit{Medrecords})), \\ & (\mathit{Doctor}, (\text{"}\mathit{UpdateDoctor}\text{"}, \mathit{EntityFullAccess}, \mathit{Doctors})), \\ & (\mathit{Doctor}, (\text{"}\mathit{UpdateMedrecord}\text{"}, \mathit{EntityUpdate}, \mathit{Medrecords})), \\ & (\mathit{Nurse}, (\text{"}\mathit{ReadMedrecord}\text{"}, \mathit{EntityRead}, \mathit{Medrecords}))\}, \end{aligned}$$

4.4.3 Action hierarchy

Permissions of Fig. 4.1 refer to abstract actions, such as *read* or *update*. These must be linked to their concrete counterparts. Our security kernel allows to express this link, as well as an action hierarchy, defining abstract actions in terms of other abstract actions (e.g. *EntityFullAccess* is defined as the set including *EntityUpdate* and *EntityRead*). These two relations are expressed in schema *ActionsRelation*. We first introduce the enumerated type of atomic actions.

$\mathit{ATM\_ACTION} ::= \mathit{MRReadMedrecord}1 \mid \mathit{DRLinkDoctors}1 \mid \mathit{MRChangeContentsPr}1$

ActionsRelation

$$\begin{array}{l}
\mathit{Sets} \\
\mathit{action\_Inherits} : \mathit{ABS\_ACTION} \leftrightarrow \mathit{ABS\_ACTION} \\
\mathit{atm\_action} : \mathbb{F}\, \mathit{ATM\_ACTION} \\
\mathit{action\_Relation} : \mathit{ABS\_ACTION} \leftrightarrow (\mathit{ATM\_ACTION} \times \mathit{RESOURCE}) \\
\hline
\mathit{action\_Inherits}^{+} \cap \mathrm{id}\ \mathit{abs\_action} = \varnothing \\
\mathrm{dom}\ \mathit{action\_Inherits} \subseteq \mathit{abs\_action} \wedge \mathrm{ran}\ \mathit{action\_Inherits} \subseteq \mathit{abs\_action} \\
\mathrm{dom}\ \mathit{action\_Relation} \subseteq \mathit{abs\_action} \wedge \mathrm{ran}\ \mathit{action\_Relation} \subseteq (\mathit{atm\_action} \times \mathit{resource})
\end{array}$$

It must be noted that the correspondance between abstract and concrete actions takes into account the class on which the abstract action is performed. For example, concrete operation *MRReadMedrecord*1 only makes sense for medical records. These relations are instantiated as follows in our example.

$$\begin{array}{ll}
\mathit{action\_Inherits}' == & \{(\mathit{EntityRead}, \mathit{EntityFullAccess}), \\
& (\mathit{EntityUpdate}, \mathit{EntityFullAccess})\}, \\
\mathit{action\_Relation}' == & \{(\mathit{EntityRead}, (\mathit{MRReadMedrecord}1, \mathit{Medrecords})), \\
& (\mathit{EntityUpdate}, (\mathit{MRChangeContentsPr}1, \mathit{Medrecords})), \\
& (\mathit{EntityUpdate}, (\mathit{DRLinkDoctors}1, \mathit{Doctors}))\},
\end{array}$$

abstract_Asmt unfolds the hierarchy of abstract actions in *comp_Asmt*. Then *concrete_Asmt* replaces abstract actions by their concrete counterparts for the given kind of resource.

ComputeAssignment

$$\begin{array}{l}
\mathit{InheritAssignment}\,;\mathit{ActionsRelation} \\
\mathit{abstract\_Asmt} : \mathit{ROLE} \leftrightarrow (\mathit{PERMISSION} \times \mathit{ABS\_ACTION} \times \mathit{RESOURCE}) \\
\mathit{concrete\_Asmt} : \mathit{ROLE} \leftrightarrow (\mathit{PERMISSION} \times \mathit{ATM\_ACTION} \times \mathit{RESOURCE}) \\
\hline
\ldots
\end{array}$$

$$\begin{array}{ll}
\mathit{concrete\_Asmt}' == & \{(\mathit{Doctor}, (\text{"}\mathit{ReadMedrecord}\text{"}, \mathit{MRReadMedrecord}1, \mathit{Medrecords})), \\
& (\mathit{Doctor}, (\text{"}\mathit{UpdateDoctor}\text{"}, \mathit{DRLinkDoctors}1, \mathit{Doctors})), \\
& (\mathit{Doctor}, (\text{"}\mathit{UpdateMedrecord}\text{"}, \mathit{MRChangeContentsPr}1, \mathit{Medrecords})), \\
& (\mathit{Nurse}, (\text{"}\mathit{ReadMedrecord}\text{"}, \mathit{MRReadMedrecord}1, \mathit{Medrecords}))\},
\end{array}$$

4.4.4 Roles, users and sessions

Actual users of the security kernel must be linked to roles, through sessions. We first define sets of users, sessions and user ids, and the corresponding types. *USERID* already appeared in the functional model and will be used to make a link between users appearing in the security part of the model, and the classes representing these users in the functional model. Injective function *accessRights* makes this link between user ids, and users.

A session corresponds to one and only one user, who has activated a set of roles. These roles must correspond to roles allowed to this particular user. This is expressed in *role_Asmt*. Several predicates, associated to these schemas, check the consistency between these variables.

$[USER, SESSION]$

$$
\begin{array}{|l}
\textit{RoleAssignment} \\
\hline
\textit{Sets} \\
user : \mathbb{F}\ USER \\
role\_Asmt : USER \leftrightarrow ROLE \\
\hline
\ldots \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{SessionRoles} \\
\hline
\textit{RoleAssignment} \\
uid : \mathbb{F}\ USERID\ ; \\
session : \mathbb{F}\ SESSION \\
accessRights : USERID \pinj USER \\
session\_User : SESSION \pfun USER \\
session\_Role : ROLE \leftrightarrow SESSION \\
\hline
\ldots \\
\hline
\end{array}
$$

Table 4.1 features several sessions with associated users, roles and ids.

4.4.5 Putting it all together

Schema *PermissionAssignment* offers an entire table of the graphical model given in Fig. 4.1. It constructs a relation between user identity, user, role and the respective permissions, atomic actions, and the resources. This is achieved using *concrete_Asmt* relation and linking roles to their users and user ids.

$$
\begin{array}{|l}
\textit{PermissionAssignment} \\
\hline
\textit{SessionRoles}\ ;\textit{RoleAssignment}\ ;\textit{ComputeAssignment} \\
perm\_Asmt : (USERID \times USER \times ROLE) \leftrightarrow \\
\qquad\qquad (PERMISSION \times ATM\_ACTION \times RESOURCE) \\
\hline
perm\_Asmt = \{uid : \mathrm{dom}\ accessRights\ ; u : \mathrm{dom}\ role\_Asmt\ ; \\
\qquad r : \mathrm{ran}\ role\_Asmt\ ; b : \mathrm{ran}\ concrete\_Asmt\ | \\
\qquad\qquad (uid, u) \in accessRights \land (u, r) \in role\_Asmt \land (r, b) \in concrete\_Asmt \bullet \\
\qquad\qquad\qquad ((uid, u, r) \mapsto b)\} \\
\hline
\end{array}
$$

In our example, *perm_Asmt* takes the following value

$$
\begin{array}{l}
perm\_Asmt' == \\
\{((\text{"003"}, \text{"}Alice\text{"}, Doctor), (\text{"}ReadMedrecord\text{"}, MRReadMedrecord1, Medrecords)), \\
\ ((\text{"003"}, \text{"}Alice\text{"}, Doctor), (\text{"}UpdateDoctor\text{"}, DRLinkDoctors1, Doctors)), \\
\ ((\text{"003"}, \text{"}Alice\text{"}, Doctor), (\text{"}UpdateMedrecord\text{"}, MRChangeContentsPr1, Medrecords)), \\
\ ((\text{"004"}, \text{"}Bob\text{"}, Doctor), (\text{"}ReadMedrecord\text{"}, MRReadMedrecord1, Medrecords)), \\
\ ((\text{"004"}, \text{"}Bob\text{"}, Doctor), (\text{"}UpdateDoctor\text{"}, DRLinkDoctors1, Doctors)), \\
\ ((\text{"004"}, \text{"}Bob\text{"}, Doctor), (\text{"}UpdateMedrecord\text{"}, MRChangeContentsPr1, Medrecords)), \\
\ ((\text{"007"}, \text{"}Jeck\text{"}, Nurse), (\text{"}ReadMedrecord\text{"}, MRReadMedrecord1, Medrecords))\},
\end{array}
$$

We can now use this table, and the information about sessions, to specify the basis for secure operations. *SecureOperation* actually does nothing : it does not update the state nor computes a result. It simply states preconditions to allow *user ?*, with id *uid ?*, acting in a given *role ?*, during a given *session ?* to perform a given *action ?* on a *resource ?*, as stated by *permission ?*.

Session	User	Role	User Id
sess1	Alice	Doctor	003
sess2	Bob	Doctor	004
sess3	Jeck	Nurse	007

Tab. 4.1 – Three sessions

SecureOperation

$\Xi SessionRoles ; \Xi PermissionAssignment$
$session? : SESSION ; resource? : RESOURCE ; atm\_action? : ATM\_ACTION$
$role? : ROLE ; user? : USER ; uid? : USERID ; permission? : PERMISSION$

$(session?, user?) \in session\_User$
$(role?, session?) \in session\_Role$
$((uid?, user?, role?), (permission?, atm\_action?, resource?)) \in perm\_Asmt$

Another use of this table is to perform queries on the access control policy. We have implemented three such queries, inspired by [BCDE09a].
– Given a role, what are the atomic actions allowed to this role ?
– Given an atomic action, which roles may perform it ?
– Given a kind of resource, which permissions apply ?
EvaluateActionsAgainstRoles corresponds to the second kind of query. It takes an atomic action as input and returns a table listing all roles allowed to perform this atomic action, and the corresponding permission.

EvaluateActionsAgainstRoles

$\Xi Sets ; \Xi ComputeAssignment$
$atm\_action? : ATM\_ACTION$
$z\_roles! : ROLE \leftrightarrow (PERMISSION \times ATM\_ACTION \times RESOURCE)$

$z\_roles! = \{ r : \mathrm{dom}\ comp\_Asmt ; p : permission ; rsrc : resource \mid$
$\quad (r \mapsto (p, atm\_action?, rsrc)) \in concrete\_Asmt \bullet (r \mapsto (p, atm\_action?, rsrc)) \}$

This can be evaluated using Jaza. For example, the following query questions about the permissions to call *MRChangeContentsPr*1. The answer tells us that only role doctor is allowed to perform this action on medical records.

$; EvaluateActionsAgainstRoles[atm\_action? := MRChangeContentsPr1]$
$\ldots$
$z\_roles! == \{(Doctor, ("UpdateMedrecord", MRChangeContentsPr1, Medrecords))\}$

4.5 Linking functional and security models

SecureOperation is meant to be included, as a precondition, in the secured version of the operations of the functional model. For example, let us consider the setter method for *contents*, named *MRChangeContentsPr*. A secured version of this operation includes the schema of the operation and *SecureOperation*. Schemas *PatientHospitalRel* and *DoctorHospitalRel* are also included to get read access to the associations between hospitals, patients and doctors.

The first predicate makes a link between this operation and the corresponding atomic action and resource in the security model. They can be generated automatically. The other predicate expresses the constraint *Same hospital as patient* : "the medical record may only be updated by a doctor working in the current hospital of the patient". It retrieves *hospital*, the hospital corresponding to the patient $x?$ of the medical record. Then it retrieves the object of class Doctor corresponding to the user id of the user of the current session. Finally, it checks that this doctor works for *hospital*. This constraint must be added manually by the analyst and cannot be extracted from the UML diagram.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{SecureMRChangeContentsPr}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\mathit{SecureOperation} \\
\mathit{MRChangeContentsPr} \\
\Xi \mathit{PatientHospitalRel}\,; \Xi \mathit{DoctorHospitalRel} \\
\hline
\mathit{atm\_action}? = \mathit{MRChangeContentsPr1} \land \mathit{resource}? = \mathit{Medrecords} \\
\exists\, \mathit{hospital} : \mathit{Hospital} \mid \mathit{hospitalOfPatient}(\mathit{patientOfMedrecord}(x?)) = \mathit{hospital} \bullet \\
\quad \exists\, \mathit{doctor} : \mathit{Doctor} \mid \mathit{accessRights}^{-1}(\mathit{session\_User}(\mathit{session}?)) = \mathit{doctor.id} \bullet \\
\quad\quad \mathit{doctor} \in \mathit{doctorsOfHospital}(\mathit{hospital}) \\
\hline
\end{array}$$

This operation inherits all input parameters of schemas *SecureOperation*. Most of these parameters can be deduced by Jaza once *session*? has been fixed. Therefore, we define a new version of the schema hiding these parameters.

$$\begin{array}{l}
\mathit{SecureMRChangeContentsPr2} == \\
\quad \mathit{SecureMRChangeContentsPr} \setminus (\mathit{uid}?, \mathit{user}?, \mathit{abs\_action}?, \mathit{atm\_action}?, \mathit{resource}?, \\
\quad\quad\quad\quad \mathit{permission}?, \mathit{role}?)
\end{array}$$

Secure versions of *ReadMedicalRecord* and *LinkDoctors* are defined similarly.

Constraint *Same hospital as patient* links information from the security model (the id of the current user) to the state of the functional model (the hospital of the patient). Its evaluation depends on the states of both models and can thus evolve if any of these models evolves. As we will see in Sect. 4.6, this makes the analyses and validation of the security policy more complex.

4.6 Validating and Animating Secure Operations

Validation of security properties is based on animation of the specification using Jaza [Utt05a]. Jaza can animate a large subset of constructs of the Z language. It uses a combination of rewriting and constraint solving to find a final state and outputs from a given initial state and inputs. If the initial state and inputs don't satisfy the precondition of the operation, the tool returns *"No Solutions"*. The tool can be further queried to find out which constraint could not be satisfied.

In the sequel, we start from the state of Fig. 5.6 and Table 4.1. We first show how normal behaviours can be experimented with the tool, demonstrating that such behaviours are permitted by the security model. We then investigate the attempts of a malicious doctor to corrupt the integrity of a medical record.

4.6.1 Normal behaviour

Our first tests aim at exercising scenarios that are expected to succeed. Their success will show that the combination of security and functional models allows such normal behaviours to take place.

Scenario I : *A doctor reads a medical record.*

$;\mathit{SecureMRReadMedrecord2}$
$\mathit{Input}\ \mathit{session}? = \text{"}\mathit{sess1}\text{"}$
$\mathit{Input}\ r? = \text{"}\mathit{meddata2}\text{"}$

This first scenario tests whether a doctor, here Alice, may read a medical record. This tests the inheritance of permission *ReadMedrecord* from nurses to doctors. Jaza animation gives the following result :

$x! == \{\langle\!|\ \mathit{contents} == \text{"}\mathit{healthy}\text{"}, \mathit{recordnb} == \text{"}\mathit{meddata2}\text{"}\ |\!\rangle\}\ |\!\rangle$

Scenario II : *A doctor updates the medical record of a patient in the same hospital.* In this scenario, doctor Alice wants to update some medical record. Since Alice belongs to the same hospital as the patient, this modification is allowed.

$;SecureMRChangeContentsPr2$
$Input\ x? = \langle\!| \ contents == "healthy", recordnb == "meddata2" \ |\!\rangle$
$Input\ newcontents? = "severe"$
$Input\ session? = "sess1"$

The output shows that contents have been changed to *"severe"*.

$$\begin{aligned} Medrecord' == \quad & \{\langle\!| \ contents == "severe", recordnb == "meddata2" \ |\!\rangle, \\ & \langle\!| \ contents == "sick", recordnb == "meddata1" \ |\!\rangle\}, \end{aligned}$$

These two examples show that the security kernel does not block licit operations. They can be shown to stakeholders of the information system to validate that the right behaviour was captured.

4.6.2 Analysing a malicious behaviour

Security analysis also involves the evaluation of the system's ability to block unauthorized behaviour. Here, let us consider a malicious doctor, Bob, who tries to corrupt the integrity of medical record *meddata*1.

Because we are worrying about integrity constraints, we can start by querying the system about which roles may perform operation *MRChangeContentsPr*1, which is the only operations allowed to modify the contents of a medical record. As we have seen in Sect. 5.6.1, this query tells us that only doctors are allowed to perform this operation. Still, animations go beyond the results of queries presented in Sect. 5.6.1, because queries don't take into account constraints such as *Same hospital as patient* which may restrict the access to some operations. We will thus try a scenario where Bob tries to modify medical record *meddata*1.

Scenario III.A : *A doctor attempts to update the medical record of a patient of another hospital.*

$;SecureMRChangeContentsPr2$
$Input\ x? = \langle\!| \ contents == "sick", recordnb == "meddata1" \ |\!\rangle$
$Input\ newcontents? = "cured"$
$Input\ session? = "sess2"$

Hopefully, Jaza answers that this execution is not allowed by the model.

No Solutions

Upon checking the constraints we come to know that hospital for this doctor is not the same as of the patient. These constraints give us added value on the standard queries. Still, similar results can be obtained using the SecureMova tool [BCDE09a]. Let us now consider a scenario that cannot be played by that tool.

The query tool told us that only doctors are allowed to change the contents of a medical record. But Jaza animation also confirmed that a constraint requires the doctor to work in the same hospital as the patient. Since Bob does not work in the same hospital, there are two ways for him to change the outcome of this constraint. Either he moves the patient to his hospital, or he joins the hospital of the patient. Let us study the latter solution, and query the model about which roles are allowed to change the affiliation of a doctor.

$;EvaluateActionsAgainstRoles[atm\_action? := DRLinkDoctors1]$
$\ldots$
$z\_roles! == \{(Doctor, ("UpdateDoctor", DRLinkDoctors1, Doctors))\}$

The query tells us that doctors are allowed to call this operation. Let us try it !

Scenario III.B : *The doctor first attempts to change his hospital association using one of the class methods and he succeeds in his attempt.*

$;SecureDRLinkDoctors2$
$Input\ session? = "sess2"$
$Input\ hospital? = \langle\!| \ name == "RedCross" \ |\!\rangle$
$Input\ doctor? = \langle\!| \ id == "004", name == "Bob" \ |\!\rangle$

The output tells us that Bob is now working for both hospitals.

$$\begin{aligned}
&doctorsOfHospital' == \\
&\quad \{(\langle\!| \ name == "BlueCare" \ |\!\rangle, \ \{\langle\!| \ id == "003", name == "Alice" \ |\!\rangle, \\
&\qquad \langle\!| \ id == "004", name == "Bob" \ |\!\rangle\}), \\
&\quad (\langle\!| \ name == "RedCross" \ |\!\rangle, \ \{\langle\!| \ id == "003", name == "Alice" \ |\!\rangle, \\
&\qquad \langle\!| \ id == "004", name == "Bob" \ |\!\rangle\})\},
\end{aligned}$$

Scenario III.C : *The doctor makes the malicious changes to the medical record*

$;SecureMRChangeContentsPr2$
$Input\ x? = \langle\!| \ contents == "sick", recordnb == "meddata1" \ |\!\rangle$
$Input\ newcontents? = "cured"$
$Input\ session? = "sess2"$

Bob did succeed and compromised the integrity of the medical record.

$$\begin{aligned}
Medrecord' == \ &\{\langle\!| \ contents == "cured", recordnb == "meddata1" \ |\!\rangle, \\
&\langle\!| \ contents == "severe", recordnb == "meddata2" \ |\!\rangle\},
\end{aligned}$$

Actually, it means that the current access control rules allow any doctor to join the hospital of any patient. Constraint *Same hospital as patient* is thus useless !

Our approach supports three kinds of validation activities : (a) answering standard queries about the access rules (leaving out the constraints), (b) checking that a given operation may be performed by a given user in a given state, (c) sequencing several operations for given users from a given state. Our scenarios show that the three kinds of activities are useful. State of the art tools such as SecureMova or OCL/USE only allow (a) and (b), which are mainly of static nature. Our tool covers (c), adding a dynamic character to validation activities.

Constructing a sufficiently complete set of scenarios is essential to perform a suitable validation. This construction is outside the scope of the current chapter that focuses on making such scenarios animatable.

4.7 Related Work

Our previous works [DLCP00],[Led06] on RoZ are the roots to our present work. Amalio [AP03] gives an overview of the alternate approaches to translate UML into Z.

SecureUML[BDT06] is a security profile for UML. It has already been presented and it is the basis of our approach. The works of Sohr [SDAG08], and the SecureMova tool [BCDE09a] are the closest to our approach, and have deeply influenced it. In Sect. 4.6, we showed several queries similar to the ones handled by these tools. In addition, our tool can handle sequences of operations involving both security and functional model. UMLSec [Jür04] is another UML profile that focuses on secrecy and cryptographic protocols. Our work does not target secrecy aspects, but addresses a more abstract level focusing on access control.

Hall [Hal94] used Z to specify a formal security policy model for an industrial project. Likewise, ISO standardized RBAC has widely been described by researchers using Z. A few notable propositions are [AK06],[MSGC07],[YHHZ06] that offer generic formal representation of RBAC. Yet, these works focus on meta-model foundations of RBAC, while we target the animation of application level models.

Various validation and verification of security properties based on RBAC are given in [MSGC07],[Bos95]. Abdallah [AK06] defines a security administration using access monitor for core RBAC and distinguishes among various concepts of RBAC. Boswell [Bos95], describes a security policy model in Z, for NATO Air Command and Control System (ACCS). The author shares learned lessons from manual validation of this large, distributed, and multi-level-secure system. This too questions manual versus automated validation/verification and creates room for tools like Jaza [Utt05a]. Morimoto et al., [MSGC07] chose a common-criteria security functional requirements taken from ISO/IEC-15408 and proposed a process to verify Z specifications by the Z/EVES theorem prover. Sohr [SDA05] has proposed protecting clinical information systems to overcome risks by using first-order LTL facilitated with Isabelle/HOL for formal verification of security policy for RBAC.

4.8 Conclusion and future work

We have presented an approach to validate security design models using Z assertions. The graphical notation of security rules is inspired by SecureUML. Our proposal goes through three steps : (a) automated generation of functional specifications using RoZ [DLCP00], (b) the use of a generic security kernel, instantiated by the security model, and specified in Z, and (c) the link between the kernel and the operations of the functional model. Animation of the specifications makes it possible to check that normal behaviours are authorized by the security model and to analyze potential attacks. This is based on the evaluation of standard queries about the security policy and the animation of user-defined scenarios. Using Jaza brings a dynamic dimension to these analyses which is not covered by state of the art tools such as SecureMova and USE.

Our current tool automatically translates the functional model, but requires manual instantiation of the security kernel, and manual definition of the link between both models. Our next step is to generate this information automatically, from the security part of the SecureUML diagram and a description of the action hierarchy. Also, the security kernel can be improved to take into account additional concepts such as delegation or organisation. Other perspectives include the automated generation of scenarios that test the model, and the definition of metrics for the coverage of the model by these scenarios.

Chapitre 5

Validation of security policies by the animation of Z specifications

Designing a security policy for an information system is a non-trivial task. In this chapter, we consider the design of a security policy based on a variant of the RBAC model, close to SecureUML. This variant includes constraints for the separation of duties, as well as contextual constraints. Contextual constraints use information about the state of the functional model of the application to grant permissions to users. These constraints add flexibility to the security policy, but make its validation more difficult. In this chapter, we first review two tools, USE and SecureMOVA, which can be used to analyse and validate a security policy. These tools focus on analyses of static aspects of the secured system. We then propose a new tool, based on the Z formal language, which uses animation of the specification to validate the static as well as dynamic aspects of the security policy, taking into account possible evolutions of the state of the functional model. We discuss how the security policy and the functional application are described to the tool, and what kind of queries and animations can be performed to analyse nominal and malicious behaviours of the system.

5.1 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 design security policies. In this chapter, we focus on role-based access control models (RBAC) [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. 5.3.2). 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. Our work focusses on validation.

With such complex models, validation 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 [Jür04, SDAG08] 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 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 validation of RBAC models. They focus on static analyses of the model. In this chapter, we propose a toolset which supports both static and dynamic analyses, allowing to study nominal and malicious behaviours of the secure system.

In Sect. 5.2, we present the meeting scheduler example which will illustrate our work. In Sect. 5.3, we review the features of two tools, USE and SecureMOVA, which are representative of the current state of the art. In Sect. 5.4, we discuss the interest of leading dynamic analyses of security policies. Sect. 5.5 discusses the translation of security and functional diagrams into a Z specification. Sect. 5.6 details the dynamic analyses that can be performed on our specification. Finally Sect. 5.7 draws the conclusions of this work.

5.2 The meeting scheduler

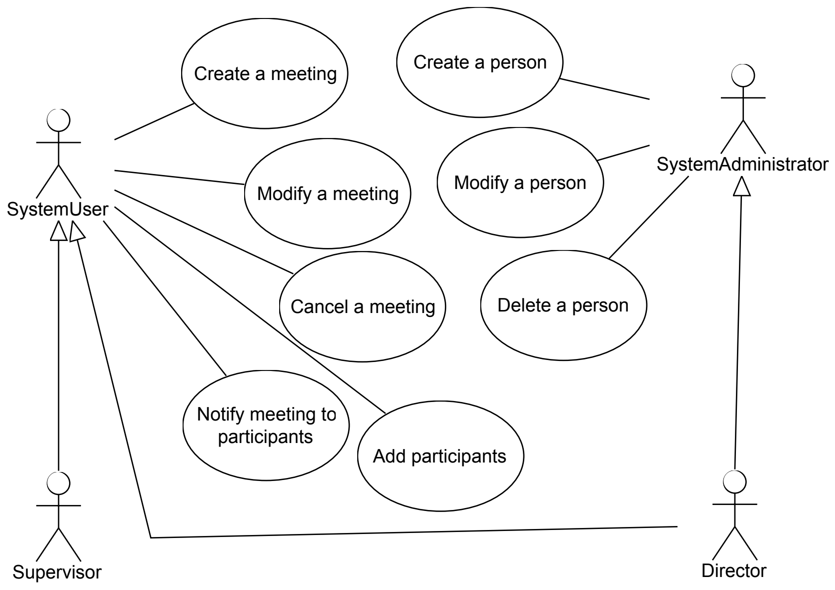


Fig. 5.1 – Use cases for the meeting scheduler

To illustrate our work, we consider a meeting scheduler example used by Basin et al to illustrate SecureUML and SecureMOVA [BCDE09b]. The meeting scheduler helps users plan a "meeting" involving several "persons". Basically, the information system records information about persons, meetings, and the links between these. These links are (a) the ownership of a meeting by a person who organizes it, and (b) the participation of a given user in a meeting. Fig. 5.1 gives the major use cases of this system and the related actors. The major kind of actor is the system user. System users can create and cancel meetings, modify the meeting's information (e.g. change the time or duration of the meeting), add participants to a meeting, and notify the participants about the meeting (which performs some side-effecting operation such as sending a mail to the participants). The system administrator is another actor. Basically, he is responsible of managing information about the persons, i.e. the potential owners of and participants to the meetings.

Fig. 5.1 gives the basis for access control rules : users are in charge of their meetings and system administrators manage the persons. An important security property is related to the integrity of information about meetings. It is expressed by the following rules : (1) a meeting may only be modified or canceled by its owner, and (2) supervisors have the privilege to modify or cancel meetings they don't own. Supervisors are thus introduced as a specialisation of system users. Another kind of actor is the Director, who is both a user and an administrator.

5.3 State of the art tools

5.3.1 RBAC and SecureUML

Our security model is based on SecureUML [BDL06, BCDE09b], an extension of RBAC (Role-Based Access Control) [DDR03]. In RBAC, access control is expressed by a set of *permissions*. A permission allows to perform a set of *actions* on a set of *resources*. Instead of directly granting permissions to *users*, RBAC abstracts users performing the same duties into *roles*. Permissions are thus granted to roles. When a user wants to access some action on a resource, he starts a *session* and activates one or several roles in this session. Based on the activated roles, he gets the permission to access the resource.

RBAC has been extended with several constraint mechanisms. One of the goals is to achieve separation of duties. Separation of duties aims at forbidding a user to take conflicting roles. In our example, one may consider that roles System Administrator and Supervisor give too much privilege and forbid that a user takes these roles. Dynamic separation of duties (DSD) requires that a given user may not take these conflicting roles simultaneously in the same session. Static separation of duties (SSD) forbids a user to take these conflicting roles even in different sessions.

SecureUML is a UML profile which expresses RBAC rules in a class diagram, using stereotyped elements. The class diagram provides a functional model of the application, and the stereotyped elements define the security model. It includes the concepts or RBAC and the possibility to associate permissions with contextual constraints. These constraints involve elements of both security and functional models and restrict the applicability of the permission to the cases where the constraint is verified. In the meeting scheduler, such a constraint is associated to the permission of system users to modify or cancel a meeting. The constraint restricts this permission to the owner of the meeting. Information about the owner of the meeting will be retrieved from the functional class diagram, while information about the user performing the action is related to the security model.

Contextual constraints give much flexibility to express a security policy, but their validation must take into account both functional and security models. Therefore, they require adequate tools. In the next sections, we briefly review 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.

It must be noted that UMLSec [Jür04] is another attempt to address security in UML, but it focusses on the security model (in particular cryptographic aspects) and does not address its interaction with the functional model.

5.3.2 USE for the validation of security policies

The USE tool (UML-based Specification Environment) [GBR07b] 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 focusses 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 brothers may not take conflicting roles.
– Context-dependent permissions, e.g. a meeting may only be modified by its owner.

The last two properties cannot be expressed on a pure security model. It must be augmented with functional information, e.g. some attribute *ownedMeetings* 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 as 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. It detects erroneous dynamic behaviours of the security policy. For example, if two roles are constrained both by a precedence and a conflict relations, it is impossible to find a sequence of RBAC administrative and system operations which leads to create the second role.

5.3.3 SecureMOVA

In [BCDE09b], 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.

SecureMOVA allows to evaluate queries about the security policy. The tool provides an extensive set of queries over a given model, possibly associated with a given initial state. In [BCDE09b], Basin et al list the queries that are supported by the tool. A first set of queries explores the relations between roles and actions.

– Given a role, what are the atomic actions that a user in this role can perform ?
– Given an atomic action, which roles can perform this action ?
– Given a role and an atomic action, under which circumstances can a user in this role perform this action ?

Other queries ask more general questions to analyse the security policy. They help identify redundant roles or permissions.

– Are there two roles with the same set of atomic actions ?
– Given an atomic action, which roles allow the least set of actions, including the atomic action ?
– Do two permissions overlap ?

– Are there atomic actions that every role, except the default role, may perform ?

With SecureMOVA it is also 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 in the current context.

– Given a functional and a security state, can a given user in a given role perform a given action on a given resource ?
– Given a user and a state, what are all actions that this user can perform ?
– Given a state, which users may perform a given action on a given resource ?
– Given a state, which role should take a given user to perform a given action on a given resource ?

This extensive set of supported queries is of great help to analyse and validate a security policy. In particular, the last set of queries, which involve a given functional state, can be very useful to study the impact of contextual constraints. Nevertheless, all reported examples [BCDE09b] 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.

5.4 The need for dynamic analyses

In the sequel, we propose to use animation techniques to further validate security policies. Animation allows to play sequences of actions from a given state. USE and SecureMOVA only report whether the first action of the sequence can be executed from the given state. Animation of sequences of actions is useful to investigate two kinds of behaviours : nominal behaviours, corresponding to the requirements of the system, and malicious behaviours, corresponding to attacks against the secure system.

In both cases, the corresponding behaviour may involve several steps, and it is not sufficient to investigate whether a given action can be performed in a given state. It is also necessary to check that the given state can be reached from the initial state, and when sequences of actions are considered, to compute the resulting state and check that the next action can be performed from this resulting state. Animation tools allow to perform a sequence of actions, starting from an initial state and to compute all intermediate states.

Such dynamic analyses require the availability of executable models. 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].

Instead of working at the implementation level, our approach favours early validation at the abstract level of a PIM (Platform-Independent Model). 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], B[Abr96b], or Alloy [Jac06].

In [TRA+09], functional and security models are merged into a single UML model which is translated into Alloy. Alloy can then be used to find a state which breaks a given property. The properties described in [TRA+09] are mainly of static nature, i.e. they focus on the search for a state which breaks a property, and don't search for sequences of actions leading to such a state. Nevertheless, Alloy can take into account the behaviour of the actions of the model, and we believe it has the potential to perform such dynamic analyses.

In this chapter, we adopt the Z language, and the Jaza tool [Utt05b] is used to animate the operations of a Z specification.

5.5 A toolset based on Z

We propose to translate the functional and security models into a Z specification, and then to use the Jaza animator to analyse this specification, using animation and queries. Actually, several attempts have already specified RBAC in Z [FSG+01], but these were not aimed to be the input of an animator.

5.5.1 Input models

Our toolset takes as input : (a) a class diagram of the functional application, possibly annotated in Z, and (b) several security diagrams, including diagrams stating the permissions, and a diagram assigning users to roles. Security diagrams are completed by a description of an action hierarchy linking abstract actions to concrete ones. From these inputs, our toolset computes a Z specification of the system which can be animated with Jaza (see sect. 5.6).

Functional model

The functional model is described by a class diagram annotated with Z assertions and we use the RoZ tool [DLCP00] to complete it and translate it into Z. Given a class diagram, RoZ automatically generates the specification of basic operations (i.e. attribute and association setters). The center of Fig. 5.2 gives the class diagram for the meeting scheduler. It includes two classes (Meeting and Person) and two associations. A meeting is characterized by its starting date and its duration, a person is simply characterized by his/her name. Most operation specifications have been created automatically. They correspond to operations to create and delete objects, update object attributes, and create links between objects. Three operations are user-defined : notify and cancel are specific to the application, createMeeting creates of an object of class meeting and simultaneously links it to its owner and first participant. This operation is necessary in order to satisfy the arity constraints related to both associations : a meeting has at least one owner and one participant.

From this annotated diagram, RoZ can generate a complete Z specification. For example, the specification of operations RemoveMeeting and cancel are as follows :

$$
\begin{array}{|l}
\textit{MeetingRemoveMeeting} \\
\hline
\Delta \textit{MeetingExt} \\
\textit{meeting}? : \textit{MEETING} \\
\hline
\textit{Meeting}' = \textit{Meeting} \setminus \{\textit{meeting}?\} \\
\hline
\end{array}
$$

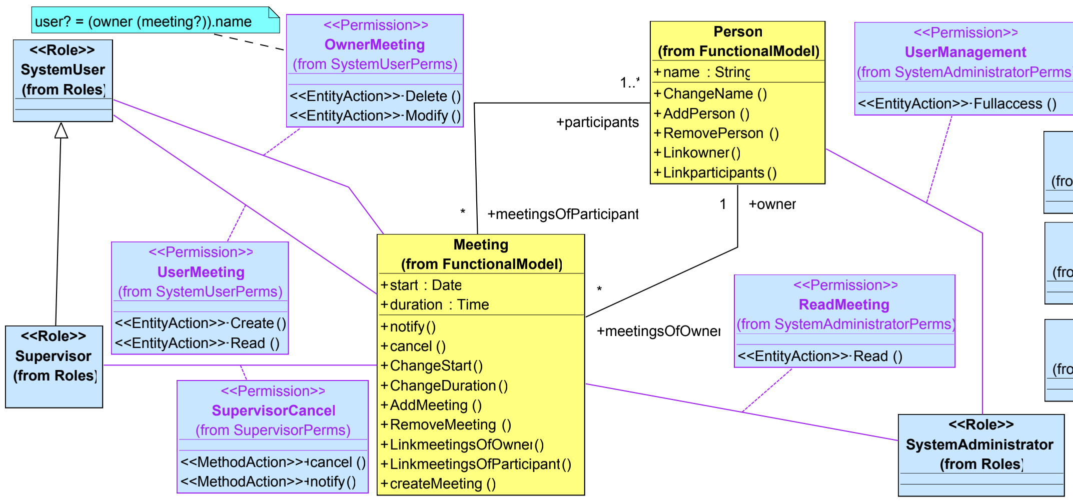


FIG. 5.2 – Class diagram and permissions for the meeting scheduler

$$
\begin{array}{|l}
\hline
\textit{meetingcancel} \\
\textit{MeetingRemoveMeeting} \\
\Xi \textit{PersonExt} \\
\Delta \textit{MeetingOwnerRel} \\
\Delta \textit{MeetingParticipantsRel} \\
\hline
\textit{owner}' = \{\textit{meeting}?\} \lhd \textit{owner} \\
\textit{participants}' = \{\textit{meeting}?\} \lhd \textit{participants} \\
\hline
\end{array}
$$

The first operation takes a meeting as input (denoted by '?') and removes it from the set of existing meetings (*Meeting*). Operation *meetingcancel* includes the first operation and extends its scope to access the owner and participant associations. The meeting is also removed from the domains of both associations. In this example, the specification of RemoveMeeting was generated automatically by the tool, but the user had to provide the details of *meetingcancel* as annotations of the corresponding operation in the class diagram.

RoZ generates specifications which can be animated with Jaza [Led06]. This animation helps to convince the analyst that he has captured the right functional model, and expressed the right specifications for application specific operations.

5.5.2 Diagrams for the security model

Permissions

The security model involves several diagrams. The main diagram (Fig. 5.2) expresses the permissions related to each role. In accordance with the use cases of Fig. 5.1, users may only access meetings. A first permission, UserMeeting, allows them to create and read objects of the class meeting. A second permission, OwnerMeeting, details the rights to update an existing meeting, i.e. to modify it or to delete it. This permission is associated to a constraint, written in the Z language, which states that the user must have the same name as the owner of the meeting.

Similar permissions are expressed for Supervisor and SystemAdministrator. Permission SupervisorCancel grants to supervisors the right to perform operations cancel and notify on any meeting. UserManagement grants to administrators full access to the class Person, and ReadMeeting grants them the right to read class Meeting.

It must be noted that these permissions refer to abstract operations (e.g. Read or Fullaccess) and that a link must be established between these abstract operations and their concrete counterparts. This will be explained in Sect. 5.5.2.

Roles and users

An additional diagram (Fig. 5.3) declares the roles of the application, and links them to several users. In this diagram, the roles correspond to the actors of the use case diagram : SystemUser, Supervisor, SystemAdministrator and Director. Four users are declared and assigned to these roles through UA (User Assignment) links. These user assignments list the roles that a user can take in a session. Yet, the user may choose to perform the session using a subset of his possible roles. The diagram also declares some separation of duties constraints between roles. Fig. 5.3 features one static separation of duty (SSD) between Supervisor and Administrator, and one dynamic separation of duties (DSD) between Director and SystemUser. It can be visually checked that the SSD constraint is respected by the user assignments. The DSD constraint, which will be enforced during a session, may only be violated by Mark who may use both roles of the DSD.

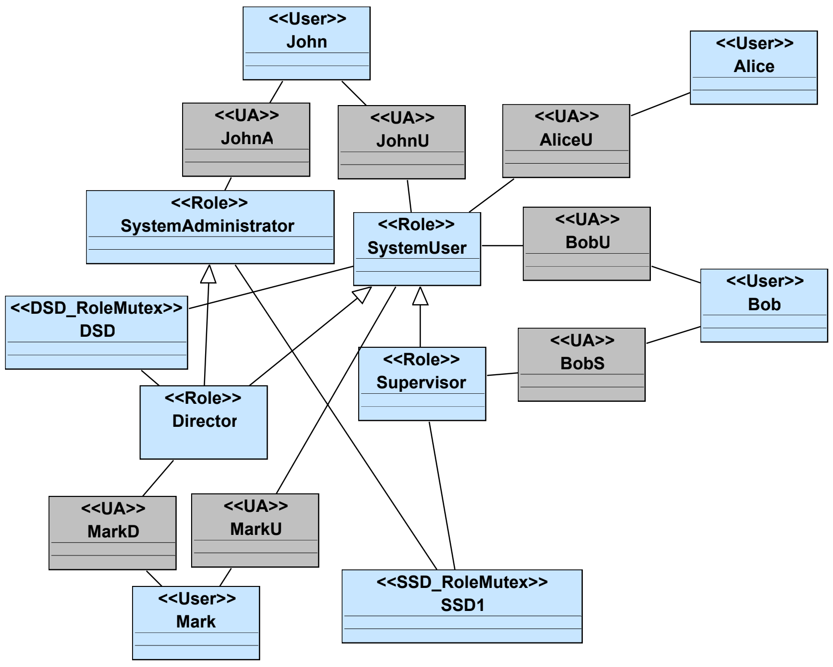


Fig. 5.3 – Users, roles and separation of duties for the meeting scheduler

Action hierarchy

As mentioned earlier, the permissions of Fig. 5.2 refer to abstract actions. A link must be established between these and the actual operations defined in the classes. Currently, our toolset does not provide a graphical notation to express this link. It must be defined directly using the Z syntax. We intend to define a graphical notation for it in a near future. The following table, *action_Relation* expresses how abstract actions are instantiated in each class. For example, action EntityDelete corresponds to Cancel in class Meeting and to RemovePerson in class Person. To avoid name conflicts in the Z specification, operation names are suffixed with "1" and class names with “s”.

$$\begin{aligned}
&\mathit{action\_Relation} = \\
&\quad \{(\mathit{EntityDelete} \mapsto (\mathit{Cancel}1, \mathit{Meetings})), \\
&\quad (\mathit{EntityDelete} \mapsto (\mathit{RemovePerson}1, \mathit{Persons})), \\
&\quad (\mathit{EntityRead} \mapsto (\mathit{Notify}1, \mathit{Meetings})), \\
&\quad (\mathit{EntityCreate} \mapsto (\mathit{AddMeeting}1, \mathit{Meetings})), \\
&\quad (\mathit{EntityCreate} \mapsto (\mathit{CreateMeeting}1, \mathit{Meetings})), \\
&\quad (\mathit{EntityCreate} \mapsto (\mathit{AddPerson}1, \mathit{Persons})), \\
&\quad (\mathit{EntityUpdate} \mapsto (\mathit{ChangeStart}1, \mathit{Meetings})), \\
&\quad (\mathit{EntityUpdate} \mapsto (\mathit{ChangeDuration}1, \mathit{Meetings})), \\
&\quad (\mathit{EntityUpdate} \mapsto (\mathit{ChangeName}1, \mathit{Persons})), \\
&\quad (\mathit{AssocEndUpdate} \mapsto (\mathit{Linkowner}1, \mathit{Persons})), \\
&\quad (\mathit{AssocEndUpdate} \mapsto (\mathit{Linkparticipants}1, \mathit{Persons})), \\
&\quad (\mathit{AssocEndUpdate} \mapsto \\
&\qquad\qquad (\mathit{LinkmeetingsOfOwner}1, \mathit{Meetings})), \\
&\quad (\mathit{AssocEndUpdate} \mapsto \\
&\qquad\qquad (\mathit{LinkmeetingsOfParticipant}1, \mathit{Meetings})), \\
&\quad (\mathit{NotifyExecute} \mapsto (\mathit{Notify}1, \mathit{Meetings})), \\
&\quad (\mathit{CancelExecute} \mapsto (\mathit{Cancel}1, \mathit{Meetings})), \}
\end{aligned}$$

It must be noted that the previous table does not explain what FullAccess stands for. This is because FullAccess corresponds to several abstract operations. This is detailed in *action_Inherits*. The table also defines AssocEndUpdate as a special case of EntityUpdate.

$$\begin{aligned}
\mathit{action\_Inherits} = \ & \{(\mathit{EntityRead} \mapsto \mathit{EntityFullAccess}), \\
& (\mathit{EntityUpdate} \mapsto \mathit{EntityFullAccess}), \\
& (\mathit{EntityCreate} \mapsto \mathit{EntityFullAccess}), \\
& (\mathit{EntityDelete} \mapsto \mathit{EntityFullAccess}), \\
& (\mathit{AssocEndUpdate} \mapsto \mathit{EntityUpdate})\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{perm\_Assignment} == \\
&\quad \{ \quad (("001", "\mathit{Alice}", \mathit{SystemUser}), ("\mathit{OwnerMeeting}", \mathit{Cancel}1, \mathit{Meetings})), \\
&\qquad\quad (("001", "\mathit{Alice}", \mathit{SystemUser}), ("\mathit{OwnerMeeting}", \mathit{ChangeDuration}1, \mathit{Meetings})), \\
&\qquad\quad (("001", "\mathit{Alice}", \mathit{SystemUser}), ("\mathit{OwnerMeeting}", \mathit{ChangeStart}1, \mathit{Meetings})), \\
&\qquad\quad (("001", "\mathit{Alice}", \mathit{SystemUser}), ("\mathit{UserMeeting}", \mathit{CreateMeeting}1, \mathit{Meetings})), \\
&\qquad\quad (("001", "\mathit{Alice}", \mathit{SystemUser}), ("\mathit{UserMeeting}", \mathit{Notify}1, \mathit{Meetings})), \\
&\ldots \\
&\qquad\quad (("002", "\mathit{Bob}", \mathit{Supervisor}), ("\mathit{OwnerMeeting}", \mathit{Cancel}1, \mathit{Meetings})), \\
&\qquad\quad (("002", "\mathit{Bob}", \mathit{Supervisor}), ("\mathit{SupervisorCancel}", \mathit{Cancel}1, \mathit{Meetings})), \\
&\qquad\quad (("002", "\mathit{Bob}", \mathit{Supervisor}), ("\mathit{UserMeeting}", \mathit{Notify}1, \mathit{Meetings})), \\
&\ldots \\
&\qquad\quad (("003", "\mathit{John}", \mathit{SystemAdministrator}), ("\mathit{UserManagement}", \mathit{AddPerson}1, \mathit{Persons})), \\
&\qquad\quad (("003", "\mathit{John}", \mathit{SystemAdministrator}), ("\mathit{UserManagement}", \mathit{Linkowner}1, \mathit{Persons})), \\
&\ldots
\end{aligned}$$

FIG. 5.4 – A subset of the *perm_Assignment* table

Z Translation of the security model

The security diagrams are prepared with the TopCased tool[1]. A meta-model and a UML profile have been defined to support the edition of these models. The graphical models are translated into Z using Acceleo[2], a MDA based code generator.

The Z specification of the security model is based on the specification of a Z security kernel, independent of a specific application, which specifies the main RBAC data structures (user assignment to roles, role hierarchy, definition of permissions, action hierarchy, session management, Static and Dynamic separation of duties) and computes a table, named *perm_Assignment* which links user ids, users, roles, permissions, actions and resources.

The translation of the security diagrams and the action hierarchy of Sect. 5.5.2 are used to instantiate these data structures, and the associated enumerated types. Using the Jaza animator, we can compute *perm_Assignment* for our example. Fig. 5.4 gives a subset of this table. For example, the first line tells us that Alice, whose user id is 001, acting as System User, may cancel a meeting due to permission OwnerMeeting. It also tells us that Bob, acting as a supervisor, has two ways to cancel a meeting, using either permission OwnerMeeting or permission SupervisorCancel. It must be noted that this table does not refer to contextual constraints. Its information is thus partial.

The security kernel defines a generic operation, named *SecureOperation*, which takes as arguments a user, its user id, a role, a session, a permission, an atomic action and a resource and checks that (a) the user is logged in the session with the given role, and (b) that table *perm_Assignment* authorizes this action for the user in the given role. This definition of SecureOperation is actually a precondition that must be satisfied for the action to take place.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{SecureOperation}} \\
\hline
\ldots \\
\mathit{user?} : \mathit{USER} \\
\mathit{userid?} : \mathit{USERID} \\
\mathit{role?} : \mathit{ROLE} \\
\mathit{session?} : \mathit{SESSION} \\
\mathit{permission?} : \mathit{PERMISSION} \\
\mathit{atm\_action?} : \mathit{ATOMIC\_ACTION} \\
\mathit{resource?} : \mathit{RESOURCE} \\
\hline
\ldots \\
(\mathit{session?}, \mathit{user?}) \in \mathit{session\_User} \\
(\mathit{role?}, \mathit{session?}) \in \mathit{session\_Role} \\
((\mathit{userid?}, \mathit{user?}, \mathit{role?}), (\mathit{permission?}, \mathit{atm\_action?}, \mathit{resource?})) \\
\qquad \in \mathit{perm\_Assignment} \\
\hline
\end{array}$$

5.5.3 Linking both formal models

The last step in the preparation of the Z specification links the Z specifications of both models. First, one must relate the types appearing in both models. Here, the constraint on OwnerMeeting compares the name of the owner to the user performing the cancel operation. This requires that name and user have compatible types. In our example, this is done by redefining type *USER* as a *STRING*.

[1]http ://www.topcased.org/
[2]http ://www.acceleo.org/pages/home/en

$USER == STRING$

At this point, secure versions of the functional operations can be defined. For example, the secure version of *meetingcancel* includes *SecureOperation* and *meetingcancel* (given in Sect. 5.5.1). What the operation actually does is completely defined in the functional operation (i.e. *meetingcancel*). So the secure operation simply adds several checks to allow the operation to take place. These checks take the form of additional preconditions. These requires the atomic action to be *Cancel*1 and the resource to be *Meetings*. They also require that input parameter *meeting*? corresponds to an existing meeting, which allows to retrieve its owner. The last condition includes the contextual constraint (user is owner). Since this constraint only applies for OwnerMeeting and not for SupervisorCancel, it only applies if the role is not Supervisor.

$$\begin{array}{|l}
\textit{Securemeetingcancel} \\
\hline
\textit{SecureOperation} \\
\textit{meetingcancel} \\
\hline
\textit{atm\_action}? = \textit{Cancel}1 \\
\textit{resource}? = \textit{Meetings} \\
\textit{meeting}? \in \textit{Meeting} \\
\textit{role}? \neq \textit{Supervisor} \implies (\textit{user}? = (\textit{owner}(\textit{meeting}?)).\textit{name}) \\
\hline
\end{array}$$

Securemeetingcancel has a large number of input parameters. Many of these parameters can be deduced from a subset of the input parameters (here *session*? and *meeting*?) and the preconditions of the operation. Operation *Securemeetingcancel*2 actually hides the useless parameters. In Z, the hide operation ($\setminus$) existentially quantifies the hidden variables. This means that the Z animator will have to find a value for each of the hidden parameters.

$$\begin{aligned}
&\textit{Securemeetingcancel}2 == \textit{Securemeetingcancel} \setminus (\textit{userid}?, \\
&\quad \textit{user}?, \textit{atm\_action}?, \textit{resource}?, \textit{permission}?, \textit{role}?)
\end{aligned}$$

Currently, secure operations are defined manually. Still this definition is completely systematic and a significant part can be automated, using additional Acceleo transformations.

5.6 Animation of the specification

Based on the resulting Z specification, we can use the Jaza animator [Utt05b] to perform static (queries) and dynamic analyses (animations) of the security policy. Jaza is a Z animation tool based on a combination of proof (simplification, rewriting) and search (generate and test) techniques. It covers a wide range of Z constructs and supports some level of non-determinism in the specifications (provided the search space is not too large).

Jaza can execute an operation whose input parameters are fully instantiated. It checks the preconditions, computes the resulting state and checks that the resulting state is in accordance with all postconditions of the operation and with the state invariants. The user may also omit some input parameters, using the hiding operator. In that case, Jaza searches for values which will satisfy the pre-conditions of the operation and chooses one of these. This requires the search space to be finite, and small enough. In the sequel we will exploit both features of Jaza to analyse the Z specification.

5.6.1 Queries on the security model

We start our analysis by asking some queries, inspired by the ones of SecureMOVA [BCDE09b](see Sect. 5.3.3). These queries are mainly based on the *perm_Assignment* table (Fig. 5.4).
– What are the atomic actions associated to a given role ?
– Which roles can perform a given atomic action ?
For each of these queries, a corresponding Z operation has been defined. Since the queries don't depend on the application, the Z operations are also reusable. For example, let us query which roles may perform the cancel operation. Jaza answers that three roles can perform this action and reports on the associated permissions. A closer look at the diagrams reveals that one of these permissions is associated to a constraint.

$EvaluateActionsAgainstRoles[atm\_action? := Cancel1]$

$$
\begin{aligned}
&z\_roleAction! == \\
&\{\quad (Director, ("OwnerMeeting", Cancel1, Meetings)), \\
&\quad (Supervisor, ("OwnerMeeting", Cancel1, Meetings)), \\
&\quad (Supervisor, ("SupervisorCancel", Cancel1, Meetings)), \\
&\quad (SystemUser, ("OwnerMeeting", Cancel1, Meetings))\}
\end{aligned}
$$

A second series of queries consider the whole set of rules. They help identify generic flaws in the security policy.
– Are there duplicate roles, i.e. two roles with the same set of atomic actions ?
– Do two permissions overlap ?
– Is there an atomic action that every role may perform ?
– Is there an atomic action that nobody may perform ?
For example, the following query reports that Supervisor and SystemUser are duplicate roles. It means that they have the same privileges in table *perm_Assignment*. Still a closer look at the diagrams shows that a contextual constraint restricts the rights of SystemUser, which justifies the existence of both roles.

$FindDuplicateRoles$

$z\_role1! == Supervisor, z\_role2! == SystemUser$

The following query looks for operations that are always blocked by the security policy. It reveals that RemoveMeeting is not accessible. Actually, RemoveMeeting is meant to be used as a part of *meetingcancel*. So it is normal that no role has access to this operation.

$AccessNobody$

$z\_action! == RemoveMeeting1$

It must be noted that the same queries are supported by SecureMOVA, but that it only answers "yes" or "no". We found it useful to provide witnesses when the answer is positive, because it speeds up the debugging process.

5.6.2 Dynamic analyses : nominal behaviours

The queries of the previous section are of static nature and do not take into account the contextual constraints associated to permissions. So they don't benefit from our integration of the functional and security models. In this section, we will perform dynamic queries, animating sequences of actions which correspond either to nominal behaviours or to possible attacks.

All animations of this section rely on an initial state where some sessions are predefined. Fig.5.5 gives information about these sessions.

First, we explore nominal behaviours. Our first goal is to find a sequence of actions which will lead us to the functional state depicted in Fig. 5.6. This requires to create two persons, one meeting, and three links. Persons must be created by the system administrator (i.e. John in

Session	User	Roles
sess1	Alice	SystemUser
sess2	Bob	Supervisor, SystemUser
sess3	John	SystemAdministrator, SystemUser
sess4	Mark	Director

FIG. 5.5 – Sessions with their users and roles

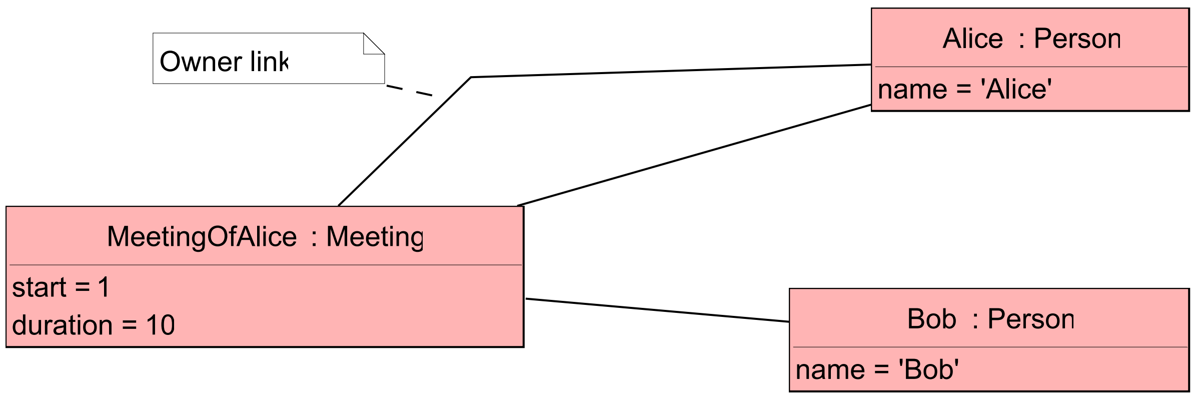


FIG. 5.6 – Object diagram for the meeting scheduler

session 3), then the meeting and its links will be created by Alice (session 1). This corresponds to the following Jaza animation.

$SecurePersonAddPerson2[session? := "sess3",$
  $person? := \lparr\ name == "Alice"\ \rparr]$
$SecurePersonAddPerson2[session? := "sess3",$
  $person? := \lparr\ name == "Bob"\ \rparr]$
$SecuremeetingcreateMeeting2[session? := "sess1",$
  $meeting? := \lparr\ start == 1, duration == 10\ \rparr,$
  $owner? := \lparr\ name == "Alice"\ \rparr]$
$SecuremeetingLinkmeetingsOfParticipant2[$
  $session? := "sess1",$
  $meeting? := \lparr\ start == 1, duration == 10\ \rparr,$
  $person? := \lparr\ name == "Bob"\ \rparr]$

This animation proceeds with success. Actually it covers a nominal behaviour which includes several use cases of Fig. 5.1 : create a person, create a meeting, add participants.

We proceed by trying to cancel the meeting. This will validate the contextual constraint. First, we use the session of John to perform this attempt. Since John is neither supervisor nor the owner of the meeting, this attempt should fail. And this is exactly what happens.

$Securemeetingcancel2[session? := "sess3",$
  $meeting? := \lparr\ start == 1, duration == 10\ \rparr]$

No solutions

We then try the same operation, using the session of Alice, the owner of the meeting. This time, the operation succeeds and the set of meetings is empty after the operation.

$Securemeetingcancel2[session? := "sess1",$
  $meeting? := \lparr\ start == 1, duration == 10\ \rparr]$

$Meeting' == \{\}, \ldots$

These animations increase our confidence that we expressed the right rule and the right constraint.

Another nominal behaviour is to delete some person. Let us consider that Alice has left the company and that we must delete object Alice, starting from the state of Fig. 5.6. Only system administrators are allowed to remove a person, so this will be performed by John in session 3.

$SecurePersonRemovePerson2[session? := "sess3",$
$\quad person? := \langle\!\langle\ name == "Alice"\ \rangle\!\rangle]$

No solutions

Jaza reports that the operation failed. Actually, this is due to the fact that deleting Alice leads to have a meeting without owner, which is forbidden by the class diagram (every meeting has one and only one owner). So the functional model requires to first cancel Alice's meeting and then remove Alice. Since John is administrator, he has no right to cancel Alice's meeting. Since Alice has left the company, we need the help of a supervisor, here Bob in session 2. Now the following sequence of operations will succeed.

$Securemeetingcancel2[session? := "sess2",$
$\quad meeting? := \langle\!\langle\ start == 1, duration == 10\ \rangle\!\rangle]$
$SecurePersonRemovePerson2[session? := "sess3",$
$\quad person? := \langle\!\langle\ name == "Alice"\ \rangle\!\rangle]$

This animation convinces us that it was useful to create role Supervisor in our security policy, otherwise, the security rule would make it impossible to remove a user who has left the company. One may wonder whether role Director could be used to cancel the meeting and then remove the person. But the animator reports that the Director, who is neither supervisor nor the meeting owner, may not cancel Alice's meeting. This may suggest to modify the definition of Director and make him inherit from Supervisor (but this will conflict with the SSD constraint).

Other analyses of nominal behaviours can test SSD and DSD constraints. For example, the following animation shows that the SSD constraint works as expected

$AddRole[user? := "Mark", role? := SystemAdministrator]$
$AddRole[user? := "Mark", role? := Supervisor]$

No solutions

and the following one gives a similar result for DSD

$NewSession[session? := "sess5",$
$\quad user? := "Mark", role? := Director]$
$AddSessionRole[session? := "sess5",$
$\quad user? := "Mark", role? := SystemUser]$

No solutions

5.6.3 Further dynamic analyses

In [BCDE09b], SecureMOVA is used to evaluate queries which depend on a given context. "Given a state, which role should take a given user to perform a given action on a given resource ?" For example, which role should take Bob to cancel Alice's meeting ?

Here, the result does not only depend on *perm_Assignment*, but also on the current state of the data. We can ask a similar query in Z, by defining the following operation.

$$RoleNeededForMeetingCancel == (NewSession \setminus (role?))\ ;\ (Securemeetingcancel2)$$

RoleNeededForMeetingCancel first creates a new session, then uses this session to cancel the meeting. It hides input parameter *role?* so that Jaza must find a role which satisfies the preconditions of both operations. When we call this operation, acting as user Bob, it actually leads to a resulting state where the set of meetings is empty. A closer look at the state shows that session 6 was created with Bob as user, and in the role of Supervisor. This answers our question : Bob may cancel Alice's meeting if he logs in as a supervisor.

$RoleNeededForMeetingCancel[session? := "sess6",$
$\quad user? := "Bob",$
$\quad meeting? := \langle\!| \; start == 1, duration == 10 \; |\!\rangle]$

$Meeting' == \{\}, \ldots$
$session\_Role' == \{\ldots, (Supervisor, "sess6")\},$
$session\_User' == \{\ldots, ("sess6", "Bob")\},$

5.6.4 Studying an attack scenario

Integrity of meetings is an important security property we want to enforce on our information system. Let us now focus on user John, who may play the role of SystemAdministrator and SystemUser. For some malicious reason, John wants to cancel the meeting of Alice. Since John may play two different roles, we can ask which role he should use to cancel the meeting (as we did for Bob in the previous section).

$RoleNeededForMeetingCancel[session? := "sess7",$
$\quad user? := "John",$
$\quad meeting? := \langle\!| \; start == 1, duration == 10 \; |\!\rangle]$

No solutions

As expected, the system answers that John is not allowed, in any of his roles to perform this action. In Sect. 5.6.1, we already queried which roles allow to perform action Cancel1 (using *EvaluateActionsAgainstRoles*), and found that it requires roles SystemUser, Supervisor, or Director. John may only use role SystemUser to cancel the meeting, but a closer look at Fig. 5.2 tells us that permission “OwnerMeeting” requires John to be the owner of the meeting. This explains why he is not allowed to cancel the meeting. This also suggests that John may get this permission if he becomes owner of the meeting. This requires a more elaborate attack where John first becomes owner of the meeting and then cancels it. The functional model provides two methods to change the owner of the meeting (see Fig. 5.2) : `LinkmeetingsOfOwner` in class `Meeting` and `Linkowner` in class `Person`.

Let us check which roles may use these operations :

$EvaluateActionsAgainstRoles[$
$\quad atm\_action? := LinkmeetingsOfOwner1]$

$z\_roleAction! ==$
$\{ \quad (Director,$
$\quad ("OwnerMeeting", LinkmeetingsOfOwner1, Meetings)),$
$\quad (Supervisor,$
$\quad ("OwnerMeeting", LinkmeetingsOfOwner1, Meetings)),$
$\quad (SystemUser,$
$\quad ("OwnerMeeting", LinkmeetingsOfOwner1, Meetings))\}$

None of these permissions apply for John, because he may only take the role SystemUser in this list, and in that case, he must be the owner of the meeting. Operation `Linkowner` corresponds to the other end of the association. A similar query may be performed.

$EvaluateActionsAgainstRoles[atm\_action? := Linkowner1]$

$z\_roleAction! ==$
$\{ \quad (Director, ("UserManagement", Linkowner1, Persons)),$
$\quad (SystemAdministrator,$
$\quad ("UserManagement", Linkowner1, Persons))\}$

So John may perform action `Linkowner` as SystemAdministrator. This action requires to first create an object of class `Person` corresponding to John. John being system administraor, he may create this object, using session sess3.

$SecurePersonAddPerson2[session? := "sess3",$
$\quad person? := \langle\!| \; name == "John" \; |\!\rangle]$
$SecurepersonLinkowner2[session? := "sess3",$
$\quad person? := \langle\!| \; name == "John" \; |\!\rangle,$
$\quad meeting? := \langle\!| \; start == 1, duration == 10 \; |\!\rangle]$

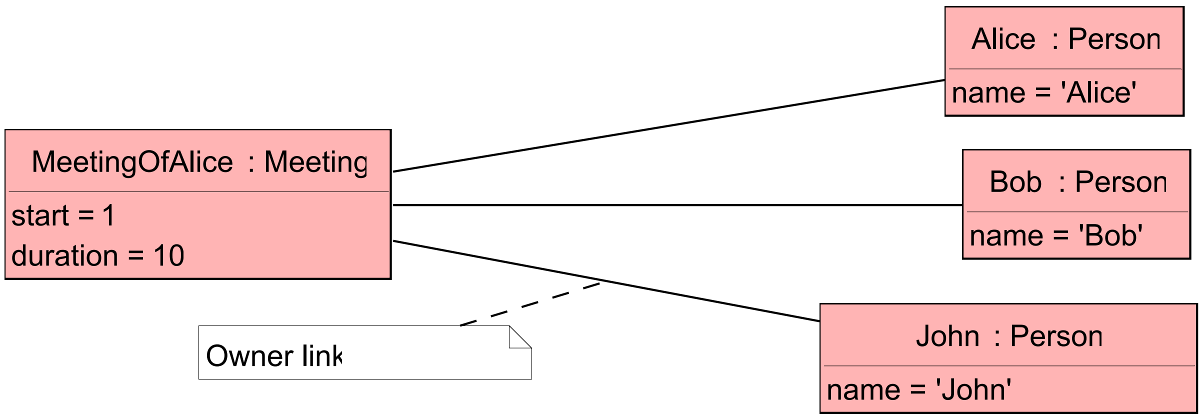


FIG. 5.7 – Another Object diagram for the meeting scheduler

John is now the owner of the meeting, as shown in Fig. 5.7. Being the owner, he may now cancel the meeting.

$Securemeetingcancel2[session? := "sess3",$
$\quad meeting? := \langle\!| \; start == 1, duration == 10 \; |\!\rangle]$

$Meeting' == \{\} \ldots$

The attack of John has succeeded! This may be considered as a flaw of the security policy. The meeting scheduler example was discussed in several articles, and defined independently of our research team. To the best of our knowledge, this problem was never reported before. We foresee that similar problems will happen in SecureUML descriptions which use contextual constraints.

The problem is that SystemAdministrator has full access to class `Person`, which includes the right to modify association ends. One solution is to add a SSD constraint between SystemAdministrator and SystemUser. Hence, John will still be able to become owner of the meeting, but will not be able to log as SystemUser in order to delete it.

5.7 Conclusion

This chapter has addressed the validation of security policies expressed as RBAC rules with contextual constraints. Such constraints refer to elements of both security and functional models, using the state of the functional model 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, validation must take both models into account.

Current state of the tools We have presented a toolset based on a variant of SecureUML and the Z specification language. It allows to perform static analyses, as done by the SecureMOVA tool, and dynamic analyses, playing sequences of actions. Such sequences of actions correspond to expected behaviours, and to attacks against the secure system. We presented these tools on a classical example, the meeting scheduler, addressed in the presentation of SecureMOVA. We identified a potential attack against the integrity of the information system that requires a sequence of actions to allow evolutions of the functional state. We believe that it is easier to analyse this sequence of actions with animation tools, than with static analyses only.

Our toolset includes a large number of the queries supported by USE and SecureMOVA, and it can be extended to support most of the remaining ones. One feature of SecureMOVA remains difficult to support. SecureMOVA is able to report the text of the conditions that are associated to a permission, due to the reflexive character of the UML model. Our Z specification does not allow such reflexivity mechanisms, and can only evaluate the condition in a given state.

Part of our translation from diagrammatic models to Z specifications is currently performed manually. Still, this manual translation is systematic, and we are confident that it will be soon handled automatically. Finally, we did not evaluate the capability of our tool to scale up, and only used it on small models, with acceptable response times (a few seconds). Further work is needed to experiment it on real-size models and, if needed, to optimize its calculations.

Other perspectives Animation is not the only way to perform dynamic analysis. Model-checking provides an interesting alternative. In this chapter, we showed a sequence of actions which compromizes the integrity of the information system. Our tools help identify such sequences, but model-checking could help find a sequence of actions which leads from a given initial state to some unwanted state. Model-checking tools are not available for the Z language, but Pro-B [LB08] provides such a tool for the B language, which is close to Z. This gives an interesting perspective for future work.

Chapitre 6

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.

6.1 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 [Fac95] 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 [Reg02].

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 [Fac95] :

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, Reg02, 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.

6.2 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 6.1). 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.

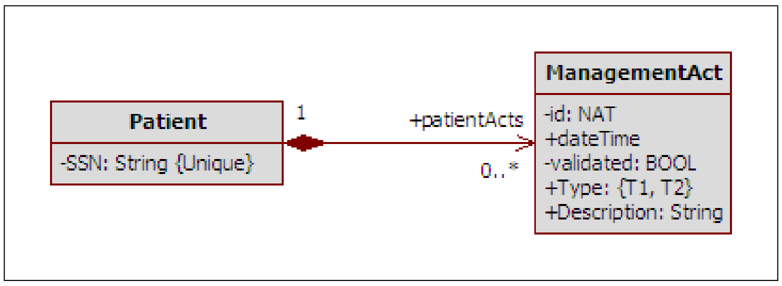


FIG. 6.1 – Extrait du diagramme de classes de l'étude de cas IFREMMONT

6.2.1 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 6.2 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].

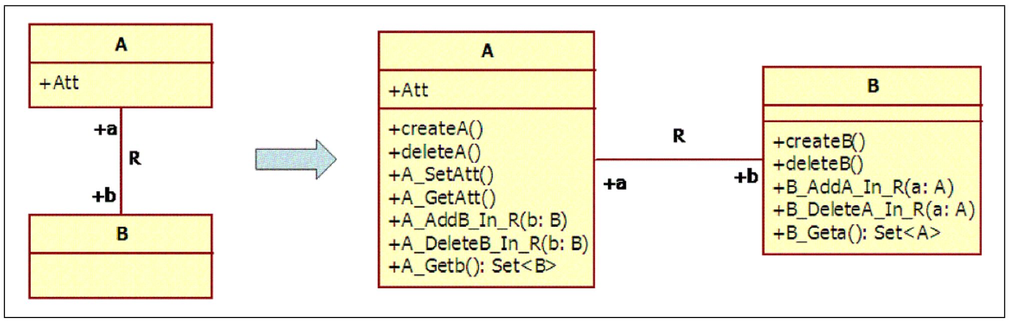


FIG. 6.2 – Intégration des opérations de base dans le diagramme de classes

- 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.

6.2.2 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, [Reg02] 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, Reg02]. 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 $\mathcal{E}_A := \varnothing$
```

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) $\hat{=}$
   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) $\hat{=}$
   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 6.1) dans la machine **Functional_Model** :

```
MACHINE
    Functional_Model
SETS
    PATIENTS
VARIABLES
    Patients
INVARIANT
    Patients ⊆ PATIENTS
INITIALISATION
    Patients := ∅
```

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 6.3.

```
createPatient(obj) ≙
    PRE obj ∈ PATIENTS ∧ obj ∉ Patients THEN
        Patients := Patients ∪ {obj}
        /* Initialisation des attributs obligatoires */
    END
```

```
deletePatient(obj) ≙
    PRE obj ∈ PATIENTS ∧ obj ∈ 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. 6.3 – 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.

6.2.3 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 6.2 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 6.1 donne les différentes valeurs de $\mathcal{R}$.

	Optionnel	Obligatoire
Unique	⤔	⤖
Non unique	⇸	$\rightarrow$

Tab. 6.1 – Relations fonctionnelles issues des attributs de classes

Par exemple, l'attribut `SSN` de la classe Patient (figure 6.1) est un attribut mono-valué, optionnel et unique. Il sera donc traduit par une injection partielle comme suit :

$$\text{Patient\_SSN} \in \mathit{Patients} ⤔ \text{LesSSNs}$$

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

$$\text{ManagementAct\_Validated} \in \mathit{ManagementActs} \rightarrow \mathbf{BOOL}$$

C. 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 6.2) défini par : 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 $obj \in \mathcal{P}_A \wedge obj \in \mathcal{E}_A$ THEN
      ANY $att$ WHERE
         $att \in \mathcal{T}_{\text{Att}}$ [$\wedge$ $att \notin$ ran($\mathcal{R}$)]
       THEN
          $\mathcal{R}(obj) := att$
       END
   END ;
```

```
$att \leftarrow$ A_GetAtt($obj$) =
   PRE $obj \in \mathcal{P}_A \wedge obj \in \mathcal{E}_A$ THEN
         $att := \mathcal{R}(obj)$
   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.

```
$ssn \leftarrow \mathbf{patient\_GetSSN}(obj)$ =
  PRE $obj \in PATIENTS \wedge obj \in Patients$ THEN
      $ssn := patient\_SSN(obj)$
  END ;
```

```
$\mathbf{patient\_SetSSN}(obj)$ =
  PRE $obj \in PATIENTS \wedge obj \in Patients$ THEN
    ANY $ssn$ WHERE
      $ssn \in LesSSNs \wedge ssn \notin \mathbf{ran}(patient\_SSN)$
    THEN
      $\mathbf{patient\_SSN}(obj) := ssn$
    END
  END ;
```

Prise en compte des attributs dans les constructeurs

Outre les getters et les setters générés automatiquement pour tous les attributs, les constructeurs d'instances effectives doivent prendre en compte la valeur par défaut des attributs ainsi que les attributs obligatoires. En effet, les attributs ayant une valeur par default sont initialisés avec cette valeur lors de la construction d'une instance. Les attributs obligatoires n'ayant aucune valeur par default sont initialisés en passant leurs valeurs en paramètre du constructeur. Par exemple, la valeur par défaut de l'attribut `Validated` de ManagementAct est `false` et l'attribut obligatoire `dateTime` n'a pas de valeur par default. Le constructeur `createManagementAct` sera donc comme suit :

```
createManagementAct(obj,dateTimeValue) $\hat{=}$
  PRE obj $\in$ MANAGEMENTACTS $\wedge$ obj $\notin$ ManagementActs $\wedge$ dateTi-
meValue $\in$ DATETIME THEN
      ManagementActs := ManagementActs $\cup$ {obj} ||
      ManagementAct_Validated(obj) := FALSE ||
      ManagementAct_DateTime(obj) := dateTimeValue
  END
```

FIG. 6.4 – 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) $\hat{=}$
  PRE obj $\in \mathcal{P}_A \wedge$ obj $\in \mathcal{E}_A$ THEN
      $\mathcal{E}_A := \mathcal{E}_A - \{obj\}$ ||
      A_Att := $\{obj\} \lhd$ A_Att
  END
```

Par exemple, la figure 6.5 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. 6.5 – Destructeur de la classe Patient prenant en compte l'attribut SSN

6.2.4 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 :

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

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 6.2 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	⇸	0	*	0	1
Fonction totale	$\rightarrow$	0	*	1	1
Injection partielle	⤔	0	1	0	1
Injection totale	$\rightarrowtail$	0	1	1	1
Surjection partielle	⤀	1	*	0	1
Surjection totale	$\twoheadrightarrow$	1	*	1	1
Bijection partielle	⤗	1	1	0	1
Bijection totale	⤖	1	1	1	1

TAB. 6.2 – 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 6.3 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 $Cmult1(assos) \ \wedge \ Cmult2(assos)$ (2^{eme} et 3^{eme} colonne du tableau 6.3). 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 6.1) peut être effectuée de deux manières. La première suit le tableau 6.2 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 6.3 et produit l'invariant suivant :

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

$$\begin{array}{l} patientActs \in Patients \leftrightarrow ManagementActs \wedge \\ patientActs^{-1} \in ManagementActs \rightarrow Patients \end{array}$$

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*).

Génération des opérations de base

Dans la section 6.2.1 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

[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.

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).

De même que pour les attributs, les liens avec une multiplicité minimale égale à 1 sont créés lors de la construction d'instances. Aussi, la destruction d'un objet impliqué dans un tel lien induit-elle la destruction de ce lien. Toutefois, avant la construction d'un lien, on doit s'assurer que les cardinalités ne dépassent pas les valeurs maximales (et réciproquement pour les valeurs minimales lors de la destructions d'un lien).

Dans la figure 6.1, l'association patientActs est une composition navigable uniquement de Patient vers ManagementAct. Par conséquent, on ne produit pas dans la spécification B le constructeur *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 6.2.2.

```
patient_AddpatientActs($obj$) =
    PRE $obj \in PATIENTS \wedge obj \in Patients$ THEN
        ANY $ma$ WHERE
            $ma \in MANAGEMENTACTS \wedge ma \notin ManagementActs \wedge$
            $ma \notin \mathbf{dom}(patientActs)$
        THEN
            $ManagementActs := ManagementActs \cup \{ma\} \mid\mid$
            $patientActs := patientActs \cup \{(ma \mapsto obj)\} \mid\mid$
            $\mathbf{managementact\_validated}(ma) := \mathbf{FALSE}$
        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 $obj \in PATIENTS \wedge obj \in Patients$ THEN
        ANY $ma$ WHERE
            $ma \in MANAGEMENTACTS \wedge ma \in ManagementActs \wedge$
            $ma \in patientActs^{-1}[\{obj\}] \wedge$
            $managementact\_validated(ma) = \mathbf{FALSE}$
        THEN
            $ManagementActs := ManagementActs - \{ma\} \mid\mid$
            $patientActs := \{ma\} \lhd\!\!\!- patientActs \mid\mid$
            $managementact\_validated := \{ma\} \lhd\!\!\!- managementact\_validated \mid\mid$
            $managementact\_dateTime := \{ma\} \lhd\!\!\!- managementact\_dateTime \mid\mid$
            $managementact\_Type := \{ma\} \lhd\!\!\!- managementact\_Type$
        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 :

```
$managementacts \leftarrow$ **patient_GetpatientActs** $(obj) =$
    **PRE** $obj \in PATIENTS \wedge obj \in Patients$ **THEN**
        $managementacts := patientActs^{-1}[\{obj\}]$
    **END** ;
```

6.2.5 Prise en compte de l'héritage de classes

L'héritage entre classes est traduit en B par une inclusion entre les ensembles d'instances effectives. Par exemple, une classe C qui hérite d'une classe A produira l'invariant suivant : $\mathcal{E}_C \subseteq \mathcal{E}_A$ (avec $\mathcal{E}_C$ et $\mathcal{E}_A$ sont les variables d'instances effectives de C et de A). Cet invariant d'inclusion permet de simuler l'héritage d'attributs, de rôles et d'opérations car toutes les operations relatives à une classe, incluant les assesseurs et les mutateurs de rôles et d'attributs, s'appliqueront systématiquement aux sous-classes.

Le constructeur d'une classe doit ajouter l'instance créée dans les ensembles des instances effectives issus de ses parents. Ce constructeur prend en compte également les attributs et les rôles hérités. Cela peut être mis en oeuvre en copiant la pré-condition et la substitution issues du constructeur de la superclasse. En effet, dans l'exemple ci-dessous, la pré-condition obj $\in \mathcal{P}_A \wedge$ obj $\notin \mathcal{E}_A$ ainsi que la substitution $\mathcal{E}_A := \mathcal{E}_A \cup \{\text{obj}\}$ sont celles de l'opération createA. Cela permet de couvrir de la même manière un héritage à plusieurs niveaux. Ce même raisonnement est effectué pour l'opération de destruction d'instances effectives deleteC.

```
**createC**(obj) $\hat{=}$
    **PRE** obj $\in \mathcal{P}_A \wedge$ obj $\notin \mathcal{E}_A$ **THEN**
        $\mathcal{E}_A := \mathcal{E}_A \cup \{obj\}$
        /* Initialisation de roles et d'attributs issus de la classe A*/
            $\mathcal{E}_C := \mathcal{E}_C \cup \{obj\}$
            /* Initialisation de roles et d'attributs issus de la classe C*/
    **END**
```

```
**deleteC**(obj) $\hat{=}$
    **PRE** obj $\in \mathcal{P}_A \wedge$ obj $\in \mathcal{E}_A$ **THEN**
        $\mathcal{E}_A := \mathcal{E}_A - \{obj\}$
        /* Mise à jour des roles et liens issus de la classe A */
            $\mathcal{E}_C := \mathcal{E}_C - \{obj\}$
            /* Mise à jour des roles et liens issus de la classe C */
    **END**
```

6.2.6 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 6.1 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.

```
$\textbf{managementact\_Validate}(obj) =$
    $\textbf{PRE}$
        $obj \in MANAGEMENTACTS \wedge$
        $obj \in ManagementActs \wedge$
        $managementact\_validated(obj) = \textbf{FALSE} \wedge$
        $obj \in \textbf{dom}(managementact\_Type) \wedge$
        $obj \in \textbf{dom}(managementact\_dateTime)$
    $\textbf{THEN}$
        $\textbf{managementact\_validated}(obj) := \textbf{TRUE}$
    $\textbf{END}$
```

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 :

$$\textbf{dom}(managementact\_validated \rhd \{\textbf{TRUE}\}) \subseteq \textbf{dom}(managementact\_Type) \wedge$$
$$\textbf{dom}(managementact\_validated \rhd \{\textbf{TRUE}\}) \subseteq \textbf{dom}(managementact\_dateTime)$$

6.3 Transformation du modèle de sécurité

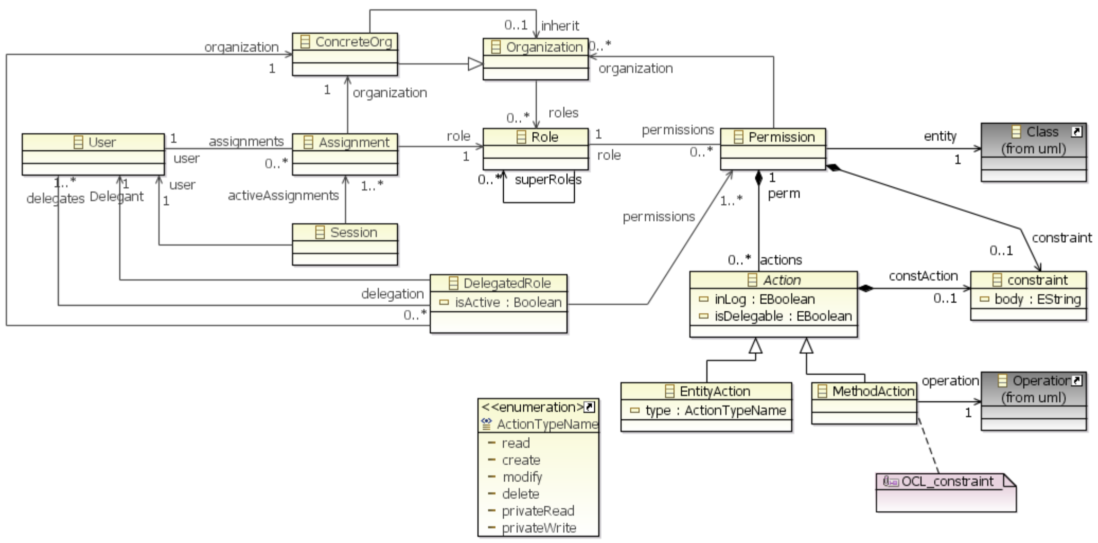


Fig. 6.6 – Méta-modèle de sécurité

6.3.1 Approche proposée

La figure 6.7 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 6.6. 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")
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.

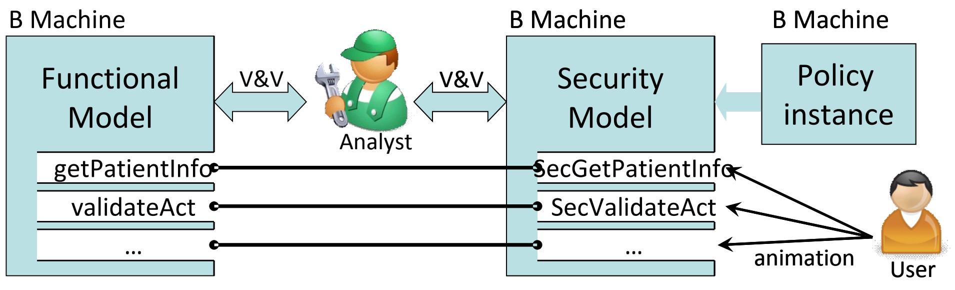


FIG. 6.7 – Principe de la traduction

6.3.2 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 6.8).

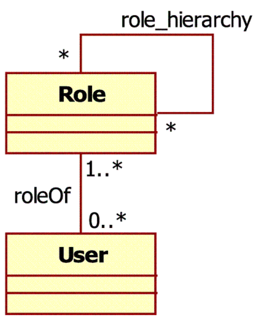


FIG. 6.8 – Portion du méta-modèle de sécurité proposé pour core-RBAC

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

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

Dans ce modèle nous considérons que la relation *roleOf* associe pour chaque utilisateur un ensemble de rôles. L'invariant "$\mathbf{closure1}(Roles\_Hierarchy) \cap \mathbf{id}(ROLES) = \varnothing$" 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
  $\mathbf{changeUser}(user) =$
  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 6.9.

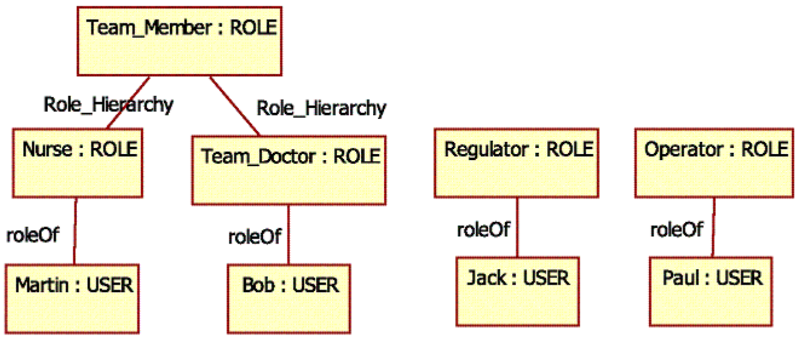


FIG. 6.9 – Instance du méta-modèle de la figure 6.8

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

SETS
  $ROLES = \{Team\_Doctor, Nurse, Operator, Team\_Member, Regulator\}$;
  $USERS = \{Bob, Paul, Martin, Jack, none\}$
  ...
INITIALISATION
  $roleOf := \{(Bob \mapsto \{Team\_Doctor\}),$
      $(Paul \mapsto \{Operator\}),$
      $(Martin \mapsto \{Nurse\}),$
      $(Jack \mapsto \{Regulator\}),$
      $(none \mapsto \varnothing\ )\} \ ||$
  $Roles\_Hierarchy := \{(Team\_Doctor \mapsto Team\_Member),$
      $(Nurse \mapsto Team\_Member)\} \ ||$
  $currentUser := none$
  ...

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

6.3.3 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

$$
\begin{array}{l}
\mathit{AttributeKind} \in \mathit{Attributes} \rightarrow \mathit{KindsOfAtt} \wedge \\
\mathit{AttributeOf} \in \mathit{Attributes} \rightarrow \mathit{ENTITIES} \wedge \\
\mathit{OperationOf} \in \mathit{Operations} \rightarrow \mathit{ENTITIES} \wedge \\
\mathit{constructorOf} \in \mathit{Operations} \rightarrowtail\!\!\!\!\!+\; \mathit{ENTITIES} \wedge \\
\mathit{destructorOf} \in \mathit{Operations} \rightarrowtail\!\!\!\!\!+\; \mathit{ENTITIES} \wedge \\
\mathit{setterOf} \in \mathit{Operations} \rightarrowtail\!\!\!\!\!+\; \mathit{Attributes} \wedge \\
\mathit{getterOf} \in \mathit{Operations} \rightarrowtail\!\!\!\!\!+\; \mathit{Attributes} \wedge \\
\mathit{StereotypeOps} \in \mathit{Stereotypes} \leftrightarrow \mathit{Operations} \wedge \\
\mathit{setterOf} \cap \mathit{getterOf} = \varnothing \wedge \\
\mathit{PermissionAssignement} \in \mathit{PERMISSIONS} \rightarrow (\mathit{ROLES} \times \mathit{ENTITIES}) \wedge \\
\mathit{EntityActions} \in \mathit{PERMISSIONS} \nrightarrow \mathbb{P}\,(\mathit{ActionsType}) \wedge \\
\mathit{MethodActions} \in \mathit{PERMISSIONS} \nrightarrow \mathbb{P}\,(\mathit{Operations}) \wedge \\
\mathit{isPermitted} \in \mathit{ROLES} \leftrightarrow \mathit{Operations}
\end{array}
$$

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 $\mathit{contructorOf}^{-1}(E)$.

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* ∈ *ROLES* et *entity* ∈ *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 6.10 exprime une politique de contrôle d'accès associée au modèle fonctionnel de la figure 6.1. Dans ce diagramme, nous distinguons les permissions associées à l'entité *Patient* de celles associées à *ManagementAct*.
- 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)

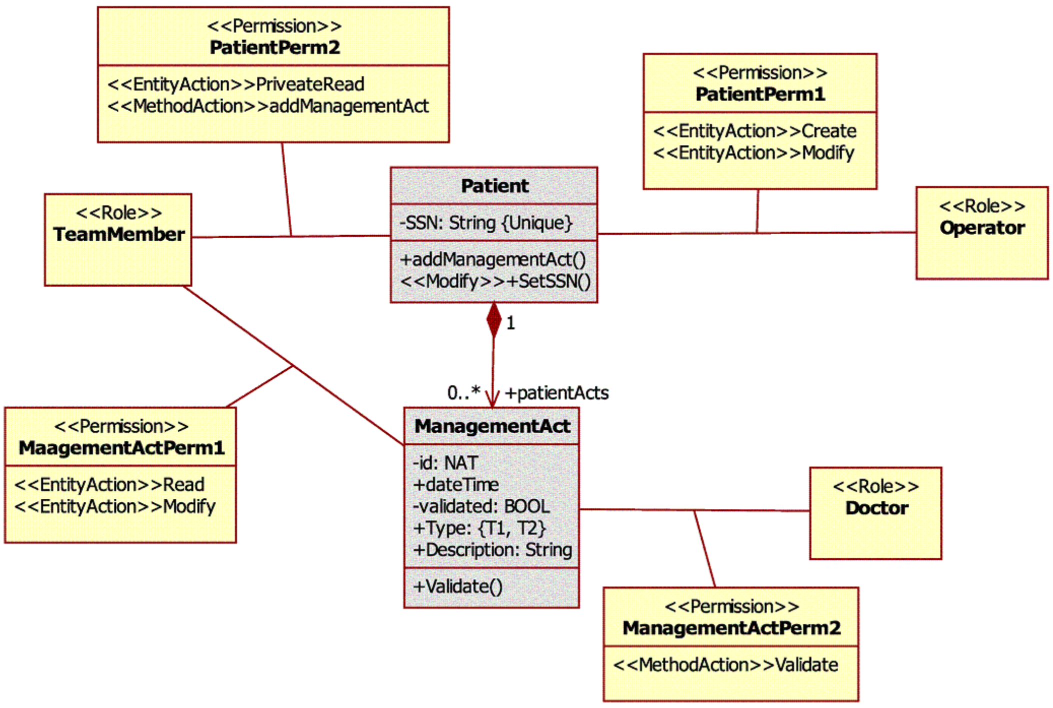


FIG. 6.10 – Expression d'une politique de sécurité sur l'exemple de la figure 6.1

- 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 ↦ private),
                    (Validated ↦ private),
                    (dateTime ↦ public),
                    (Type ↦ public)}
    ||
    AttributeOf := {(SSN ↦ Patient),
                  (Validated ↦ ManagementAct),
                  (dateTime ↦ ManagementAct),
                    (Type ↦ ManagementAct)}
    ||
    OperationOf := {(CreatePatient ↦ Patient),
                  (DeletePatient ↦ Patient),
                  (Patient_AddManagementAct ↦ ManagementAct),
                  (Patient_SetSSN ↦ Patient),
                  (Patient_GetSSN ↦ Patient),
                  (Patient_DeleteManagementAct ↦ Patient),
                  (Patient_GetManagementActs ↦ Patient),
                  (Managementact_SetdateTime ↦ ManagementAct),
                  (Managementact_GetdateTime ↦ ManagementAct),
                  (Managementact_SetValidated ↦ ManagementAct),
                  (Managementact_GetValidated ↦ ManagementAct),
                  (Managementact_SetType ↦ ManagementAct),
                  (Managementact_GetType ↦ ManagementAct),
                  (Managementact_Validate ↦ ManagementAct)}
    ||
    constructorOf := {(CreatePatient ↦ Patient)}
    ||
    destructorOf := {(DeletePatient ↦ Patient)}
    ||
    StereotypeOps := {(modifyOp ↦ Patient_SetSSN),
                    (readOp ↦ Managementact_GetValidated)}
    ||
    setterOf := {(Patient_SetSSN ↦ SSN),
               (Managementact_SetdateTime ↦ dateTime),
               (Managementact_SetValidated ↦ Validated),
               (Managementact_SetType ↦ Type)}
    ||
    getterOf := {(Patient_GetSSN ↦ SSN),
               (Managementact_GetdateTime ↦ dateTime),
               (Managementact_GetValidated ↦ Validated),
               (Managementact_GetType ↦ 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
...
  $PermissionAssignement := \{(PatientPerm1 \mapsto (Operator \mapsto Patient)),$
                          $(PatientPerm2 \mapsto (Team\_Member \mapsto Patient)),$
                          $(ManagementActPerm1 \mapsto (Team\_Member \mapsto ManagementAct)),$
                          $(ManagementActPerm2 \mapsto (Team\_Doctor \mapsto ManagementAct))\}$
  ||
  $EntityActions := \{(PatientPerm1 \mapsto \{create, modify\}),$
                  $(PatientPerm2 \mapsto \{privateRead\}),$
                  $(ManagementActPerm1 \mapsto \{read, modify\})\}$
  ||
  $MethodActions := \{(PatientPerm2 \mapsto \{Patient\_AddManagementAct\}),$
                  $(ManagementActPerm2 \mapsto \{Managementact\_Validate\})\}$
  ||
  $isPermitted := \varnothing$
```

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 :

$$\textbf{setPermissions} = \textbf{PRE } isPermitted = \varnothing \textbf{ THEN } isPermitted := permissions \textbf{ END};$$

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 $obj \in PATIENTS \wedge obj \notin Patients$ THEN
    SELECT
      $CreatePatient \in isPermitted[currentRole]$
    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 7

Formalising dynamic access control rules

7.1 Integrating ASTD into the security metamodel

The metamodel shown in Fig. 6.6 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. 7.1. This metamodel is based on the metamodel described previously and on the ASTD metamodel shown in Fig. 7.2 in order to link common concepts. New classes are also introduced in order to describe elements from obligation or SoD concept.

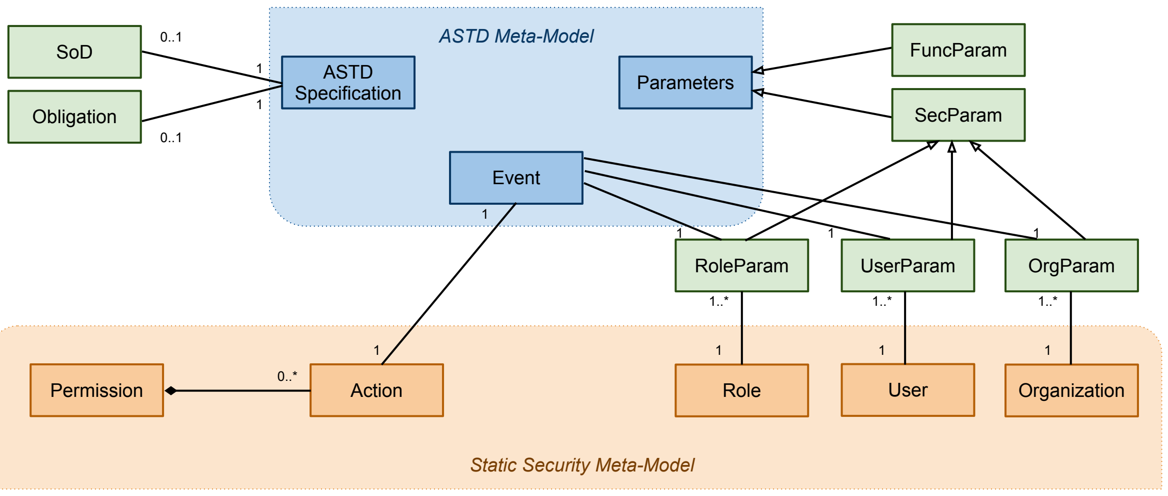


Fig. 7.1 – Méta-modèle de sécurité.

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.

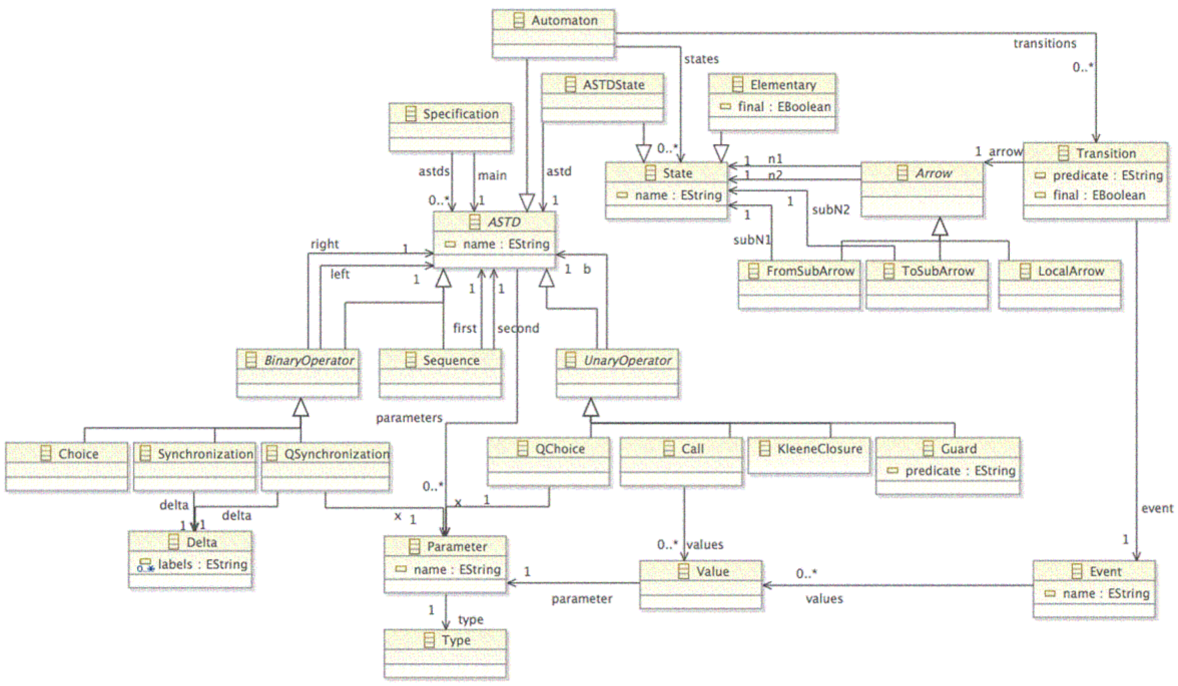


FIG. 7.2 – Méta-modèle des ASTD.

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.

7.2 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 [?]

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 7.3. 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 7.4. 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 7.5. Finally, future work and evolution perspectives will be presented.

7.3 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.

7.4 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.

7.4.1 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 7.4.2 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 $\mid$ 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 $\mid\ x : \mathrm{T}$, 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).

7.4.2 An ASTD Case Study

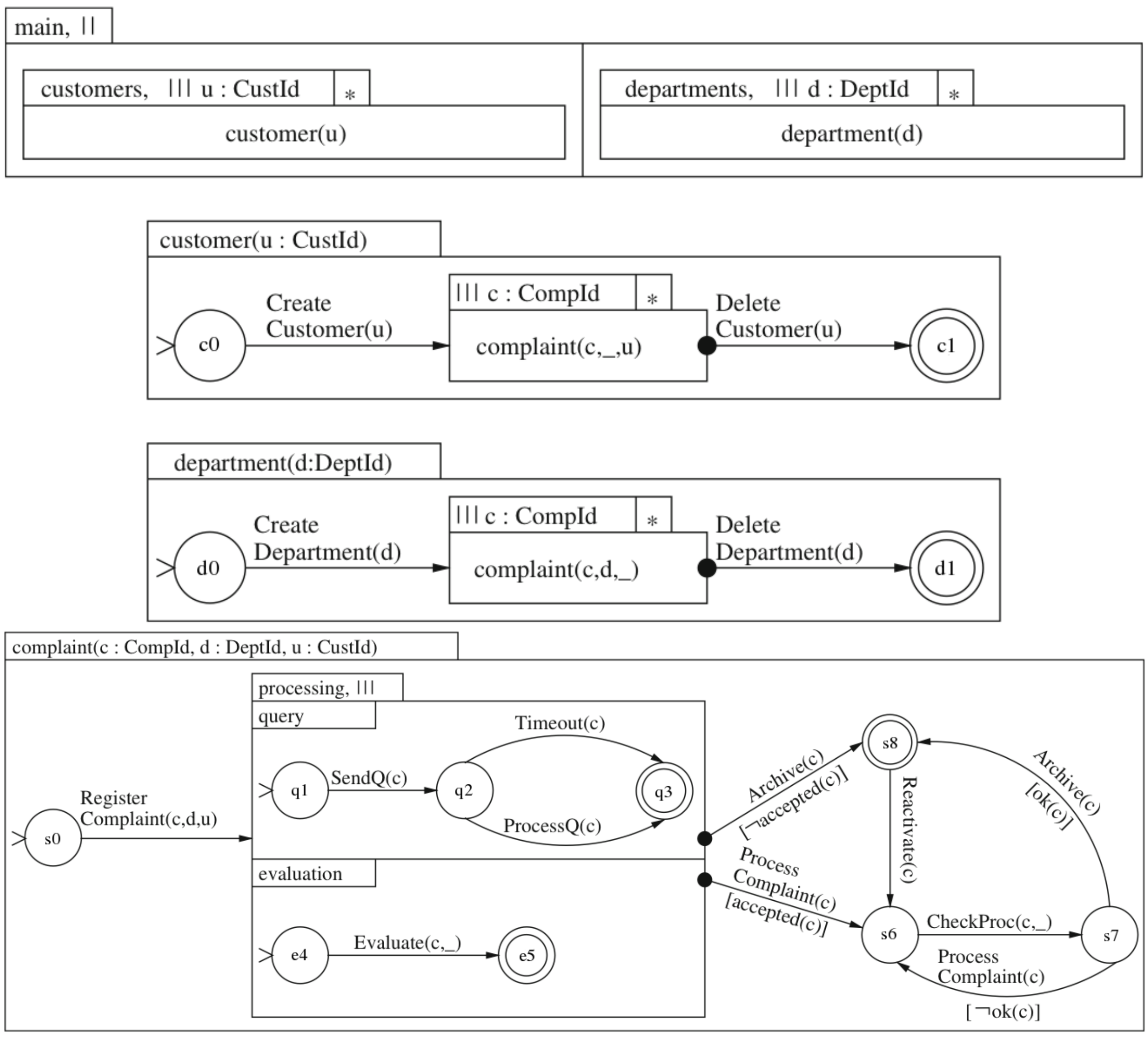


FIG. 7.3 – An ASTD specification describing a complaint management system.

In order to present features and expressiveness of the ASTD notation to the reader, Fig. 7.3 introduces the case study that will be used throughout this chapter. This ASTD models an information 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 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 Archive(c) actions.

7.4.3 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. 7.1 – 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	-	-

7.5 Translation

Translation from ASTD to Event-B is achieved in several steps. Fig. 7.4 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 7.1. 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.

7.5.1 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.

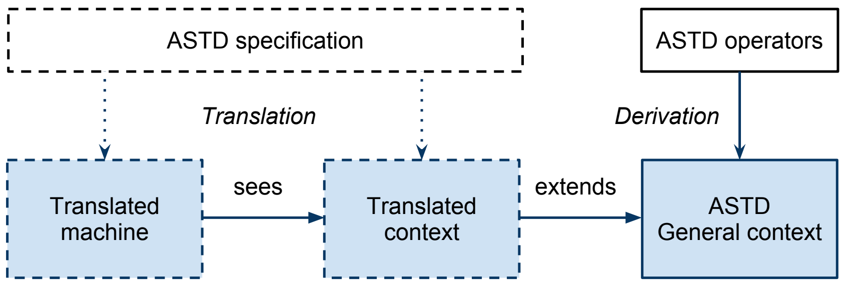


Fig. 7.4 – The architecture resulting from the translation process.

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 7.2. 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 7.2 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. 7.3. 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 $TransCreateCustomer$.

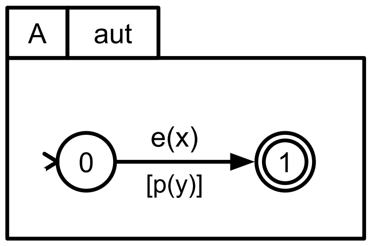
```
Event  CreateCustomer ≙
    any
        u
    where
        grd1 :  u ∈ USERSET
        grd2 :  StateCustomer(u) ∈ dom(TransCreateCustomer)
    then
        act1 :  StateCustomer(u) := TransCreateCustomer(StateCustomer(u))
    end
```

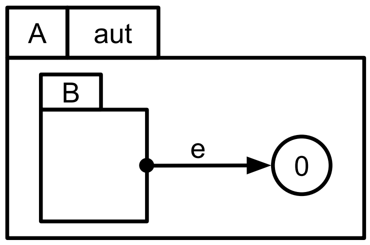
7.5.2 Sequence

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

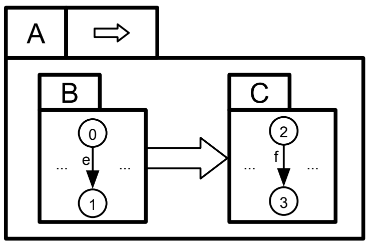
TAB. 7.2 – Automata ASTD to Event-B translation rules

Automata ASTD pattern	Added to the context	Modifications on the machine

	**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 $e \mathrel{\widehat{=}}$
		any
		x
		where
		g1 : $x \in XSET$
		g2 : $P(y)$
		g3 : $StateA \in dom(TransE)$
		then
		a1 : $StateA := TransE(StateA)$
		end

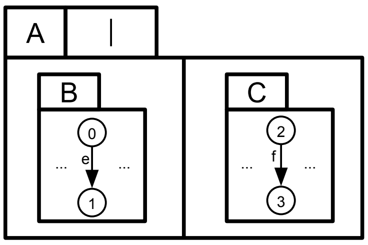
	**CONSTANTS**
	`isFinalB`
	AXIOMS
	ax1 : $isFinalB = \ldots$
	// Depends on B type
		Event $e \mathrel{\widehat{=}}$
		where
		g1 : $isFinalB(StateB) = TRUE$
		$\ldots$

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** $switchSequenceA \mathrel{\widehat{=}}$ **where** g1 : $isFinalB(StateB) = TRUE$ **then** a1 : $StateA := right$ **end** **Event** $e \mathrel{\widehat{=}}$ **where** g2 : $StateA = left$ $\ldots$ **Event** $f \mathrel{\widehat{=}}$ **where** g3 : $StateA = right$ $\ldots$

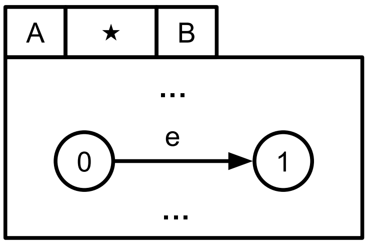
7.5.3 Choice

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
	**Event** $e \mathrel{\widehat{=}}$ **where** `g1` : $StateA = first \lor StateA = none$ . . . **then** `a1` : $StateA := first$ . . . **Event** $f \mathrel{\widehat{=}}$ **where** `g2` : $StateA = second \lor StateA = none$ . . . **then** `a2` : $StateA := second$ . . .

7.5.4 Kleene Closure

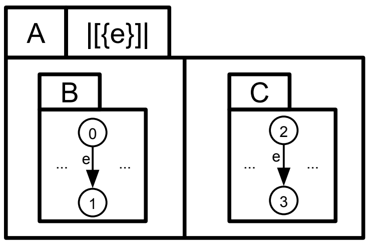
When an iteration of a Kleene closure ASTD is completed, its operand must be reset to initial state. For this reason, an additional event is generated. In the IS, this event is internal and hidden, in the ASTD specification, the semantics off Kleene operator handles the process, but in Event-B the reset 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
	**Event** $lambdaA \mathrel{\widehat{=}}$ **where** `g1` : $isFinalB(StateB) = TRUE$ **then** `a1` : $StateB := initB$ And all sub states . . . **end** **Event** $e \mathrel{\widehat{=}}$ **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.

7.5.5 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
 A |[{e}]| B C 0 e 1 2 e 3 ...	**Event** $e \mathrel{\widehat{=}}$
	where
	gB : $\mathit{guardsfromB}$ ASTD
	gC : $\mathit{guardsfromC}$ ASTD
	$\ldots$
	then
	a1 : $\mathit{StateB} := \ldots$
	a2 : $\mathit{StateC} := \ldots$
	$\ldots$

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.

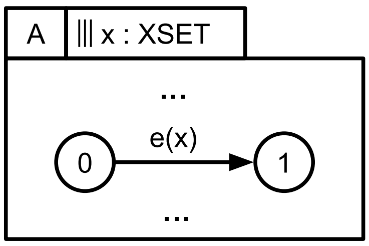
`readyInCustomer` : $\mathit{StateCustomer}(\mathit{AssociationCustomer}(c)) = cc$

`readyInDepartment` : $\mathit{StateDepartment}(\mathit{AssociationDepartment}(c)) = dc$

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.

7.5.6 Quantified Interleaving

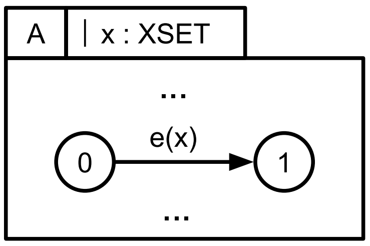
The quantified interleaving does not introduce additional constraints to events. The following table shows how variables induced by quantified interleaving are handled in events.

Quantified interleaving ASTD pattern	Modifications on the machine
 A ||| x : XSET ... 0 e(x) 1 ...	**Event** $e \mathrel{\widehat{=}}$
	any
	x
	where
	g1 : $x \in \mathit{XSET}$
	g2 : $\mathit{StateA}(x) = \ldots$
	$\ldots$
	then
	a1 : $\mathit{StateA}(x) := \ldots$
	$\ldots$

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.

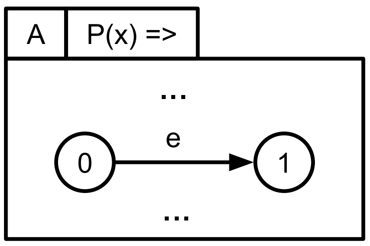
7.5.7 Quantified Choice

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** $e \mathrel{\widehat{=}}$ **any** x **where** g1 : $x \in XSET$ g2 : $StateA = (qNone \mapsto 0)$ $\lor\ StateA = (qSome \mapsto x)$ ... **then** a1 : $StateA := (qSome \mapsto x)$ ...

7.5.8 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** $e \mathrel{\widehat{=}}$ **any** x **where** g1 : $StateA = checked \lor$ $(StateA = notChecked \land P(x))$ ... **then** a1 : $StateA := checked$ ...

7.5.9 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.

7.6 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.

7.7 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.

Chapitre 8

Efficient Execution of ASTD Specifications

The following sections present the implementation of *i*ASTD, an interpreter for Algebraic State Transition diagrams (ASTD) in OCaml. Information system expected behavior can be specified using ASTD, a graphical and formal method based on automata, statecharts and process algebra. *i*ASTD can directly execute a formal model of the system, acting as a controller checking if actions are executed in accordance with ordering properties expressed in the specification.

8.1 Introduction

Informations Systems (IS) play a central role in today businesses. Their usage, complexity but also their cost keep increasing. We strongly believe that formal methods can improve the quality and reduce the cost of IS design. Formal notations being amenable to automated analysis, they can be used for model-based software development, code generation and efficient interpretation, automated verification and testing, thereby reducing development costs, fostering reuse and increasing quality. Originally designed for IS, but reusable for other kind of systems, the Algebraic State Transition Diagrams (ASTD) notation aims to provide designers with a formal and graphical method in order to model acceptable behavior of their system. It can be used to model workflows and its semantics is adapted to model entity life cycle in the system.

In order to provide a tool acting as a controller based on the ASTD notation, without any need to adapt or modify the model, we developed *i*ASTD, an interpreter for ASTD. In the following sections we introduce the ASTD notation and we present the current implementation of *i*ASTD, its features and its usages. Finally we conclude by addressing the current limitations of *i*ASTD and future works.

8.2 The ASTD Notation

In formal methods, there are several specification paradigms such as process algebra, logic-based or state-based languages. In order to specify information systems, the EB^3 [FSD03] method was developed. It features a process algebra similar to CSP [Hoa85] with some IS-oriented additions such as quantifications. However, it lacks a graphical representation that can help during the modeling process and that is one of the advantage of UML statecharts.

The ASTD notation [FGL$^+$08] is a graphical representation linked to a formal semantics created to specify systems such as IS. An ASTD defines a set of traces of actions accepted by the system. ASTD was introduced as an extension of Harel's Statecharts [Har87] and is based on operators from EB^3. An ASTD is built from transitions, denoting action labels and parameters, and places that can be elementary (as states in automata theory) 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. Each type of ASTD is associated with a formal semantics similar to regular expressions, automata, statecharts and EB^3. One of the main important features of ASTD is to allow parametrized instances and quantifications, aspects missing from original statecharts. This means that ASTD can describe not only the behavior of one instance but of all the entities and relationships of the system. Reader is invited to consult a formal, mathematical and complete description of the ASTD notation in [FGLF08].

Fig. 8.1 is an example of ASTD specification describing the behavior of a library IS used for managing members, books and loans of books by members. It was designed with E/R diagram in mind. The life cycle of a *member* and a *book* are described by the corresponding ASTD. The relationship between a given book and a member is described in the *loan* ASTD. Finally, the *main* ASTD describes the overall behavior, linking all members and all books together.

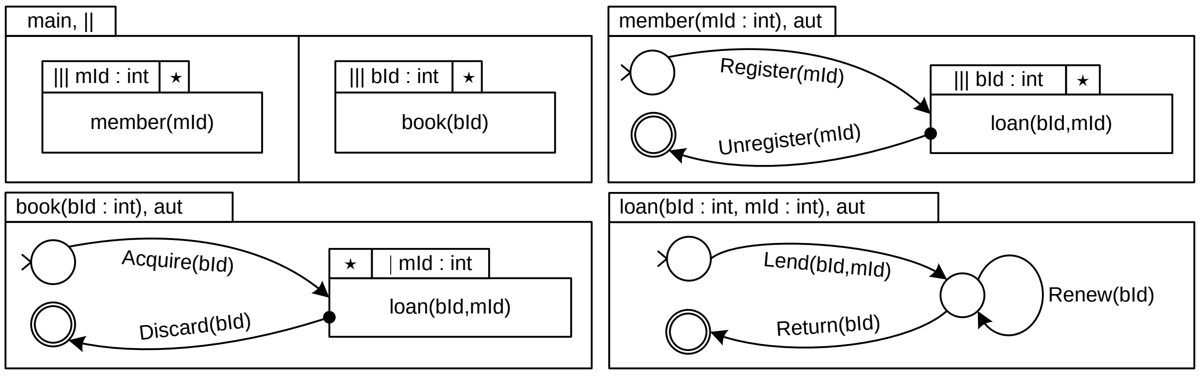


Fig. 8.1 – ASTD specification for a library IS managing members, books and loans.

Analogously to a labelled transition system (LTS), the representation of a system in ASTD is composed of two important parts : (i) a static definition of the possible transitions between acceptable states, which we call the ASTD *topology* ; (ii) the current state of the system. The topology of an ASTD refers to the type, the transitions and the way sub-ASTD are combined. State is defined for each ASTD as a mathematical structure denoting the previously executed events. If an event can be executed, the state of the ASTD evolves. However, the topology of the ASTD remains constant. Once modeled, an ASTD can be executed by providing an event *i.e.* a transition label and its parameters. Before any execution occurs, the ASTD is in an initial state defined by its topology. In our library example modeled in Fig. 8.1, at the initial state, the action register(1) can be executed in order to register the first member of the library. Then a book can be acquired by executing register(10). Since both book 10 and member 1 are now registered in the system, lend(10, 1) can be executed if this member wants to lend this book.

8.3 Control Using *i*ASTD

Once the designer has specified its system using the ASTD notation, this specification can be executed with no modification or any other additional work. The ASTD specification is one of the inputs of *i*ASTD, an interpreter for the ASTD notation. *i*ASTD was developed using the OCaml language [Ler98], a functional language of the ML family.

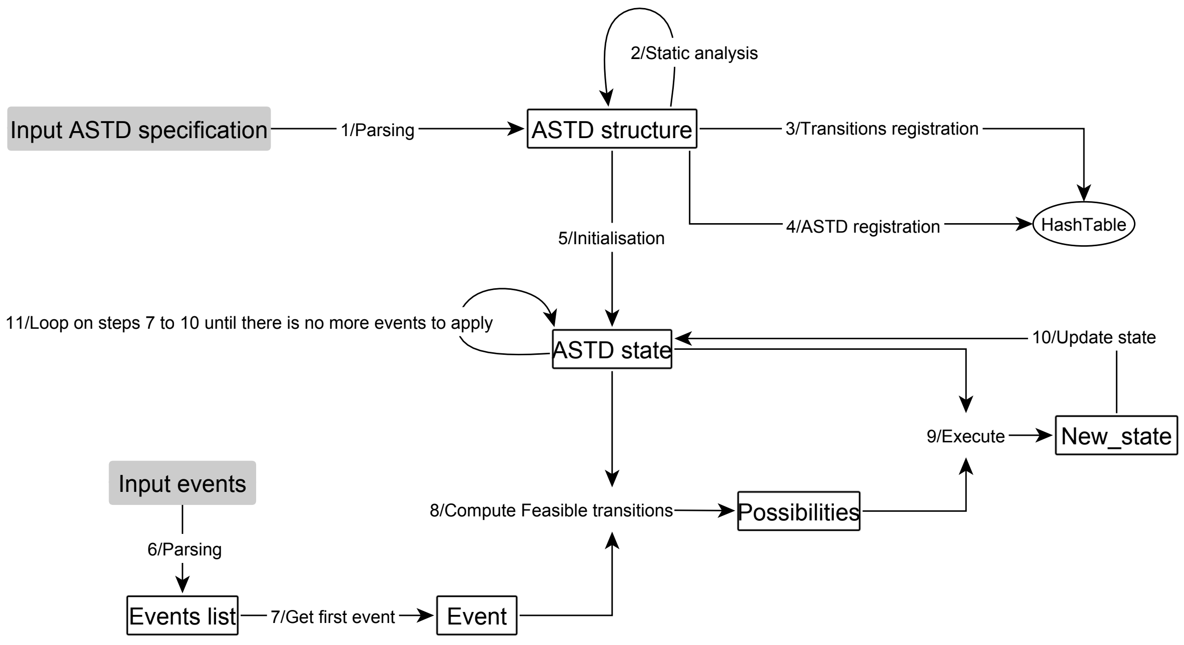


FIG. 8.2 – Main behavior of *i*ASTD

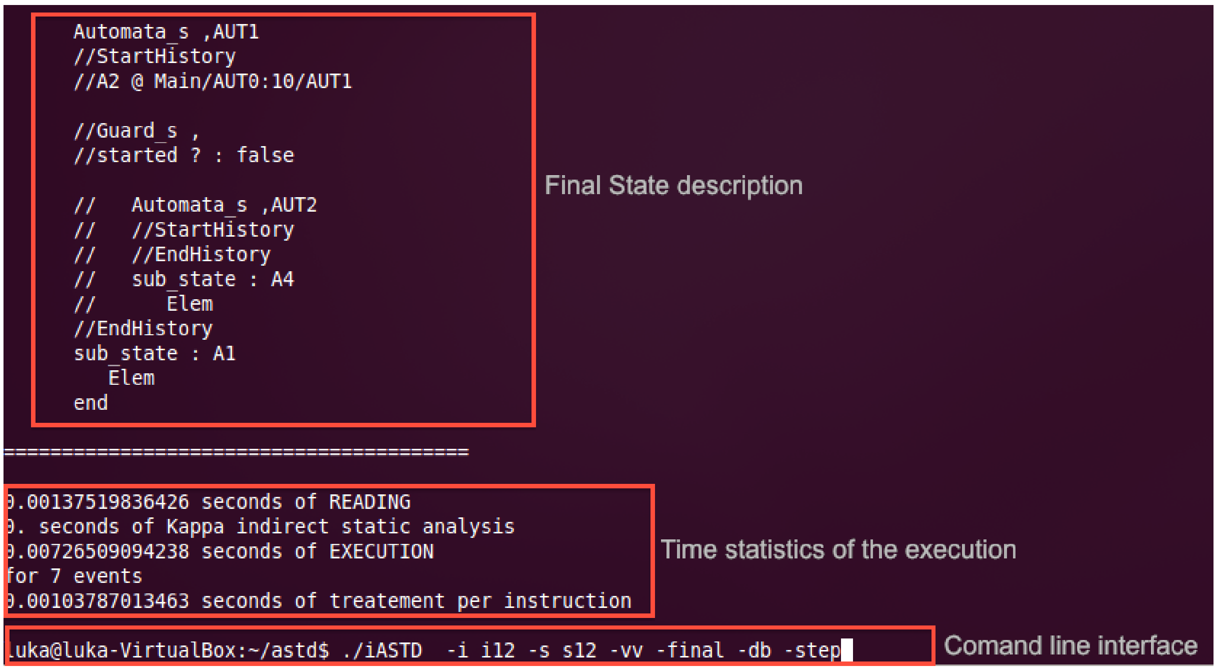
8.3.1 Main Algorithm

Fig. 8.2 describes the main behavior of *i*ASTD. Parsing of the ASTD specification is performed as step 1. Some optimizations for faster execution such as transition registration are performed during steps 2, 3 and 4. Then, the initial state of the specification is computed during step 5. *i*ASTD is now ready to accept input event. *i*ASTD can parse a trace of events that will be executed. For each event and regarding to current state, *i*ASTD computes the list of active transitions. If the event can be executed, *i*ASTD computes the new state of the ASTD and updates data. In the case that event is not executable, state does not evolve, and *i*ASTD skip to the next event. Step 8 is the most time and resources consuming step. It evaluates all guards and environment in order to check if the event can be executed or not. In case of non-determinism *i.e.* if multiple transitions corresponding to an event can be executed, *i*ASTD executes arbitrarily the first active transition found.

In order to improve performances of step 8, some optimizations have been introduced. These algorithms are referred as kappa optimization and were originally developed for EB^3 in [FF06]. Direct kappa optimization is a method allowing *i*ASTD to select the good transition to execute in a quantification. It efficiently selects the right transition to execute by analyzing the value of the parameter and the topology of the ASTD. This optimization reduces the size of the set of possible values for the quantified variable. Indirect kappa optimization is useful in case transitions do not have all parameters that could be needed in order to execute an event. In our example described in Fig. 8.1, the transition `return`(*bId*) does not require the parameter *mid* since there is an implicit link between a book and the member who lends it. However, regarding execution efficiency, this parameter can improve performance if it is provided. Indirect kappa optimization computes the value of such missing parameters.

8.3.2 Other Features

Other features were implemented in *i*ASTD. *i*ASTD can use a database in order to save the current state of the interpreted ASTD in a persistent way. At step 10 of the algorithm of the interpreter,



```
    Automata_s ,AUT1
    //StartHistory
    //A2 @ Main/AUT0:10/AUT1

    //Guard_s ,
    //started ? : false

    //   Automata_s ,AUT2
    //   //StartHistory
    //   //EndHistory
    //   sub_state : A4
    //      Elem
    //EndHistory
    sub_state : A1
       Elem
    end

========================================

).00137519836426 seconds of READING
). seconds of Kappa indirect static analysis
).00726509094238 seconds of EXECUTION
for 7 events
).00103787013463 seconds of treatement per instruction

.uka@luka-VirtualBox:~/astd$ ./iASTD  -i i12 -s s12 -vv -final -db -step
```

FIG. 8.3 – Screen capture of *i*ASTD after execution

*i*ASTD stores the value of the new state in the database. This implementation was realized using a Sqlite database engine [New04]. In case the information system has to be temporally offline, database persistency can prevent data losses. On the other hand, database storage and retrieval is less efficient than memory storage in terms of time for data access. Database storage also allows multiple instances of *i*ASTD to distributedly execute the same specification with a shared state stored in the database. A command line interface was also developed with several flags in order to activate or disable some of the functionalities of *i*ASTD such as database, kappa optimization or verbose mode for example. We are also working on the implementation of *i*ASTD as a service in order to integrate it in a SOA (Service Oriented Architecture) [BMH05] environment.

Fig. 8.3 is a capture of the command line interface of *i*ASTD after the execution of seven events listed in file $i12$ over the specification described in file $s12$. Parameters were added prior to the execution. The $-vv$ option enables a moderate verbose level (five levels, from silent to debug, are available). By adding $-final$, the state of the ASTD is written after the last event is executed. In order to activate database storage of states, the $-db$ parameter was used. Finally, the $-step$ parameter activates a step-by-step mode, where a break point is placed between the execution of each event.

8.3.3 Usages

*i*ASTD can be used as an animator for ASTD specification in order to test it. One can write acceptable traces and test that they are correctly executed by the specification. *i*ASTD can also be used as a controller for a production information system. With very few modifications, the implemented system can call *i*ASTD in order to verify that the event can be executed before any modification to the system is performed. In the case of a new IS, with no implementation available, *i*ASTD can also be used as the core engine of the system but is not sufficient. Indeed, *i*ASTD does not update attributes values of the system. This limitation can be removed by implementing functions that handle these updates. Another usage for *i*ASTD is as an access control engine that computes, according to a security policy written using the ASTD notation, if an user is allowed to execute an event in the system. This kind of usage is detailed in the French SELKIS project[1].

8.4 Conclusion, Current Limitations and Future Works

We have presented *i*ASTD, an interpreter for the ASTD notation. Implemented in OCaml, it can interpret formal specifications with no need to modify, adapt or rewrite them. It can be used as a controller for IS since the ASTD notation is adapted to model workflow-based and E/R-based specifications.In its current implementation, *i*ASTD does not accept input during execution. A complete list of all to-be-executed events must be provided before starting computation. This is a serious limitation regarding expected usages of *i*ASTD, but this problem can be easily addressed with very few modifications to the code of *i*ASTD. Another limitation is regarding attributes of the system. Since *i*ASTD does not have access natively to attributes of the system, one must code by hand some of the functions used to evaluate guards. We are currently developing a graphical editor for ASTD specification [AFLM10]. By linking *i*ASTD and such editor, it would be possible to animate ASTD specification during the specification phase, improving modeling quality. It would also be possible to implement a graphical interface for *i*ASTD in order to visualize the current state of a specification during execution. Since the ASTD notation and EB^3 approach share several characteristics, it would be possible to integrate automatic error message generation [MFF09] in *i*ASTD in order to provide the user with useful information such as the reason why his event was not executed.

[1]http ://lacl.fr/selkis/

Chapitre 9

Proof of Translation Rules

9.1 Introduction

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 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. 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 answer 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. Interpreter *i*ASTD can efficiently execute ASTD specifications. However, there is no tool allowing proof of invariants or property check over an ASTD specification. In [MFGL10] we introduced 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]. In the following we prove that our translation is a good by simulation.

9.2 State translation

In [MFGL10], we introduced a translation mechanism for ASTD topology. In order to prove that this mechanism is correct, we introduce a link between the state of an ASTD and the state of translated Event-B machine.

9.2.1 Automata

The states of an automaton are of type $\langle \mathsf{aut}_\circ, n, h, s \rangle$ where

- $n \in \mathsf{Name}$ denotes the name of the state.
- $h \in \mathsf{Name} \nrightarrow \mathsf{State}$ is a partial function that denotes the last visited sub-state of an automaton ; it is used to implement the notion of *history* state introduced in statecharts.
- $s \in \mathsf{State}$ is the current state of the automaton. It can be a compound state, denoted by type State, or an elementary state, denoted by elem.

In the Event-B machine corresponding to this ASTD, a variable was created to model the state of the automata. Let *StateAutomata* be the name of this variable. Then, in order to translate the state of an ASTD automata in the corresponding Event-B machine, we must ensure that $\mathit{StateAutomata} = n$. Regarding h, the history partial function, there is no need to create a similar function in Event-B since the value of states is still stored in variables event after leaving an automaton to an higher state. State s is then translated according to its type.

9.2.2 Sequence

The type of a sequence state, is $\langle \Rightarrow_{\circ}, \mathit{sideS}, s\rangle$ where $\mathit{sideS} \in [\mathsf{fst} \mid \mathsf{snd}]$ and $s \in \mathsf{State}$. In the Event-B machine corresponding to this ASTD, a variable was created to model the state of the sequence. Let *StateSequence* be the name of this variable. Then $\mathit{StateSequence} = \mathit{sideS}$. State s is then translated according to its type.

9.2.3 Choice

The type of a choice state is $\langle |_{\circ}, \mathit{sideC}, s\rangle$ where $\mathit{sideC} \in (\perp \mid \langle \mathsf{left}\rangle \mid \langle \mathsf{right}\rangle)$ and $s \in (\mathsf{State} \mid \perp)$. Let *StateChoice* be the name of the variable in the Event-B machine corresponding to this ASTD created to model the state of the choice. Then $\mathit{StateChoice} = \mathit{sideC}$.

9.2.4 Kleene Closure

The type of a closure state is $\langle \star_{\circ}, \mathit{started?}, s\rangle$ where $s \in \mathsf{State}$ and $\mathit{started?} \in \mathsf{Boolean}$ Let *StateKleene* be the name of the variable in the Event-B machine corresponding to this ASTD created to model the state of the choice. Then $\mathit{StateKleene} = \mathit{started?}$.

9.2.5 Synchronization

The type of a synchronization state is $\langle |\,[]\,|_{\circ}, s_l, s_r\rangle$ where $s_l, s_r \in \mathsf{State}$. There is no variable associated to the state of a synchronization.

9.2.6 Quantified choice

The type of a quantification choice state is $\langle |:_{\circ}, \mathit{value}, s\rangle$ where $\mathit{value} \in [\perp \mid v]$ with $v \in \mathsf{Term}$. Let *StateQChoice* be the name of the variable in the Event-B machine corresponding to this ASTD created to model the state of the choice. Then $\mathit{StateQChoice} = (\mathit{qNone} \mapsto 0)$ if $\mathit{value} = \perp$ or $\mathit{StateQChoice} = (\mathit{qSome} \mapsto v)$.

9.2.7 Quantified interleaving

The state of a quantified synchronization is of type $\langle |\,[]\,|:_{\circ}, f\rangle$ where $f \in T \to \mathsf{State}$. State of a quantified synchronization in stored as a function such as $\mathit{StateQSynch}(x)$ returns the state associated to value x if the quantification state.

9.2.8 Guard

The type of a guard state is $\langle \Rightarrow_{\circ}, \mathit{started}, s\rangle$ where $\mathit{started} \in \mathsf{Boolean}$ and $s \in \mathsf{State}$. If *StateGuard* is the name of the variable denoting the state of the ASTD in the Event-B machine then $\mathit{StateGuard} = \mathit{started}$.

9.3 Proof by Simulation

Let $\mathbf{A}$ be an ASTD, $\mathbf{A}_t$ its topology and $\mathbf{A}_s$ its state. Let σ be an user query and its parameters that $\mathbf{A}$ can execute, and let $\mathbf{A}'_s$ be the resulting state of this execution. Let τ_t and τ_s be translation functions from ASTD topology and structure to Event-B machine and state, let $M = \tau_t(\mathbf{A}_t)$ the translated machine $S = \tau_s(\mathbf{A}_s)$ and $S' = \tau_s(\mathbf{A}'_s)$ be the current and future state of M. Let e be an event of M. For any Event-B state s, we define s' such as $s \overset{t}{\rightsquigarrow} s'$. s' is the result of the execution of a trace of events $t = e_1, \ldots, e_n$ with s as initial state.

In order to prove the correctness of our translation, we have to prove the following property :

$$\exists\, t\ \exists\, S',\ t = e_1, \ldots, e_n,\ \tau_s(\mathbf{A}_s) \overset{t}{\rightsquigarrow} S'\ :\ \tau_s(\mathbf{A}'_s) = S' \tag{9.1}$$

In other words, there is a sequence of events in M (the translation of ASTD $\mathbf{A}$) that can modify S (the translated state of ASTD $\mathbf{A}$) into S' (the translation of the state of ASTD $\mathbf{A}$ after the execution of σ). Fig. 9.1 illustrates the simulation proof.

$$\begin{tikzcd}
\mathbf{A}_s \arrow[r, "\tau_s"] \arrow[ddd, "\sigma"'] & S \arrow[d, "e_1"] \\
 & S_1 \arrow[d] \\
 & \cdots \arrow[d, "e_n"] \\
\mathbf{A}'_s \arrow[r, "\tau_s"] & S'
\end{tikzcd}$$

FIG. 9.1 – Simulation proof.

In the following sub-sections we describes for each transition type in the ASTD semantics the simulation proof of the translation. We always consider an ASTD which name is $\mathbf{A}$. In the translated Event-B machine, its state is denoted by *StateA*. If we have a transition labelled $e(\overrightarrow{p})$ on the ASTD, we suppose that event e will encode the transition in the Event-B machine. We consider that the event executed by the ASTD is $\sigma = e(\overrightarrow{x})$. In the following, paragraph titles refer to inference rules of [FGLF08]

9.3.1 Automata

In the following, we describe the six transition types of automata ASTD. Each one will use the translation of ASTD transition into functions in the context of the Event-B machine that are reused in the events (in both guards and actions). Regarding history functions, since Event-B variables encoding sub states are not modified when leaving a nested ASTD, there is no need to save the state into a dedicated history function.

aut1

Inference rule aut1 describes the execution of a local transition. In order to prove (9.1), we have to exhibit a trace of events t that complies with the constraints defined in the previous section.

$$\begin{aligned}
\mathbf{A}_s &= \langle \mathsf{aut}_\circ, n_1, h, s\rangle \\
\mathbf{A}'_s &= \langle \mathsf{aut}_\circ, n_2, h', init(\nu(n_2))\rangle \\
\tau_s(\mathbf{A}_s) &= \{StateA = n_1\ ,\ StateN2 = ?\} \\
\tau_s(\mathbf{A}'_s) &= \{StateA = n_2\ ,\ StateN2 = initN2\}
\end{aligned} \tag{9.2}$$

We know that we have $\delta((\mathsf{loc}, n_1, n_2), \sigma, g, final?)$ since the ASTD can execute σ. The Event-B event e is thus coded as following (omitting the parameters that do not interfere here) :

Event $e \mathrel{\widehat{=}}$

where

g1 : $StateA \in dom(TransE)$

g2 : g

g3 : $isFinalA(StateA)$ // depending on the value of $final?$

then

a1 : $StateA := TransE(StateA)$

a2 : $StateN2 := initN2$

Thus, the transition function $TransE$ associated to event e is such as

$$(n_1 \mapsto n_2) \in TransE$$

For that reason, we can execute event e when in state $\tau_s(\mathbf{A}_s)$. The new state is defined by actions $a1$ and $a2$. Thus $S' = \{StateA = n_2\ ,\ StateN2 = initN2\}$. For that reason we have $\tau_s(\mathbf{A}_s) \overset{\{e\}}{\rightsquigarrow} S' = \tau_s(\mathbf{A}'_s)$. Thus, $t = \{e\}$ is an acceptable trace for the simulation.

aut2

Inference rule aut2 describes the execution of a transition going to a deeper ASTD but not to an history state. In this case, we have the following system :

$$\begin{aligned}
\mathbf{A}_s &= \langle \mathsf{aut}_\circ, n_1, h, s\rangle \\
\mathbf{A}'_s &= \langle \mathsf{aut}_\circ, n_2, h', (\mathsf{aut}_\circ, n_{2_\flat}, h_{init}, init(\nu(n_{2_\flat}))))\rangle \\
\tau_s(\mathbf{A}_s) &= \{StateA = n_1\ ,\ StateN2 = ?\} \\
\tau_s(\mathbf{A}'_s) &= \{StateA = n_2\ ,\ StateN2 = n_{2_\flat}\}
\end{aligned} \tag{9.3}$$

Event e resulting from the translation of the transition is :

Event $e \mathrel{\widehat{=}}$

where

g1 : $StateA \in dom(TransE)$

g2 : g

g3 : $isFinalA(StateA)$

g4 : $StateA \in dom(toSubE)$

then

a1 : $StateA := TransE(StateA)$

a2 : $StateN2 := toSubE(StateA)$

with

$$(n_1 \mapsto n_2) \in \mathit{TransE}$$
$$(n_1 \mapsto n_{2_b}) \in \mathit{toSubE}$$

Thus, we have $S' = \{\mathit{StateA} = n_2\ ,\ \mathit{StateN2} = n_{2_b}\}$ as current state after the execution of e. $t = \{e\}$ is an acceptable trace for the simulation.

aut3

Inference rule aut3 describes the execution of a transition going to a deeper ASTD in shallow history state. In this case, we have the following system :

$$\begin{array}{rcl} \mathbf{A}_s & = & \langle \mathsf{aut}_\circ, n_1, h, s \rangle \\ \mathbf{A}'_s & = & \langle \mathsf{aut}_\circ, n_2, h', (\mathsf{aut}_\circ, n_{2_b}, h_{init}, init(\nu(n_{2_b})))\rangle \\ \tau_s(\mathbf{A}_s) & = & \{\mathit{StateA} = n_1\ ,\ \mathit{StateN2} = n_{2_b},\ \mathit{StateN2b} = ?\} \\ \tau_s(\mathbf{A}'_s) & = & \{\mathit{StateA} = n_2\ ,\ \mathit{StateN2} = n_{2_b},\ \mathit{StateN2b} = \mathit{InitN2b}\} \end{array} \quad (9.4)$$

Event e resulting from the translation of the transition is :

```
**Event**   $e \mathrel{\widehat{=}}$
    **where**
        g1 : $StateA \in dom(TransE)$
        g2 : $g$
        g3 : $isFinalA(StateA)$
    **then**
        a1 : $StateA := TransE(StateA)$
        a2 : $StateN2b := InitN2b$
```

with

$$(n_1 \mapsto n_2) \in \mathit{TransE}$$

Thus, we have $S' = \{\mathit{StateA} = n_2\ ,\ \mathit{StateN2} = n_{2_b},\ \mathit{StateN2b} = \mathit{InitN2b}\}$ as current state after the execution of e. $t = \{e\}$ is an acceptable trace for the simulation.

aut4

Inference rule aut4 describes the execution of a transition going to a deeper ASTD in deep history state. In this case, we have the following system :

$$\begin{array}{rcl} \mathbf{A}_s & = & \langle \mathsf{aut}_\circ, n_1, h, s \rangle \\ \mathbf{A}'_s & = & \langle \mathsf{aut}_\circ, n_2, h', h(n_2)\rangle \\ \tau_s(\mathbf{A}_s) & = & \{\mathit{StateA} = n_1\ ,\ \mathit{StateN2} = n_{2_b},\ \mathit{StateN2b} = x\} \\ \tau_s(\mathbf{A}'_s) & = & \{\mathit{StateA} = n_2\ ,\ \mathit{StateN2} = n_{2_b},\ \mathit{StateN2b} = x\} \end{array} \quad (9.5)$$

Event e resulting from the translation of the transition is :

Event $e \,\widehat{=}$
    where
        g1 : $StateA \in dom(TransE)$
        g2 : g
        g3 : $isFinalA(StateA)$
    then
        a1 : $StateA := TransE(StateA)$

with

$$(n_1 \mapsto n_2) \in TransE$$

Thus, we have $S' = \{StateA = n_2\ ,\ StateN2 = n_{2_b},\ StateN2b = x\}$ as current state after the execution of e since e does not modify $StateN2$ and other sub-states. $t = \{e\}$ is an acceptable trace for the simulation.

aut5

Inference rule aut5 describes the execution of a transition going to a shallower ASTD. In this case, we have the following system :

$$\begin{array}{rcl} \mathbf{A}_s & = & \langle \mathsf{aut}_\circ, n_1, h, s \rangle \\ \mathbf{A}'_s & = & \langle \mathsf{aut}_\circ, n_2, h', init(\nu(n_2)) \rangle \\ \tau_s(\mathbf{A}_s) & = & \{StateA = n_1\ ,\ StateN1 = x,\ StateN2 = ?\} \\ \tau_s(\mathbf{A}'_s) & = & \{StateA = n_2\ ,\ StateN1 = x,\ StateN2 = InitN2\} \end{array} \tag{9.6}$$

Event e resulting from the translation of the transition is :

Event $e \,\widehat{=}$
    where
        g1 : $StateA \in dom(TransE)$
        g2 : g
        g3 : $isFinalA(StateA)$
        g4 : $StateN1 \in dom(fromSubE)$
    then
        a1 : $StateA := TransE(StateA)$
        a2 : $StateN2 := InitN2$

with

$$(n_1 \mapsto n_2) \in TransE$$

Thus, we have $S' = \{StateA = n_2\ ,\ StateN2 = InitN2\}$ as current state after the execution of e, with no modification of $StateN1$, in order to keep state in history. $t = \{e\}$ is an acceptable trace for the simulation.

aut6

Inference rule aut6 describes the execution of a transition staying into a deeper ASTD. It does not modify state of current ASTD itself. A new inference rule is called right after this one.

9.3.2 Sequence

seq 1

Inference rule seq 1 describes the execution of a transition staying in the first side of a sequence ASTD. In this case, we have the following system :

$$\begin{aligned}
\mathbf{A}_s &= \langle \Rightarrow_\circ, \mathsf{fst}, s \rangle \\
\mathbf{A}'_s &= \langle \Rightarrow_\circ, \mathsf{fst}, s' \rangle \\
\tau_s(\mathbf{A}_s) &= \{StateA = \mathsf{fst}\} \\
\tau_s(\mathbf{A}'_s) &= \{StateA = \mathsf{fst}\}
\end{aligned} \tag{9.7}$$

Event e resulting from the translation of the transition is :

```
Event  e ≙
    where
        g... : ...
        gSeq : $StateA = \mathsf{fst}$
    then
        a... : ...
```

with guards and actions from sub-ASTD translations.

Thus, we have $S' = \{StateA = \mathsf{fst}\}$ as current state after the execution of e. $t = \{e\}$ is an acceptable trace for the simulation. A new inference rule is called right after this one.

seq 2

Inference rule seq 2 describes the execution of a transition switching from the first side of a sequence ASTD (called B) to the second side (called C). In this case, we have the following system :

$$\begin{aligned}
\mathbf{A}_s &= \langle \Rightarrow_\circ, \mathsf{fst}, s \rangle \\
\mathbf{A}'_s &= \langle \Rightarrow_\circ, \mathsf{snd}, s' \rangle \\
\tau_s(\mathbf{A}_s) &= \{StateA = \mathsf{fst}\} \\
\tau_s(\mathbf{A}'_s) &= \{StateA = \mathsf{snd}\}
\end{aligned} \tag{9.8}$$

Event e resulting from the translation of the transition is :

```
Event  e ≙
    where
        g... : ...
        gSeq : $StateA = \mathsf{snd}$
    then
        a... : ...
Event  switchSequenceA ≙
    where
```

```
        g1 : $isFinalB(StateB) = TRUE$
        g2 : $StateA = \mathsf{fst}$
    then
        a1 : $StateA := \mathsf{snd}$
        a2 : $StateC := InitC$
    end
```

with guards and actions from sub-ASTD translations.

Since { StateA $=$ fst}, we have to execute *switchSequenceA* in order to have { StateA $=$ snd}. Then we can execute e in order to modify the sub-state of the ASTD. Thus, we have $S' = \{StateA = \mathsf{snd}\}$ as current state after the execution of *switchSequenceA* followed by e. Condition from the inference rule ensuring that fstis final is verified by guard $g1$ $t = \{switchSequenceA,\ e\}$ is an acceptable trace for the simulation. A new inference rule is called right after this one.

seq 3

Inference rule seq 3 describes the execution of a transition staying in the second side of a sequence ASTD. It is similar to seq 1 and only differs by the substitution of fst by snd. A new inference rule is called right after this one.

9.3.3 Choice

|1

Inference rule |1 describes the execution of the first transition of a choice ASTD located in the left side. In this case, we have the following system :

$$\begin{aligned}
\mathbf{A}_s &= \langle |_\circ, \bot, \bot\rangle \\
\mathbf{A}'_s &= \langle |_\circ, \mathsf{left}, s'\rangle \\
\tau_s(\mathbf{A}_s) &= \{StateA = none\} \\
\tau_s(\mathbf{A}'_s) &= \{StateA = \mathsf{left}\} \qquad (9.9)
\end{aligned}$$

Event e resulting from the translation of the transition is :

```
Event  $e \mathrel{\widehat{=}}$
    where
        g1 : $StateA = \mathsf{left} \lor StateA = none$
            ...
    then
        a1 : $StateA := \mathsf{left}$
            ...
```

Thus, we have $S' = \{StateA = \mathsf{left}\}$ as current state after the execution of e. $t = \{e\}$ is an acceptable trace for the simulation. A new inference rule is called right after this one.

|2

Inference rule |2 describes the execution of the first transition of a choice ASTD located in the right side. In this case, we have the following system :

$$\begin{aligned}\mathbf{A}_s &= \langle |_\circ, \perp, \perp\rangle \\ \mathbf{A}'_s &= \langle |_\circ, \mathsf{right}, s'\rangle \\ \tau_s(\mathbf{A}_s) &= \{StateA = none\} \\ \tau_s(\mathbf{A}'_s) &= \{StateA = \mathsf{right}\}\end{aligned} \tag{9.10}$$

Event e resulting from the translation of the transition is :

```
Event  $e \mathrel{\widehat{=}}$
    where
        g1 : $StateA = \mathsf{right} \lor StateA = none$
             ...
    then
        a1 : $StateA := \mathsf{right}$
             ...
```

Thus, we have $S' = \{StateA = \mathsf{right}\}$ as current state after the execution of e. $t = \{e\}$ is an acceptable trace for the simulation. A new inference rule is called right after this one.

|3

Inference rule |3 describes the execution a transition of a choice ASTD located in the left side that is not the first transition executed. In this case, we have the following system :

$$\begin{aligned}\mathbf{A}_s &= \langle |_\circ, \perp, \perp\rangle \\ \mathbf{A}'_s &= \langle |_\circ, \mathsf{left}, s'\rangle \\ \tau_s(\mathbf{A}_s) &= \{StateA = \mathsf{left}\} \\ \tau_s(\mathbf{A}'_s) &= \{StateA = \mathsf{left}\}\end{aligned} \tag{9.11}$$

Proof is similar to |1. $t = \{e\}$ is an acceptable trace for the simulation. A new inference rule is called right after this one.

|4

Inference rule |4 describes the execution of a transition of a choice ASTD located in the right side that is not the first transition executed. In this case, we have the following system :

$$\begin{aligned}\mathbf{A}_s &= \langle |_\circ, \perp, \perp\rangle \\ \mathbf{A}'_s &= \langle |_\circ, \mathsf{right}, s'\rangle \\ \tau_s(\mathbf{A}_s) &= \{StateA = \mathsf{right}\} \\ \tau_s(\mathbf{A}'_s) &= \{StateA = \mathsf{right}\}\end{aligned} \tag{9.12}$$

Proof is similar to |2. $t = \{e\}$ is an acceptable trace for the simulation.

9.3.4 Kleene Closure

$\star$1

Inference rule $\star$1 describes the execution of the first transition of a kleene closure ASTD or the execution of a transition as a new iteration. In this case, we have the following system :

$$\begin{aligned}
\mathbf{A}_s &= \langle \star_\circ, started?, s\rangle \\
\mathbf{A}'_s &= \langle \star_\circ, \mathbf{true}, s'\rangle \\
\tau_s(\mathbf{A}_s) &= \{StateA = started?\} \\
\tau_s(\mathbf{A}'_s) &= \{StateA = \mathbf{true}\}
\end{aligned} \tag{9.13}$$

Events e and $lambdaA$ resulting from the translation of the transition is :

```
Event  $lambdaA \mathrel{\widehat{=}}$
    where
        g1 : $isFinalB(StateB) = TRUE$
    then
        a1 : $StateB := initB$
             And all sub states ...
    end
Event  $e \mathrel{\widehat{=}}$
    then
        a1 : $StateA := TRUE$
             ...
```

Depending on the value of *started*?, the proof is different.

If *started*? = **true** then the execution of e is part of a new iteration of the ASTD B since we have the premiss $final_b(s)$ in the inference rule. This means that the ASTD B must be reseted before e can be executed. e can only be executed when B is in its initial state. After the execution of *labdaA*, we have { StateA = **true**, StateB = initB }. We can now execute e. Hence $t = \{lambdaA,\ e\}$ is an acceptable trace for the simulation.

If *started*? = **false** then the execution of e is the first transition to be executed by the ASTD. For that reason $S = \{StateA = \mathbf{false},\ StateB = initB\}$. e can be executed. Hence $t = \{e\}$ is an acceptable trace for the simulation.

$\star$2

Inference rule $\star$2 describes the execution of a transition staying into a deeper ASTD. It does not modify state of current ASTD itself. A new inference rule is called right after this one.

9.3.5 Synchronization

$|[]|$1

Inference rule $|[]|$1 describes the execution of a transition in the left side of the ASTD. Since this event is not synchronized, it is only present in the left side of the ASTD. Hence, only the state corresponding to the left ASTD will evolve. A new inference rule is called right after this one.

$|\,[]\,|$2

Inference rule $|\,[]\,|$2 describes the execution of a transition in the left side of the ASTD. Since this event is not synchronized, it is only present in the left side of the ASTD. Hence, only the state corresponding to the left ASTD will evolve. A new inference rule is called right after this one.

$|\,[]\,|$3

Inference rule $|\,[]\,|$3 describes the execution of a transition in both the left and right sides of the ASTD. Since this event is synchronized, guards from the sub-ASTD are put in conjunction and its actions modify both states as if there were executed separately. Two inference rules are called right after this one.

9.3.6 Quantified choice

$|$:1

Inference rule $|$:1 describes the execution of the first transition of a quantified choice ASTD, when the quantified variable value has not been chosen yet. In this case, we have the following system :

$$\begin{array}{rcl} \mathbf{A}_s & = & \langle |{:}_\circ, \bot, \_\rangle \\ \mathbf{A}'_s & = & \langle |{:}_\circ, v, s'\rangle \\ \tau_s(\mathbf{A}_s) & = & \{StateA = (qNone \mapsto 0)\} \\ \tau_s(\mathbf{A}'_s) & = & \{StateA = (qSome \mapsto v)\} \end{array} \tag{9.14}$$

Event $e(x)$ resulting from the translation of the transition is :

```
Event  e ≙
    any
        x
    where
        g1 : $x \in T$
        g2 : $StateA = (qNone \mapsto o) \vee StateA = (qSome \mapsto x)$
             ...
    then
        a1 : $StateA := (qSome \mapsto x)$
             ...
```

Since $S = \{StateA = (qNone \mapsto 0)\}$ satisfy $g2$ and because $v \in T$ is a premiss of the inference rule, we have $S' = \{StateA = (qSome \mapsto v)\}$ as current state after the execution of $e(v)$. $t = \{e(v)\}$ is an acceptable trace for the simulation. A new inference rule is called right after this one.

$|$:2

Inference rule $|$:2 describes the execution of a transition of a quantified choice ASTD when the quantified variable value has already been chosen. In this case, we have the following system :

$$\begin{aligned}
\mathbf{A}_s &= \langle |{:}_\circ, v, s\rangle \\
\mathbf{A}'_s &= \langle |{:}_\circ, v, s'\rangle \\
\tau_s(\mathbf{A}_s) &= \{StateA = (qSome \mapsto v)\} \\
\tau_s(\mathbf{A}'_s) &= \{StateA = (qSome \mapsto v)\}
\end{aligned} \tag{9.15}$$

Event $e(x)$ resulting from the translation of the transition is similar to the previous section Since $S = \{StateA = (qSome \mapsto v)\}$ satisfy $g2$ and because $v \in T$ is a premiss of the inference rule, we have $S' = \{StateA = (qSome \mapsto v)\}$ as current state after the execution of $e(v)$. $t = \{e(v)\}$ is an acceptable trace for the simulation. A new inference rule is called right after this one.

9.3.7 Quantified interleaving

$|\,[]\,|$:1

Inference rule $|{:}1$ describes the execution of a transition of a quantified interleaving ASTD. In this case, we have the following system :

$$\begin{aligned}
\mathbf{A}_s &= \langle |\,[]\,|\,{:}_\circ, f\rangle \\
\mathbf{A}'_s &= \langle |\,[]\,|\,{:}_\circ, f \mathbin{\lhd\!\!\!+} \{v \mapsto s'\}\rangle \\
\tau_s(\mathbf{A}_s) &= \{StateA(v) = ?\} \\
\tau_s(\mathbf{A}'_s) &= \{StateA(v) = s'\}
\end{aligned} \tag{9.16}$$

Event $e(x)$ resulting from the translation of the transition is :

```
Event  $e \mathrel{\widehat{=}}$
    any
        $x$
    where
        g1 :  $x \in XSET$
        g2 :  $StateA(x) = \ldots$
              $\ldots$
    then
        a1 :  $StateA(x) := \ldots$
              $\ldots$
```

Hence, if $e(v)$ is executed, only $StateA(v)$ is modified. The new value of $StateA(v)$ depends on the next inference rule to be called.

$|\,[]\,|$:2

This inference rule is currently not supported by the translation mechanism. We only translate the quantified interleaving *i.e.* with no synchronization of common actions.

9.3.8 Guard

⇒1

Inference rule ⇒1 describes the execution of the first transition of a guard ASTD. In this case, we have the following system :

$$\begin{array}{rcl} \mathbf{A}_s & = & \langle \Rightarrow_\circ, \mathbf{false}, init(b) \rangle \\ \mathbf{A}'_s & = & \langle \Rightarrow_\circ, \mathbf{true}, s' \rangle \\ \tau_s(\mathbf{A}_s) & = & \{StateA = \ \mathbf{false}\} \\ \tau_s(\mathbf{A}'_s) & = & \{StateA = \mathbf{true}\} \end{array} \tag{9.17}$$

Event $e(x)$ resulting from the translation of the transition is :

```
Event   $e \mathrel{\widehat{=}}$
    where
        g1 : $(StateA = \mathbf{true}) \lor g$
            ...
    then
        a1 : $StateA := \mathbf{true}$
            ...
```

In this case, $S = \{StateA = \ \mathbf{false}\}$, hence the guard g must hold. After the execution of e StateA $= \mathbf{true}$. $t = \{e\}$ is an acceptable trace for the simulation. A new inference rule is called right after this one.

⇒2

Inference rule ⇒2 describes the execution of any but the first transition of a guard ASTD. In this case, we have the following system :

$$\begin{array}{rcl} \mathbf{A}_s & = & \langle \Rightarrow_\circ, \mathbf{true}, s \rangle \\ \mathbf{A}'_s & = & \langle \Rightarrow_\circ, \mathbf{true}, s' \rangle \\ \tau_s(\mathbf{A}_s) & = & \{StateA = \ \mathbf{true}\} \\ \tau_s(\mathbf{A}'_s) & = & \{StateA = \mathbf{true}\} \end{array} \tag{9.18}$$

Hence, StateA is not modified by the execution of e. $t = \{e\}$ is an acceptable trace for the simulation. A new inference rule is called right after this one.

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.

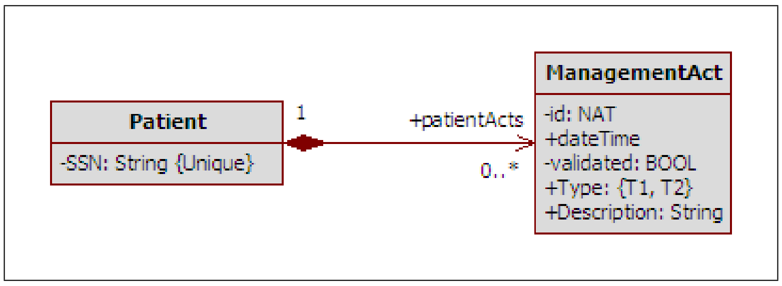


Fig. A.1 – Extrait du diagramme de classes de l'étude de cas IFREMMONT

MACHINE

$\quad Functional\_Model$

SETS

$\quad PATIENTS$;

$\quad MANAGEMENTACTS$;

$\quad SSNs$;

$\quad Types$;

$\quad dateTimes$

VARIABLES

$\quad Patients,\ patient\_SSN,$

$\quad ManagementActs,$

$\quad patientActs,$

$\quad managementact\_validated,$

$\quad managementact\_Type,$

$\quad managementact\_dateTime$

INVARIANT

$\quad Patients \subseteq PATIENTS \wedge$

$\quad ManagementActs \subseteq MANAGEMENTACTS \wedge$

```
    $patientActs \in ManagementActs \rightarrow Patients \wedge$
    $patient\_SSN \in Patients ⤔ SSNs \wedge$
    $managementact\_validated \in ManagementActs \rightarrow \mathbf{BOOL} \wedge$
    $managementact\_Type \in ManagementActs ⇸ Types \wedge$
    $managementact\_dateTime \in ManagementActs ⇸ dateTimes \wedge$
    $\mathbf{dom}(managementact\_validated \rhd \{\mathbf{TRUE}\}) \subseteq \mathbf{dom}(managementact\_Type) \wedge$
    $\mathbf{dom}(managementact\_validated \rhd \{\mathbf{TRUE}\}) \subseteq \mathbf{dom}(managementact\_dateTime)$
INITIALISATION
    $Patients := \varnothing \;||$
    $ManagementActs := \varnothing \;||$
    $patientActs := \varnothing \;||$
    $patient\_SSN := \varnothing \;||$
    $managementact\_validated := \varnothing \;||$
    $managementact\_Type := \varnothing \;||$
    $managementact\_dateTime := \varnothing$
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
        $Patients := Patients - \{obj\} \;||$
        $ManagementActs := ManagementActs - patientActs^{-1}[\{obj\}] \;||$
        $managementact\_validated := patientActs^{-1}[\{obj\}] ⩤ managementact\_validated \;||$
        $managementact\_Type := patientActs^{-1}[\{obj\}] ⩤ managementact\_Type \;||$
        $managementact\_dateTime := patientActs^{-1}[\{obj\}] ⩤ managementact\_dateTime \;||$
        $patientActs := patientActs ⩥ \{obj\} \;||$
        $patient\_SSN := \{obj\} ⩤ patient\_SSN$
    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
```

```
            $\mathbf{patient\_SSN}(obj) := ssn$
        END
    END ;
$ssn \leftarrow \mathbf{patient\_GetSSN}(obj) =$
    PRE $obj \in PATIENTS \land obj \in Patients$ THEN
        $ssn := patient\_SSN(obj)$
    END ;

$\mathbf{patient\_AddManagementAct}(obj) =$
    PRE $obj \in PATIENTS \land obj \in Patients$ THEN
        ANY $ma$ WHERE
            $ma \in MANAGEMENTACTS \land ma \notin ManagementActs \land$
            $ma \notin \mathbf{dom}(patientActs)$
        THEN
            $ManagementActs := ManagementActs \cup \{ma\} \parallel$
            $patientActs := patientActs \cup \{(ma \mapsto obj)\} \parallel$
            $\mathbf{managementact\_validated}(ma) := \mathbf{FALSE}$
        END
    END ;
$\mathbf{patient\_DeleteManagementAct}(obj) =$
    PRE $obj \in PATIENTS \land obj \in Patients$ THEN
        ANY $ma$ WHERE
            $ma \in MANAGEMENTACTS \land ma \in ManagementActs \land$
            $ma \in patientActs^{-1}[\{obj\}] \land$
            $managementact\_validated(ma) = \mathbf{FALSE}$
        THEN
            $ManagementActs := ManagementActs - \{ma\} \parallel$
            $patientActs := \{ma\} \lhd patientActs \parallel$
            $managementact\_validated := \{ma\} \lhd managementact\_validated \parallel$
            $managementact\_dateTime := \{ma\} \lhd managementact\_dateTime \parallel$
            $managementact\_Type := \{ma\} \lhd managementact\_Type$
        END
    END ;
$managementacts \leftarrow \mathbf{patient\_GetManagementActs}\ (obj) =$
    PRE $obj \in PATIENTS \land obj \in Patients$ THEN
        $managementacts := patientActs^{-1}[\{obj\}]$
    END ;
$validated \leftarrow \mathbf{managementact\_Getvalidated}(obj) =$
    PRE $obj \in MANAGEMENTACTS \land obj \in ManagementActs$ THEN
        $validated := managementact\_validated(obj)$
    END ;
$\mathbf{managementact\_SetType}(obj) =$
```

```
    PRE
        $obj \in MANAGEMENTACTS\ \wedge$
        $obj \in ManagementActs\ \wedge$
        $managementact\_validated(obj) = \textbf{FALSE}$
     THEN
        ANY $type$ WHERE
            $type \in Types$
        THEN
            $\textbf{managementact\_Type}(obj) := type$
        END
    END;
$type \leftarrow \textbf{managementact\_GetType}(obj) =$
    PRE $obj \in MANAGEMENTACTS \wedge obj \in ManagementActs$ THEN
        $type := managementact\_Type(obj)$
    END;
managementact_SetdateTime($obj$) =
    PRE
        $obj \in MANAGEMENTACTS\ \wedge$
        $obj \in ManagementActs\ \wedge$
        $managementact\_validated(obj) = \textbf{FALSE}$
    THEN
        ANY $datetime$ WHERE
            $datetime \in dateTimes$
        THEN
            $\textbf{managementact\_dateTime}(obj) := datetime$
        END
    END;
$datetime \leftarrow \textbf{managementact\_GetdateTime}(obj) =$
    PRE $obj \in MANAGEMENTACTS \wedge obj \in ManagementActs$ THEN
        $datetime := managementact\_dateTime(obj)$
    END;
managementact_Validate($obj$) =
    PRE
        $obj \in MANAGEMENTACTS\ \wedge$
        $obj \in ManagementActs\ \wedge$
        $managementact\_validated(obj) = \textbf{FALSE}\ \wedge$
        $obj \in \textbf{dom}(managementact\_Type)\ \wedge$
        $obj \in \textbf{dom}(managementact\_dateTime)$
    THEN
        $\textbf{managementact\_validated}(obj) := \textbf{TRUE}$
    END
END
```

Annexe B

Spécifications B issues du modèle de sécurité

MACHINE
  $UserAssignements$
SETS
  $ROLES = \{Team\_Doctor, Nurse, Operator, Team\_Member, Regulator\};$
  $USERS = \{Bob, Paul, Martin, Jack, none\}$
VARIABLES
  $roleOf,$
  $Roles\_Hierarchy,$
  $currentUser$
INVARIANT
  $Roles\_Hierarchy \in ROLES \leftrightarrow ROLES \wedge$
  $roleOf \in USERS \to \mathbb{P}\ (ROLES) \wedge$
  $\mathbf{closure1}(Roles\_Hierarchy) \cap \mathbf{id}(ROLES) = \varnothing \wedge$
  $currentUser \in USERS$
INITIALISATION
  $roleOf := \{(Bob \mapsto \{Team\_Doctor\}),$
    $(Paul \mapsto \{Operator\}),$
    $(Martin \mapsto \{Nurse\}),$
    $(Jack \mapsto \{Regulator\}),$
    $(none \mapsto \varnothing\ )\}\ ||$
  $Roles\_Hierarchy := \{(Team\_Doctor \mapsto Team\_Member),$
    $(Nurse \mapsto Team\_Member)\}\ ||$
  $currentUser := none$
OPERATIONS
  $\mathbf{changeUser}(user) =$
  PRE
    $user \in USERS$
  THEN
    $currentUser := user$
  END
END

MACHINE

RBAC_Model

INCLUDES

Functional_Model,

UserAssignements

SETS

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

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

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

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

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

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

$\mathit{Stereotypes} = \{\mathit{readOp}, \mathit{modifyOp}\}$

VARIABLES

AttributeKind, *AttributeOf*, *OperationOf*,
constructorOf, *destructorOf*, *setterOf*, *getterOf*,
PermissionAssignement, *EntityActions*,
MethodActions, *StereotypeOps*,
isPermitted

INVARIANT

$\mathit{AttributeKind} \in \mathit{Attributes} \rightarrow \mathit{KindsOfAtt} \land$

$\mathit{AttributeOf} \in \mathit{Attributes} \rightarrow \mathit{ENTITIES} \land$

$\mathit{OperationOf} \in \mathit{Operations} \rightarrow \mathit{ENTITIES} \land$

$\mathit{constructorOf} \in \mathit{Operations} \mathrel{\rightarrowtail\!\!\!\!\!\!+} \mathit{ENTITIES} \land$

$\mathit{destructorOf} \in \mathit{Operations} \mathrel{\rightarrowtail\!\!\!\!\!\!+} \mathit{ENTITIES} \land$

$\mathit{setterOf} \in \mathit{Operations} \mathrel{\rightarrowtail\!\!\!\!\!\!+} \mathit{Attributes} \land$

$\mathit{getterOf} \in \mathit{Operations} \mathrel{\rightarrowtail\!\!\!\!\!\!+} \mathit{Attributes} \land$

$\mathit{StereotypeOps} \in \quad \mathit{Stereotypes} \leftrightarrow \mathit{Operations} \land$

$\mathit{setterOf} \cap \mathit{getterOf} = \varnothing \land$

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

$\mathit{EntityActions} \in \mathit{PERMISSIONS} \mathrel{\rightarrow\!\!\!\!\!+} \mathbb{P}\ (\mathit{ActionsType}) \land$

$\mathit{MethodActions} \in \mathit{PERMISSIONS} \mathrel{\rightarrow\!\!\!\!\!+} \mathbb{P}\ (\mathit{Operations}) \land$

$\mathit{isPermitted} \in \mathit{ROLES} \leftrightarrow \mathit{Operations}$

DEFINITIONS

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$currentRole == (roleOf(currentUser) \cup \mathbf{ran}(roleOf(currentUser) \lhd \mathbf{closure1}(Roles\_Hierarchy)));$

$$
\begin{aligned}
permissions == {} & PermOpCreation \cup \\
& PermOpDestruction \cup \\
& PermOpPRead \cup \\
& PermOpReadOps \cup \\
& PermOpRead \cup \\
& PermOpPModify \cup \\
& PermOpModifyOps \cup \\
& PermOpModify \cup \\
& PermOpMethodActions
\end{aligned}
$$

INITIALISATION

$$
\begin{aligned}
AttributeKind := \{ & (SSN \mapsto private), \\
& (Validated \mapsto private), \\
& (dateTime \mapsto public), \\
& (Type \mapsto public)\}
\end{aligned}
$$

$$\|$$

$$\begin{aligned}
&\mathit{AttributeOf} := \{(\mathit{SSN} \mapsto \mathit{Patient}),\\
&\quad (\mathit{Validated} \mapsto \mathit{ManagementAct}),\\
&\quad (\mathit{dateTime} \mapsto \mathit{ManagementAct}),\\
&\quad (\mathit{Type} \mapsto \mathit{ManagementAct})\}
\end{aligned}$$

$$\|$$

$$\begin{aligned}
&\mathit{OperationOf} := \{(\mathit{CreatePatient} \mapsto \mathit{Patient}),\\
&\quad (\mathit{DeletePatient} \mapsto \mathit{Patient}),\\
&\quad (\mathit{Patient\_AddManagementAct} \mapsto \mathit{ManagementAct}),\\
&\quad (\mathit{Patient\_SetSSN} \mapsto \mathit{Patient}),\\
&\quad (\mathit{Patient\_GetSSN} \mapsto \mathit{Patient}),\\
&\quad (\mathit{Patient\_DeleteManagementAct} \mapsto \mathit{Patient}),\\
&\quad (\mathit{Patient\_GetManagementActs} \mapsto \mathit{Patient}),\\
&\quad (\mathit{Managementact\_SetdateTime} \mapsto \mathit{ManagementAct}),\\
&\quad (\mathit{Managementact\_GetdateTime} \mapsto \mathit{ManagementAct}),\\
&\quad (\mathit{Managementact\_SetValidated} \mapsto \mathit{ManagementAct}),\\
&\quad (\mathit{Managementact\_GetValidated} \mapsto \mathit{ManagementAct}),\\
&\quad (\mathit{Managementact\_SetType} \mapsto \mathit{ManagementAct}),\\
&\quad (\mathit{Managementact\_GetType} \mapsto \mathit{ManagementAct}),\\
&\quad (\mathit{Managementact\_Validate} \mapsto \mathit{ManagementAct})\}
\end{aligned}$$

$$\|$$

$$\mathit{constructorOf} := \{(\mathit{CreatePatient} \mapsto \mathit{Patient})\}$$

$$\|$$

$$\mathit{destructorOf} := \{(\mathit{DeletePatient} \mapsto \mathit{Patient})\}$$

$$\|$$

$$\begin{aligned}
&\mathit{StereotypeOps} := \{(\mathit{modifyOp} \mapsto \mathit{Patient\_SetSSN}),\\
&\quad (\mathit{readOp} \mapsto \mathit{Managementact\_GetValidated})\}
\end{aligned}$$

$$\|$$

$$\begin{aligned}
&\mathit{setterOf} := \{(\mathit{Patient\_SetSSN} \mapsto \mathit{SSN}),\\
&\quad (\mathit{Managementact\_SetdateTime} \mapsto \mathit{dateTime}),\\
&\quad (\mathit{Managementact\_SetValidated} \mapsto \mathit{Validated}),\\
&\quad (\mathit{Managementact\_SetType} \mapsto \mathit{Type})\}
\end{aligned}$$

$$\|$$

$$\begin{aligned}
&\mathit{getterOf} := \{(\mathit{Patient\_GetSSN} \mapsto \mathit{SSN}),\\
&\quad (\mathit{Managementact\_GetdateTime} \mapsto \mathit{dateTime}),\\
&\quad (\mathit{Managementact\_GetValidated} \mapsto \mathit{Validated}),\\
&\quad (\mathit{Managementact\_GetType} \mapsto \mathit{Type})\}
\end{aligned}$$

$$\|$$

$$\begin{aligned}
&\mathit{PermissionAssignement} := \{(\mathit{PatientPerm1} \mapsto (\mathit{Operator} \mapsto \mathit{Patient})),\\
&\quad (\mathit{PatientPerm2} \mapsto (\mathit{Team\_Member} \mapsto \mathit{Patient})),\\
&\quad (\mathit{ManagementActPerm1} \mapsto (\mathit{Team\_Member} \mapsto \mathit{ManagementAct})),\\
&\quad (\mathit{ManagementActPerm2} \mapsto (\mathit{Team\_Doctor} \mapsto \mathit{ManagementAct}))\}
\end{aligned}$$

$$\|$$

$$\begin{aligned}
&\mathit{EntityActions} := \{(\mathit{PatientPerm1} \mapsto \{\mathit{create}, \mathit{modify}\}),\\
&\quad (\mathit{PatientPerm2} \mapsto \{\mathit{privateRead}\}),\\
&\quad (\mathit{ManagementActPerm1} \mapsto \{\mathit{read}, \mathit{modify}\})\}
\end{aligned}$$

$$\|$$

$$\begin{aligned}
&\mathit{MethodActions} := \{(\mathit{PatientPerm2} \mapsto \{\mathit{Patient\_AddManagementAct}\}),\\
&\quad (\mathit{ManagementActPerm2} \mapsto \{\mathit{Managementact\_Validate}\})\}
\end{aligned}$$

$$\|$$

```
    $isPermitted := \varnothing$
OPERATIONS
setPermissions = PRE $isPermitted = \varnothing$ THEN $isPermitted := permissions$ END ;
secure_createPatient($obj$) =
    PRE $obj \in PATIENTS \wedge obj \notin Patients$ THEN
        SELECT
            $CreatePatient \in isPermitted[currentRole]$
        THEN
            createPatient($obj$)
        END
    END ;
secure_deletePatient($obj$) =
    PRE $obj \in PATIENTS \wedge obj \in Patients$ THEN
        SELECT
            $DeletePatient \in isPermitted[currentRole]$
        THEN
            deletePatient($obj$)
        END
    END ;
secure_patient_SetSSN($obj$) =
    PRE $obj \in PATIENTS \wedge obj \in Patients$ THEN
        SELECT
           $Patient\_SetSSN \in isPermitted[currentRole]$
        THEN
            patient_SetSSN($obj$)
        END
    END ;
$ssn \longleftarrow$ secure_patient_GetSSN($obj$) =
    PRE $obj \in PATIENTS \wedge obj \in Patients$ THEN
        SELECT
            $Patient\_GetSSN \in isPermitted[currentRole]$
        THEN
            $ssn \longleftarrow$ patient_GetSSN($obj$)
        END
    END ;
secure_patient_AddManagementAct($obj$) =
    PRE $obj \in PATIENTS \wedge obj \in Patients$ THEN
        SELECT
            $Patient\_AddManagementAct \in isPermitted[currentRole]$
        THEN
            patient_AddManagementAct($obj$)
        END
     END ;
secure_patient_DeleteManagementAct($obj$) =
    PRE $obj \in PATIENTS \wedge obj \in Patients$ THEN
        SELECT
            $Patient\_DeleteManagementAct \in isPermitted[currentRole]$
        THEN
            patient_DeleteManagementAct($obj$)
        END
    END ;
$managementacts \leftarrow$ secure_patient_GetManagementActs($obj$) =
    PRE $obj \in PATIENTS \wedge obj \in Patients$ THEN
        SELECT
            $Patient\_GetManagementActs \in isPermitted[currentRole]$
        THEN
            $managementacts \leftarrow$ patient_GetManagementActs($obj$)
        END
     END ;

$validated \leftarrow$ secure_managementact_Getvalidated($obj$) =
    PRE $obj \in MANAGEMENTACTS \wedge obj \in ManagementActs$ THEN
        SELECT
            $Managementact\_GetValidated \in isPermitted[currentRole]$
        THEN
            $validated \leftarrow$ managementact_Getvalidated($obj$)
        END
    END ;

secure_managementact_SetType($obj$) =
    PRE $obj \in MANAGEMENTACTS \wedge obj \in ManagementActs$ THEN
        SELECT
            $Managementact\_SetType \in isPermitted[currentRole]$
        THEN
            managementact_SetType($obj$)
        END
    END ;
$type \leftarrow$ secure_managementact_GetType($obj$) =
    PRE $obj \in MANAGEMENTACTS \wedge obj \in ManagementActs$ THEN
        SELECT
            $Managementact\_GetType \in isPermitted[currentRole]$
        THEN
            $type \leftarrow$ managementact_GetType($obj$)
        END
    END ;
secure_managementact_SetdateTime($obj$) =
    PRE $obj \in MANAGEMENTACTS \wedge obj \in ManagementActs$ THEN
        SELECT
            $Managementact\_SetdateTime \in isPermitted[currentRole]$
        THEN
            managementact_SetdateTime($obj$)
        END
    END ;
$datetime \leftarrow$ secure_managementact_GetdateTime($obj$) =
    PRE $obj \in MANAGEMENTACTS \wedge obj \in ManagementActs$ THEN
        SELECT
            $Managementact\_GetdateTime \in isPermitted[currentRole]$
        THEN
            $datetime \leftarrow$ managementact_GetdateTime($obj$)
        END
    END ;
secure_managementact_Validate(obj) =
    PRE $obj \in MANAGEMENTACTS \wedge obj \in ManagementActs$ THEN
        SELECT
            $Managementact\_Validate \in isPermitted[currentRole]$
        THEN
            managementact_Validate(obj)
        END
    END ;

changeCurrentUser(user) =
    PRE $user \in USERS \wedge isPermitted \neq \varnothing$ 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 Univ. 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. Sécurité des Architectures Réseaux et des Systèmes d'Information*, 2008.

[AFLM10] Paul Amar, Marc Frappier, Cecile Lartaud, and Jeremy Milhau. Integrating ASTD in the Rodin platform. In *Rodin User and Developer Workshop 2010*. University of Duesseldorf, September 2010.

[AK06] A. E. Abdallah and E. J. Khayat. Formal Z Specifications of Several Flat Role-Based Access Control Models. In *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop (SEW'06)*, pages 282–292, 2006.

[AP03] N. Amálio and F. Polack. Comparison of Formalisation Approaches of UML Class Constructs in Z and Object-Z. In *Proceedings of the Formal Specification and Development in Z and B (ZB'03)*, pages 339–358. LNCS 2651 Springer, 2003.

[Arg03] In *AFADL'2003*, pages 3–18, January 2003.

[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.

[BCDE09a] D. A. Basin, M. Clavel, J. Doser, and M. Egea. Automated Analysis of Security Design Models. *Information and Software Technology, Special issue on Model Based Development for Secure Information Systems, Elsevier*, 51, Issue 5, 2009.

[BCDE09b] David A. Basin, Manuel Clavel, Jürgen Doser, and Marina Egea. Automated analysis of security-design models. *Inf. & Softw. 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.

[BDT06] D. Basin, J. Doser, and T. Lodderstedt. Model Driven Security : From UML Models to Access Control Infrastructures. *Proceedings of the ACM Transactions on Software Engineering and Methodology (TOSEM'06)*, 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.

[BMH05] P.F. Brown, R. Metz, and B.A. Hamilton. Reference Model for Service Oriented Architecture 1.0. 2005.

[Bos95] A. Boswell. Specification and Validation of a Security Policy Model. *Proceedings of the IEEE Transactions on Software Engineering*, 21(2) :63–68, 1995.

[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 Symp. on Security and Privacy*, 1987.

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

[DLCP00] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. An Overview of RoZ : A Tool for Integrating UML and Z Specifications. In *Proc. 12th Conf. on Advanced information Systems Engineering (CAiSE'2000)*, pages 417–430. LNCS, Vol. 1789, 2000.

[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.

[Fac95] 1995.

[FF06] Benoît Fraikin and Marc Frappier. Efficient interpretation of large quantifications in a process algebra. In *4th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS-2006), Proceedings.* INSTICC Press, May 2006.

[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.

[FGLF08] Marc Frappier, Frédéric Gervais, Régine Laleau, and Benoît Fraikin. Algebraic state transition diagrams. Technical Report 24, Université de Sherbrooke, Département d'informatique, Sherbrooke, Québec, Canada, June 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.

[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.

[FSG$^+$01] D. F. Ferraiolo, R. S. Sandhu, S. I. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for Role-based Access Control. In *ACM Transactions on Information and System Security (TISSEC'-01)*, pages 224–274, 2001.

[GBR07a] M. Gogolla, F. Büttner, and M. Richters. USE : A UML-based Specification Environment for Validating UML and OCL. *Sci. of Comput. Program.*, 69 :27–34, 2007.

[GBR07b] 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.

[Hal94] A. Hall. Specifying and Interpreting Class Hierarchies in Z. In *Proceedings of the Z User Workshop*, pages 120–138. Springer/BCS, 1994.

[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. In Jacques Julliand, editor, *AFADL'2004 - Session Outils*, 2004.

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

[Ida06] Akram Idani. PhD thesis, Université de Grenoble 1 – France, Novembre 2006.

[IL10] Akram Idani and Mohamed-Amine Labiadhand Yves Ledru. 15(3), 2010.

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

[Jac06] Daniel Jackson. *Software Abstractions : logic, language and analysis.* MIT Press, 2006.

[Jür04] J. Jürjens. *Secure Systems Development with UML.* Springer, 2004.

[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.

[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. PhD thesis, 2002.

[Led06] Y. Ledru. Using Jaza to Animate RoZ Specifications of UML Class Diagrams. In *Proc. 30th Annual IEEE/NASA Software Engineering Workshop (SEW-30 2006).* IEEE CS Press, 2006.

[Ler98] X. Leroy. The OCaml programming language. *At http ://caml. inria. fr*, 1998.

[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.

[Mam02] Amel Mammar. PhD thesis, CNAM-Paris, Novembre 2002.

[Mar02] Rafael Marcano. PhD thesis, 2002.

[Mar03] S. Martin. The Best of Both Worlds Integrating UML with Z for Software Specifications. *Journal of Computing and Control Engineering*, 14 :8–11, 2003.

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

[MFF09] J. Milhau, B. Fraikin, and M. Frappier. Automatic Generation of Error Messages for the Symbolic Execution of EB^3 Process Expressions. In *Integrated Formal Methods : 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009, Proceedings*, volume 5423 de Lecture Notes in Computer Science, pages 337–351. Springer Berlin/Heidelberg, 2009.

[MFGL10] Jérémy Milhau, Marc Frappier, Frédéric Gervais, and Régine Laleau. Systematic translation rules from astd to event-b. In *Integrated Formal Methods - 8th International Conference, IFM 2010, Nancy, France, October 11-14, 2010. Proceedings*, volume 6396 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2010.

[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.

[MSGC07] S. Morimoto, S. Shigematsu, Y. Goto, and J. Cheng. Formal verification of security specifications with common criteria. In *Proceedings of the 22nd Annual ACM Symposium on Applied Computing (SAC'07)*, pages 1506–1512, 2007.

[New04] Chris Newman. *SQLite (Developer's Library).* Sams, Indianapolis, IN, USA, 2004.

[Ngu98] Hong Phuong Nguyen. PhD thesis, CNAM-Paris, 1998.

[OSS06] D. OKALAS OSSAMI. *Construction Simultanée de Spécifications Multi-Vues UML et B.* PhD thesis, LORIA -Université Nancy2, 2006.

[Pas95] PhD thesis, Université de Rennes 1, Juillet 1995.

[PSS08] D. Power, M. Slaymaker, and A. Simpson. On Formalizing and Normalizing Role-Based Access Control Systems. *The Computer Journal*, 2008.

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

[Reg02] Hdr, 2002.

[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.

[SDA05] K. Sohr, M. Drouineaud, and G. Ahn. Formal Specification of Role-based Security Policies for Clinical Information Systems. In *Proceedings of the 20th Annual ACM Symposium on Applied Computing*, pages 332–339, 2005.

[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 *AFADL 2010, Poitiers, France*, 2010.

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

[Spi92] J. M. Spivey. The Z Notation : A reference manual (2nd ed.). Prentice Hall, 1992.

[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.

[TRA+09] Manachai Toahchoodee, Indrakshi Ray, Kyriakos Anastasakis, Geri Georg, and Behzad Bordbar. Ensuring spatio-temporal access control for real-world applications. In *SACMAT 2009, 14th ACM Symp. on Access Control Models and Technologies.* ACM, 2009.

[UML04] In Jacques Julliand, editor, *AFADL'2004 - Session Outils*, 2004.

[Utt05a] M. Utting. JAZA : Just Another Z Animator. 2005.

[Utt05b] 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.

[VDM92] 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.

[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.

[YHHZ06] C. Yuan, Y. He, J. He, and Z. Zhou. A Verifiable Formal Specification for RBAC Model with Constraints of Separation of Duty. In *Proceedings of the Information Security and Cryptology (Inscrypt'06)*, pages 196–210. LCNS 4318, Springer, 2006.