Université Bretagne Loire
Laboratoire Lab-STICC
UMR CNRS 6285
ENSTA Bretagne, Brest

Validation Formelle d'Implantation de Patrons de Sécurité

Thèse soutenue le 22 mai 2018 à l'ENSTA Bretagne

Fadi OBEID

Jury :

- **Rapporteur : Frédéric Boniol, Professeur, ONERA**
- **Rapporteur : Roland Groz, Professeur, INP Ensimag, Université de Grenoble**
- **Antoine Beugnard, Professeur, IMT Atlantique**
- **Caroline Fontaine, Professeur, IMT Atlantique**
- **Joaquin Alvaro, Professeur, Telecom Paris**
- **David Eudline, DGA-MI**
- **Directeur : Philippe DHAUSSY, Professeur, ENSTA Bretagne**

Thèse financée par :

- **DGA-MI : Direction Générale d'Armement - Maîtrise de l'Information**
- **Brest Métropole**
- **European Project "Environmental Research Infrastructures Providing Shared Solutions for Science and Society" (Envri +).**

Préambule

Les systèmes de contrôle et d'acquisition de données (SCADA) posent aujourd'hui des défis pour les experts de sécurité. De nombreux travaux ont eu pour objectif d'élaborer des solutions théoriques, des guides méthodologiques et recommandations, pour renforcer la sécurité et la protection des services offerts.

Une des solutions proposées est d'intégrer des mécanismes de sécurité qui répondent à des problèmes de sécurité récurrents. Ces mécanismes sont décrits sous la forme de patrons de sécurité comme solutions méthodologiques à adapter aux spécificités des architectures considérées. L'utilisation des patrons de sécurité nécessite de vérifier que les contre-mesures implantées dans les architectures sont fiables. Une contre-mesure est considérée fiable si elle résout un problème de sécurité sans affecter les autres exigences du système.

Une fois un modèle d'architecture modifié pour intégrer des mécanismes de sécurité, il est nécessaire de valider formellement ce nouveau modèle au regard des exigences attendues. Les techniques de *model checking* permettent cette validation en vérifiant, d'une part, que les propriétés du modèle initial sont préservées et, d'autre part, que les propriétés associées aux mécanismes de sécurité sont respectées.

Dans ce travail, nous cherchons à étudier les méthodes et les concepts pour générer des modèles architecturaux respectant des politiques de sécurité et des exigences de sécurité spécifiques. À partir d'un modèle d'architecture SCADA, d'une politique de sécurité et d'une librairie des patrons de sécurité, nous souhaitons générer, automatiquement, une architecture que nous nommons *Architecture sécurisée*. Chaque patron de sécurité est décrit par une description formelle de sa structure et de son comportement, ainsi qu'une description formelle des propriétés de sécurité associées à ce patron.

Cette thèse rend compte des travaux sur l'exploitation de techniques de vérification formelle des propriétés, par model-checking. L'idée poursuivie est de pouvoir générer un modèle d'architecture qui implémente des patrons de sécurité, et de vérifier que les propriétés de sécurité sont respectées dans l'architecture résultante.

Mots clés : Sécurité de l'information, Patrons de Sécurité, Propriétés de Sécurité, Vérification formelle, Model Checking, SCADA.

Remerciements

Tout d'abord, je remercie les rapporteurs de la thèse, les professeurs Frédérique Boniol et Roland Groz, pour leurs commentaires et leur remarques pertinentes me permettant d'améliorer mon travail et le rapport final de la thèse. Je les remercie aussi d'avoir participé au jury de la thèse et de leurs questions pertinentes.

Je remercie également les membres du jury, les professeurs Antoine Beugnard, Caroline Fontaine, Joaquin Alvaro, et David Eudline. Je les remercie aussi pour leurs commentaires et questions relative à la présentation de mes travaux.

Je remercie Antoine Beugnard d'avoir bien voulu présider ce jury.

Un grand remerciement à mon directeur de thèse, Philippe DHAUSSY, avec qui le travail était un grand plaisir. J'ai été chanceux d'avoir Philippe comme directeur de mes travaux.

Je remercie tous mes collègues de l'ENSTA Bretagne pour la bonne ambiance et une expérience de recherche trés agréable et inoubliable. Je remercie en particulier Ciprian TEODOROV et Luka LE ROUX pour leurs soutiens tout le long de ma thèse, et pour tout ce qu'ils m'ont appris au niveau recherche ainsi qu'au niveau personnel.

Finalement, je remercie mes amis et ma famille, surtout mes parents, pour tout leur soutien. Je leurs dois tout mon succès.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Objectif du travail de recherche

Les systèmes de contrôle et d'acquisition de données (SCADA) posent aujourd'hui des défis pour les experts de sécurité. De nombreux travaux [ZS10, FCM10, TLM08, DPP+17, MP16] ont eu pour objectif d'élaborer des solutions théoriques, des guides méthodologiques et recommandations, pour renforcer la sécurité en SCADA et la protection des services offerts.

Une des solutions proposées est d'intégrer des mécanismes de sécurité qui répondent à des problèmes de sécurité récurrents. Ces mécanismes sont décrits sous la forme de patrons de sécurité [FLP10, YB98] comme solutions méthodologiques à adapter aux spécificités des architectures considérées à sécuriser. L'utilisation des patrons de sécurité nécessite de vérifier que les contre-mesures implantées dans les architectures sont fiables. Une contre-mesure est considérée fiable si elle résout un problème de sécurité sans affecter les autres exigences du système.

Une fois un modèle d'architecture modifié pour intégrer des mécanismes de sécurité, il est nécessaire de valider formellement ce nouveau modèle au regard des exigences attendues. Les techniques de *model checking* permettent cette validation en vérifiant, d'une part, que les propriétés du modèle initial sont préservées et, d'autre part, que les propriétés associées aux mécanismes de sécurité sont respectées.

Dans ce travail, nous cherchons à étudier les méthodes et les concepts pour générer des modèles architecturaux respectant des politiques de sécurité et des exigences de sécurité spécifiques. À partir d'un modèle d'architecture SCADA, d'une politique de sécurité et d'une librairie des patrons de sécurité, nous souhaitons générer, automatiquement, une architecture que nous nommons *Architecture sécurisée*. Chaque patron de sécurité est décrit par une description formelle de sa structure et de son comportement, ainsi qu'une description formelle des propriétés de sécurité associées à ce patron.

Pour mener les vérifications formelles de propriétés, nous générons les modèles au format FIACRE [1] qui permet de spécifier les comportements de notre architecture ainsi que ses interactions avec son environnement. Celui-ci inclut les interactions légitimes vis à vis de l'architecture, mais également le comportement d'attaquants potentiels. Le but est de prouver que le modèle d'architecture généré, et donc doté des mécanismes basés sur les patrons de sécurité, respecte les exigences décrites dans la politique de sécurité choisie au regard d'attaques. Dans notre travail, les vérifications formelles sont réalisées avec l'outillage OBP[2] qui a été développé dans l'équipe d'accueil à l'ENSTA Bretagne. Nous formalisons les exigences de sécurité à vérifier avec le langage CDL [DBRL12] associé à cet outil. Des scénarios nominaux, ainsi que des scénarios d'attaques peuvent aussi être décrits en CDL, ce qui permet de profiter des travaux sur la réduction de la complexité lors des explorations des modèles [CT16], ce qui est un aspect problématique bien connu du $model - checking$.

Pour valider les concepts étudiés, nous décrivons deux expérimentations sur des exemples simples de systèmes SCADA intégrant des patrons de sécurité. Mais l'idée poursuivie par ce travail est d'étudier, l'impact de l'intégration des patrons de sécurité sur des modèles de complexité plus réaliste.

[1] Langage défini dans le cadre du projet TopCased (http://www.topcased.org).
[2] OBP est accessible librement sur http://www.obpcdl.org.

1.2 Problématiques abordée dans ce travail

Pour aborder cette recherche, nous résumons la problématique suivant trois enjeux :

- **Enjeu 1 : Formalisation des règles de sécurité.** Il nous faut, tout d'abord, aborder la formalisation des politiques de sécurité. Une politique peut être, par exemple, une politique de contrôle d'accès. La politique, dans ce cas, exprime un ensemble d'hypothèses et d'exigences devant être respectées par le système lors de son utilisation. Les hypothèses expriment le contexte dans lequel se situe le système, c'est-à-dire, d'une part, les conditions ou configurations du système durant son utilisation et, d'autre part, les scénarios d'attaques pouvant survenir sur le système. Les exigences, quant à elles, sont relatives au comportement du système. Elles sont exprimées sous la forme des propriétés formelles. La question se pose donc quant au choix du ou des bons formalismes pour décrire une politique de sécurité. Comment la définir à l'aide des propriétés formelles ?

  De cette formalisation, doit en découler l'identification de mécanismes de sécurité à implanter au sein de l'architecture pour satisfaire les exigences de sécurité. Ces mécanismes logiciels ont pour but de sécuriser l'architecture considérée. Celle-ci, dotée des protections de sécurité, doit pouvoir répondre aux exigences de sécurité dans le contexte qui a été décrit (configuration, scénarios d'attaque). Nous choisissons, dans ce travail, d'orienter nos études sur l'application de patrons de sécurité largement décrits dans la littérature.

  Ce premier enjeu fait l'objet des chapitres 2 et 3.

- **Enjeu 2 : Intégration de mécanismes de sécurité au sein d'une architecture.** Dans notre approche, un mécanisme de protection est modélisé par des éléments de modèles intégrés dans le modèle de l'architecture du système à protéger. Plusieurs mécanismes ou patrons peuvent cohabiter sur une même architecture et se composer, entre eux, et avec les éléments de l'architecture non sécurisée initiale. Il faut donc pouvoir effectuer une composition entre la spécification de la politique de sécurité, l'ensemble des modèles de patrons qui sont choisis et devant être intégrés dans l'architecture et les éléments de l'architecture candidate du système à protéger Le résultat de cette composition ou transformation est une nouvelle architecture générée sécurisée répondant aux exigences de sécurité choisies.

  La figure 1.1 illustre le processus de transformation pour générer un modèle sécurisé en prenant en entrée un modèle non sécurisé, une librairie des patrons de sécurité et une politique de sécurité. Pour l'instant dans notre approche, c'est le concepteur qui choisit, en fonction de la politique de sécurité qu'il souhaite implanter, les patrons qui seront utilisés.

  Le choix de la composition mise en œuvre est spécifié sous forme d'un ensemble de règles de transformation, qui sont appliquées lors de la génération du nouveau modèle de l'architecture.

  Ce deuxième enjeu fait l'objet du chapitre 4.

- **Enjeu 3 : Validation formelle des modèles d'architectures générées.** Le dernier enjeu consiste à vérifier formellement le modèle obtenu par le processus de transformation. Cette validation implique l'identification de critères et de mesures permettant de valider les bonnes performances de cette architecture

  Ce troisième enjeu fait l'objet du chapitre 5.

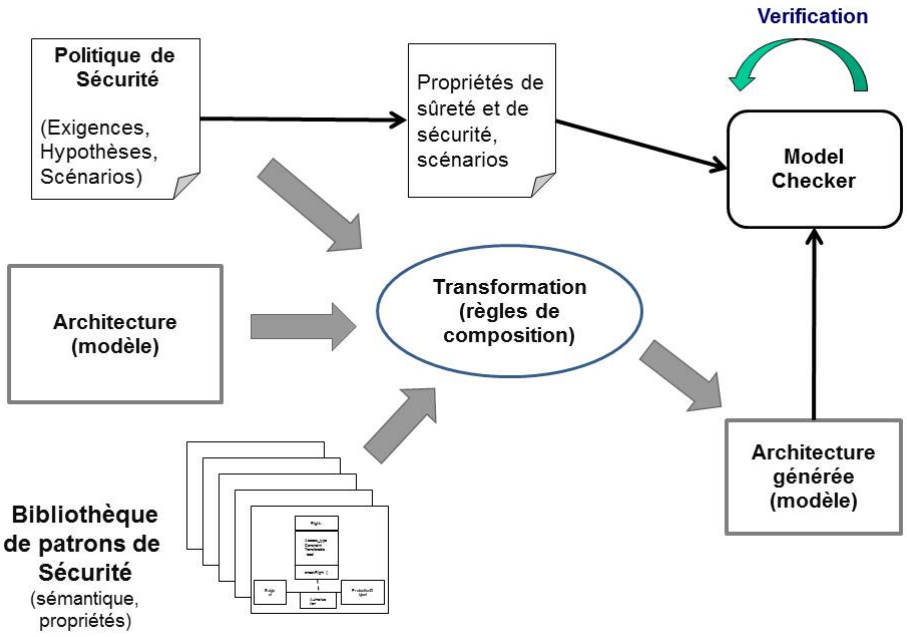

Figure 1.1: Processus de transformation pour la génération d'un modèle architecture.

1.3 Bilan

Dans le cadre de ce travail, nous avons répondu à ces trois enjeux qui peuvent se résumer en notre capacité à :

- Produire un modèle d'architecture avec des mécanismes de sécurité basés sur l'intégration des patrons de sécurité.
- Valider formellement le modèle de l'architecture selon les propriétés de la politique de sécurité choisie.

Pour ce faire, nous avons décrit un cadre formel permettant l'intégration des mécanismes de sécurité au sein de modèles d'architecture et la validation de cette intégration. Ce cadre formel peut être étendu dans le cas, d'une part, par l'ajout de fonctionnalités au sein de chaque patron, ce qui impliquerait d'enrichir ses propriétés formelles en vue de les vérifier sur le modèle d'architecture sécurisée généré. D'autre part, de nouveaux patrons peuvent être impliqués et intégrés suivant la méthode décrite dans ce rapport pour les quatres patrons choisis.

1.4 Plan du mémoire

Ce mémoire est organisé comme suit : Le chapitre 2 introduit un état de l'art et les politiques de sécurité. Le chapitre 3 décrit les patrons de sécurité de la littérature. Le processus et les principes d'intégration des mécanismes de sécurité au sein des architectures sont décrits au chapitre 4. Le chapitre 5 décrit les expérimentations permettant d'illustrer les concepts proposés lors de cette thèse et la technique utilisées pour la vérification formelle

des propriétés. Finalement, nous concluons ce rapport, chapitre 6, par un bilan et des perspectives de recherche.

CHAPTER 2

Etat de l'Art

2.1 Introduction

Le monde de l'information se développe de façon continue et exponentielle, dans notre vie quotidienne et professionnelle. Avec ce développement, les besoins croissants et les nouveaux champs d'utilisation, les attaques que subissent les systèmes informatiques sont de plus en plus nombreuses. Le besoin de protéger nos systèmes et les informations est aujourd'hui essentiel et les protections deviennent incontournables. Les exemples sont nombreux. Un pace-maker, machine dédiée à aider les personnes qui ont des problèmes cardiaques, s'appuie sur des informations qui, si elles devaient être corrompues, causeraient la perte d'une vie humaine. Les installations nucléaires, dépendent beaucoup des systèmes de contrôles automatisés, ne peuvent être corrompues, ce qui pourrait conduire à une catastrophe irrémédiable.

L'augmentation actuelle de la cybercriminalité augmente le besoin de la sécurité de l'information. Les dommages causés par les attaques sur les entreprises dans le monde ont atteint 400 milliards de dollars par an en 2015. Les études prévoient que la cybercriminalité continuera d'augmenter et coûtera aux entreprises dans le monde plus de 6 billions de dollars par an d'ici 2021. La figure 2.1 représente la distributions des cibles des attaques cybers pour 2016 d'après le site *hackmageddon* (https://www.hackmageddon.com/) spécialisé dans les statistiques sur les attaques cyber. On remarque que un quart des attaques vise les installations industrielles.

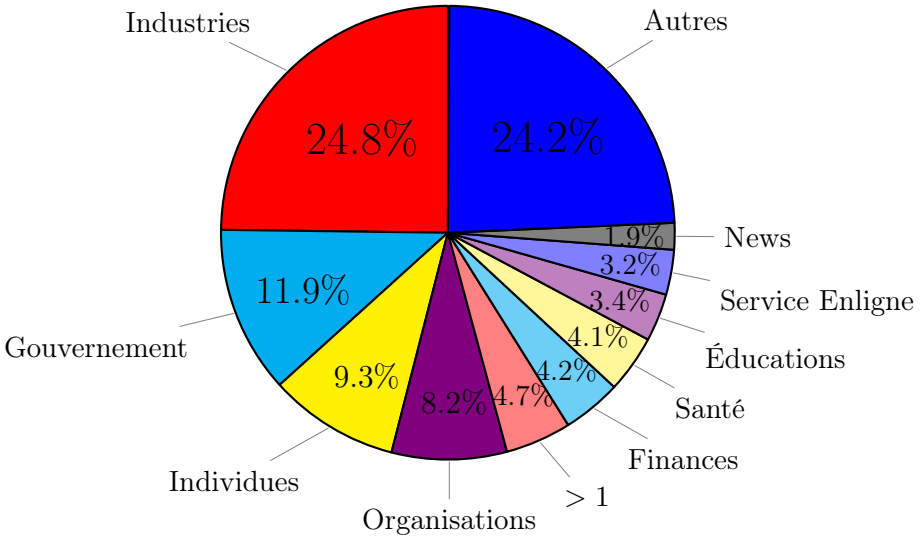

Figure 2.1: Distribution des Cibles des Attaques Cybers (2016).

Pour concevoir des systèmes sécurisés, il faut définir et spécifier des politiques de sécurité adéquates. Celles-ci sont constituées d'un ensemble d'exigences ou propriétés de sécurité qui doivent être prises en compte lors de la conception des systèmes et respectées tout au long de la vie du système. Par exemple, une politique spécifie que l'objet *obj* doit être restreint en écriture à un sujet *subj*. L'application de cette politique assure le respect de l'intégrité de *obj*. Dans un monde parfait, l'application correcte des politiques de sécurité assure la bonne fonctionnalité et la sécurité d'un système selon les lois de sécurité définies. Le problème qui persiste peut être les attaques qui profitent des failles de conceptions et des erreurs d'implantation pour compromettre les politiques de sécurité. Des mesures de sécurité spécifiques devront alors être mises en place pour augmenter la résistance d'un

système contre les attaques [PPK17]. Un système parfaitement sécurisé est un système qui n'offre aucun service mais manque d'intérêt. L'objectif de la sécurité est de rendre de plus en plus robustes les systèmes et d'augmenter au maximum le coût des attaques (en matière de temps, d'effort, et d'argent).

2.2 Les politiques de Sécurité

Une politique de sécurité représente le cœur de la sécurité d'un système d'information. Elle spécifie les contraintes à haut niveau d'un système. La définition d'une politique de sécurité adoptée en général est une spécification d'objectifs de sécurité qui explicitent ce qu'un système doit respecter. Celles-ci définissent ce qui est permis ou interdit dans l'utilisation et l'accès au système. Dans la plupart des cas, les politiques considèrent implicitement que tout ce qui n'est pas autorisé, est interdit. Cependant, dans certains cas, il peut être utile d'expliciter ce qui est interdit. Il est donc souvent indispensable, et c'est ce qui se fait en pratique, de vérifier ou de garantir des interdictions autant que des autorisations. Les politiques doivent être précises, non ambiguës, explicites, cohérentes et consistantes. Par exemple la politique, associée à un composant du type $firewall$, décrit les configurations du $firewall$, les permissions d'accès des utilisateurs et le contrôle du flot de données.

2.2.1 Conception de Politique de Sécurité

Une politique de sécurité est décrite par des objectifs de sécurité qui se déclinent en un ensemble de propriétés de sécurité. Ces objectifs correspondent à un ensemble de propriétés qui peuvent être définies sous une forme littéraire ou formelle. Ces propriétés sont regroupées en trois grandes classes [52085] : confidentialité, intégrité et disponibilité que nous décrivons plus en détail en sous-section 2.3. Chaque propriété représente les conditions que le système doit respecter pour rester dans un état considéré comme sûr. Une définition incorrecte, ou l'application partielle d'une politique, peut entraîner le système dans un état non sûr. Une politique de sécurité n'implante pas seulement des restrictions et des contrôles, elle est définie d'une manière abstraite où les règles sont établies sans tenir compte des mécanismes d'implantation [ASL02]. Une formalisation permet le dialogue entre des concepteurs, des développeurs et des experts du domaine lors de la construction et la validation d'un système à sécuriser.

[RC10] décrit des politiques de sécurité et introduit un formalisme ou un méta-modèle pour la définition d'objectifs de sécurité sur lequel est basé un langage pour la spécification des propriétés de sécurité.

2.2.2 Difficultés de définition des politiques de sécurité

Pour s'assurer d'une bonne définition des politiques de sécurité, nous devons nous assurer que celles-ci soient cohérentes. Ceci signifie que les politiques sont consistantes et n'ont aucun conflit. En outre, ces politiques doivent être complètes en couvrant tous les cas et toutes les possibilités. La difficulté dans ce domaine est que nous ne pouvons pas décider, une fois pour toute, un comportement parfait car chaque système, entreprise ou organisation possède sa propre infrastructure, ses configurations et ses propres besoins et objectifs. Un autre problème est le fait que, dans la plupart des cas, les politiques sont modifiées en permanence et toute mise à jour ne doit nuire ni à la cohérence ni à l'exhaustivité des politiques. Pour cela, nous recherchons la possibilité d'obtenir des garanties de sécurité dans chaque cas et pour chaque configuration.

Les conflits de politique dans la gestion des systèmes distribués sont étudiés dans [LS99], où les raffinements et les modifications sont pris en considération également. [DIR02] traite de la détection de conflits de politiques dynamiques et statiques dans de grands systèmes en évolution. Dans [ASH03], l'objectif était d'obtenir une découverte automatique des anomalies de la stratégie du pare-feu pour révéler les conflits de règles et les problèmes potentiels dans les pare-feu hérités, en tenant compte de l'insertion, du retrait et de la modification des règles. [BLR03] transforme les spécifications de comportement des règles et du système en une notation formelle basée sur Event Calculus (EC) [KS89] qui prend en compte la notion de temps. [Jus03] considère les problèmes rencontrés lors de la mise en œuvre de politiques de contrôle d'accès obligatoire dans des systèmes distribués complexes. Ceci est très intéressant pour les grandes entreprises qui souhaitent garantir l'exactitude de leurs politiques de sécurité. Enfin, [MXYJ10] propose une méthode pour vérifier si le comportement du système satisfait les règles de sécurité. Dans notre travail, nous ne regardons pas la cohérence et l'intégralité des politiques choisies. Nous examinons si le comportement du système respecte ces politiques.

2.3 Propriétés de Sécurité

Une propriété de sécurité est un attribut que l'on donne à un système, ou un élément du système comme un composant, un espace mémoire ou une donnée, un canal de communication, un message, etc. Par exemple, considérons un système qui contient des sujets et des objets, et considérons une propriété qui spécifie que seulement une entité cliente *clt* a le droit d'accéder en lecture à un objet *obj*. Pendant la conception du système, cette propriété doit être prise en compte dans la politique et les mesures de sécurité à mettre en place pour ce système. Cette propriété devient un attribut de ce système. Dans le cas ci-dessus, une telle propriété est appelée propriété de confidentialité.

Dans la suite, nous décrivons différents types de propriétés de sécurité. Ceux-ci peuvent être organisés sous la forme d'un triangle CID (*Confidentialité*, *Intégrité*, et *Disponibilité*).

2.3.1 Confidentialité

La confidentialité concerne le contrôle de l'accès à l'information en lecture. Elle correspond à une non-occurrence de divulgations non autorisées de l'information. Le principe est d'empêcher des entités de lire une information confidentielle, sauf s'ils y sont autorisés.

Une définition formelle peut être donnée ici :

Confidentialité : Soit une information I et soit $\mathcal{X}$ un ensemble d'entités non autorisées à accéder à I. La propriété de confidentialité de $\mathcal{X}$ envers I est respectée si aucun membre de $\mathcal{X}$ ne peut obtenir de l'information de I. (noté $\mathcal{X}\ confidentiality\ I$).

D'un point vue concret, nous pouvons formaliser cette propriété comme suit. Nous notons $\mathcal{C}lt$ l'ensemble d'entités clientes, $\mathcal{O}bj$ l'ensemble des objets du système et $\mathcal{O}per$ l'ensemble des opérations possibles.

Un accès en lecture $READ$ sur un objet obj détenu par un composant c est réalisable par un client clt si ce client a les droits $hasRight$ en lecture sur cette objet.

Formellement :

$$\begin{array}{l} \forall\ c\ \in\ \mathcal{C}omp,\ \forall\ clt\ \in\ \mathcal{C}lt,\ \forall\ obj\ \in\ \mathcal{O}bj, \\ \Box\ [evt\_access\ (c,\ clt,\ (READ,\ obj))\ \Rightarrow \\ hasRight\ (c,\ clt,\ (READ,\ obj))] \end{array} \tag{2.1}$$

La propriété signifie que si *clt* a réussi à lire *obj* détenue par *c*, alors *clt* a le droit d'exercer l'opération *READ* sur *obj*. Nous notons *hasRight* (*c*, *clt*, (*READ*, *obj*)) la fonction qui retourne *true*, si et seulement si le client *clt* a le droit d'exécuter l'opération *oper* sur l'objet *obj* détenu par *c*. Nous notons le couple (*oper*, *obj*) qui englobe une opération sur un objet et $\mathcal{O}pObj$ l'ensemble de tous les instances de ce couple. De plus, nous notons *evt_access* (*c*, *clt*, (*READ*, *obj*)) l'événement détectable au moment où la fonction *access* (*clt*, (*READ*, *obj*)) est appelée par *c*.

Pour la suite du document, nous utilisons les notations suivantes : Soit une fonction *f* (*params*), exécutable par des instances *e* d'une classe *E* et *params* une liste de paramètres. Nous notons *evt_f* (*e*, *params*), l'événement détectable à l'instant où *e* exécute la fonction *f* (*params*). Nous notons *pre_f* (*e*, *params*) le prédicat vrai à partir de l'instant où l'événement *evt_f* (*e*, *params*) a été détecté.

2.3.2 Intégrité

L'intégrité concerne, les accès en écriture. Elle correspond à une non-occurrence d'altérations inappropriées de l'information. C'est-à-dire, des modifications par des clients non autorisés, ou des modifications incorrectes par des clients autorisés (faute accidentelle).

D'une manière générale, l'intégrité désigne le fait que les données, lors de leur traitement, de leur conservation ou de leur transmission, ne doivent subir aucune altération non autorisée. La garantie que des informations proviennent bien des propriétaires est connue sous le nom d'intégrité de l'origine, plus communément appelée authenticité.

Une définition formelle peut être donnée ici :

Intégrité : Soit $\mathcal{X}$ un ensemble d'entités et soit *I* de l'information. La propriété d'intégrité de $\mathcal{X}$ envers *I* est respectée si aucun membre de $\mathcal{X}$ ne peut modifier *I* (noté $\mathcal{X}$ *integrity I*).

Les membres qui sont en dehors de l'ensemble $\mathcal{X}$ sont en général explicitement nommés contrairement aux éléments de $\mathcal{X}$ implicitement désignés. D'un point vue concret, nous pouvons formaliser cette propriété comme suit.

$$\begin{array}{l} \forall\ c\ \in\ \mathcal{C}omp,\ \forall\ clt\ \in\ \mathcal{C}lt,\ \forall\ obj\ \in\ \mathcal{O}bj, \\ \Box\ [evt\_access\ (c,\ clt,\ (WRITE,\ obj))\ \Rightarrow \\ hasRight\ (c,\ clt,\ (WRITE,\ obj))] \end{array} \tag{2.2}$$

Cela signifie que si *clt* a réussi à écrire ou de modifier *obj*, alors *clt* a le droit d'exercer l'opération *WRITE* sur *obj*.

2.3.3 Disponibilité

La disponibilité concerne essentiellement l'accessibilité et les temps de réponse des services. Un système doit être prêt quant à l'utilisation des services qu'il offre aux clients : Fourniture d'un accès à une information pour que les clients autorisés puissent la lire ou la modifier,

faire en sorte qu'aucun client (ou faute accidentelle) ne puisse empêcher les clients autorisés d'accéder à l'information.

La disponibilité est à mettre en parallèle avec la fiabilité, et plus spécifiquement, la sûreté de fonctionnement. La disponibilité traite différents modèles de fautes : pannes franches, faute d'omissions, fautes temporelles et fautes byzantines.

Une définition formelle peut être donnée ici :

Disponibilité : Soit $\mathcal{X}$ un ensemble d'entités et soit I de l'information. La propriété de disponibilité de $\mathcal{X}$ envers I est respectée si tous les membres de $\mathcal{X}$ peuvent accéder à I dans un temps donné s'ils ont le droit, et qu'ils font la demande. (noté $\mathcal{X}$ *disponibility* I).

D'un point vue concret, nous pouvons formaliser cette propriété comme suit.

$$\begin{aligned} &\forall\ clt\ \in\ \mathcal{C}lt,\ \forall\ obj\ \in\ \mathcal{O}bj,\ opObj\ \in\ \mathcal{O}pObj, \\ &\Box\ [evt\_request\ (c,\ clt,\ opObj)\ \wedge\ hasRight\ (c,\ clt,\ opObj)\ \Rightarrow \\ &\Diamond\ evt\_access\ (c,\ clt,\ opObj)] \end{aligned} \tag{2.3}$$

Cela signifie que $\forall\ opObj \in \mathcal{O}pObj$, si clt a le droit d'exercer l'opération $opObj.oper$ sur l'objet $opObj.obj$ et si la demande d'accès ($request(clt, opObj)$ a été exécutée par c ($evt\_request\ (c,\ clt, opObj)$ a donc été détecté), alors clt exécutera l'opération d'accès $opObj.oper$ sur $opObj.obj$. Une propriété de disponibilité est fortement liée au contexte et doit notamment prendre en compte les temps de réponses et les modèles de fautes envisagés.

2.3.4 Propriétés Dérivées

Certaines propriétés de sécurité, sont dérivées d'une ou de plusieurs propriétés relatives au classement CID. Parmi celles-ci sont :

- Intimité (privacy) : respect des libertés individuelles, protection de la vie privée privée (confidentialité)
- Authenticité (authenticity) : l'origine de l'information est authentique (intégrité).
- Non-répudiation (non-repudiation) : l'entité, authentique, qui a écrit, modifié, ou envoyé une information ne peut pas nier avoir fait cette action (intégrité).
- Pérennité (sustainability) : disponibilité de l'information à long terme (disponibilité)
- Exclusivité (exclusivity) : ne fournir l'accès qu'aux clients autorisés (intégrité, confidentialité et disponibilité).

2.4 Les types d'attaques

Différents types d'attaque peuvent être consultés dans la littérature [CSD04].

2.4.1 Écoute Passive

On considère comme écoute passive toutes attaques qui ne modifient pas le système, ce qui rend l'attaque difficile à découvrir. Cela consiste à lire les informations transmises sur un réseau, autrement appeler 'Packets Sniffing'. Le balayage de port est considéré aussi comme une écoute passive, sachant qu'un tel balayage prépare en général pour d'autres attaques. Sous la catégorie de l'écoute passive, on peut considérer aussi, les attaques par canaux auxiliaires qui consistent à une analyse des données pour déduire d'autres informations. Une attaque par canaux auxiliaires peut analyser des nombreux paquets (sniffer) pour déduire la clé de chiffrement par exemple. D'autres attaques par canaux auxiliaires analysent des informations telles que la consommation électrique, le temps d'exécution, la température, etc. Une écoute passive peut compromettre la confidentialité, mais aussi, est souvent utilisée dans la phase de préparation pour d'autres attaques.

2.4.2 Interférence ou Interception

On considère comme attaques d'interférence toutes destructions, modifications, rejeu, ou création des messages. Ce type d'attaque le plus connu est le $Man - in - the - Middle$ (MITM) qui consiste pour un attaquant de modifier la communication entre 2 entités. Ces 2 entités croient être en communication l'une avec l'autre mais en réalité toutes les deux le sont avec l'attaquant. L'attaquant peut lire les informations transmises (confidentialité), modifier, détruire ou envoyer les messages qu'il souhaite (intégrité). Les messages authentiques envoyés à une entité sont interceptés (disponibilité).

2.4.3 DoS et DDoS

Le déni de service (DoS) consiste à rendre une machine ou une ressource indisponible pour ses utilisateurs intentionnels en surchargeant le système. Cela est accompli en saturant la machine ou les ressources avec des demandes superflues, empêchant l'accomplissement de certaines ou toutes les demandes légitimes. Le DoS distribué (DDos) est un DoS où les requêtes superflues proviennent de nombreuses origines différentes. Ces origines peuvent être différents attaquants, mais aussi différents systèmes déjà piratés et contrôlés par un ou plusieurs attaquants. Les DoS, et le DDoS compromettent directement la disponibilité des systèmes. Ils peuvent être utilisés aussi pour produire des vulnérabilités dans un système et ensuite utiliser ces vulnérabilités pour réaliser d'autres attaques, tel que l'installation de logicielles malveillants.

2.4.4 Autres types d'attaque

D'autres types d'attaque peuvent être mentionnés. La *répudiation* consiste à refuser de reconnaître une opération qui a été effectuée (intégrité). L'émetteur d'un message refuse de reconnaître qu'il l'a émis ou le récepteur d'un message refuse de reconnaître qu'il l'a reçu. La *cryptanalyse* consiste à obtenir des informations secrètes à partir des informations publiques (confidentialité). La *déduction* par inférence ou furetage consiste à recouper des informations auxquelles on a légitimement accès pour obtenir des informations confidentielles (confidentialité) comme par exemple des dossiers médicaux. Le *déguisement* ou *masquerade* consiste à se faire passer pour un autre utilisateur (intégrité) et à tromper les mécanismes d'authentification. La *porte dérobée (trap-door)* consiste à contourner des contrôles d'accès. L'utilisation de canaux cachés (covert channels) consiste à transmettre des informations confidentielles à un utilisateur non autorisé (confidentialité...). Ceci s'effectue via les canaux de mémoire directs (réutilisation de buffers, fichiers temporaires,

secteurs disques, ...) ou indirects, les canaux temporels (modulation de l'utilisation de ressources communes, UC, disques, réseaux, périphériques, ...). Les bombes logiques sont des fonctions dévastatrices déclenchées à retardement ou par des conditions particulières (par exemple, lors de la présence de certains utilisateurs, logiciels, matériels) ou après un certain nombre d'activations. Elles peuvent provoquer la destruction d'informations stockées (données, programmes, infos de sécurité), la diffusion de fausses information de diagnostic, des dégâts matériels (usure anormale de périphériques mécaniques), destruction d'écrans, flash-BIOS, ...

Les malwares répertoriés sont des logiciels malveillantes qui prennent une variété de formes de logiciels hostiles ou intrusifs. Les types de malwares, les plus connus sont :

- Virus : peuvent s'attacher à un autre programme ou fichier afin de se reproduire automatiquement.
- Ver ou Worm : autosuffisant, il se reproduit sur un système ou sur un réseau pour toucher d'autres systèmes.
- Ransomware : bloque l'accès aux données, chiffre ces données, ou aussi menace de les publier, à moins qu'une rançon ne soit payée.
- Cheval de Troie : réalise des actions légitimes pour se cacher, et aussi des actions non légitimes dont l'installation des virus, worms, et ransomwares.

Dans notre travail, nous ne couvrons pas tous les types d'attaques. Nous nous focalisons sur les attaques qui peuvent être prises en charge par les types de mécanismes de sécurité que nous considérons. En l'occurrence, nous considérerons les attaques de types écoute passive qui posent les problèmes de confidentialité, d'interférence ou d'interception qui posent les problèmes d'intégrité et les dénis de service qui posent les problèmes de disponibilité.

2.5 Modèles de Protection des systèmes

Plusieurs types de modèles de sécurité ont été proposés et en particulier des types de contrôle d'accès peuvent être consultés dans [Com09]. Ces modèles définissent des politiques abstraites de sécurité pour garantir la satisfaction des propriétés spécifiques à implanter dans les systèmes. Dans la suite, nous présentons quelques modèles de bases.

2.5.1 Contrôle d'accès discrétionnaire (DAC)

Considéré comme le plus ancien modèle de protection, le modèle DAC (*Contrôle d'Accès Discrétionnaire*) a pour caractéristique que le propriétaire des ressources définit les droits d'accès sur celles-ci. Ils sont donc à la discrétion du propriétaire. Par exemple, dans un système d'exploitation, le propriétaire d'un fichier définit les droits d'accès en lecture, d'écriture, exécution de ses fichiers pour les utilisateurs du système.

2.5.2 OrBac

Le modèle OrBAC (*Organization Based Access Control*) [CCC08, Com09] est une extension du modèle RBAC [RSK04], ayant pour objectif de diminuer le nombre de règles de sécurité à spécifier. Dans un modèle OrBAC, une politique de sécurité est un ensemble de permissions, d'obligations et d'interdictions correspondant aux règles de contrôle d'accès

et d'usage en vigueur dans l'organisation. Ces règles dépendent en général du contexte et de l'environnement du système [CCC08].

Dans le modèle OrBac, l'organisation est une entité qui définit et qui gère la politique de sécurité. L'organisation peut être une entreprise ou un composant de sécurité tel qu'un pare-feu. La spécification d'une politique OrBAC se spécifie sur 2 niveaux :

- Un premier niveau, organisationnel ou abstrait, est indépendant de son implantation. Ce niveau contient les rôles, les activités et les vues.
- Le deuxième niveau, concret, est dérivé de la politique organisationnelle. Il spécifie les sujets, les actions et les objets. Un sujet est une entité (ex. personne, processus, utilisateur) pour lequel l'accès à un objet doit être permis. Un objet est une ressource (ex. répertoire, fichier, donnée), pour laquelle le droit d'accès doit être donné. Une action est une opération (ex. lecture, écriture) exécutée par un sujet sur un objet.

Au niveau abstrait, le rôle est un ensemble de sujets sur lesquels sont appliquées les mêmes règles de sécurité. Le rôle est indépendant des fonctionnalités autres que celles de la politique de sécurité. Une activité est ensemble d'actions et une vue est un ensemble d'objets sur lesquels sont appliquées les mêmes règles de sécurité.

Les mécanismes de contrôle d'accès mis en œuvre dans les modèles OrBac sont les suivants :

- **Décision de contrôle d'accès :** une décision, suite à une demande d'accès, peut rendre deux réponses possibles : soit la permission, soit le déni. En cas d'autres réponses, elles doivent être en nombre fini.
- **Règles de sécurité concrètes :** Au niveau concret, une règle décrit les relations parmi les sujets, actions, objets et décisions. Par exemple, la règle $allow$ $(s,\ a,\ o)$ établit que le sujet s peut exécuter l'action a sur un objet o.

Une des difficultés est de décrire l'ensemble des règles au niveau concret car, dans un système opérationnel, réaliste, elles peuvent être très nombreuses. En général, les politiques de sécurité sont exprimées au niveau abstrait via des définitions de haut niveau. Le but est de réduire le nombre de règles.

Ces abstractions sont les suivantes :

- **Abstraction d'un sujet :** Un sujet peut être abstrait par l'utilisation d'attributs et par le rôle qu'il joue dans l'organisation.
- **Abstraction d'une action :** De même, une action peut être abstraite par l'utilisation d'attributs et par le rôle qu'elle joue dans l'organisation. Un activité est l'ensemble de ces attributs.
- **Abstraction d'un objet :** Un objet peut être abstrait par l'utilisation d'attributs et par le rôle qu'il joue dans l'organisation. Une vue est l'ensemble de ces attributs.

Après avoir défini les abstractions des entités, les relations entre les entités abstraites et concrètes doivent être exprimées, implicitement ou explicitement. Un mapping est effectué entre plusieurs sujets et plusieurs rôles, plusieurs actions et plusieurs activités, plusieurs objets et plusieurs vues. Le résultat est l'obtention d'ensembles finis de sujets assignés à chaque rôle, d'actions assignés à chaque activité, d'objets assignés à chaque vue.

Les règles de sécurité doivent être exprimées selon les mappings en vigueur. A un niveau abstrait, une règle de sécurité décrit une relation entre des éléments abstraits et des décisions de contrôle d'accès. Les éléments peuvent être organisés soit à plat, soit de manière hiérarchique pour profiter de propriétés d'héritage en vue de simplifier la spécification de la politique de sécurité. Par exemple, on doit pouvoir spécifier : Soit $role_2$ (resp. $activite_2$ ou vue_2) est un sous-rôle de $role_1$ (resp. $activite_1$ ou vue_1) alors $role_2$ (resp.$activite_2$ ou vue_2) hérite des règles abstraites de $role_1$ (resp. $activite_1$ ou vue_1).

OrBAC possède des prédicats ($empower, consider, use$) pour affecter des entités à l'organisation (rôle, activité, vue) mais aussi afin de formaliser les associations entre les différentes entités du modèle. L'affectation d'un sujet s dans un rôle r au sein de l'organisation org est exprimée de la manière suivante :

$$hasProperty\ (org,\ empower\ (s,\ r))$$

Des règles de sécurité abstraites sont définies et liées à l'organisation, aux rôles, aux activités et aux vues. Ces règles abstraites sont exprimées de la manière suivante :

$$securityRule\ (org,\ permission\ (r,\ a,\ v,\ ctx),\ rule\_name)$$

Cette règle stipule que org donne la permission au rôle r de réaliser l'activité a sur la vue v dans le contexte ctx. A partir des règles de sécurité abstraites et des affectations des entités concrètes aux entités abstraites dans l'organisation, des règles de sécurité concrètes peuvent être dérivées et qui sont liées à un sujet, une action et un objet donnés.

Ainsi nous pouvons exprimer la permission d'un sujet s de réaliser une action a sur un objet o :

$$is\_permitted\ (s,\ a,\ o)$$

OrBAC permet aussi de définir une politique de sécurité mixte. Celle-ci peut combiner des permissions et des interdictions. Le fait de pouvoir à la fois exprimer des permissions et des interdictions peut engendrer des conflits entre règles de sécurité. Afin de les éviter, des priorités peuvent être affectées aux règles de sécurité [FCG07]. OrBAC enrichit l'expression de ses politiques de sécurité. Un troisième type de privilège (obligation) enrichit l'expression de ses politiques de sécurité. A partir d'obligations, des contreparties peuvent être exprimées, par exemple lors de la mise en place de contextes (initialisation, urgence, attaque) ou de dérogations ne rentrant pas dans les cadres conventionnels (mode dégradés, dépassement de forfait, ...).

L'approche OrBAC de modélisation par niveau rend reproductible et évolutive toute politique exprimée. En effet, en cas de modification au niveau organisationnel d'une politique Orbac, celle-ci ne nécessite aucun réajustement au niveau concret qui pourrait introduire des difficultés lors des tests ou des incohérences difficilement décelables. Le niveau concrêt est le niveau que les autres modèles de contrôle d'accès possèdent.

Pour pouvoir augmenter la capacité d'une politique à s'adapter en dynamique à des exigences ou un environnement d'utilisation, il a été proposé [FC08] de lui associer la notion de contexte. Dans OrBAC, le contexte correspond à un état logique. Les contextes sont définis par des règles logiques qui caractérisent dans quelles conditions le contexte est actif. Un contexte est défini pour une organisation, un sujet, une action et un objet. Les règles de sécurité ne sont exécutées que si les conditions associées au contexte sont

remplies (*hold*). Par exemple, si on veut définir un contexte pour lequel une politique spécifique est appliquée. Par exemple, si on veut définir un contexte *initContext* exprimant une phase d'initialisation d'un système *monSysteme* où un sujet *Paul* configure un objet *module* du système avec deux conditions pour que ce contexte soit valide : *Paul* a le rôle d'administrateur et le module est dans un état connecté :

$$\begin{aligned} hold \quad & (monSysteme,\ initContext\ (Paul,\ configure,\ module)) \\ & hasProperty\ (monSysteme, empower\ (Paul,\ admin)) \wedge \\ & hasProperty\ (module,\ connecte) : \end{aligned}$$

La notion de contexte apporte la possibilité d'exprimer des permissions ou des obligations dans certaines circonstances qui par nature évolue dans le temps.

2.5.3 Autres modèles de protection

Bell-LaPadula (BLP) [BL73] : BLP est apparu en 1973 et a été le premier modèle de sécurité. Ce modèle portait sur les préoccupations de confidentialité, et en particulier sur la confidentialité des données de la force aérienne des États-Unis dans les systèmes de main-d'œuvre partageant le temps. Ce modèle fonctionne de la façon suivante : Le niveau de sécurité des sujets et des objets est classifié selon plusieurs niveaux, par exemple, publique, confidentiel ou secret. Tout élément a un niveau de sécurité (classification + compartiment). On définit la propriété $NoReadUp$ (NRU) qui signifie qu'un sujet (exp. X) n'a jamais le droit de lire des objets (exp. A) avec une classification plus forte que la sienne. À partir de cette propriété, nous en déduisons une seconde intitulée $NoWriteDown$ (NWD), qui signifie qu'un sujet n'a jamais le droit d'écrire sur des objets avec une classification plus faible que la sienne. Une variante de ce modèle, est d'utiliser, en plus de la classification, des étiquettes d'autorisation pour résoudre les problèmes de confidentialité au même niveau de secret. Cela oblige tout sujet qui veut lire un objet A d'avoir, en plus d'un niveau de secret suffisant, une liste d'étiquettes contenant toutes les étiquettes de A.

Biba [Bib77] : À l'inverse de BLP, les préoccupations de Biba concernent l'intégrité. Avec les mêmes principes, on crée ce modèle en utilisant $NoWriteUp$ (NWU) (les sujets sont interdits d'écrire sur des objets d'une classification plus forte) et $NoReadDown$ (NRD) (les sujets sont interdits de lire des objets d'une classification plus faible). Si nous regroupons BLP et Biba ensemble dans un système où la confidentialité et l'intégrité sont tout aussi importantes, l'information ne peut pas circuler ni vers le haut ni vers le bas, ce qui signifie qu'il ne peut circuler qu'horizontalement dans le même niveau de confidentialité/d'intégrité. D'autre part, il est avantageux de regrouper BLP et BIBA dans un système où la confidentialité est l'inverse de l'intégrité, car les deux politiques imposeraient les mêmes directions pour les flux de données [Amo94].

Modèle de Clarck-Wilson [CW87] : Repose sur les responsabilités de fractionnement. Par exemple, une transaction bancaire importante nécessiterait l'approbation d'au moins 2 gestionnaires bancaires (ce concept est utilisé jusqu'à aujourd'hui dans les banques). Ou bien la confirmation est effectuée par plusieurs personnes en même temps, ou bien elle est effectuée à travers d'un processus, durant lequel, plusieurs personnes jouent un rôle.

Modèle de Chinese Wall [BN89] : Traite des conflits d'intérêts, ce qui signifie qu'un concessionnaire ayant un objet d'un type, ne peut traiter aucun autre objet du même type. Par exemple, un consultant qui a travaillé pour une entreprise qui travaille dans un domaine, ne peut jamais être consulté par une autre société qui travaille dans le même domaine.

Politique de BMA [AA+96] : Considère une structure de contrôle d'accès horizontale, où chaque objet a une liste de sujets assignés. Par exemple, ce modèle est utilisé dans les hôpitaux, où chaque patient peut avoir une liste qui définisse son médecin principal, infirmières, etc.

The Resurrecting Duckling [SA99] : Unifie le contrôle et la responsabilité. Un objet a 2 états, *Controlled* (est actuellement contrôlé), et *Uncontrolled* (et actuellement libre de se synchroniser avec un contrôleur). Un seul contrôleur existe à la fois, ce contrôleur peut à tout moment décider de mettre l'objet dans l'état *Uncontrolled*. Une illustration, utilisée dans les contrôles domotiques à distance, est le contrôle d'une télévision, de climatiseurs, des enceintes, etc. Certaines variations de ce protocole considèrent le passage naturel de *Uncontrolled* vers *Controlled* immédiatement (ou quelque temps) après l'absence de contrôleur.

Jikzi [AL00] : Propose une utilisation pour les publications sécurisées, avec deux préoccupations principales : la différenciation entre les documents contrôlés et incontrôlés (intégrité) et le maintien de toutes les versions de documents contrôlés (Disponibilité).

2.6 La sécurité dans les systèmes SCADA

Les systèmes d'acquisition et de contrôle de données (SCADA) sont mis en œuvre dans de nombreuses installations industrielles modernes telles que les centrales électriques, les systèmes de gestion de l'eau, les raffineries de pétrole, les installations nucléaires, etc. Ils sont dotés de moyens informatiques qui contrôlent et pilotent les procédés industriels.

Les systèmes industriels sont aujourd'hui très souvent connectés à l'internet des objets industriels [Puy18]. Ceci les rend très vulnérables face aux attaques potentielles. La question de la sécurité est donc un sujet essentiel à prendre en compte lors de leur conception. Compte tenu de leur criticité, les systèmes industriels sont dotés de protection dans le domaine de la sûreté de fonctionnement. Mais sur le plan de la protection face à des attaques, il y a encore beaucoup d'efforts à porter pour concevoir des systèmes qui répondent aux exigences de sécurité.

Un SCADA contient différentes composantes réparties en 5 niveaux (cf figure 2.2) :

- Niveau 0 : Des dispositifs de type capteurs, actionneurs.
- Niveau 1 : Les contrôleurs logiques programmables (PLC) assurent les fonctionnalités de terminaux distants (RTU) qui relient les informations (signaux) entre les contrôleurs (numériques) et les dispositifs de bas niveau. Ils exercent des calculs de base, interprètent les informations récoltées, et émettent les commandes locales. Un

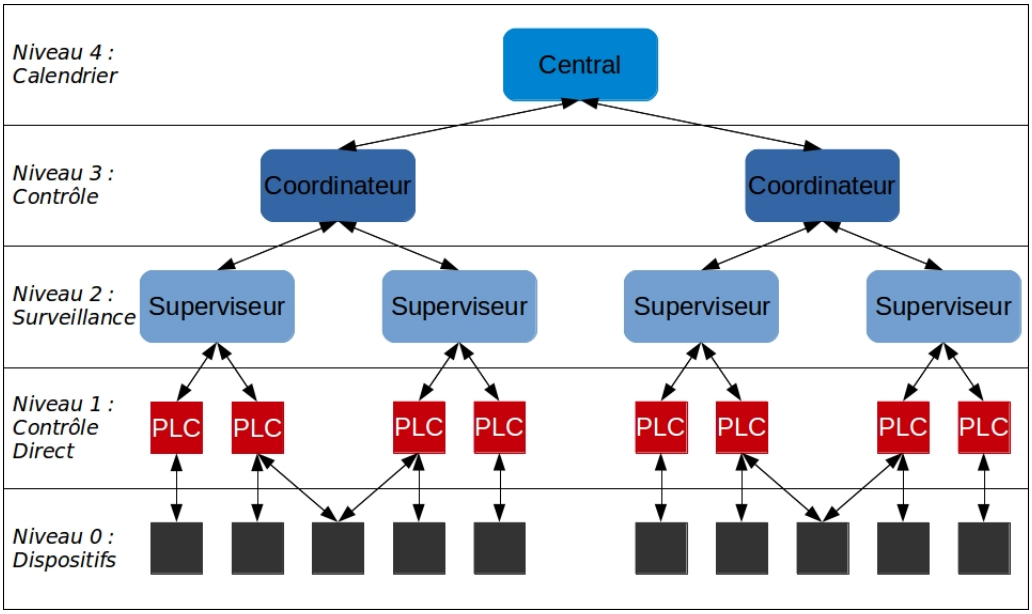

Figure 2.2: Les 5 niveaux d'un système SCADA.

PLC peut collectionner les informations de plusieurs sources, et envoyer des commandes à plusieurs destinataires. Le nombre maximal des sources et destinations dépend du nombre maximal d'entrés et sorties d'un PLC.

- Niveau 2 : Machines de surveillances qui contiennent les logiciels SCADA chargés de contrôler et de surveiller les organes de niveau inférieur (PLC).
- Niveau 3 : Des machines de contrôle et coordination qui gèrent le contrôle des différentes sections de niveau inférieur.
- Niveau 4 : La machine centrale chargée de gérer le plan de production.

A ces quatre niveaux, se superposent, sur les niveaux 2, 3 et 4, des bases de données où les informations sont sauvegardées et accédées. Des interfaces homme-machine simplifient la supervision et le contrôle et donnent accès à des rapports, génèrent des alertes et envoient des notifications.

L'évolution des systèmes SCADA a été marqué par plusieurs phases. Une phase initiale où les systèmes étaient essentiellement monolithiques, largement basés sur des calculateurs (mini-ordinateurs) sans aucune communication entre eux. Ensuite, une phase où les systèmes ont été distribués et basés sur des réseaux (LAN). Les taches exécutées sont partagées entre différents composants. Les systèmes se sont ensuite interconnectés, simplifiant la communication entre les composants, spécialement dans les cas de composants distants et de réseaux multiples. Le domaine de l'*internet des objets* étend la possibilité d'un meilleur partage des informations et donne accès à des contrôles plus complexes.

Mais quelle que soit la complexité des systèmes, nous pouvons, dans notre approche, considérer des modèles d'architectures SCADA de manière abstraite, illustrée par le modèle figure 2.3 inspiré par [FLP10].

La sécurité des systèmes SCADA a toujours été négligée. Pour les premières générations de SCADA, ce n'était pas un problème, car la sécurité était assurée par des dispositifs isolés

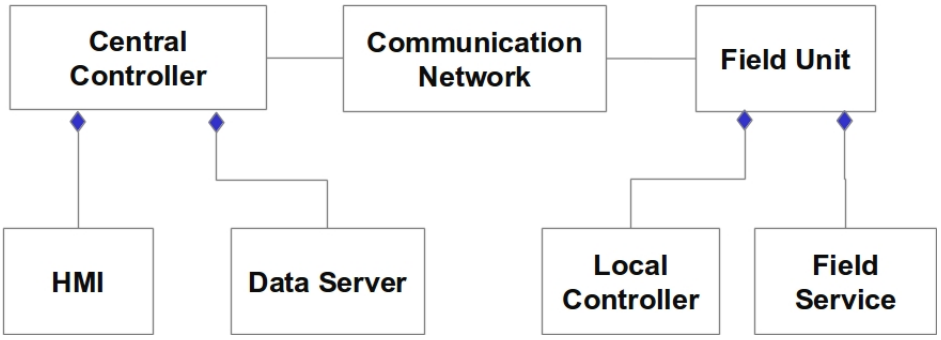

Figure 2.3: Modèle abstrait d'une architecture SCADA.

les uns des autres. Les SCADA aujourd'hui utilisent des composants commercialisés, ainsi que des protocoles standards, ce qui élimine la sécurité par isolation car les composants sont beaucoup plus ouverts à leur environnement. Par ailleurs, les SCADA aujourd'hui mettent en œuvre des liens de communication vers des réseaux locaux et vers l'internet. Ces liens augmentent largement les propriétés des systèmes SCADA, mais aussi, augmentent les problèmes de sécurité.

Beaucoup de problèmes et de difficultés sont présents lorsqu'il s'agit de sécuriser les systèmes SCADA, principalement en raison de contraintes physiques, informatiques et de disponibilité. La sécurité dans les systèmes SCADA actuels est faible voire inexistante, ce qui soulève de nombreuses questions. Une attaque contre un système SCADA peut entraîner des problèmes ayant des répercussions extrêmement graves (pertes de vies humaines, dommages économiques) dans le cas d'installation critique (cas du nucléaire ou du chimique).

En général, un SCADA a cinq types de propriétés de sécurité déjà énoncées en 2.3. L'importance de ces propriétés varie d'un système SCADA à l'autre. L'ordre d'importance décroissant le plus courant est :

- Opportunité (Timeliness) : Réactivité du système, et fraîcheur des données.
- Disponibilité : Travail continu, et toujours disponible.
- Intégrité : Les messages ne sont pas modifiés malicieusement ni générés par des entités non autorisées.
- Confidentialité : les messages et les signaux ne doivent pas être lisible par des entités non autorisées, dans le cas particulier de SCADA, ils ne devraient pas être prévisibles.
- Limitation harmonieuse : les problèmes et les attaques devraient avoir un impact limité et local sans se propager à l'ensemble du système.

Pour sécuriser un système SCADA, il faut assurer que ses propriétés de sécurité sont toujours respectées. Le problème principal est dû aux contraintes des systèmes SCADA en raison de certaines spécificités de ces systèmes. Parmi celles-ci :

- C1 : La plupart des nœuds SCADA sont des dispositifs informatiques à usage spécial avec une puissance, une mémoire et une taille limitées.

- C2 : La communication et le traitement sont généralement en temps-réel-dur (hard real-time), ce qui signifie que l'information devrait non seulement arriver à son destinataire dans un temps limité, mais aussi dans le bon ordre. Autrement dit, toutes les informations doivent arriver au bon moment, ce qui est une propriété essentielle dans les systèmes critiques.
- C3 : Les systèmes SCADA peuvent avoir besoin de travailler pendant de longues périodes.
- C4 : Les nœuds peuvent être géographiquement répartis.
- C5 : Certaines entités peuvent avoir besoin de travailler dans des conditions physiques difficiles, comme les usines chimiques.
- C6 : Les protocoles de communications sont divers selon les niveaux dans le SCADA. Ils peuvent être temps réel pour les niveaux 0 et 1 (par exemple MODBUS, CAN Bus [ISO-8802, 1998], Profibus [DIN-19245, 1991]. Ces bus de terrain sont, pour la plupart, standardisés. Pour les niveaux supérieurs, le protocole OPC-UA [IEC-62541], entre autre, est mis en œuvre auquel viennent cohabiter des protocoles tels que FTP ou HTTPS.
- C7 : SCADA utilise habituellement la technique de sûreté qui oblige le système à partir dans un état sûr en cas de détection de problèmes.

Les conséquences de ces spécificités sur la sécurité sont nombreuses, par exemple :

- C1 fait que les messages sont prévisibles, puisque la plupart de ces messages sont des signaux spécifiques.
- C1 et C2 rendent l'implémentation de la cryptographie très difficile.
- C3 et C4 posent une difficulté à mettre à jour/renouveler certains nœuds ou la sécurité de ces nœuds.
- Alors qu'aujourd'hui un SCADA peut avoir des connexions normales vers l'extérieur, C4 peut provoquer des liens inattendus vers l'extérieur.
- C4 et C5 rendent difficile de sécuriser les nœuds face aux effractions physiques, permettant les techniques des attaques par canaux auxiliaires.
- C6 offre aux attaquants plus de connaissances sur le système utilisé.
- C7 peut être utilisé malicieusement par un attaquant qui force le système à passer à un état sûr, puis l'attaquer dans cet état contrôlé et limité.

[Boy09] offre un aperçu sur les différentes techniques qui composent un SCADA. Il contient une partie dédiée aux problèmes de la sécurité des systèmes SCADA et leurs solutions. [Kru05] décrit les vulnérabilités des SCADA et les conséquences possibles en cas d'attaques, ainsi que des solutions pour sécuriser les systèmes SCADA.

[ILW06] décrit les principaux défis de la sécurité SCADA ainsi que des solutions techniques et des orientations de recherche prises par la communauté de sécurité. [ZJS11] souligne une taxonomie des cyberattaques sur les systèmes SCADA en caractérisant les

attaques selon la couche ciblée (matériel, logiciel, communication). [ZS10] montre une taxonomie des techniques de détection et prévention d'intrusions spécifiques aux SCADA. [FCM10] présente une taxonomie des solutions de sécurité applicables aux SCADA. [TLM08] propose un technique d'évaluation systématique des vulnérabilités des systèmes SCADA au niveau du système, des points d'accès et des scénarios.

[CDF+07] considère des techniques de détection d'intrusion basées sur les spécifications de modèle, ses caractéristiques et son comportement attendu. À ce propos, les auteurs décrivent trois techniques de détection et un prototypage d'implémentation sur un réseau qui utilise $ModbusTCP$ qui est un des protocoles les plus utilisés dans les systèmes SCADA. Enfin, [FLP10] démontre comment nous pouvons utiliser des patrons de sécurité pour créer un système SCADA sécurisé.

2.7 Patrons de Sécurité

Un patron de sécurité est une solution réutilisable qui s'est révélée efficace pour un problème de sécurité fréquent. Étant donné que tous les développeurs ne sont pas des experts en sécurité, un patron de sécurité offre aux développeurs un guide sur la façon d'utiliser et de mettre en œuvre une solution de sécurité en fonction du contexte précis d'un problème de sécurité.

Christopher Alexander était le premier à introduire le concept de patrons [AIS+77, Ale79]. L'application directe de ces patrons était du domaine de l'architecture, de conception urbaine et des habitations communautaires. Dix ans plus tard, Kent Beck et Ward Cunningham ont commencé à appliquer la méthodologie des patrons dans le domaine de la programmation orienté objet [BC87]. Les patrons de conception sont devenus populaires grâce à *The "Gang of Four"* (Gamma et al.) [Gam95]. Quelques années plus tard, il y a eu l'apparition des premiers patrons de sécurité.

Les patrons de sécurité sont souvent utilisés dans la phase de conception de modèle. Dans nos travaux, nous avons étudié plus spécifiquement certains types de patrons qui sont détaillés dans le chapitre 3. Mais la méthodologie, proposée dans cette étude, pourrait s'appliquer à d'autres types de patrons.

Un patron de sécurité est identifié en quatre étapes :

- E1 : Identification d'un problème de sécurité fortement récurrent, et de la meilleure solution pour résoudre ce problème.
- E2 : Modélisation du résultat de $E1$ dans un format spécifique (différents formats ont été proposé dans la littérature) qui regroupe les détails tels que : titre, contexte, solution, relations, utilisations connues, etc.
- E3 : Adapter le modèle proposé.
- E4 : Publication du modèle révisé au public : livre, Internet, article, etc.

L'utilisation d'un patron de sécurité est faite aussi en plusieurs étapes :

- A1 : Reconnaissance d'un problème de sécurité à résoudre.
- A2 : Sélection d'un patron de sécurité qui résout ce problème, s'il existe.
- A3 : Application (adaptation) du motif sélectionné au problème.

- A4 : Évaluation du résultat et mise à jour du patron publié en modifiant la section *Utilisations Connues*.

Les patrons de sécurité sont devenus populaires à la fin des annés 90. Ils ont fait l'objet de beaucoup de travaux décrits dans la littérature. On cite ici les plus intéressants pour nos travaux. Parmi les premières publications sur les patrons de sécurité, [YB98] décrit comment l'accès à une application peut être sécurisée en utilisant une combinaison des patrons de sécurité. Fernandez [Fer00, FP01] s'est focalisé sur les aspects autorisation d'accès à l'aide des patrons. Dans [WC03], les auteurs proposent, après une présentation des 10 principes de sécurité de Viega et McGraw, différents patrons de sécurité. Les auteurs décrivent les aspects structurels et comportementaux en utilisant le langage de modélisation UML. Cette publication décrit des contraintes formelles associées aux différents patrons dans le but de pouvoir les vérifier formellement.

La sélection d'un patron de sécurité dépend largement des exigences de sécurité souhaitées. Cependant, choisir un patron ou une combinaison des patrons, implique des compromis dans le modèle. [WM08] propose une approche pour sélectionner des patrons de sécurité en recherchant une combinaison correcte et satisfaisante des patrons. [FB13] étudie la structure et le but des patrons de sécurité, en utilisant UML pour décrire les patrons. L'auteur illustre l'utilisation des patrons avec des conseils détaillés de mise en œuvre. [SFBH+13] se focalise sur l'intégration des patrons de sécurité dans les systèmes. Le point fort de cette publication est de s'adresser, non seulement aux experts de sécurité, mais aussi à tous développeurs qui n'ont pas une grande expertise en sécurité. Les deux dernières publications livrent une explication détaillée sur les patrons de sécurité et catégorisent de nombreux patrons selon leurs objectifs abstraits (contrôle d'accès, authentification, sécurité de réseaux, etc.). [YWM08] présente une étude sur les patrons de sécurité, et propose une classification selon la phase de développement ciblée. [HAJ12] présente un langage de patron de sécurité unificateur et une classification de tous les patrons de sécurité publiés à l'époque. Enfin, [BS11] montre comment les modèles de sécurité peuvent être détectés à l'aide d'outils de *reverse engineering* (au niveau du code).

2.8 Discussion

Dans notre travail, nous cherchons à exploiter des techniques de validation formelle pour valider des modèles d'architectures qui sont dotées de mécanismes de protection de sécurité. Pour l'instant, nous avons considéré des modèles abstraits d'architectures à sécuriser. Mais nous pensons que nos travaux, à terme, doivent pouvoir être exploités dans le cadre de la modélisation des architectures SCADA. La littérature mentionne [FLP10] de nombreux patrons qui peuvent être intégrés dans les modèles des systèmes SCADA et ces différents travaux nous ont inspirés.

Le choix des patrons à intégrer dépend de la politique de sécurité choisie. Celle-ci, comme vu dans l'état de l'art, correspond à un ensemble de règles (ou propriétés) qui doivent être assurées dans des contextes d'attaques, qui forment les hypothèses d'utilisation des architectures. Les techniques de vérification formelle peuvent alors être exploitées pour s'assurer, dans les phases amont de conception des modèles d'architecture, que les exigences de sécurité sont respectées.

Notre objectif est de pouvoir générer, à partir d'une politique de sécurité donnée et d'un modèle d'architecture SCADA non sécurisée, une architecture sécurisée respectant la politique choisie. Cette génération est basée sur une bibliothèque de patrons de sécurité. Nous nous basons sur un ensemble de patrons de sécurité décrits dans la littérature. Dans

notre approche, le modèle généré est validé formellement par une technique de *model − checking*.

Par contre, dans nos travaux, nous ne nous intéressons pas à la qualité du modèle d'architecture SCADA. Nous supposons que ce modèle est correct et est le résultat d'une expertise de concepteurs avertis.

Dans la suite du document (chapitre 3), nous décrivons quatre patrons. Chaque patron est formalisé, tant pour sa structure que son comportement. Les propriétés formelles attachées à ce patrons sont décrites. Ensuite, nous décrivons au chapitre 4, nos choix d'implantation des mécanismes de sécurité, basés sur les patrons, au sein du modèle d'architecture. Nous composons alors les éléments de l'architecture initiale avec les éléments invoqués dans les patrons. Ceci permet d'aboutir à la génération d'une nouvelle architecture, basée sur des règles de composition. Finalement chapitre 5, nous illustrons cette intégration de patrons au sein des modèles d'architecture sur deux cas d'études simples. Nous montrons comment la vérification des propriétés fonctionnelles (nominales) et des propriétés de sécurité peut s'effectuer, par *model-checking*, sur les modèles générés.

CHAPTER 3

Patrons de Sécurité

3.1 Introduction

Ce chapitre correspond à la description de l'enjeu 1 décrit en section 1.2.

Un patron de sécurité [FP01, SFBH+13, HAJ12, WC03] est une solution réutilisable à un problème de sécurité récurrent. Il fournit des lignes directrices détaillées sur la façon de mettre en œuvre une solution architecturale pour un problème de sécurité spécifique. Les patrons doivent être considérés comme des outils méthodologiques pour décrire des solutions techniques liées à la sécurité dans un contexte métier en capturant les expertises métier. Ils imposent des décisions qui doivent être prises en considération lors de la conception des architectures. Ils facilitent ainsi la communication entre experts et non experts.

Dans la littérature, beaucoup du patrons ont été proposés [SFBH+13]. Associées aux patrons de sécurité, des auteurs ont proposés des méthodologies pour leur mise en œuvre et leur intégration au sein de modèles d'architectures logicielles durant les phases de conception. Les patrons encapsulent ainsi les aspects de sécurité à intégrer dans une architecture. La description des patrons inclut une description de propriétés qui précisent les exigences qui doivent être respectées par le patron une fois intégré dans une architecture.

Pour la description des patrons, différents formats ont été proposés dans la littérature [SFBH+13, HAJ12, WC03] Dans ce travail, nous choisissons le format simplifié suivant :

- *Name*: Nom du patron.
- *Functioning* : Le fonctionnement attendue du patron et les contraintes associées.
- *Structure* : Structure architecturale du patron. Nous choisissons de décrire la structure par un diagramme de classes (UML).
- *Behavior* : Comportement du patron. Le comportement du patron est décrit par un ou plusieurs diagrammes de séquences et éventuellement un automate qui précise le comportement et les fonctions s'exécutant au sein du patron.
- *Properties* : Propriétés de sécurité attendues du patron : Ces propriétés sont formalisées à l'aide d'une logique temporelle de type LTL.
- *Example* : Exemple illustrant l'utilisation du patron, avec des scénarios nominaux et des scénarios d'attaque.

Dans la description des patrons que nous présentons, dans la suite du document, nous identifions un fonctionnement type pour chacun. Les architectures cibles que nous considérons sont des architectures de type SCADA. Nous en présentons un modèle abstrait qui servira, par la suite et avec différentes versions, à nos expérimentations décrites au chapitre 5.

Nous considérons que l'ensemble des mesures de sécurité à implanter pour une architecture ne sont pas, en général, gérée par un seul dispositif. Dû aux spécificités des patrons, la responsabilité de gestion de l'ensemble de la politique de sécurité à implanter est partagée entre plusieurs dispositifs basées sur différents patrons.

Mais nous pouvons résumer ici l'essentiel qu'apporte chaque patron qui sont détaillés dans la suite du document.

- Le patron $SingleAccessPoint(SAP)$ unifie les ports d'accès pour effectuer des *vérifications* sur un seul point d'accès.
- Le patron $CheckPoint(CHP)$ est dédié à réaliser certaines *vérifications* en se basant sur une *politique* de sécurité, et en appliquant des mesures dans les cas où la politique n'est pas respectée.
- Le patron $Authorization(AUTH)$ implante des mesures de sécurité consacrées aux droits d'accès à des ressources. Celles-ci sont vues, par exemple ici, comme des données, une fonction ou un traitement à exécuter.
- Le patron $FireWall$ implante des mesures de sécurité consacrées aux filtrages des messages et à l'application de restrictions sur des messages sortants et entrants dans les composants de l'architecture.

Nous considérons deux objectifs principaux en terme de sécurité nécessitant l'application de patrons sur une architecture. Un premier objectif (Objectif 1) qui concerne les accès à des ressources détenues par un composant. Un deuxième (Objectif 2) qui concerne les accès à un ensemble de composants. Pour chaque objectif, la responsabilité de mécanismes basés sur les patrons diffère.

- Objectif 1 : Sécurisation des accès aux ressources d'un composant :
  - Le mécanisme de type SAP limite les points d'accès aux ressources d'un composant en un seul point d'accès, et vérifie la disponibilité des ressources.
  - Le mécanisme de type CHP, sollicité par SAP, exécute des contrôles complémentaires liés à la politique de sécurité souhaitée.
  - Le mécanisme de type $AUTH$, sollicité par CHP est responsable du respect des droits d'accès à une ressource d'un composant en lecture/écriture.
- Objectif 2 : Sécurisation des accès à un ensemble de composants :
  - Le mécanisme de type SAP limite les points d'accès à un ensemble de composants en un seul point, et vérifie la disponibilité des entités destinataires des requêtes.
  - Le mécanisme de type CHP est appelé par le SAP pour faire d'autres vérifications liées à la politique de sécurité.
  - Le mécanisme de type $FireWall$ précise si tel requête peut transiter ou non en se basant sur les restrictions et les exceptions.

La section 3.2 introduit les notations formelles nécessaires à la description des patrons. Ensuite, les patrons SAP, CHP, $AUTH$ et $FireWall$ sont décrits dans la suite de ce chapitre.

3.2 Notations pour la formalisation

Cette section introduit des notations en vue de formaliser la description des patrons de sécurité. Des notations élémentaires, utiles dans la suite du document, sont présentées en 3.2.2. Puis d'autres notations sont décrites en 3.2.3 pour la formalisation des propriétés, basée sur la logique temporelle linéaire de type LTL.

3.2.1 Architecture Abstraite

Nous supposons, dans notre approche, que les éléments d'architecture pris en compte dans notre processus (figure 1.1) respectent le méta modèle d'architecture type illustré figure 3.1. Nous considérons ici qu'une architecture est composée d'entités clients,$\mathcal{C}lt$ appartenant à un environnement, et des entités composants $\mathcal{C}omp$.

Les clients envoient des requêtes vers les composants de l'architecture. Les composants reçoivent les requêtes, accèdent à des ressources de l'architecture (mémoire, fichier, ...), transfèrent les requêtes à d'autres composants ou répondent aux clients. Les entités communiquent via des canaux de communications. Chaque canal peut contenir des messages et est géré en mode $FIFO$.

Dans notre approche, ce sont uniquement les composants qui sont impactés par l'intégration des patrons de sécurité.

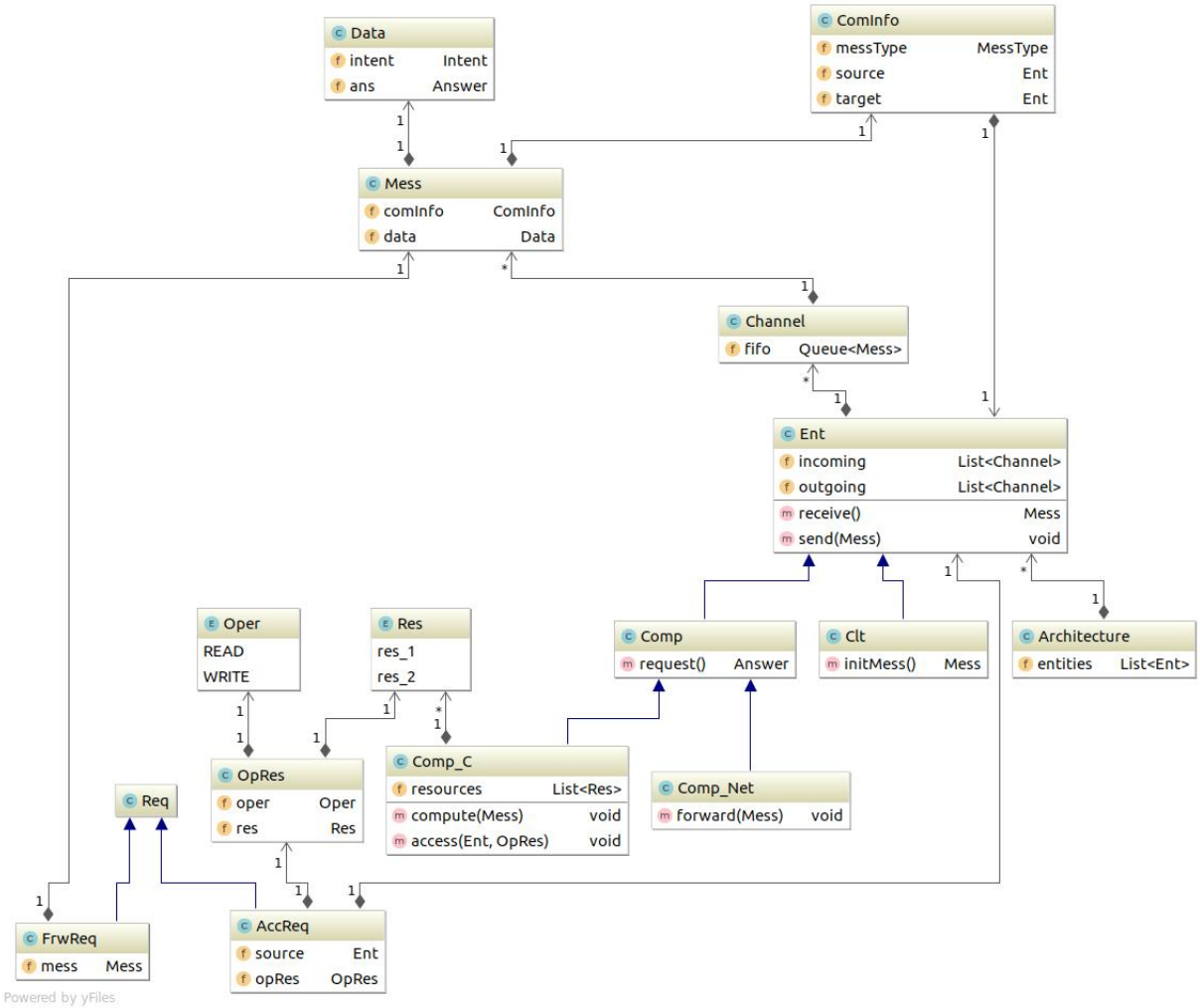

Figure 3.1: Méta-modèle d'une architecture type.

3.2.2 Notations élémentaires

3.2.2.1 Notations pour la description des architectures

- Une architecture $\mathcal{A}rchitecture$, illustrée figure 3.1, est un ensemble $\mathcal{E}nt$ d'entités qui sont de deux types : des composants et des clients. L'ensemble des composants est noté $\mathcal{C}omp$ [1]. L'ensemble des clients est noté $\mathcal{C}lt$ ($\mathcal{E}nt = \mathcal{C}omp \bigcup \mathcal{C}lt$).

  Les composants communiquent entre eux et avec l'environnement (les clients) par des messages $\mathcal{M}ess$ via des canaux de communication $\mathcal{C}hannel$. Les clients de l'ensemble $\mathcal{C}lt$ initialise des messages $initMess()$, et les envoi vers les composants. L'ensemble $\mathcal{C}lt$ simule donc un environnement qui interagit avec l'ensemble $\mathcal{C}omp$. L'ensemble $\mathcal{C}omp$ des composants représente la partie sécurisable de l'architecture $\mathcal{A}rchitecture$.

  Un composant est un composant dit *d'accès* ou un composant dit *de communication*.

  - Les composants d'accès ($\mathcal{C}omp\_C$) sont des composants qui contiennent chacun un ensemble de ressources. Nous notons $\mathcal{R}es_c$ l'ensemble de ressource de composant c ($Res_c \subset \mathcal{R}es$).
  - Les composants de communications ($\mathcal{C}omp\_Net$) ne contiennent pas de ressources.

3.2.2.2 Notations pour la communication entre entités

- Une entité $e \in \mathcal{E}nt$ détient un ensemble de canaux de communications entrants ($incoming$) et un ensemble de canaux de communications sortants ($outgoing$).
- Nous notons $\mathcal{C}hannel$ l'ensemble des canaux de communication. Un canal est une $FIFO$ contenant des messages appartenant à l'ensemble des messages considérés pour l'architecture ($m \in \mathcal{M}ess$).
- Un message $m \in \mathcal{M}ess$ est composé de :

  - $ComInfo$: Information sur la communication contenant la source $source$, la destination $target$, et le type de message $type$. Nous considérons cette partie de message d'être intègre (source authentique). Nous considérons aussi que ce champ n'est pas confidentiel (tout entité peut le lire).
  - $Data$: Le cœur d'un message, contenant l'objet de ce message noté $intent$. Par exemple, cela peut signifier une commande qui doit être exécutée. La partie ans peut contenir ACK ou NAK pour confirmer que la commande a bien été exécutée.

L'opération d'envoi d'un message m par une entité e est notée $send$ ($e : \mathcal{E}nt$, $m : \mathcal{M}ess$)[2]. L'opération de réception d'un message m par une entité e se note $receive$ ($e : \mathcal{E}nt$, $m : \mathcal{M}ess$).

A la réception d'un message, un composant communication peut transférer ce message. Pour cela, ce composant exécute une requête de transfert de la forme $request$ ($req : FrwReq$). $FrwReq$ contient simplement des informations relatives au message à transférer pour savoir si ce transfert est possible.

[1] Pour simplifier, nous utilisons le même nom pour dénommer un ensemble d'instances et la classe associée du méta modèle. Par exemple, $\mathcal{C}omp$ dénote l'ensemble des composants de type $Comp$ du modèle. Un composant c appartient à l'ensemble $\mathcal{C}omp$ est une instance de la classe $Comp$.

[2] e peut être ici différent de $m.comInfo.source$ en cas de diffusion d'un message via des composants intermédiaires.

Un composant d'accès contient la fonction *compute*() qui décrit le traitement exécuté par ce composant. Dans nos travaux, nous intéressons à la partie du composant nécessitant des accès à ses ressources. Un traitement peut faire objet de plusieurs demande d'accès. Chaque demande d'accès est donc fait à partir d'une requête d'accès de la forme *request* ($req : AccReq$) $AccReq$ contient des informations sur l'entité qui demande l'accès (la source du message) et les informations sur l'opération et la ressource concernée (OpRes référence les deux informations).

Notons que, toutes les instances d'un composant de même type ont le même traitement. Pour définir des traitements différents pour les composants, des classes héritant de $Comp\_C$ sont nécessaires.

3.2.2.3 Exemple de communication entre entités

La figure 3.2 illustre un exemple de communication entre une entité e (client ou composant) et un composant c. L'entité e envoie (*send*) une requête req au composant c. Le composant reçoit (*receive*) la requête et répond à la requête en exécutant la commande, puis renvoie une réponse $resp$ à l'entité e avec la valeur ans égale à ACK ou NAK.

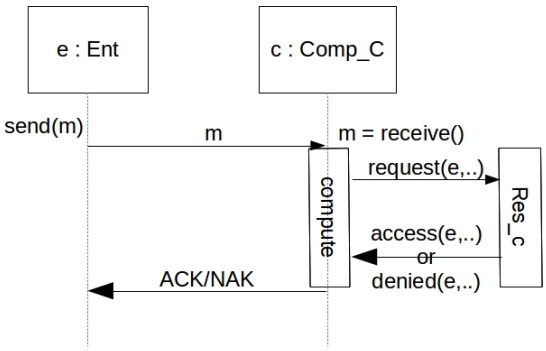

Figure 3.2: Exemple d'une communication entre une entité et un composant d'accès.

Le composant c exécute la commande en faisant appel à plusieurs opérations. Nous nous intéressons uniquement aux opérations d'accès.

La figure 3.3 donne un exemple sur la manière d'extraire les opérations d'accès. Soit un exemple d'un composant d'accès qui spécifie la fonction $Compute$ pour répondre à des commandes (vue $Original$). Ce comportement est modélisé de la façon suivante (vue $Model$). Au niveau modèle, nous ne nous intéressons pas aux valeurs des variables, ou aux résultats des traitements. Dans le cas illustré figure 3.3, si un composant de type $Sensor\_T1$, de type d'accès, reçoit un message, lui étant destiné, et contenant la commande $COMMAND\_1$, tous les accès, liés à cette commande, sont vérifiés. Par exemple, $READ(x)$ provoque la requête $request(req)$ avec $req = (comInfo.source,\ opRes)$ et $opRes = (READ, x)$. La réponse à cette requête est $access(req)$ ou $denied(req)$. Dans le cas de $denied(req)$, le reste des accès n'est pas contrôlé, une réponse négative peut être envoyée selon les spécifications. Si toutes les requêtes ont été permises, une réponse positive est envoyée.

La figure 3.4 illustre la comparaison entre la communication décrite dans la vue $Original$ avec la modélisation, vue $Model$. Nous remarquons que quelque soit le résultat du traitement de la vue $Original$, les réponses envoyées, dans la vue $Model$, aux clients contiennent

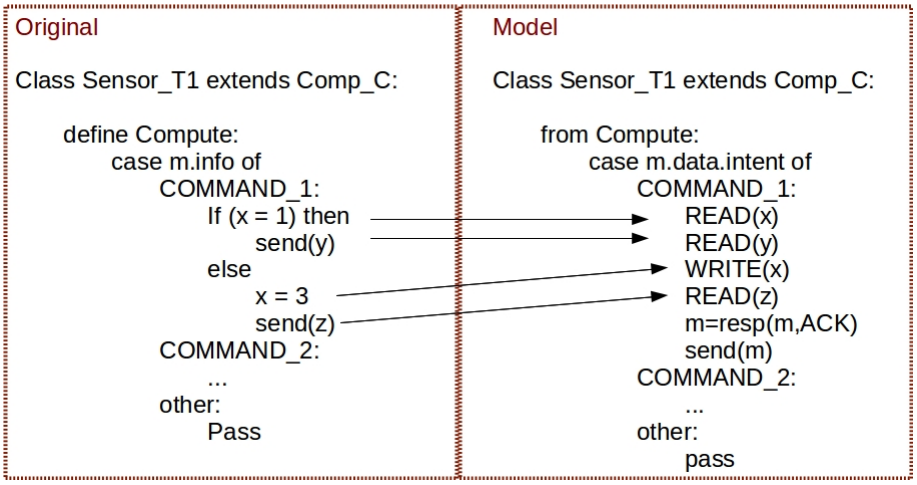

Figure 3.3: Exemple d'abstraction de composant.

toujours ACK ou NAK.

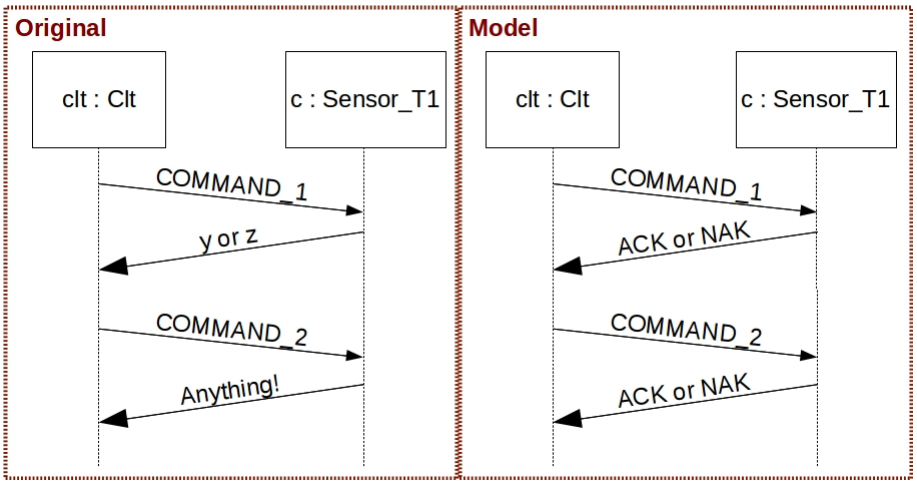

Figure 3.4: Illustration des ommunications dans les vues *original* et $Model$.

3.2.3 Formalisation des propriétés

L'intégration de patrons de sécurité dans une architecture a pour conséquence d'ajouter des mécanismes architecturaux liés au besoin de sécurité décrit dans la politique de sécurité à implanter. Ces mécanismes sont associés à des exigences qui se rajoutent aux exigences de l'architecture initiale, c'est à dire l'architecture qui accueillent ces mécanismes. Ces exigences peuvent être formalisées par des propriétés formelles dont l'expression est basée en général sur des logiques temporelles.

Dans notre approche, nous nous intéressons à des propriétés de deux ordres :

- les propriétés initiales d'une architecture.

- les propriétés ajoutées suite à l'intégration des patrons dans l'architecture. Ces propriétés correspondent aux exigences fonctionnelles du patron et à ses contraintes d'implantation.

Lors du processus d'intégration des patrons, il est nécessaire de valider la nouvelle architecture par une vérification des comportements du modèle de l'architecture. L'ensemble des propriétés formelles est alors vérifié sur le comportement du modèle généré. Dans notre approche, ces vérifications s'effectuent par une technique de $model - checking$. Pour la formalisation des propriétés, nous utilisons des notations formelles. Nous rappelons que $evt\_send(e\_1, m)$ est l'événement d'envoi du message m par e_1 qui exécute la fonction $send(m)$. $pre\_sent(e\_1, m)$ est un prédicat qui est vrai dès l'occurrence de $evt\_send(e_1, m)$.

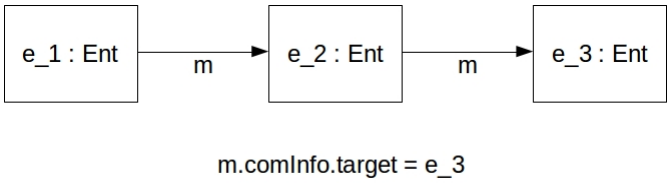

Figure 3.5: Exemple d'une communication via une entité intermédiaire.

Considérons le cas de communication dans la figure 3.5. Si nous souhaitons définir une propriété qui spécifie que l'envoi du message m par l'entité e_1 sera suivi fatalement par sa réception par l'entité e_2, elle s'exprime de la manière suivante:

$$\Box\ [evt\_send\ (e_1,\ m)\ \Rightarrow\ \Diamond\ evt\_receive\ (e_2,\ m)] \tag{3.1}$$

Nous pouvons utiliser le joker ANY pour exprimer des ensembles d'entités, de messages, d'opérations, etc. Par exemple, l'envoi d'un message m, quelque soit son émetteur, est spécifié par l'événement $evt\_send\ (ANY,\ m)$. La réception de m, quelque soit son récepteur, est noté $evt\_receive\ (ANY,\ m)$.

La propriété suivante :

$$\Box\ [evt\_send\ (ANY,\ m)\ \Rightarrow\ \Diamond\ evt\_receive\ (ANY,\ m)] \tag{3.2}$$

exprime que l'envoi de m, quelque soit son émetteur, sera suivi fatalement par sa réception, quelque soit son récepteur. Dans l'exemple de la figure 3.5, nous supposons que le destinataire final de m est e_3, c'est-à-dire : $m.comInfo.target = e_3$ et $m.comInfo.source = e_1$. Nous remarquons que la propriété 3.2 est satisfaite, même si le message m n'arrive pas à sa destination finale e_3. Si nous voulons exprimer que le message doit arriver à sa destination finale, nous devons exprimer la propriété 3.3 :

$$\Box\ [evt\_send\ (ANY,\ m)\ \Rightarrow\ \Diamond\ evt\_receive\ (m.comInfo.target,\ m)] \tag{3.3}$$

Dans notre exemple, la propriété 3.3 est satisfaite seulement si $m.comInfo.target$, donc e_3, reçoit m.

Finalement, la propriété 3.4 exprime que si l'entité e_3 reçoit le message m, alors m a déjà été émis par e_2. Lors des vérifications, deux événements d'envoi ou de réception, même s'ils impliquent le même message m, sont vus comme des événements différents.

$$\Box\ [evt\_receive\ (e\_3,\ m)\ \Rightarrow\ \Diamond\ pre\_sent\ (e\_2,\ m)] \tag{3.4}$$

3.3 Single Access Point (SAP)

Le patron *Single Access Point* (SAP) a été introduit par [YB97]. C'est un patron qui peut être intégré dans différents types d'implantation, au niveau d'un système, d'une application, d'un serveur, etc. L'objectif du patron SAP est d'implanter un point d'accès unique pour toutes les communications provenant de l'extérieur de la partie à sécuriser afin d'améliorer le contrôle et la surveillance des entrées.

Nous notons qu'un accès à un composant ou un ensemble de composants est *contrôlé* si les données contenues dans le message (ou requête) provoquant l'accès au composant sont analysées (ou testées) au regard de la politique de sécurité appliquée à ce composant. Différents types de contrôle peuvent être effectués par un SAP[3] :

- Dans le cas de demande d'accès à une ressource res détenu par un composant c, le contrôle vérifie si res est disponible.
- Dans le cas de demande de relayage à un composant c appartenant à un ensemble de composants, le contrôle vérifie si c est disponible dans cet ensemble.

De plus, une signature est apposée par SAP dans un champ des requêtes. Diverses implantations peuvent être envisagées pour les signatures (en clair, chiffrées) et l'accumulation des signatures le long d'une chaîne de transmission de requêtes dans un réseau de composants d'une architecture.

L'application de ce patron empêche donc les entités externes à une architecture de communiquer directement avec les composants de cette architecture. Toute requête entrante est acheminée via un canal unique, où la surveillance peut être effectuée facilement. Le point d'accès unique est un endroit approprié pour éventuellement capturer un journal de l'historique (*Log*) des accès. Ces données peuvent être utiles pour vérifier le séquencement des accès en fonction de leurs droits.

Ce patron peut être utilisé à différents niveaux d'une architecture : A l'entrée de l'architecture considérée ou au niveau d'un composant de cette architecture.

3.3.1 Fonctionnement

Le patron SAP donne un cadre général pour appliquer les fonctionnement de contrôle des accès à un composant ou un ensemble des composants. Nous identifions deux types de SAP :

- Un SAP qui unifie les points d'accès à un ensemble de composants. Nous le nommons SAP_NET. Le mécanisme de type SAP_NET est implanté dans un composant spécifique qui implante la fonctionnalité du patron. L'ensemble protégé peut contenir

[3] Dans le reste du document, nous utilisons l'expression "SAP" qui signifie "mécanisme de type SAP".

d'autres composants de type SAP_NET. Nous notons $\mathcal{S}ap\_Net$ l'ensemble de tous les composants de communications de type SAP_NET ($\mathcal{S}ap\_Net \subset \mathcal{C}omp\_Net$).

- Un SAP qui unifie les points d'accès aux ressources d'un seul composant. Nous le nommons SAP_C. Un mécanisme de type SAP_C est intégré dans un composant qui détient des ressources. Dans ce cas, toute requête à destination d'un composant protégé est prise en charge par ce mécanisme avant l'accès aux ressources. Nous notons $\mathcal{S}ap\_C$ l'ensemble des composants de type SAP_C ($\mathcal{S}ap\_C \subset \mathcal{C}omp\_C$).

La figure 3.6 illustre ces deux types de SAP. Soient les entités $e\_1, \ldots e\_n$ clientes externes à l'ensemble $\{c\_2, c\_3\}$. c_1 est un composant de communication contenant le mécanisme SAP_NET protégeant l'ensemble $\{c\_2, c\_3\}$. Donc, tout message transitant entre des entités externes à cet ensemble et des entités appartenant à cet ensemble doit passer par le composant c_1 et être contrôlé par son mécanisme SAP_NET. Ceci est le cas des messages entrants comme sortants. Un message transitant entre deux composants dans l'ensemble $\{c\_2, c\_3\}$, ne passe pas par c_1 car ils n'ont pas besoin d'être contrôlés. De plus, tout message entrant, passant par c_1, doit être signé, pour assurer aux composants internes que ce message est passé par c_1.

Tous les composants protégés par un composant comportant des mécanismes de sécurité tel que SAP_NET doit pouvoir vérifier la signature d'un message. Une telle vérification peut faire objet d'un patron dédié à la gestion des signatures. Dans nos travaux nous ne nous intéressons pas à de tels patrons. Nous considérons que la vérifications d'une signature est faite d'une façon simple à la réception d'un message.

De même, c_2 contient le mécanisme SAP_C pour controler l'accès à ses ressources $\{r\_1, \ldots r\_m\}$. Toute demande d'accès à des ressources dans c_2 doit passer par le mécanisme SAP_C de c_2.

Nous décrirons plus tard dans le document la manière dont nous implantons les concepts de SAP_NET et SAP_C. Pour simplifier ici l'illustration, nous indiquons les liens de communications directement entre les SAP.

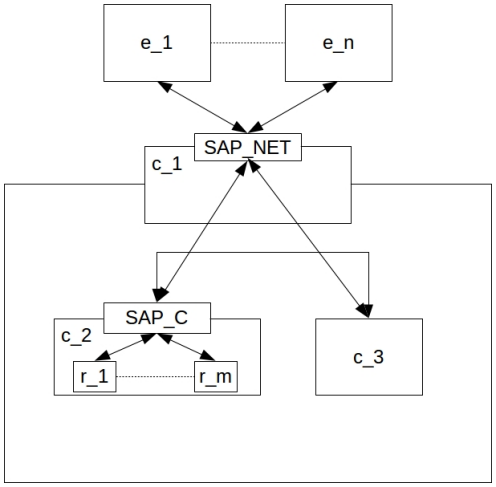

Figure 3.6: Fonctionnement du patron SAP.

Les fonctionnalités du SAP sont définis comme suit :

- Fonctionnalité 1 : Si le SAP reçoit un message, ce message est *contrôlé* et une signature est apposée.

  - Cas de SAP_NET : contrôle les demandes de transfert des messages au niveau de l'accessibilité de leur destinataire, avant de les relayer ou d'appeler le mécanisme CHP si celui-ci est présent.
  - Cas de SAP_C : contrôle les demandes d'accès au niveau de l'accessibilité de la ressource demandée, avant autoriser cet accès ou d'appeler le mécanisme CHP si celui-ci est présent.

- Fonctionnalité 2 : Dans le cas d'un contrôle infructueux, une réponse négative est renvoyée.

  - Cas de SAP_NET : la réponse négative est envoyée à l'entité source du message, faisant l'objet d'une demande de transfert, si le composant destinataire n'est pas accessible.
  - Cas de SAP_C : la réponse négative est envoyée à l'entité source de la demande d'accès, si la ressource concernée par cet accès n'est pas accessible.

3.3.2 Structure

La figure 3.7 décrit le modèle de patron SAP.

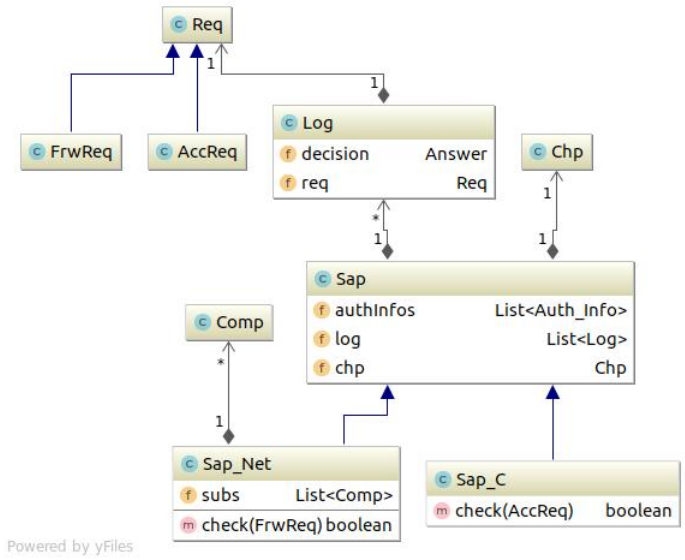

Figure 3.7: Structure du patron SAP.

- Un SAP référence, de manière générique :

  - $authInfos$: Des informations permettant au SAP de prendre des décisions.
  - log : Un historique des décisions prises pour les requêtes traitées.
  - chp : Un patron de type CHP pour appliquer des contrôles additionnels liés à la politique de sécurité.

- Un SAP de type SAP_C est intégré dans un composant c et applique les fonctionnalités de type SAP pour contrôler les accès aux ressources.

  - La fonction *check* ($accReq$) : vérifie si la ressource demandée $accReq.opRes.res$ est accessible pour l'opération $opRes.oper$.

- Un SAP de type SAP_NET est intégré dans un composant c et applique les fonctionnalités de type SAP pour contrôler le transfert des messages.

  - $subs$: ensemble des composants protégés par ce SAP. $c \in c_s.subs$ avec $c_s \in Sap\_Net$, signifie que c est un composant protégé par c_s.
  - La fonction $check(frwReq)$: SAP vérifie si le destinataire frwReq.mess.comInfo.target est accessible.

- Log : donnée relative à une décision prise par rapport à une requête req. $decision$ est de type $Answer$ avec deux valeurs possibles : ACK et NAK.

Nous imposons une règle concernant l'inclusion d'ensemble de composants protégés par un SAP_NET comme illustrée figure 3.8. Soient deux composants sap_net1 et sap_net2 $\in$ $\mathcal{S}ap\_Net$ avec $\{c1, c2\} \in sap\_net1.subs$, $\{c3, c4\} \in sap\_net2.subs$ et $sap\_net2 \in sap\_net1.subs$.

Alors $\forall$ c $\in$ $sap\_net2.subs$, c $\in$ $sap\_net1.subs$. Ceci a pour conséquence que les règles de protection implantées dans sap_net1 s'appliquent non seulement à $\{c1, c2\}$ mais également à $\{c3, c4\}$.

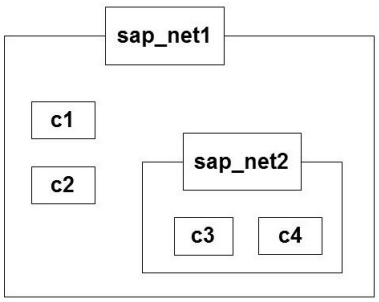

Figure 3.8: Inclusion de SAP_NET.

3.3.3 Comportement

Le diagramme de séquence, figure 3.9, illustre le cas suivant : Soit un composant c, muni d'un mécanisme de type SAP_C. Considérons le fonctionnement suivant : c doit traiter deux requêtes req_1 et req_2 émises par l'entité e. La première requête req_1 déclenche le contrôle par le SAP_C, et supposons que la ressource demandée par la requête est accessible. L'accès peut donc s'exécuter et le comportement de c continue en séquence. Pour la deuxième requête req_2, supposons que le SAP_C en déduit que le destinataire de la requête ($req\_2.opRes.res$) est inaccessible. SAP_C répond directement à la source de la requête ($req\_2.source = e$), avec une réponse négative (NAK). Dans ce cas, le traitement se termine.

Le diagramme de séquence, figure 3.10, illustre le cas suivant : Soit un composant c, muni d'un mécanisme de type SAP_NET. c doit exécuter deux requêtes de transfert req_1 et req_2 émises par l'entité e_2. La première requête req_1 tel que $req\_1.mess = m$. Cette

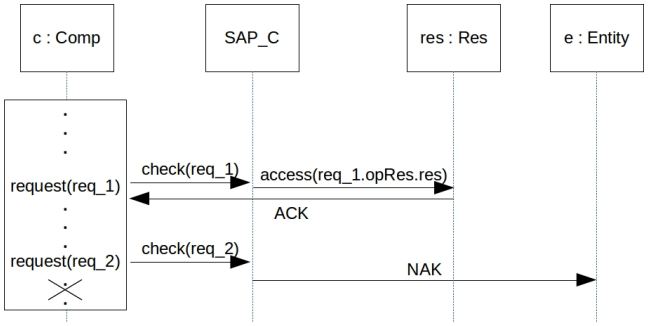

Figure 3.9: Comportement du patron SAP_C.

requête est contrôlée par le SAP_NET, et supposons que le destinataire de la requête est accessible. Le message est transféré. Notons ici que e_1 n'est pas nécessairement le destinataire de message dans la requête, e_1 peut être une autre entité intermédiaire. La deuxième requête req_2 est contrôlée par le SAP_NET. Supposons que le contrôle indique que le destinataire n'est pas accessible. SAP_NET envoit une réponse négative à la source du message ($req\_2.mess.comInfo.source = e\_2$) ayant causé la requête.

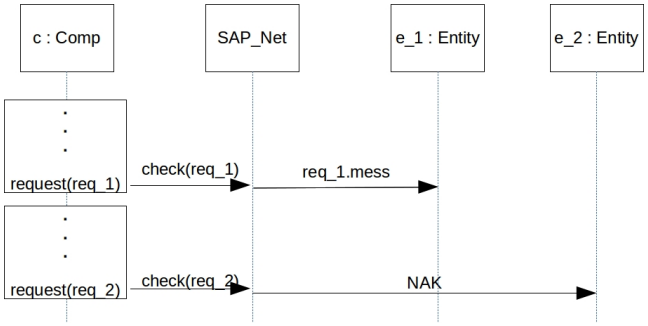

Figure 3.10: Comportement du patron SAP_NET.

3.3.4 Propriétés

Les exigences associées à un patron de type SAP sont reprises ici d'une classification proposée dans [WC03].

- Authenticité : une requête qui est directement envoyée à un composant interne au système, provient soit d'un autre composant interne, soit du composant SAP (P1).
- Disponibilité : si le destinataire d'une requête, provenant de l'extérieur du système, est indisponible, un message d'erreur est retourné à l'émetteur de la requête (P2).

  Si une entité externe envoie plus de requêtes au composant SAP qu'il ne peut en gérer en un temps donné, leur envoi peut être retardé pour maintenir la disponibilité de service pour d'autres entités (P3).
- Confidentialité : La communication entre les composants internes n'est pas divulguée aux entités extérieures (P4).

- Intégrité : Les requêtes internes au système ou provenant du composant SAP ne peuvent pas être modifiées par des entités externes (P5).
- Existence interne : Si le destinataire interne à une requête externe est connu, le composant SAP lui expédie cette requête (P6).
- Absence interne : Si le destinataire interne à une requête externe est inconnu, un message d'erreur est retourné à l'entité externe appelante (P7).

Nous notons ici, que les propriétés de disponibilité P6, pour le cas du SAP_NET (3.9) ou du SAP_C (3.14), ne précisent pas si l'accès aux ressources sont autorisés, au sens de la politique de sécurité souhaitée. Elles expriment uniquement les cas où les ressources sont accessibles. D'autres propriétés seront exprimées, lors de l'intégration d'autres patrons (patrons *Auth* et *Firewall*) pour vérifier les conditions d'accès.

Pour formaliser ces propriétés, nous complétons les notations formelles élémentaires, vues au paragraphe 3.2, par les définitions et prédicats qui suivent. Nous rappelons que $c_s \in \mathcal{S}ap\_C$ signifie que c_s est un composant d'accès implantant un SAP de type SAP_C. Aussi, $c_s \in \mathcal{S}ap\_Net$ signifie que c_s est un composant de communication implantant un SAP de type SAP_NET. $c \in sap.subs$ signifie que c appartient à l'ensemble protégé par c_s.

- **Accessibilité d'un composant :**

  Nous considérons que le destinataire est toujours accessible d'un point de vue d'un Sap_Net s'il est inclus dans l'ensemble des composants protégés par ce Sap_Net ou si la source de message est inclus dans l'ensemble protégé.

  $\forall\ c_s \in \mathcal{S}ap\_Net,\ c \in c_s.subs \Rightarrow c$ est un composant protégé par c_s.

- **Accessibilité des ressources :**

  Nous considérons qu'une ressource est toujours accessible d'un point de vue du composant si le composant inclut cette ressource.

  $\forall\ c_s \in \mathcal{S}ap\_C,\ res \in c_s.resources \Rightarrow res$ est une ressource détenue par c_s. Nous rappelons que $\mathcal{R}es_c$ est l'ensemble de toutes les ressources de c.

3.3.4.1 Propriétés générales d'un SAP

- Disponibilité (P3) : L'application de SAP ne bloque aucun message. Tout message envoyé à un composant $c \in Comp$, est fatalement reçu, soit par son destinataire, soit par un Sap_Net si celui-ci est localisé entre la source et le destinataire. Il y a toujours une possibilité que le message n'arrive pas à sa destination vu qu'un composant de type Sap_Net peut ne pas laisser passer ce message.

  Nous formalisons la propriété prt_sap_1 (3.5) comme suit :

prt_sap_1 :

$$\begin{aligned}&\forall\ m \in \mathcal{M}ess,\\ &\Box\ [evt\_send\ (ANY,\ m)\ \Rightarrow\ \Diamond\ (evt\_receive\ (m.comInfo.target,\ m)\ \vee\\ &(evt\_receive\ (c_s,\ m)\ \wedge\ c_s \in Sap\_Net\ \wedge\\ &(m.comInfo.target \in c_s.subs\ \vee\ m.comInfo.source \in c_s.subs)))]\end{aligned} \tag{3.5}$$

3.3.4.2 Propriétés d'un SAP_NET

- Authenticité (P1) :

  – Tout message, provenant de l'extérieur d'un ensemble de composants protégés par un Sap_Net, doit être contrôlé par son protecteur avant d'être transmis aux composants internes à l'ensemble.

  Nous formalisons la propriété $prt\_sap\_net\_1.a$ (3.6) comme suit :

$$\begin{array}{l}\textbf{prt\_sap\_net\_1.a :}\\ \forall\, m \in \mathcal{M}ess,\ \forall\, c_s \in \mathcal{S}ap\_Net,\ \forall\, c \in c_s.subs,\\ \Box\,[pre\_receive\ (c,\ m)\ \wedge\ m.comInfo.source \notin c_s.subs\ \Rightarrow\\ pre\_check\ (c_s,\ FrwReq(m))]\end{array} \tag{3.6}$$

  – Tout message qui est transmis à un composant interne à un ensemble de composants protégés par un Sap_Net, et sans passer par le SAP_NET, provient d'un composant interne à l'ensemble.

  Nous formalisons la propriété $prt\_sap\_net\_1.b$ (3.7) comme suit :

$$\begin{array}{l}\textbf{prt\_sap\_net\_1.b :}\\ \forall\, m \in \mathcal{M}ess,\ \forall\, c_s \in \mathcal{S}ap\_Net,\ \forall\, c \in c_s.subs,\\ \Box\,[evt\_receive\ (c,\ m)\ \wedge\ \neg\, pre\_check\ (c_s,\ FrwReq(m))\ \Rightarrow\\ m.comInfo.source \in c_s.subs]\end{array} \tag{3.7}$$

- Disponibilité (P2, P7) : Tout requête de transfert de message dans un Sap_Net, et ayant comme destination un composant non accessible, doit produire une réponse négative.

  Nous formalisons la propriété $prt\_sap\_net\_2$ (3.8) comme suit :

$$\begin{array}{l}\textbf{prt\_sap\_net\_2 :}\\ \forall\, req \in \mathcal{F}rwReq,\ \forall\, c_s \in \mathcal{S}ap\_Net,\\ \Box\,[evt\_request\ (c_s,\ req)\ \wedge\\ req.mess.comInfo.source \notin c_s.subs\ \wedge req.mess.comInfo.target \notin c_s.subs\ \Rightarrow\\ \Diamond\ evt\_send\ (sap,\ neg\ (sap,\ req.mess))]\end{array} \tag{3.8}$$

  Nous notons $neg(c, m)$ la réponse négative à m de la part du composant c. Remarquons ici que, si la source est protégée par un Sap_Net, le destinataire est supposé accessible parce qu'il peut etre un composant à l'extérieur de ce Sap_Net.

- Disponibilité (P6) : Tout requête de transfert de message par un SAP_NET, doit être contrôlée.

Nous formalisons la propriété $prt\_sap\_net\_3$ (3.9) comme suit :

$$\begin{aligned}&\textbf{prt\_sap\_net\_3 :}\\&\forall\ req\ \in \mathcal{F}rwReq,\ \forall\ c_s\ \in \mathcal{S}ap\_Net,\\&\Box\ [evt\_request\ (c_s,\ req)\ \Rightarrow\\&\Diamond\ evt\_check\ (c_s,\ req)]\end{aligned} \tag{3.9}$$

- Confidentialité (P4) :

  La notion de confidentialité implique qu'une communication entre deux composants internes à un ensemble protégé par un SAP_NET ne doit pas être divulguée à l'extérieur de cet ensemble. Tout message échangé en interne de l'ensemble ne doit pas être vu de l'extérieur.

  Nous formalisons la propriété $prt\_sap\_net\_4$ (3.10) comme suit :

$$\begin{aligned}&\textbf{prt\_sap\_net\_4 :}\\&\forall\ m \in \mathcal{M}ess,\ \forall\ e \in \mathcal{E}nt,\ \forall\ c_s \in \mathcal{S}ap\_Net,\\&\Box\ [evt\_receive\ (e, m)\ \wedge\\&(m.comInfo.source \in c_s.subs \wedge m.comInfo.target \in c_s.subs) \Rightarrow\ e \in c_s.subs]\end{aligned} \tag{3.10}$$

- Intégrité (P5) :

  La notion d'intégrité implique que les messages provenant de composants internes ou d'un SAP_NET ne peuvent pas être modifiés par une entité externe.

  Nous formalisons la propriété $prt\_sap\_net\_5$ (3.11) comme suit :

$$\begin{aligned}&\textbf{prt\_sap\_net\_5 :}\\&\forall\ m \in \mathcal{M}ess,\ \forall\ e \in \mathcal{E}nt,\ \forall\ c_s \in \mathcal{S}ap\_Net,\\&\Box\ [evt\_send(e, m)\ \wedge\\&(m.comInfo.source \in c_s.subs \wedge m.comInfo.target \in c_s.subs) \Rightarrow\ e \in c_s.subs]\end{aligned} \tag{3.11}$$

3.3.4.3 Propriétés d'un SAP_C

- Authenticité (P1) :

  – Tout accès à un composant de type Sap_C est contrôlé auparavant.
  Nous formalisons la propriété $prt\_sap\_c\_1$ (3.12) comme suit :

$$\begin{aligned}&\textbf{prt\_sap\_c\_1 :}\\&\forall\ c\ \in\ \mathcal{S}ap\_C,\ \forall\ e\ \in\ \mathcal{E}nt,\ \forall\ opRes\ \in\ \mathcal{O}pRes,\\&\Box\ [evt\_access\ (c,\ e,\ opRes)\ \Rightarrow\\&pre\_check\ (c,\ AccReq\ (e,\ opRes))]\end{aligned} \tag{3.12}$$

- Disponibilité (P2, P7) : Lors d'une demande d'accès à un composant c de type Sap_C, si la ressource n'est pas accessible, un message d'erreur (NAK) est renvoyé par le composant vers l'entité demandeuse.

  Nous formalisons la propriété $prt\_sap\_c\_2$ (3.13) comme suit :

$$\begin{aligned}&\textbf{prt\_sap\_c\_2 :}\\&\forall\ c\ \in\ \mathcal{S}ap\_C,\ \forall\ req\ \in\ \mathcal{A}ccReq,\\&\Box\ [evt\_request\ (c,\ req)\ \wedge\ req.opRes.res\ \notin\ \mathcal{R}es_c\ \Rightarrow\\&\Diamond\ evt\_send\ (c,\ m)\ \wedge\\&m.comInfo.source = c\ \wedge\ m.comInfo.target = req.source\ \wedge\ m.data.ans = NAK]\end{aligned} \tag{3.13}$$

- Disponibilité (P6) : Toute requête dans un Sap_C, demandant l'accès à une ressource, doit être contrôlée.

  Nous formalisons la propriété $prt\_sap\_c\_3$ (3.14) comme suit :

$$\begin{aligned}&\textbf{prt\_sap\_c\_3 :}\\&\forall\ c\ \in\ \mathcal{S}ap\_C,\ \forall\ req\ \in\ \mathcal{A}ccReq,\\&\Box\ [evt\_request\ (c,\ req)\ \Rightarrow\\&\Diamond\ evt\_check\ (c,\ req)]\end{aligned} \tag{3.14}$$

3.3.5 Exemple

Nous illustrons dans la figure 3.11, un exemple simple d'utilisation de patron SAP sans le patron CHP.

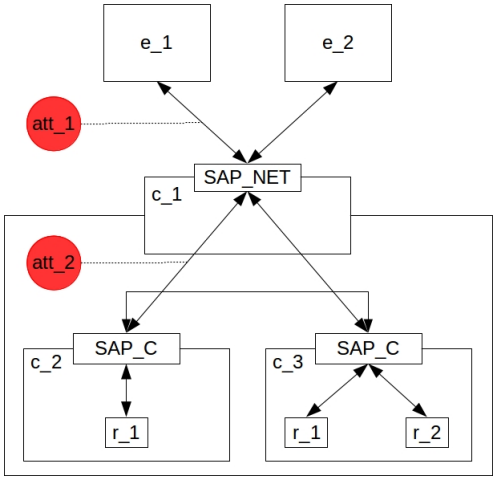

Figure 3.11: Exemple d'utilisation du patron SAP.

Dans les exemples présentés, nous faisons des hypothèses sur l'attaquant, hypothèses souvent utilisées lors de la vérification des protocoles de sécurité :

- (H1) l'attaquant peut insérer des messages sur n'importe quel canal de communication.

- (H2) l'attaquant ne peut pas supprimer un message sur un canal.

- (H3) l'attaquant ne peut pas modifier un message signé par un SAP, ni un message ayant pour source une autre entité que lui.

Dans le cas de la figure 3.11, le composant $c\_1 \in \mathcal{S}ap\_Net$ applique le mécanisme SAP_NET pour protéger l'ensemble des composants $\{c\_2, c\_3\}$. Il garantit un seul point d'accès (*accès légitime*) à cet ensemble.

En complément, sur ce composant, nous pouvons intégrer des mesures de sécurité complémentaires faisant appel à d'autres patrons. Ces mesures permettent de filtrer des messages, détecter des intrusions, etc. Aussi, nous pouvons spécifier une politique de sécurité qui concerne plusieurs composants. Par exemple, nous pouvons décider qu'un client qui envoie un message à un composant, n'a pas le droit d'envoyer un message à un autre composant.

Ce type d'implantation permet de protéger les composants des attaques provenant de l'extérieur de l'ensemble $\{c\_2, c\_3\}$. Par exemple, si l'attaque att_1 essaie d'envoyer des messages, ceux-ci seront arrêtés par c_1. Nous rappelons que le SAP_NET contrôle les messages seulement au niveau d'accessibilité de leur destinataire. Pour arrêter les attaques, d'autres patrons seront nécessaires. SAP garantit une isolation (un seul point d'accès) concernant les canaux légitimes.

Cette protection peut ne pas suffire dans certains cas, lorsqu'un accès aux canaux de communications est possible (*accès non légitime*) comme, par exemple, l'intrusion (att_2) figure 3.11. Celle-ci peut compromettre la politique de sécurité par l'envoi de messages aux composants c_2 et c_3. Cette attaque viole deux propriétés de SAP_NET de c_1, qui sont $prt\_sap\_net\_1.a$ et $prt\_sap\_net\_1.b$, car des messages sont émis vers des composants, sans qu'ils soient contrôlés au préalable par SAP_NET de c_1. Pourtant, cela ne pose aucun problème parce que c_2 et c_3 vérifient les signatures des messages, et les messages de att_2 seront ignorés (ou provoqueront une alerte si cela est spécifié). De plus, nous avons les composants c_2 et c_3 qui appliquent chacun le mécanisme de SAP_C pour contrôler les accès à leurs ressources.

Nous considérons l'intérieur d'un composant isolé. Il ne peut être accédé par une entité que par l'envoi de messages à ce composant. Par exemple, un attaquant ne peut pas modifier le code de c_2 pour éviter passer par son SAP_C lors de l'accès aux données. Mais si un attaquant réussit à accéder à une ressource sans passer par le mécanisme SAP_C, la propriété 3.12 sera violée et nous pouvons tracer la faille qui est la cause une telle vulnérabilité.

Cet exemple simple met en évidence des choix d'implantation des mécanismes de sécurité qui peuvent se poser au concepteur. Un premier choix est de concentrer les protections de sécurité en un seul composant. Mais il faut alors être sûr qu'aucun accès non légitime ne soit possible sur les canaux menant aux composants qu'il protège. L'ajout de fonctions de contrôle sur un composant peut aussi avoir un impact négatif sur sa complexité et sa performance. Un deuxième choix est de distribuer les fonctions de contrôle sur chaque composant. Nous montrons, au chapitre 5, comment les méthodes d'exploration des modèles et de vérification de propriétés peuvent contribuer à évaluer les différentes stratégies d'intégration des patrons au sein des architectures.

3.4 Patron Check Point (CHP)

L'objectif du patron *Check Point* [YB97] est de renforcer une politique de sécurité choisie et l'activation de contre mesures en cas de violation d'un accès. Nous rappelons que, dans le cas du patron SAP, l'objectif principal est d'avoir un seul point d'accès. L'idée ici est d'appliquer des mesures de sécurité sur ce point d'accès.

Nous considérons que les contrôles liés à l'accessibilité d'un composant (cas du SAP_NET) ou l'accessibilité d'une ressource (cas du SAP_C) sont prises en charge par le SAP. Le CHP[4], quant à lui, est utilisé pour appliquer des mesures de sécurité complémentaires référencées dans la politique de sécurité. Lorsqu'un SAP exécute un contrôle d'accès (cf les fonctions $check_c$ et $check_{net}$), il fait appel au CHP pour exécuter ces vérifications et déclencher des actions (contre-mesures) en cas de besoin. Dans notre approche, nous considérons que les deux patrons SAP et CHP sont appliqués ensemble au sein des architectures.

Nous notons, $\mathcal{C}hp$, l'ensemble des composants contenant le mécanisme de type CHP ($\mathcal{C}hp \subset \mathcal{C}omp$).

3.4.1 Fonctionnement

Un CHP assure la vérification de la politique de sécurité et gère le déclenchement de contre-mesure en cas d'une demande qui viole une exigence de sécurité. Il est fortement conseillé d'appliquer le mécanisme de CHP accompagné du mécanisme SAP. Dans notre approche, nous utilisons ensemble CHP et SAP.

La figure 3.12 illustre le fonctionnement du patron CHP dans les deux cas où respectivement le CHP est connecté au mécanisme SAP_NET ou à un SAP_C. Le composant de communication c_1, contient le mécanisme SAP_NET. Le mécanisme CHP est appelé par SAP_NET (après contrôle de l'accessibilité du destinataire) pour vérifier si le message concerné respecte la politique de sécurité. Une politique peut, par exemple, définir une loi qui interdit tous les messages provenant d'une source spécifique ou ayant une destination spécifique.

Le composant d'accès c_2, contient le mécanisme SAP_C. Le mécanisme CHP est appelé par SAP_C (après contrôle de l'accessibilité de la ressource demandée). CHP vérifie si la requête respecte la politique de sécurité. Dans le cas d'accès, une politique peut interdire tous les accès pour une opération et une ressource donnée.

Dans le cas où une requête viole les exigences de sécurité, une contre-mesure peut être déclenchée (pour ignorer un message ou envoyer d'une réponse négative). Une contre-mesure peut provoquer le déclenchement d'une alerte (numérique ou physique comme dans le cas des architectures SCADA), l'arrêt temporaire ou permanent d'un composant ou d'un ensemble de composants.

3.4.2 Structure

La figure 3.13 illustre la structure du modèle du patron CHP.

Nous nommons $\mathcal{C}hp$ l'ensemble de tous les composants implantant le mécanisme de type CHP. Un CHP référence de manière générique :

[4]Comme pour le SAP, dans le document, nous utilisons l'expression "CHP" qui signifie "mécanisme de type CHP".

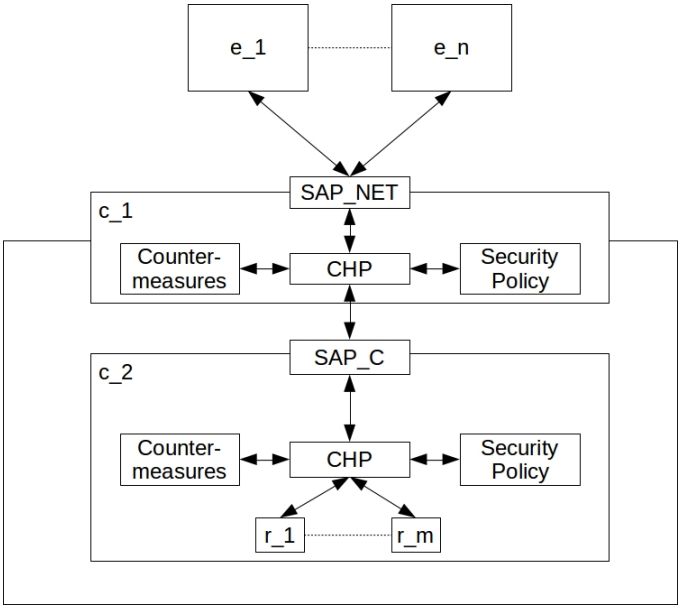

Figure 3.12: Fonctionnement du patron CHP.

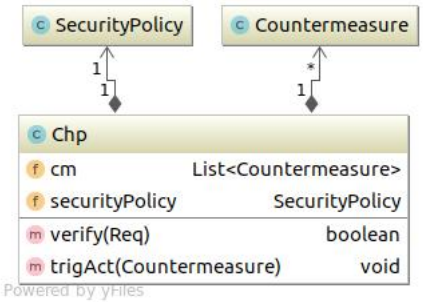

Figure 3.13: Structure du patron CHP.

- *SecurityPolicy* : décrit les règles de la politique de sécurité (par exemple, des autorisation d'accès).
- *Countermeasure* : décrit une contre-mesure spécifique en relation avec une violation d'une propriété).

Les fonctions sont :

- $verify$ (req : Req) vérifie si la requête req respecte la politique de sécurité.
- $trigAct$ (cm : $Countermeasure$) déclenche la contre-mesure cm.

3.4.3 Comportement

Le diagramme de séquence de la figure 3.14 illustre un exemple de comportement de CHP dans le cas d'une cohabitation avec le mécanisme SAP (SAP_NET ou SAP_C) dans le même composant. La requête est contrôlée par le mécanisme SAP. Après le contrôle positif de cette requête (destinataire accessible dans le cas de SAP_NET, ressource accessible dans le cas de SAP_C), le SAP demande au CHP de vérifier la requête. Le CHP vérifie si cette requête est conforme avec la politique de sécurité.

Dans le premier cas (req_1), la requête respecte la politique de sécurité, elle est exécutée (en accédant à la ressource ou en transférant le message).

Dans le deuxième cas (req_2), la requête viole la politique de sécurité, le CHP déclenche une contre-mesure appropriée à cette requête. Une contre-mesure peut être une réponse négative envoyée à l'entité source de la demande.

Nous notons ici que l'exécution de la requête ne se passe pas à intérieur du mécanisme CHP. Une explication plus détaillée de comportement exacte de CHP et du reste des patrons est décrite dans le chapitre 4.

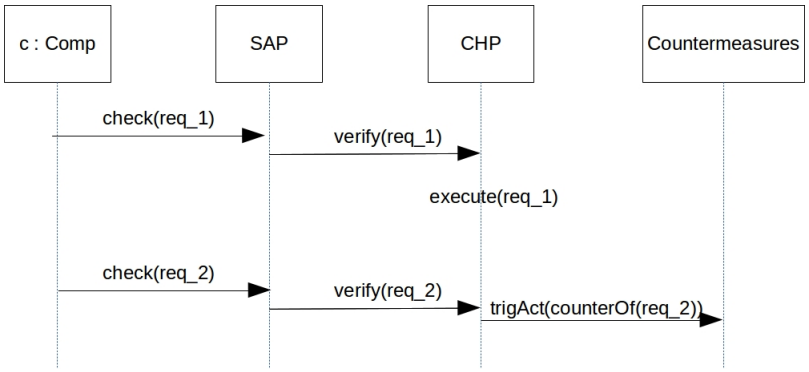

Figure 3.14: Comportement du patron CHP cohabitant avec un SAP.

3.4.4 Propriétés

Comme pour le SAP, nous reprenons les exigences associées à CHP décrites dans [WC03].

- Authenticité : Les contrôles liés à la politique de sécurité sont définis dans *SecurityPolicy* (P1).
- Confidentialité : Les communications internes à la structure CHP en particulier avec le module *SecurityPolicy* ne peuvent être vues par des entités externes (P2)

- Sécurité générale :
  - Une demande (transfert d'un message ou accès à une ressource), vérifiée et conforme avec la politique de sécurité, est acceptée (P3).
  - Une demande, vérifiée et non conforme avec la politique de sécurité, déclenche une action (P4).
  - Une demande, vérifiée et non conforme avec la politique de sécurité, est refusée (P5).
- Intégrité : Les messages véhiculés entre CHP et $SecurityPolicy$ ne peuvent pas être modifiés par des entités externes (P6).

Compte tenu de notre approche de conception des entités sécurisées, certaines propriétés sont intrinsèquement vérifiées car nous les considérons comme des hypothèses.

Les hypothèses que nous considérons sont :

- H1 : Les règles de la politique de sécurité sont encodées dans une fonction correspondante à l'objet $SecurityPolicy$ de la figure 3.12. Cette hypothèse valide la propriété précédente d'authenticité P1.
- H2 : Un attaquant ne peut pas avoir accès aux communications internes entre CHP et le module $SecurityPolicy$. Cette hypothèse valide la propriété précédente de confidentialité P2.
- H3 : Un attaquant ne peut pas modifier les données véhiculées entre CHP et $SecurityPolicy$ (P6). Cette hypothèse valide la propriété précédente d'intégrité P6.

Pour formaliser ces propriétés, nous avons besoin des notations suivantes associées au composant c tel que $c \in \mathcal{C}hp$.

- $policy\_conform\ (c,\ req)$: vrai si la requête req est conforme à la politique de sécurité de composant c.
- $evt\_execute\ (c,\ req)$: événement survenant quand c exécute la requête req.
  - Dans le cas d'un composant d'accès, $evt\_execute\ (c,\ req)$ est équivalent à $evt\_access\ (req.source, req.opRes)$.
  - Dans le cas d'un composant d'accès, $evt\_execute\ (c,\ req)$ est équivalent à $evt\_forward\ (req.mess)$.
- $counterOf\ (c,\ req)$ est la contre-mesure à déclencher par c et associée à la requête req.

Avec ces notations, les propriétés sont formalisées comme suit.

- Sécurité Générale (P3) : Toute demande vérifiée par un chp de type CHP, et conforme à la politique de sécurité, est prise en charge.

  Nous formalisons la propriété prt_chp_1 (3.15) comme suit :

$$\begin{aligned} &\textbf{prt\_chp\_1 :} \\ &\forall\ c \in \mathcal{C}hp,\ \forall\ req \in \mathcal{R}eq, \\ &\Box\ [evt\_verify\ (c,\ req)\ \wedge\ policy\_conform\ (c,\ req) \\ &\Rightarrow\ \Diamond\ evt\_execute\ (c, req)] \end{aligned} \tag{3.15}$$

- Sécurité Générale (P4) : Toute demande vérifiée par un *chp* de type CHP, et non conforme à la politique de sécurité, déclenche un appel de contre-mesure appropriée.

  Nous formalisons la propriété prt_chp_2 (3.16) comme suit :

$$\begin{aligned}&\textbf{prt\_chp\_2 :}\\&\forall\ c \in \mathcal{C}hp,\ \forall\ req \in \mathcal{R}eq,\\&\Box\ [evt\_verify\ (c,\ req)\ \wedge\ (\neg\ policy\_conform\ (c,\ req))\\&\Rightarrow\ \Diamond\ (evt\_trigAct\ (c, cm) \wedge cm = counterOf\ (c,\ req))]\end{aligned} \tag{3.16}$$

- Sécurité Générale (P5) : Toute requête prise en charge, a été vérifié par CHP, et est conforme à la politique de sécurité.

  Nous formalisons la propriété prt_chp_3 (3.17) comme suit :

$$\begin{aligned}&\textbf{prt\_chp\_3 :}\\&\forall\ c \in \mathcal{C}hp,\ \forall\ req \in \mathcal{R}eq,\\&\Box\ [evt\_execute\ (c,\ m)\\&\Rightarrow\ policy\_conform\ (c,\ req)\ \wedge\ pre\_verify\ (c,\ req))]\end{aligned} \tag{3.17}$$

- Nous ajoutons une propriété qui exprime que si une contre-mesure est déclenchée, alors il y a eu une demande non conforme à la politique de sécurité.

  Nous formalisons la propriété prt_chp_4 (3.18) comme suit :

$$\begin{aligned}&\textbf{prt\_chp\_4 :}\\&\forall\ c \in \mathcal{C}hp,\ \forall\ cm \in \mathcal{C}ountermeasure,\\&\Box\ [evt\_trigAct\ (c,\ cm)\\&\Rightarrow\ (\exists\ req\ \in\ \mathcal{R}eq\ \wedge\ cm = counterOf\ (c,\ req)\ \wedge\\&\neg\ policy\_conform\ (c,\ req))\ \wedge\ pre\_verify\ (c,\ req))]\end{aligned} \tag{3.18}$$

3.4.5 Exemple

La figure 3.15 illustre l'application des mécanismes CHP et SAP sur un composant de communication et un composant d'accès. En complément du contrôle réalisé par les SAP, nous avons maintenant la possibilité d'appliquer une politique de sécurité. Nous rappelons que dans l'exemple de la figure 3.11, att_2 ne posait pas un problème grâce à la gestion des signatures. Mais, att_1 pouvait passer s'il cible un composant accessible. Avec CHP appliqué sur c_1, nous pouvons disposer de règles de sécurité qui filtrent les messages pour empêcher att_1 de passer jusqu'à c_2. De plus, CHP s'applique à c_2 pour vérifier les accès aux ressources. Nous pouvons avoir des messages qui respectent le filtrage, mais qui peuvent causer des accès indésirables aux ressources.

Nous avons aussi le choix de disposer d'un filtrage au niveau de c_1 pour ces messages. c_1 peut protèger un ensemble important de composants. Un contrôle de la politique de sécurité concernant l'accès aux ressources doit alors être implanté sur chacun de ces composants. De plus, si c_1 est compromis, tous ces composants le seront. Un autre inconvenient d'une telle approche est la charge de travail sur c_1 qui va devoir effectuer

des vérifications sur chaque message. Nous utilisons donc dans nos travaux (et conseillons) la sécurité en profondeur où les responsabilités sont dispersées le plus possible. Dans des cas particuliers, nous pouvons avoir des composants qui ne peuvent pas avoir de mesures de sécurité (nous verrons un tel cas dans le chapitre 5). Dans de tel cas, nous serons obligés d'implanter les mesures de sécurité sur le composant protégeant le groupe, ou sur d'autre composants en choisissant correctement la politique de sécurité.

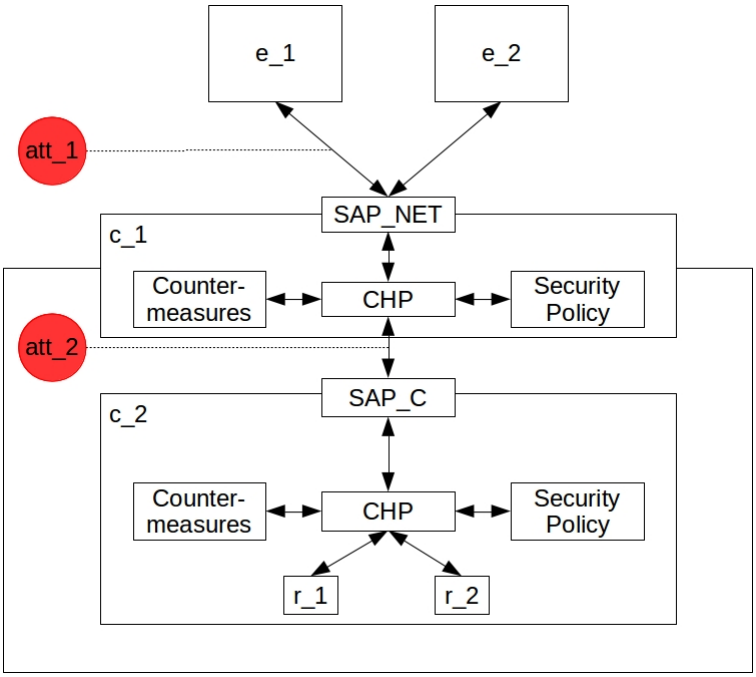

Figure 3.15: Exemple d'utilisation du patron CHP avec les deux cas de SAP.

3.5 Patron Authorization (AUTH)

Le patron *Autorization* ($AUTH$) a été décrit dans [FP01]. L'objectif de ce patron est d'appliquer une politique d'accès sécurisé aux objets ou ressources. Ce patron étend le fonctionnement du CHP. Il a pour objectif d'affecter des droits d'accès à une ressource pour une entité et pour une opération (ex. lecture, écriture, exécution).

Le principe est d'avoir une relation entre des sujets et des objets, qui spécifie les droits de chaque sujet sur chaque objet. Dans nos travaux, les sujets sont les entités qui envoient des requêtes, les objets sont les ressources possédées par des composants.

Précédemment, nous avons décrit le CHP pour assurer l'application correcte d'une politique de sécurité. Cette politique de sécurité implante des exigences d'autorisation d'accès qui sont de la responsabilité de $AUTH$.

3.5.1 Fonctionnement

La figure 3.16 illustre le fonctionnement du patron $AUTH$. Il spécifie un modèle d'implantation de la politique de sécurité dédié à la sécurité d'accès

Le fonctionnement de $AUTH$ permet de protéger l'accès aux ressources par rapport à certaines opérations, dépendant de la source de la demande d'accès. Tout accès à une

ressource protégée et demandant une opération sur cette ressource nécessite une permission (droit) explicite confirmant que l'entité demandeuse peut réaliser cet accès.

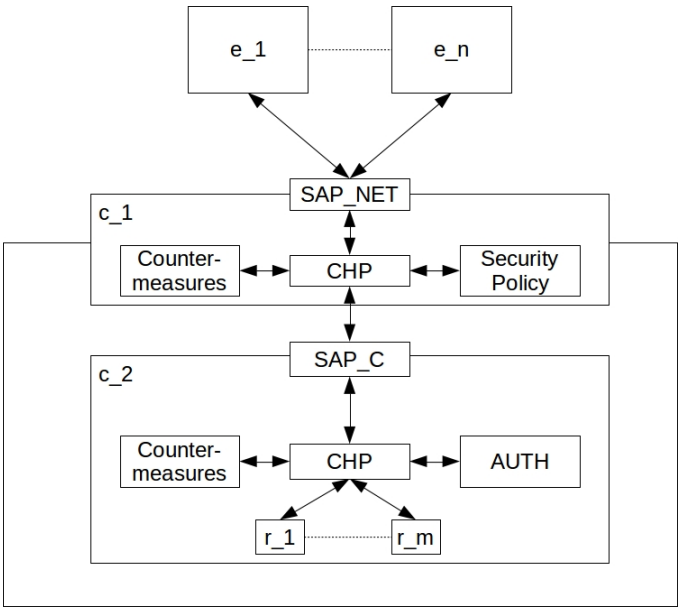

Figure 3.16: Fonctionnement du patron $AUTH$.

3.5.2 Structure

Le patron $AUTH$ implante la structure, illustrée figure 3.17, qui décrit les droits d'accès à une ressource.

- $protections\_list$: Liste des protections sur les ressources pour des opérations.
- $permissions\_list$: Liste des permissions d'accès.
- Chaque permission de type $Permission$ explicite que l'entité ent, peut réaliser l'opération $opRes.oper$ sur la source $opRes.res$.
- De plus, une permission spécifie les éléments suivants que nous ne considérons pas dans ce document :
  - $constraint$: contrainte associée à la permission pour être active.
  - $transferable$: vrai si cette permission est transférable, c'est à dire, si l'entité concernée par cette permission peut la transférer à d'autres entités.

Nous utilisons ANY dans la spécification des permissions et des protections, pour spécifier n'importe quelle opération $oper$ ou n'importe quelle ressource res. Nous introduisons les fonctions suivantes :

- $protect$ (c : SAP_C, $opRes$: $OpRes$) : ajout de l'élément $opRes$ dans la liste $protections\_list$ du composant c.

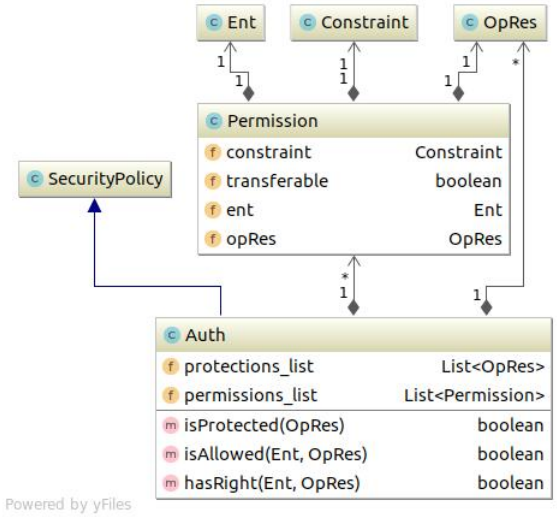

Figure 3.17: Structure du patron $AUTH$.

- $allow\ (c:\ SAP\_C,\ ent:\ Ent,\ opRes:\ OpRes)$: ajout de l'élément $(ent,\ opRes)$ dans la liste $permissions\_list$ du composant c.

Nous définissons les fonctions suivantes associées au patron $AUTH$:

- $isProtected\ (opRes:\ OpRes)$: retourne $true$ si la ressource $opRes.res$ est protégé pour l'opération $opRes.oper$.
- $isAllowed\ (e:\ \mathcal{E}nt,\ opRes:\ OpRes)$: retourne $true$ si l'entité e a une permission explicite de réaliser l'opération $opRes.oper$ sur la ressource $opRes.res$.
- $hasRight\ (e:\ \mathcal{E}nt,\ opRes:\ OpRes)$: retourne $true$ si l'entité e a le droit de réaliser l'opération $opRes.oper$ sur la ressource $opRes.res$ de c. Autrement dit, si la ressource n'est pas protégée pour cette opération (c'est à dire $isProtected\ (opRes) = false$), ou si l'entité a une permission explicite (c'est à dire $isAllowed\ (e,\ opRes) = true$).

$$hasRight\ (e,\ opRes) = \neg\ isProtected\ (opRes) \ \vee\ isAllowed\ (e,\ opRes)$$

3.5.3 Comportement

Le diagramme de la figure 3.18 illustre le comportement du patron $AUTH$ à partir du moment que CHP vérifie une requête. A ce niveau, les requêtes sont uniquement des requêtes d'accès. Soit la requête req avec $req.source = e$ et $req.opRes = opRes$.

Dans le premier cas, si e a le droit $(hasRight\ (e,\ opRes))$ d'accéder à la ressource $opRes.res$ du composant c avec l'opération $opRes.oper$, le CHP est notifié et l'accès est réalisé $(access\ (e,\ opRes))$. c est le composant muni des mécanismes CHP et $AUTH$ et dans lequel cette requête d'accès s'exécute.

Dans le deuxième cas, si l'accès viole la politique de sécurité, le CHP est informé pour qu'il déclenche une contre-mesure approprié $(trigAct\ (counterOf\ (req)))$.

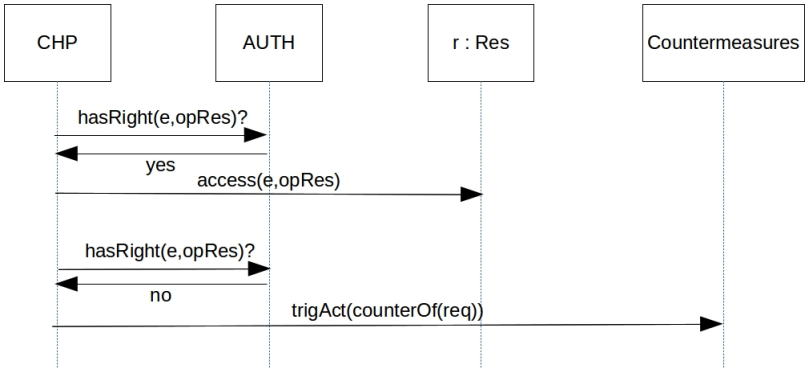

Figure 3.18: Comportement du patron $AUTH$.

3.5.4 Propriétés

Nous rappelons que $\mathcal{A}uth$ est l'ensemble des composants implantant le mécanisme $AUTH$.

La propriété essentielle associée au patron *Authorization* spécifie que l'accès à une ressource res par une entité e, pour l'opération $oper$, n'est permis que si les droits d'accès sont validés pour cette entité et cette opération. Une telle demande sera gérée par un composant qui implante les fonctionnalités du SAP_C intégrant un CHP et un $AUTH$. Cette propriété est en lien avec la propriété $prt\_sap\_C\_3$ de disponibilité (P6) du SAP_C (3.14).

Nous décomposons cette propriété en deux propriétés :

- S'il y a une vérification de demande d'accès, et que cette demande respecte les droits d'accès, alors fatalement, l'accès est réalisé. Une demande d'accès conduit à la vérification de la demande ($verify$).

  Nous formalisons la propriété prt_auth_1 (3.19) comme suit :

$$\begin{aligned} &\textbf{prt\_auth\_1 :} \\ &\forall\ c\ \in \mathcal{A}uth,\ \forall\ e\ \in \mathcal{E}nt,\ \forall\ opRes\ \in \mathcal{O}pRes, \\ &\Box\ [evt\_verify\ (c,\ AccReq(e,\ opRes))\ \wedge\ right\ (c,\ e,\ opRes) \\ &\Rightarrow\ \Diamond\ evt\_access\ (c,\ e,\ opRes)] \end{aligned} \tag{3.19}$$

  le prédicat $right\ (c : \mathcal{A}uth,\ e : \mathcal{E}nt,\ opRes : OpRes)$ est vrai si les droits d'accès sont accordés par les règles de sécurité.

- Tout accès à une ressource respecte les droits d'accès.

  Nous formalisons cette propriété prt_auth_2 (3.20) de la façon suivante :

$$\begin{aligned} &\textbf{prt\_auth\_2 :} \\ &\forall\ c\ \in \mathcal{A}uth,\ \forall\ e\ \in \mathcal{E}nt,\ \forall\ opRes\ \in \mathcal{O}pRes, \\ &\Box\ [evt\_access\ (c,\ e,\ opRes)\ \Rightarrow right\ (c,\ e,\ opRes)] \end{aligned} \tag{3.20}$$

Nous n'avons pas ici à vérifier que la contre-mesure est déclenchée, dans le cas de refus d'accès, puisque c'est de la responsabilité de CHP.

3.5.5 Exemple

La figure 3.19 illustre un exemple d'utilisation du patron $AUTH$.

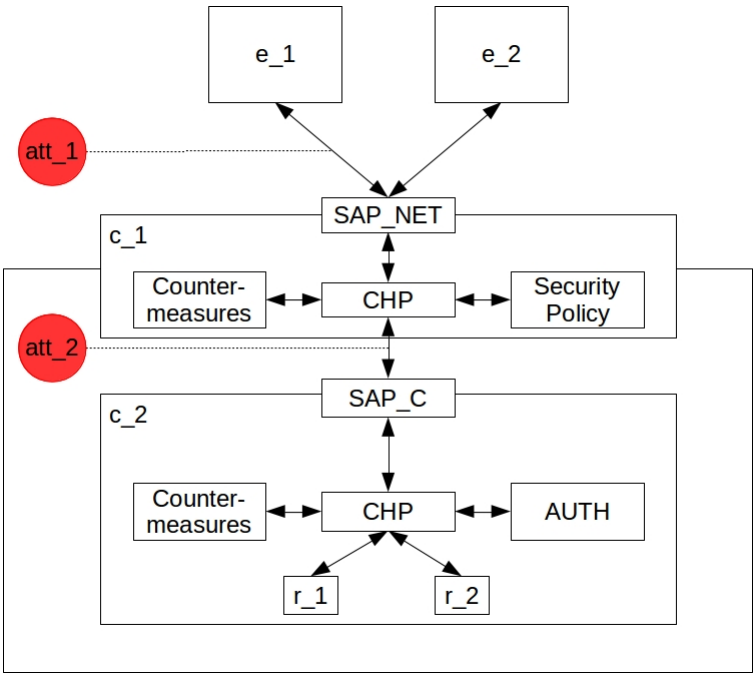

Figure 3.19: Exemple d'utilisation du patron $AUTH$.

Nous utilisons $AUTH$ pour sécuriser l'accès au ressource de c_2. de manière à ce que r_1 soit protégée quelques soient les opérations, et que r_2 soit protégée pour les opérations d'écriture. Dans ce cas, nous initialisons la politique de sécurité par les appels de fonctions suivants :

- *protect* $(c\_2,\ (ANY,\ r\_1))$: protège r_1 de c_2 pour toutes les opérations.
- *protect* $(c\_2,\ (WRITE,\ r\_2))$: protège r_2 de c_2 pour les opérations d'écriture.

Considérons également que e_1 et e_2 ont le droit d'accès en lecture à r_1, et que de plus e_1 a le droit de faire n'importe quelle opération sur r_2. Les appels de fonctions sont les suivants :

- *allow* $(c\_2,\ e\_1,\ (READ,\ r\_1))$: permet à e_1 l'accès en lecture à r_1.
- *allow* $(c\_2,\ e\_2,\ (READ,\ r\_1))$: permet à e_1 l'accès en lecture à r_1.
- *allow* $(c\_2,\ e\_1,\ (ANY,\ r\_2))$: permet à e_1 de faire n'importe quel accès à r_2.

Nous rappelons que, si un attaquant réussit à avoir le contrôle sur une entité (composant ou client), il aura toutes les permissions de cette entité.

3.6 Patron Firewall

Plusieurs descriptions du patron $FireWall$ sont disponibles dans la littérature [FLPS+03, SFBH+13, DGFRLP04]. L'objectif principal d'un $FireWall$ est de filtrer des messages

pour autoriser leur transfert. Un $FireWall$ peut fonctionner sur le niveau physique, le niveau transport ou le niveau application. Le $FireWall$ est similaire à un $AUTH$ mais intervient sur un ensemble de composants, alors que $AUTH$ intervient sur un ensemble de ressources.

Au niveau physique, le filtre du firewall s'applique sur des paquets de données transmis sur un réseau. Il peut contrôler, par exemple, que tout paquet fourni par une entité source spécifique est refusé. Au niveau transport, le filtre collectionne les paquets jusqu'à avoir assez d'informations pour décider si ces paquets sont transférables. Au niveau application, le $FireWall$ évalue l'ensemble des messages au regard de lois associés à des services. Par exemple, un service donné peut ou non envoyer des messages à telle entité destinataire.

3.6.1 Fonctionnement

Dans notre approche, nous considérons un $FireWall$ au niveau application. Plus précisément, à l'aide de $FireWall$, le composant vérifie tous les messages transitant par lui et décide si un message peut passer ou pas dépendant des lois spécifiées dans la politique de sécurité. Les lois peuvent référencer la source ou la destination d'un message, ou le type d'un message, mais jamais son contenu. Nous rappelons que le contenu d'un message $m.data$ est confidentielle. Nous considérons donc un $Firewall$ abstrait au niveau des fonctionnalités.

La figure 3.20 illustre le fonctionnement du patron $FireWall$. Il spécifie un modèle d'implantation de la politique de sécurité dédiée à la sécurité de la communication. Le fonctionnement de $FireWall$ permet de protéger la communication entre les entités. Nous utilisons ce patron pour filtrer les messages entrants ou sortants. Le $firewall$ définit une liste de règles qui doivent être respectées par un message. Pour chaque règle, nous pouvons avoir une liste d'exceptions. Un message respecte une règle sauf s'il est inclus dans une des exceptions associées cette règle. Finalement, un message peut passer s'il respecte toutes les règles.

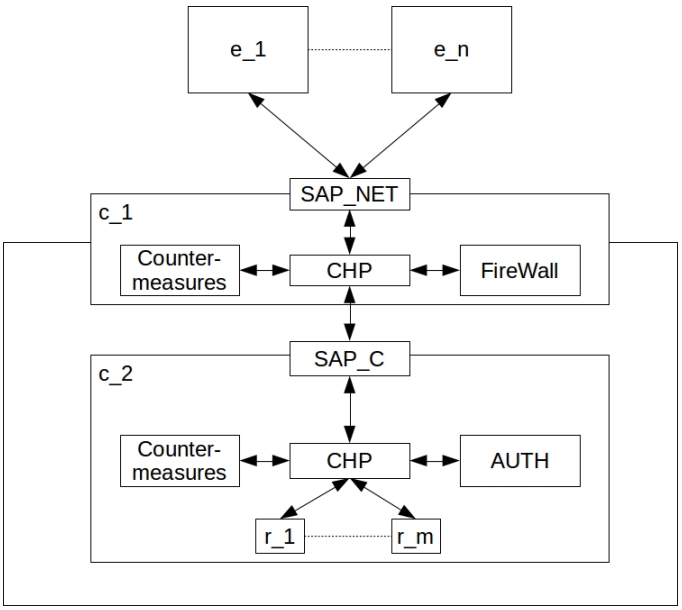

Figure 3.20: Fonctionnement du patron $FireWall$.

3.6.2 Structure

La figure 3.21 illustre la structure du patron $FireWall$ qui implante une politique de sécurité.

- Un $Firewall$ contient une liste de règles $rules\_list$ qui contient des instances de types $Rule$.
- Une règle $rule$ de type $Rule$ contient :
  - $refused$ de type $comInfo$ qui précise qu'un message m tel que $m.comInfo$ correspondant à $refused$ doit être refusé.
  - $exceptions$: liste des exceptions pour cette règle $rule$. Chacune de ces exceptions est de type $ComInfo$.

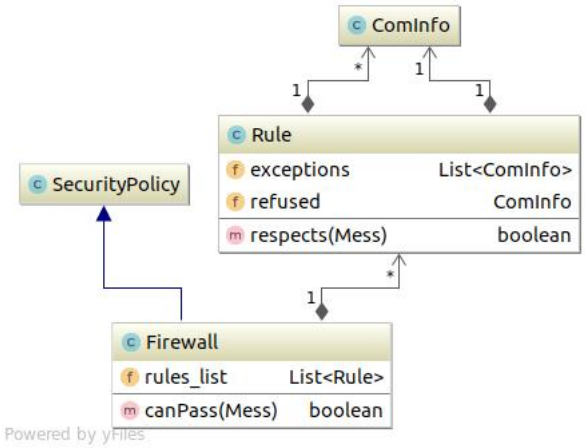

Figure 3.21: Structure du patron $FireWall$.

Un message qui respecte la politique du $FireWall$ doit respecter toutes ses règles ($rules\_list$). Ceci est testé par les fonctions suivantes :

- $respects$ (m : $\mathcal{M}ess$) : retourne vrai si $m.comInfo$ ne correspond pas à $refused$ ou s'il correspond à une des exceptions dans la liste $exceptions$.
- $canPass$ (m : $\mathcal{M}ess$) : retourne vrai si $\forall$ $rule \in$ $rules\_list$, $rule.respects$ (m),

3.6.3 Comportement

Le diagramme de la figure 3.22 illustre le comportement du patron $FireWall$ à partir du moment où CHP vérifie une requête. A ce niveau, les requêtes sont uniquement des requêtes de transfert. Soit la requête req avec $req.mess = m$. Dans le premier cas, si m peut être relayé ($canPass(m)$), le CHP est notifié et m est transféré ($forward(m)$). c est le composant muni du mécanisme CHP et $FireWall$ et par lequel cette requête de transfert transite.

Dans le deuxième cas, si le message ne doit pas être relayé (par exemple si il viole la politique de sécurité), le CHP est informé pour qu'il déclenche une contre-mesure appropriée ($trigAct$ ($counterOf$ (req))).

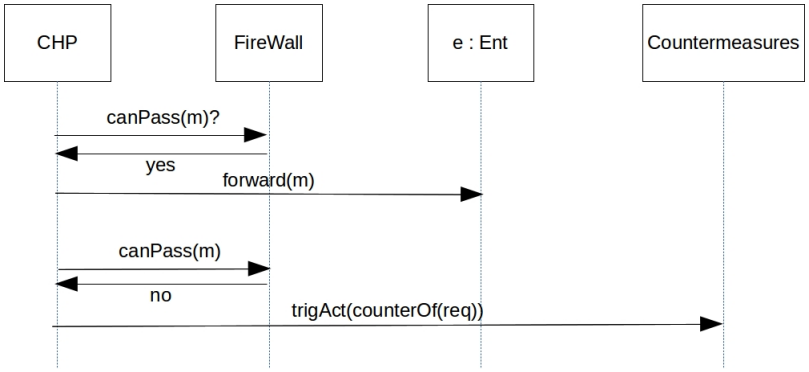

Figure 3.22: Comportement du patron $FireWall$.

3.6.4 Propriétés

Nous rappelons que $\mathcal{F}irewall$ est l'ensemble des composants implantant un mécanisme de $FireWall$.

Une propriété essentielle au patron $FireWall$ spécifie qu'un message reçu est relayé s'il respecte les règles du $FireWall$. Nous décomposons cette propriété en deux parties :

- Si un message reçu respecte les règles, le message est fatalement relayé.

  Nous exprimons la propriété $prt\_firewall\_1$ (3.21) comme suit :

$$\begin{aligned} &\textbf{prt\_firewall\_1 :} \\ &\forall\, c \in \mathcal{F}irewall,\ \forall\, m \in \mathcal{M}ess, \\ &\Box\,[evt\_receive\ (c,\ m)\ \wedge\ pass\ (c,\ m) \\ &\Rightarrow\ \Diamond evt\_send\ (c,\ m)] \end{aligned} \tag{3.21}$$

  Le prédicat $pass\ (c :\ \mathcal{F}irewall,\ m :\ \mathcal{M}ess)$ est vrai si le droit est accordé par les règles de sécurité.

- Si un message est reçu et relayé par le $FireWall$, ce message respecte les règles du firewall.

  Nous exprimons la propriété $prt\_firewall\_2$ (3.22) comme suit :

$$\begin{aligned} &\textbf{prt\_firewall\_2 :} \\ &\forall\, c \in \mathcal{F}irewall,\ \forall\, m \in \mathcal{M}ess, \\ &\Box\,[evt\_receive\ (c,\ m)\ \wedge\ evt\_send\ (c,\ m) \\ &\Rightarrow\ pass\ (c,\ m)] \end{aligned} \tag{3.22}$$

3.6.5 Exemple

La figure 3.23 illustre un exemple d'utilisation du patron $FireWall$. Cette figure illustre aussi l'ensemble des patrons que nous définissons et utilisons dans ce document.

Nous utilisons $FireWall$ pour filtrer les messages entre les composants inclus (c_2) par la protection de c_1 et ceux non inclus (e_1 et e_2). Nous pouvons créer des règles et les ajouter à la liste des règles d'un $FireWall$ comme suit :

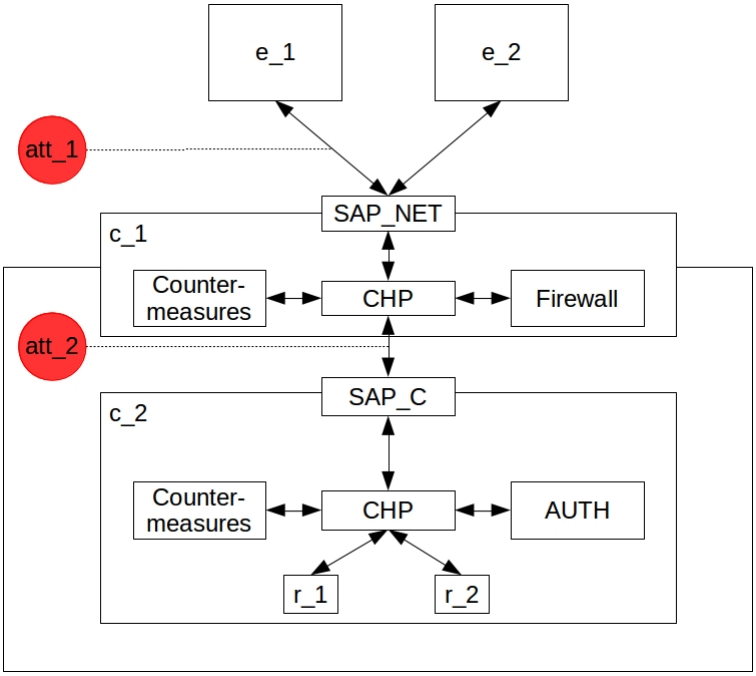

Figure 3.23: Exemple d'utilisation du patron $FireWall$.

- $comInfo\_1 : ComInfo = ComInfo(ANY, e\_1, c\_2)$: N'importe quel type de message, qui a e_1 comme source et c_2 comme destination.
- $comInfo\_2 : ComInfo = ComInfo(ANY, ANY, c\_2)$: N'importe quel type de message, de n'importe quelle source et c_2 comme destination.
- $rule\_1 : Rule = Rule(comInfo\_2, \{comInfo\_1\})$: Une règle qui précise que tout message qui correspond à $comInfo\_2$ est interdit, sauf s'il correspond à $comInfo\_1$.
- Finalement, $c\_1.rules\_list.add(rule\_1)$ pour ajouter cette règle à la politique de sécurité de c_1.

Après avoir décrit formellement cas quatres patrons, nous décrivons, dans le chapitre suivant, une technique d'implantation des mécanismes au sein des composants des architectures.

CHAPTER 4

Intégration des patrons dans les architectures sécurisées

Ce chapitre correspond à la description de l'enjeu 2 décrit en section 1.2. Il introduit la notion d'entité sécurisée embarquant les mécanismes décrits dans les patrons de sécurité. L'objectif est de générer des architectures sécurisées basées sur un réseau d'entités sécurisées.

Nous décrivons en particulier les algorithmes de composition des patrons et de transformation des modèles pour la génération du code simulant les architectures. Le code du modèle de l'architecture est généré dans un format (Fiacre) qui permet sa simulation au sein de l'outil de vérification *OBP* décrit au chapitre 5. Ce modèle intègre les mécanismes de sécurité conformes à la politique de sécurité choisie. Ils sont basés sur les patrons décrits au chapitre 3.

4.1 Approche pour l'intégration des patrons sur une architecture

La sécurisation d'une architecture logicielle implique un processus méthodologique d'intégration des mécanismes basés sur les patrons de sécurité. Cette intégration peut s'effectuer à différents niveaux de l'architecture, que ce soit au niveau interface avec l'environnement ou en interne à l'architecture, au niveau des composants qui y sont inclus. Notre objectif est d'étudier une technique et une méthodologie d'intégration des patrons dans un modèle d'architecture et de permettre la validation formelle de l'architecture générée. L'intégration suppose une adaptation des patrons pour s'intégrer au modèle de l'architecture. Nous supposons qu'en amont de ce processus, l'utilisateur possède un modèle de l'architecture logicielle non sécurisée. Cette architecture inclut la liste des composants en interne à l'architecture et la description des relations, d'une part, entre composants et, d'autre part, entre les composants et l'environnement.

Une fois les mécanismes de sécurité intégrés dans le modèle de l'architecture, ce nouveau modèle doit être simulable. En effet, le modèle doit pouvoir être exploité pour vérifier formellement les propriétés de sécurité. Dans notre approche (cf. figure 4.1), l'utilisateur spécifie ou prend en compte une politique de sécurité qui doit être appliquée. Cette politique provient d'une analyse de sécurité réalisée en amont de ce processus d'intégration. Cette politique comporte des notions générales. Elle doit donc être précisée, sous forme d'un ensemble d'exigences de sécurité, et être adaptée spécifiquement à l'architecture cible. Par exemple, une politique peut exprimer le fait de sécuriser tous les accès à l'architecture. Mais aussi, elle peut spécifier l'obligation de sécuriser l'accès à un composant spécifique de l'architecture. La politique de sécurité doit être adaptée à l'architecture concernée ce qui donne lieu à l'expression d'un ensemble d'exigences associées à cette architecture.

A partir des exigences de sécurité, le choix des mécanismes de sécurité à intégrer est réalisé par le concepteur. Ce choix peut être difficile du fait du caractère non fonctionnel des exigences de sécurité et de la difficulté à prendre en compte ces exigences à différents niveaux d'abstraction du modèle de l'architecture. Une aide est apportée au concepteur par la fourniture d'une librairie commentée des patrons pour aiguiller son choix. A partir de relations existantes entre certains patrons, le modèle d'architecture peut être construit en les intégrant pour aboutir à un modèle d'architecture dite "sécurisée".

Dans notre approche d'intégration des mécanismes de sécurité au sein des modèles d'architecture invoque donc des éléments (en grisé sur la figure 4.1) pris en entrée de ce processus et qui

sont les suivants:

- Nous considérons un modèle d'architecture non sécurisée. Nous supposons que celle-ci est modélisée sous la forme d'un réseau d'entités, communicant par des canaux de communications de type fifo. Chaque entité a un comportement décrit par un automate.

- La politique de sécurité choisie : C'est la politique à mettre en œuvre sur l'architecture de base non sécurisée. Elle est décrite sous la forme d'exigences de sécurité comme décrites au chapitre 2. Ces exigences sont ensuite spécifiées sous la forme d'un ensemble de propriétés formelles adaptées au modèle de l'architecture à sécuriser. La réécriture des exigences constituant une politique de sécurité en propriétés formelles adaptées à l'architecture dotée des patrons de sécurité s'effectue selon la méthodologie préconisée au chapitre 3. En complément des propriétés, un ensemble de scénarios sont identifiés. Ils décrivent les scénarios nominaux d'interaction entre l'architecture et son environnement ainsi que les scénarios d'attaque contre lesquels l'architecture est supposée se protéger.

- Une librairie de patrons à intégrer conformément à la politique de sécurité choisie.

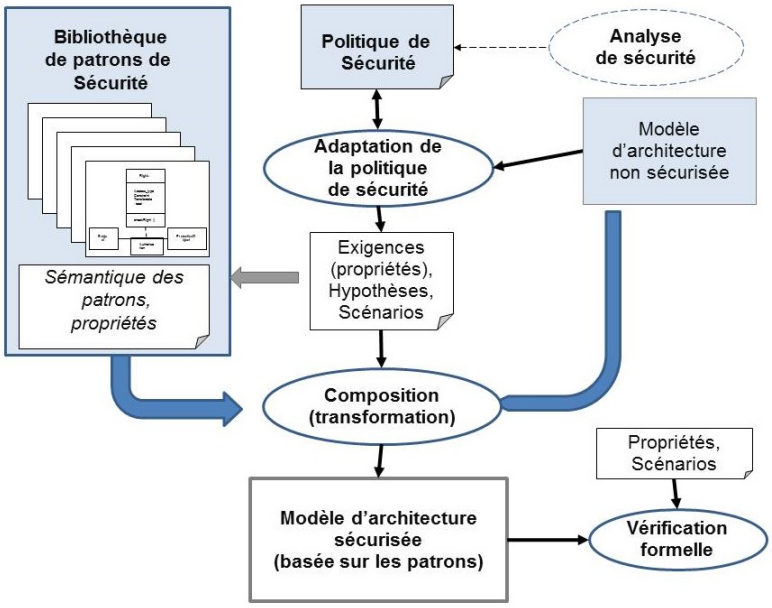

Figure 4.1: Notre approche pour la composition et l'intégration des patrons de sécurité.

A partir de ces éléments d'entrée, architecture de base, politique de sécurité choisie, librairie de patrons de sécurité, le modèle d'architecture sécurisée est construit en intégrant les mécanismes de sécurité au sein des entités de l'architecture.

Sur le modèle de cette nouvelle architecture, une validation formelle doit être réalisée. Nous choisissons, dans notre approche, de la réaliser à l'aide d'une technique de $model-checking$. Celle-ci permet de vérifier, lors d'une simulation exhaustive, toutes les propriétés formelles identifiées sur ce nouveau modèle (cf chapitre 5). Les propriétés sont de deux ordres: les propriétés initiales de l'architecture non sécurisée et les propriétés ajoutées par les patrons

qui correspondent aux contraintes de sécurité. Pour cette vérification, les scénarios d'emploi nominaux seront exploités pour valider le fonctionnement de l'architecture. Mais aussi, des scénarios simulant des attaques doivent être exploités pour pouvoir tester la robustesse de l'architecture.

Nous rendons compte, au chapitre 5, d'expérimentations basées sur cette approche et l'outillage (OBP) permettant des vérifications de propriétés. Les modèles d'architectures sécurisées sont programmées en langage Fiacre. Les propriétés formelles du type sûreté sont converties en invariants et en observateurs de rejet. Les propriétés du type vivacité sont exprimées par des formules logiques de type $SE-LTL$ [KV98, CCG+04, CCO+05], une extension de la LTL. Toutes ces propriétés sont exprimées avec la syntaxe du langage CDL associé à l'outil OBP. Les scénarios nominaux et d'attaques sont modélisés également en langage CDL. Le modèle d'architecture, spécifié en langage Fiacre, peut être simulé et exploré dans l'outillage OBP, en tenant compte de l'ensemble des scénarios.

Au cours de l'exploration exhaustive, les propriétés formelles de sûreté (invariants et observateurs) sont vérifiées. Les propriétés de vivacité ne sont pas actuellement prises en charge en tant que formules SE-LTL par OBP. Elles seront bientôt prises en charge par un model-checker, nommé $Plug$, qui est une extension d'OBP, actuellement en développement. Aujourd'hui $Plug$ n'implante que la vérification des formules logiques LTL, basées sur des predicats et non des événements qui ne sont pas encore pris en compte.

Pour les propriétés de vivacité, nous procédons donc de la manière suivante. Actuellement, une propriété de vivacité, pour être vérifiée par OBP, est encodée, d'une part, sous la forme d'un automate observateur CDL. Cet observateur est spécifié de manière à être sensible à la séquence d'événements référencés dans la propriété. Par exemple, pour la propriété $\Box[e1 \Rightarrow \Diamond e2]$, l'automate observateur a deux transitions et deux états $Start$ et $Wait$. L'occurrence de $e1$ déclenche la transition $Start$ vers $Wait$ et celle de $e2$ déclenche la transition $Wait$ vers $Start$.

D'autre part, pour que la vérification de la propriété soit correcte, des conditions sont requises et doivent être vérifiées. Une première condition est de considérer que, lors de l'exploration, les séquences d'exécution du modèle sont finies, ce qui est rendu possible par le couplage entre les scénarios CDL et le modèle. Chaque scénario, acyclique et de taille finie, interagit avec le modèle. Ensuite, une deuxième condition est que l'observateur doit avoir été sensibilisé, lors de l'exploration du modèle, par l'événement de déclenchement de l'observateur ($e1$ dans l'exemple ci-dessus), c'est à dire avoir quitter son état initial $Start$. OBP renseigne, en fin d'exploration, sur les observateurs qui n'ont pas été sensibilisés. Une troisième condition à vérifier est que l'observateur ne soit pas rester indéfiniment dans un état d'attente (par exemple $Wait$ dans l'exemple ci-dessus. Ceci pourrait se produire en cas de boucle dans le système de transition généré lors de l'exploration du modèle. Cett vérification peut être faite par un contrôle des configurations finales dans le graphe d'exploration et un examen de l'état des observateurs donné par OBP. Si le nombre de configurations finales est grand, suivant la forme des scénarios, nous pouvons fournir le graphe d'exploration à l'outil $Plug$ qui est capable de vérifier la propriété suivante : aucun observateur de ce type ne reste indéfiniment dans un état d'attente. Par exemple, pour un observateur obs, la propriété LTL suivante doit être vérifiée : $\Box[obs.Wait \Rightarrow \Diamond \neg obs.Wait]$ ($obs.Wait$ est le prédicat vrai si obs est dans l'état $Wait$).

Nous pouvons synthétiser le processus d'intégration (cf. figure 4.2) par les phases suivantes:

- Phase 1 : La politique de sécurité est adaptée à l'architecture, et fournit un ensemble

d'exigences, d'hypothèses et de scénarios (nominaux et d'attaques).

- Phase 2 : Ces exigences permettent de choisir le ou les patrons adéquats pour implanter les propriétés dans l'architecture à sécuriser. Les patrons choisis sont intégrés, par composition, dans l'architecture initiale, ce qui donne lieu à la construction d'un nouveau modèle d'architecture qui est supposée être sécurisé.
- Phase 3 : Des vérification formelles sont exécutées par $model-checking$ pour vérifier toutes les propriétés sur l'ensemble de la nouvelle architecture incluant les patrons.

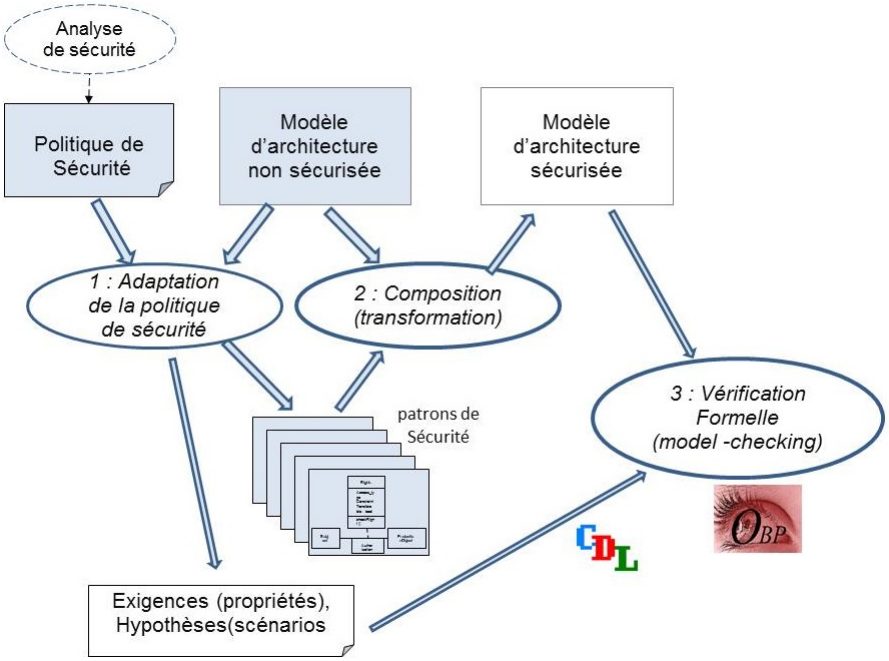

Figure 4.2: Processus d'intégration des patrons de sécurité dans une architecture.

4.2 Modèle des composants d'une architecture non sécurisée

Les architectures que nous considérons incluent deux types de composants : (i) Les composants qui ont pour mission de relayer les messages vers d'autres composants. Nous disons que ces composants sont de type NET (ou de communication). Nous notons, pour un composant c, $type\ (c) = NET$; (ii) Les composants incluant des ressources dont les accès doivent être protégés. Nous disons que ces composants sont de type $ACCESS$. Ceux-ci sont connectés à des composants de type NET mais aussi à des composants de même type $ACCESS$. Nous notons, pour un composant c, $type\ (c) = ACCESS$.

Le comportement de chacun de ces composants est modélisé par un automate.

4.2.1 Comportement d'un composant de type NET

L'automate générique décrivant un composant de type NET est illustré (cf. figure 4.3) :

- De l'état $Idle$, le composant acquiert un message ($receive$ ()) dans une de ses fifos d'entrée et passe dans l'état $Received$.

- Si le message n'est pas destiné à lui ($mine() = false$), il peut donc le relayer ($forward$) et passer dans l'état $Sending$. Sinon, le message est ignoré et le composant revient dans l'état $Idle$ pour recevoir d'autres messages. De l'état $Sending$, le message est envoyé avant que le composant revienne dans l'état $Idle$.

- Sans sécurité, tout message est relayé, sauf si le destinataire de ce message est le composant lui même. Dans ce cas, le composant ignore le message parce que nous considérons que les composants de communications ne font que relayer des messages.

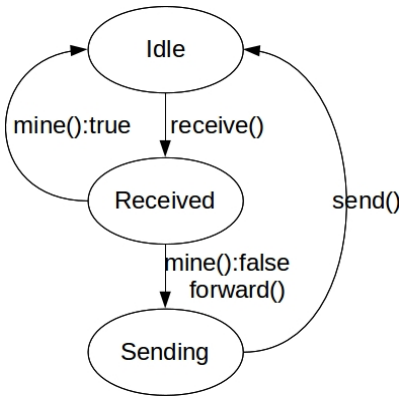

Figure 4.3: Automate de comportement d'un composant de type NET non sécurisé.

4.2.2 Comportement d'un composant de type $ACCESS$

L'automate générique décrivant un composant de type $ACCESS$ est illustré (cf. figure 4.4) :

- De l'état $Idle$, le composant acquiert un message ($receive$ ()) dans une de ses fifos d'entrée et passe dans l'état $Received$.

- Si le message a pour destination le composant lui même ($mine() = true$), le composant passe dans l'état $Compute$ pour traiter le message. Sinon, le composant passe dans l'état $Sending$ pour relayer le message.

- Durant le traitement, nous pouvons avoir plusieurs requêtes d'accès de la forme $request$ ($accReq$: $AccReq$) dont $accReq.ent$ la source du message demandant d'accéder à la ressource $accReq.opRes.res$ avec l'opération $accReq.opRes.oper$. L'accès est exécuté à partir de l'état $Access$ qui répond avec ACK ou NAK. Nous considérons que si une de ces demandes retourne NAK, le reste du traitement est abandonné et une réponse négative (NAK) peut être envoyée. Nous verrons que dans le cas de composant sécurisé, une réponse négative dans un tel cas fait partie des exigences de sécurité et pas du comportement initial du composant.

- A la fin du traitement ($done$), le composant peut envoyer une réponse ou simplement revenir dans $Idle$ dépendamment du traitement spécifié. Pour envoyer une réponse, le composant passe dans l'état $Sending$. Après que le message est envoyé, il revient dans $Idle$ pour recevoir d'autres messages.

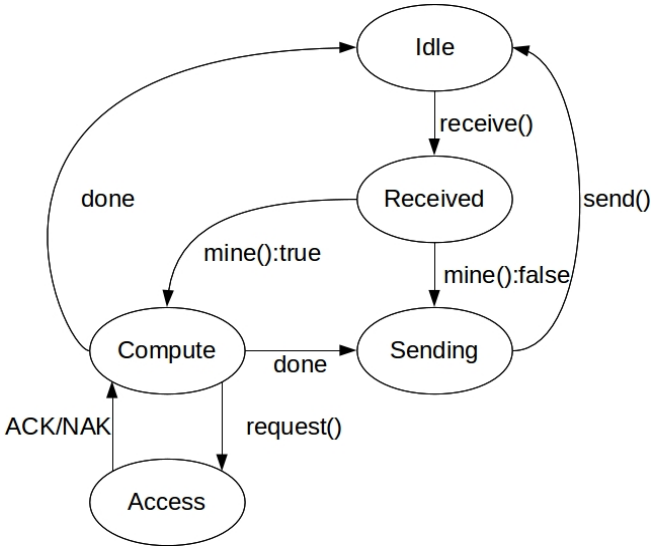

Figure 4.4: Automate de comportement d'un composant de type $ACCESS$ non sécurisé

4.3 Principes d'intégration des mécanismes de sécurité

Les principes de sécurisation des composants d'une architecture ont été introduits dans la section 3.1. Nous rappelons que la sécurité concerne deux objectifs principaux : Un premier (**objectif 1**) concerne les accès à des ressources détenues par un composant (par exemple une mémoire). Cet objectif concerne les composants de type $ACCESS$ vus précédemment. Un deuxième (**objectif 2**) concerne les accès à un ensemble de composants. Cet objectif concerne les composants de type NET vus précédemment.

Conformément à la description des patrons décrits au chapitre 3, l'objectif 1 implique l'intégration des mécanismes de type SAP, CHP et $AUTH$. L'objectif 2 implique l'intégration des mécanismes de type SAP, CHP et $FireWall$. L'intégration des mécanismes de sécurité donne lieu à différents types d'implantation. Un mécanisme de sécurité peut être implanté :

- soit sous la forme d'un composant à part entière.
- soit par l'ajout d'un ou plusieurs états spécifiques au sein de l'automate d'un composant.
- soit par l'ajout d'un simple appel de fonction.

4.3.1 Cas d'un composant de type NET

Dans le cas d'un composant de type NET, ayant pour mission de relayer les messages vers d'autres composants, nous choisissons d'intégrer, conformément à l'objectif 2, les mécanismes SAP, CHP et $FireWall$. Ce composant est transformé en un composant de type SEC_NET. Dans l'exemple illustratif figure 4.5, le composant net est transformé en un composant sec_net qui relaye les messages aux composants sec_c1 et sec_c2, eux même issus de la transformation de $c1$ et $c2$. Les mécanismes SAP, CHP et $AUTH$ sont intégrés dans les composants sec_c1 et sec_c2 comme précédemment (objectif 1).

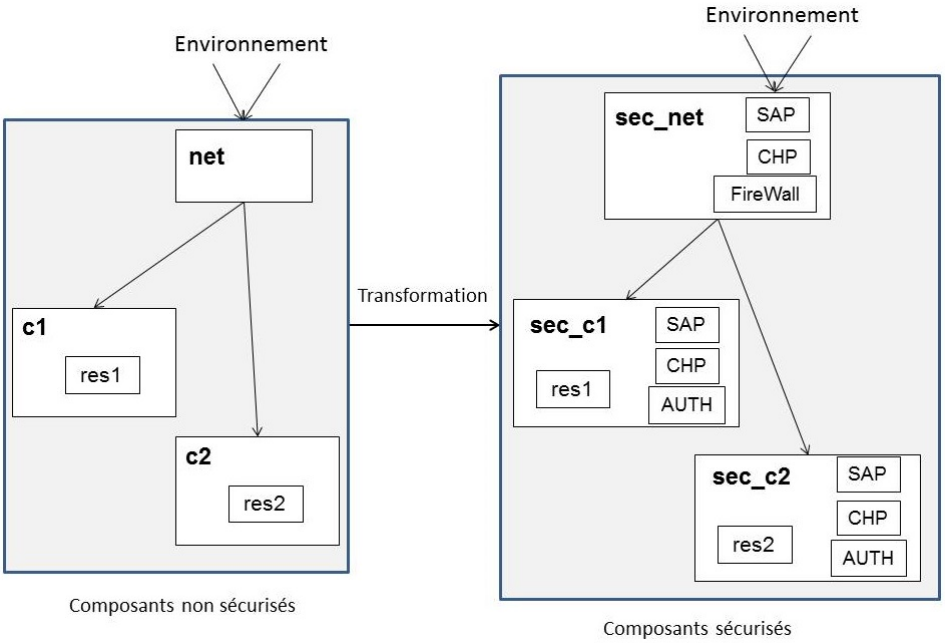

Figure 4.5: Transformation des composants non sécurisés en composants sécurisés.

Le composant sec_net relaye les messages vers les composants sec_c1 et sec_c2 après filtrage (application des règles du $FireWall$).

Les principes de transformation d'un composant de type NET en composant de type SEC_NET reposent sur l'ajout d'états et des appels de fonctions qui implantent les fonctionnalités des mécanismes SAP, CHP et $FireWall$.

L'automate de comportement d'un composant de type SEC_NET, issu de cette transformation, est illustré figure 4.6. Cet automate intègre les mécanismes de sécurité SAP, CHP et $FireWall$. Nous le décrivons comme suit :

- De l'état $Idle$, le composant acquiert un message ($receive$ ()) et passe dans l'état $Received$.

- Si le message est lui destiné ($mine() = true$), ou si la signature n'est pas correcte ($sign() = false$), le composant ignore le message et revient dans $Idle$. Sinon, il doit relayer le message. Pour cela, il fait un appel à $request(FrwReq(m))$ et passe dans l'état SAP pour contrôler cette requête.

- De SAP, si le contrôle valide la requête, ($check() = true$), il passe dans l'état CHP pour vérifier si la requête de relayage respecte la politique de sécurité. Sinon ($check() = false$), une réponse négative est préparée, en utilisant le message concerné par la requête, et le composant passe dans l'état $Sending$.

- De CHP, si la requête respecte la politique de sécurité ($verify() = true$), le composant passe dans l'état $Sending$ pour relayer le message. Sinon, le composant passe dans l'état $TrigAct$ pour appliquer la contre-mesure appropriée. Dans notre approche, pour simplifier, la contre-mesure c'est toujours l'envoi d'une réponse négative à l'entité source du message faisant objet de la requête de relayage.

- De *Sending*, le composant envoi le message, soit la requête de transfert, soit une réponse négative

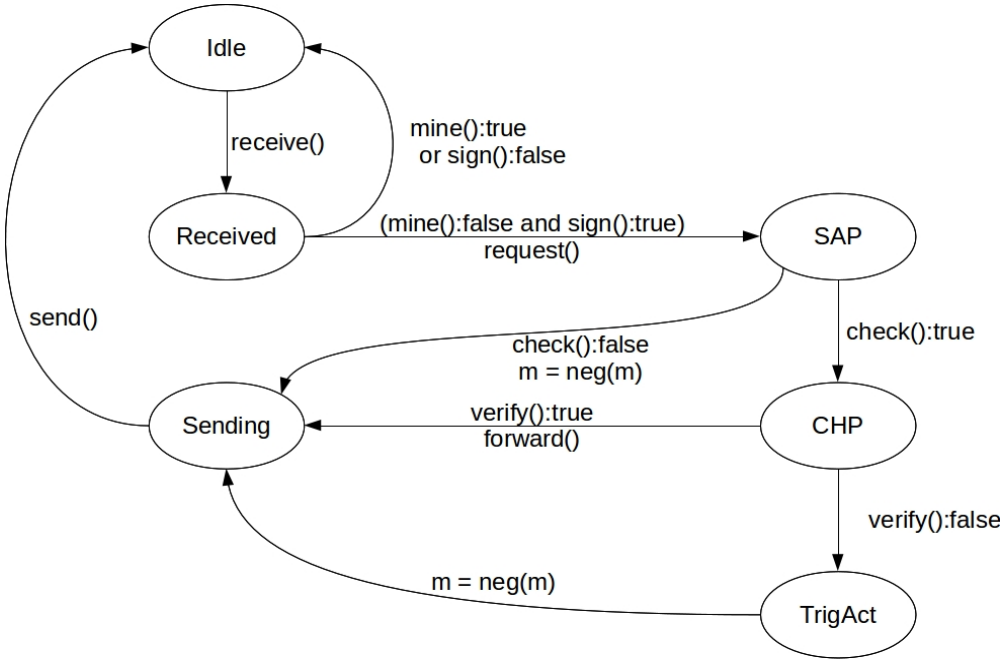

Figure 4.6: Automate de comportement de composant de type SEC_NET.

Nous faisons ici des choix particuliers dans cette implantation (par exemple le traitement de message lorsque le destinataire n'est pas le composant ou lorsque la signature n'est pas correcte). Ces choix peuvent être discutés et modifiés sans remettre en cause le raisonnement sur la manière de transformer l'automate de départ non sécurisé.

4.3.2 Cas d'un composant de type $ACCESS$

Dans le cas d'un composant de type $ACCESS$, incluant une ressource à protéger, nous choisissons d'intégrer, conformément à l'objectif 1, les mécanismes SAP, CHP et $AUTH$. Ce composant est transformé en un composant de type SEC_ACCESS. Par exemple, sur la figure 4.7, le composant c est transformé en un composant sec_c qui intègre les 3 patrons ci-dessus.

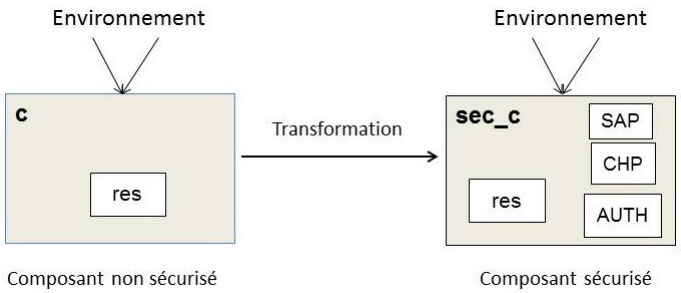

Figure 4.7: Transformation d'un composant de type $ACCESS$ (objectif 1).

Les principes de transformation d'un composant de type $ACCESS$ en composant de type SEC_ACCESS reposent sur l'ajout d'états et des appels de fonctions qui implantent les fonctionnalités des mécanismes SAP, CHP et $AUTH$. L'automate de comportement d'un composant de type SEC_ACCESS, issu de cette transformation, est illustré figure 4.8. Nous le décrivons comme suit :

- De l'état $Idle$, le composant acquiert un message ($receive$ ()) et passe dans l'état $Received$.
- Si la signature n'est pas correcte ($sign() = false$), le composant ignore le message et revient dans $Idle$. Sinon ($sign() = true$), si le message ne lui est pas destiné ($mine() = false$), il passe dans $Sending$ pour relayer ce message (il n'est pas concerné par le relayage sécurisé). Sinon, si le message est bien signé, et qu'il lui est destiné, il passe dans l'état $Compute$ pour traiter le message.
- Durant le traitement, nous pouvons avoir plusieurs requêtes d'accès de la forme $request$ ($accReq$: $AccReq$) (comme vu au parapgraphe 3.2) avec $accReq.ent$ étant la source de message demandant d'accéder à la ressource $accReq.opRes.res$ avec l'opération $accReq.opRes.oper$. Pour valider une requête d'accès, le composant passe dans l'état SAP. L'automate passe de l'état $Compute$ à $Idle$ si toutes les requêtes sont acceptées et s'il n'y a pas besoin d'envoyer une réponse positive.
- De SAP, si le contrôle valide la requête, ($check() = true$), il passe à l'état CHP pour vérifier si la requête d'accès respecte la politique de sécurité. Sinon ($check() = false$), une réponse négative est préparée en utilisant le message concerné par le traitement, et le composant passe dans l'état $Sending$.
- De CHP, si la requête respecte la politique de sécurité ($verify() = true$), le composant peut enfin accéder à la ressource $accReq.opRes.res$. Il passe alors dans l'état $Access$. Sinon, le composant passe dans l'état $TrigAct$ pour appliquer la contre-mesure appropriée (qui est l'envoi d'une réponse négative).
- Nous remarquons que de l'état $Access$, nous passons à $Compute$ toujours avec ACK, car tous les contrôles et les vérifications nécessaires sont exécutées avant d'arriver à l'état $Access$.
- Lors d'un arrêt des traitements invoqués, une réponse négative peut être envoyée lors du passage de l'état SAP à $Sending$, ou des états CHP puis $TrigAct$ à $Sending$.
- De $Sending$, le composant envoit le message qui peut être une réponse positive ou négative.
- A la fin du traitement ($done$), le composant peut envoyer une réponse ou simplement revenir dans $Idle$ dépendamment du traitement spécifié. Pour envoyer une réponse, le composant passe dans l'état $Sending$. Une fois le message envoyé, il revient dans $Idle$ pour accepter d'autres messages.

Notons que d'autres types d'implantation auraient pu être choisies. Mais, dans ce travail, nous rappelons que nous proposons une méthodologie de validation formelle des choix d'implantation. Pour un choix donné, nous proposons une méthode de vérification des propriétés de l'architecture générée. Une perspective de ce travail est d'étudier les critères de comparaison des différentes stratégies d'implantation. L'objectif à terme serait d'aboutir à des choix optimaux conformément aux politiques de sécurité à mettre en œuvre sur les architectures.

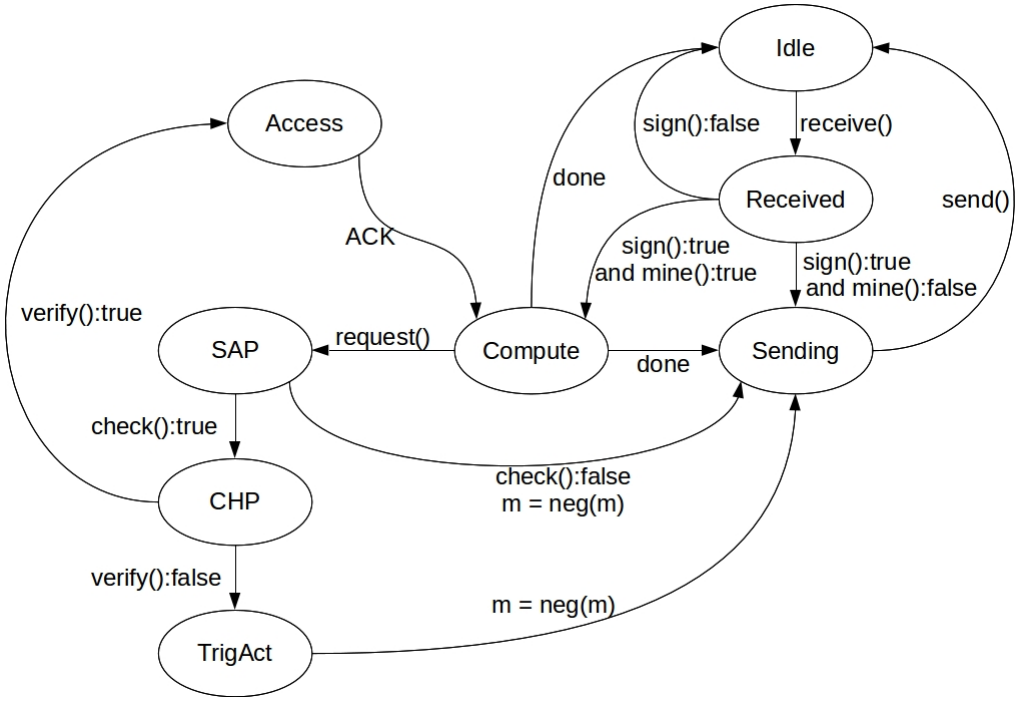

Figure 4.8: Automate d'un composant intégrant les mécanismes de type SEC_ACCESS.

4.3.3 Hypothèses sur les composants à sécuriser

Nous formalisons maintenant les transformations d'un modèle de composant non sécurisée en un modèle de composant sécurisé dans la suite de cette section. Mais nous considérons que, pour établir ces règles, des hypothèses doivent respectées. Elles sont établies sur les composants à transformer et doivent être vérifiées avant la transformation.

- *Hypothese* 1 : La réception des messages s'effectuent par une lecture des fifo d'entrée du composant et uniquement dans l'état *Idle*.
- *Hypothese* 2 : L'émission des messages s'effectuent par une écriture de fifos de sorties du composant et uniquement dans l'état *Sending*.
- *Hypothese* 3 : Dans l'état *Sending*, le composant ne peut changer d'état que s'il a émis un message.
- *Hypothese* 4 (Cas d'un composant de type $ACCESS$) : Toute transition vers l'état *Compute* a pour source l'état *Received*.

4.4 Formalisation des transformations

Nous spécifions ici les algorithmes de transformation, sous la forme de règles formelles, pour chaque architecture non sécurisée en une architecture sécurisée. Pour cela, nous définissons des règles de transformation pour chaque composant de l'architecture à transformer. Nous considérons, dans les architectures, les 2 types de composants (type $ACCESS$ et type NET). Auparavant, nous proposons des notations pour les spécifications formelles des règles.

4.4.1 Notations pour la spécification des règles de transformation

4.4.1.1 Transformation d'une architecture

Une architecture $arch$, non sécurisée, est transformée en une architecture $arch\_sec$ sécurisée par la transformation (4.1) spécifiée comme suit :

$$arch\_sec \ = \ trf \ (arch, \ PS) \tag{4.1}$$

avec : PS, la politique de sécurité à implanter sur l'architecture $arch$.

Le principe, comme nous l'avons décrit précédemment, est de convertir tous les composants de l'architecture en composants sécurisés en fonction de la politique choisie PS. Nous considérons $arch$ comme un ensemble de composants c interconnectés et inclus dans $arch$.

$$\forall c \in Comp \ includes \ (arch) \Rightarrow \ c \in compList \ (arch)$$

Chaque composant c de l'architecture est transformé en un composant sécurisé c_sec tel que :

$$\begin{aligned} &\forall c \in compList \ (arch) \ \wedge \ \ type \ (c) \ = \ NET \ \Rightarrow \\ &c\_sec \ = \ trf\_net \ (c, \ PS) \end{aligned} \tag{4.2}$$

et

$$\begin{aligned} &\forall c \in compList \ (arch) \ \wedge \ \ type \ (c) \ = \ ACCESS \ \Rightarrow \\ &c\_sec \ = \ trf\_access \ (c, \ PS) \end{aligned} \tag{4.3}$$

4.4.1.2 Transformation des composants

Chaque composant de l'architecture a un comportement décrit par un automate. Le principe de transformation des composants consiste donc en la transformation de son automate de comportement, en fonction de la politique choisie PS.

Nous formalisons l'automate de comportement d'un composant c sous la forme d'un graphe $behavior(c)$.

$behavior(c)$ est un n-uplet ($States$, $Trans$, $Labels$, src, tgt, lab) avec :

- $State$: un ensemble des états de l'automate,
- $Trans$: un ensemble de transitions,
- $Labels$: un ensemble de labels,
- une fonction src : $Trans \rightarrow State$ qui associe une transition $t \ \in Trans$ à un état $s \ \in States$ tel que s est la source de t,
- une fonction tgt : $Trans \rightarrow State$ qui associe une transition $t \ \in Trans$ à un état $s \ \in States$ tel que s est la cible de t,
- une fonction lab : $Trans \rightarrow Labels$ qui associe une transition $t \ \in Trans$ à un label $l \ \in Labels$.

Au préalable, nous introduisons la règle mk_trans qui permet de construire une transition d'un automate. Par exemple, considérons les états $s1,\ s2 \in State$ et l'étiquette $e \in Labels$, nous pouvons construire une transition t telle que

$$t = mk\_trans\ (s1,\ e,\ s2) \qquad [mk\_trans]$$

Dans la suite, notons (s1, e, s2) le résultat de l'application de la règle mk_trans aux valeurs $s1$, $s2$ et e.

Nous formalisons maintenant la transformation des composants pour les deux types de composant, $ACCESS$ et NET.

4.4.2 Cas d'un composant de type NET

La fonction de transformation trf_bev_net pour un composant c de type NET génère l'automate du composant sécurisé c_sec à partir de l'automate du composant c non sécurisé.

La transformation trf_net est définie formellement de la manière suivante :

$$\begin{aligned} &\forall c \in compList\ (arch)\ \wedge\ \ type\ (c) = NET\ \Rightarrow \\ &behavior\ (c\_sec)\ =\ trf\_bev\_net\ (behavior\ (c),\ PS\_net) \end{aligned} \tag{4.4}$$

avec :

- $behavior\ (c)$: automate du composant c non sécurisé,
- PS_net : ensemble des patrons à intégrer. Pour l'instant, nous considèrons : PS = {SAP, CHP, FireWall},
- $behavior\_sec\ (c)$: automate du composant c sécurisé, résultat de la transformation. Cet automate corrrespond à l'automate à générer conformément à la figure 4.6.

4.4.3 Cas d'un composant de type $ACCESS$

La fonction de transformation trf_bev_access pour un composant c de type $ACCESS$ génère l'automate du composant sécurisé c_sec à partir de l'automate du composant c non sécurisé.

La transformation trf_access est définie formellement de la manière suivante :

$$\begin{aligned} &\forall c \in compList\ (arch)\ \wedge\ \ type\ (c) = ACCESS\ \Rightarrow \\ &behavior\ (c\_sec)\ =\ trf\_bev\_access\ (behavior\ (c),\ PS\_access) \end{aligned} \tag{4.5}$$

avec :

- $behavior\ (c)$: automate du composant c non sécurisé,
- PS_access : ensemble des patrons à intégrer. Pour l'instant, nous considèrons : PS_access = {SAP, CHP, AUTH},
- $behavior\_sec\ (c)$: automate du composant c_sec sécurisé, résultat de la transformation. Cet automate corrrespond çà l'automate à générer conformément à la figure 4.8.

4.5 Prototype de génération de modèle d'architecture sécurisée

Le développement d'un prototype de générateur de modèle d'architecture sécurisée est en cours de développement. Une première version, prend déjà en entrée la description de l'architecture non sécurisée (ensemble des composants et liens de communication) et la liste des mécanismes de sécurité à intégrer sur chaque composant de l'architecture. Cette version permet de générer une architecture au format Fiacre avec les automates de chaque composant tels qu'ils ont été décrits précédemment. Les données d'entrée du générateur sont aujourd'hui spécifiées dans un fichier au format XML. Les scénarios et les propriétés à vérifier sont rédigés manuellement. Elles sont vérifiées par l'outil OBP en prenant en compte les scénarios considérés.

Une version plus élaborée est en cours de développement. Elle permettra de générer automatiquement des ensembles de scénarios et propriétés de sécurité à vérifier. Nous décrivons, au chapitre 5, les types de propriétés, spécifiques à chaque mécanisme de sécurité et pour chaque composant, qui doivent être générées.

A partir de la description XML, la génération va exécuter les phases suivantes (cf. figure 4.9) :

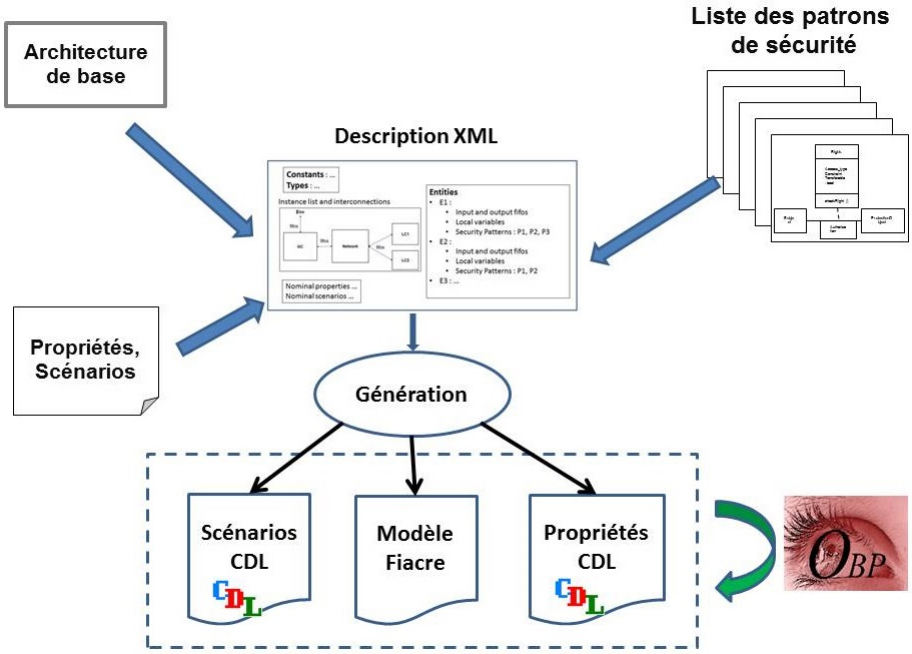

Figure 4.9: Génération du modèle de l'architecture sécurisé.

- Génération du modèle (format Fiacre) sous la forme :
  - d'un réseau de processus communicants (instances des entités). Le comportement de chaque processus est régi par un automate. Cet automate est configuré selon le paramétrage qui lui affecte des patrons. Les fifos sont affectées à chaque processus.
  - d'une liste de connexions entre les instances.
- Génération d'un ensemble de scénarios (nominaux et d'attaque) au format CDL.

- Génération d'un ensemble de propriétés (invariants, observateurs, formule LTL) au format CDL.

Une fois les modèles CDL et Fiacre générés, des propriétés qui n'ont pas été générées sont ajoutées par le concepteur ainsi que des scénarios complémentaires.

La description d'une architecture, spécifiée dans un format XML, inclut (cf. figure 4.10) :

- La liste des entités (*Entities*) composant l'architecture. Chaque entité détient :
  - une liste de fifo d'entrée (*input fifos*) et de sortie (*output fifos*).
  - une liste de variables locales
  - la liste des patrons de sécurité intégrés dans l'entité.
- La liste des instances de ces entités avec les interconnexions entre les instances (fifos).
- La liste des propriétés nominales (*nominal properties*) liées à l'architecture de base (non sécurisée) devant être vérifiées.
- La liste des constantes et types utiles pour la génération du code Fiacre.

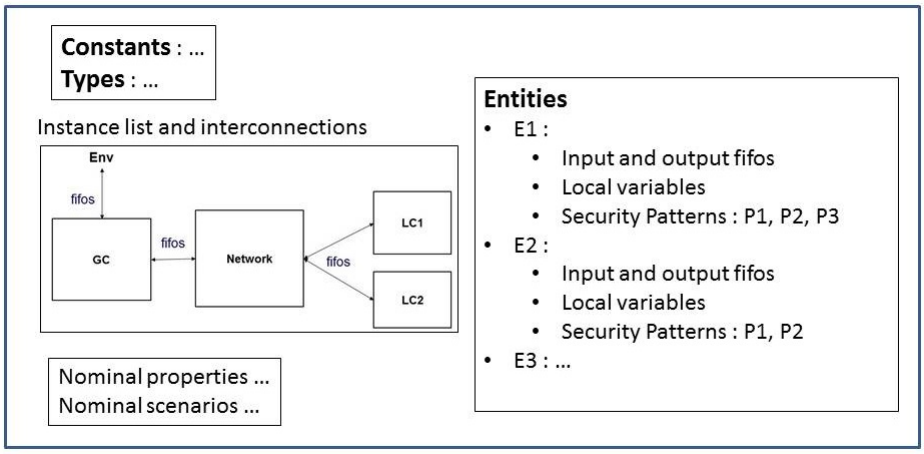

Figure 4.10: Description (XML) d'une architecture avec patrons de sécurité.

CHAPTER 5

Expérimentations

Ce chapitre correspond à l'enjeu 3 décrit au paragraphe 1.2.

Nous présentons dans ce chapitre des expérimentations qui ont été menées sur deux cas de modèles d'architecture de complexité différente. Lors de ces expérimentations, nous exploitons une approche de vérification par *model-checking*. Nous l'exploitons pour vérifier formellement les propriétés correspondantes à chaque modèle d'architecture.

En section 5.1, nous décrivons la technique de vérification et l'outillage OBP utilisé pour les expérimentations ainsi que les bases du langage CDL pour l'expression des propriétés et des scénarios. En section 5.2, la première architecture simple est décrite composée de 4 entités. Sur cette architecture, nous faisons l'hypothèse de scénarios d'attaques. Pour protéger l'architecture, nous intégrons, dans l'architecture (section 5.3), des mécanismes de sécurité basés sur les patrons décrits précédemment. Un deuxième cas d'architecture plus complexe est décrit section 5.4. Cette architecture représente une abstraction d'une architecture vétronique [Sti06].

5.1 Méthode et Outillage pour la vérification de propriétés

Le but de cette section est de décrire la technique de vérification formelle des propriétés utilisée durant les expérimentations. Nous décrivons la chaîne d'outil OBP utilisée et la spécification en langage CDL, d'une part, des propriétés et, d'autre part, des scénarios. Le but de cette section est d'introduire le formalisme CDL pour l'expression des scénarios et des propriétés et qui sera utilisé lors des expérimentations.

5.1.1 L'outil OBP

Pour établir la vérification d'un ensemble d'exigences sur un modèle[1], il faut disposer d'un modèle simulable et des exigences formalisées sous la forme, par exemple, de formules logiques ou d'automates observateurs (Figure 5.1). Le modèle simulable, les exigences et les interactions avec l'environnement (le contexte) constituent les données pertinentes et suffisantes pour pouvoir mener les vérifications. Pour pouvoir exécuter une exploration du modèle, il faut intégrer dans le modèle le comportement de l'environnement.

Pour les vérifications, nous exploitons la technique de model-checking et l'outillage *OBP* [DBRL12, CT16] (http:www.obpcdl.org) développé dans l'équipe d'accueil (Laboratoire Lab-STICC de l'ENSTA-Bretagne). Les modèles d'architectures logicielles sont actuellement codés dans le langage Fiacre. Les environnements (*contextes*) interagissant avec les modèles sont codés dans le langage CDL[2] (*Context Description Language*) ainsi que les propriétés formelles.

Le modèle d'architecture est ensuite simulé et exploré par *OBP*. L'exploration génère un système de transitions (SdT). Celui-ci représente tous les comportements du modèle dans son environnement sous la forme d'un graphe de configurations et de transitions. Sur ce SdT, une vérification des propriétés (invariants, observateurs de rejet, propriété de vivacité sous certaines conditions) peut être conduite comme expliqué en section 4.1. La difficulté liée à cette technique est la production du SdT qui peut être de grande taille, dépassant la taille mémoire disponible (explosion combinatoire). Ces dernières années, des travaux ont été menés par l'équipe d'accueil pour optimiser la gestion de la complexité

[1] Nous notons par la suite "modèle", le modèle du système à valider.
[2] La documentation du langage CDL est disponible sur http://www.obpcdl.org.

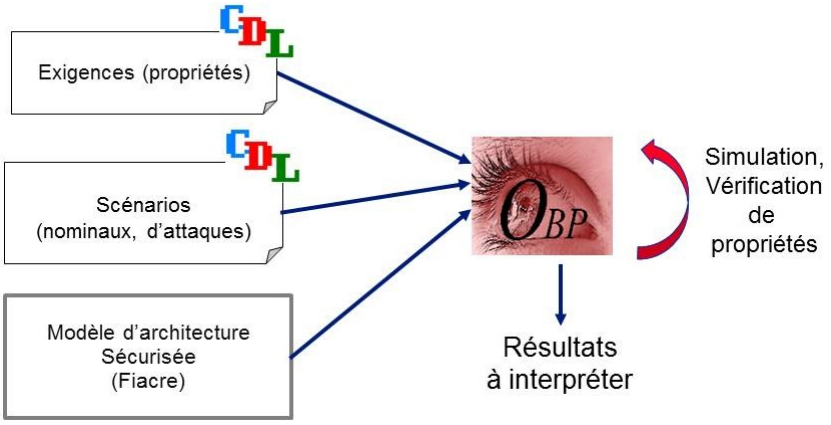

Figure 5.1: Vérification de propriétés par exploration de modèle.

des explorations des modèles, notamment par l'implantation d'algorithmes de *splitting* [DRB11, DBRL12, DT14, TCP14] et de Past-Free[ze] [CTD16, CT16].

5.1.2 Spécification des scénarios CDL

Lors des vérifications, les états du modèle d'architecture sont explorés par simulation. Lors de ces explorations, nous appliquons des scénarios qui simulent le comportement de l'environnement qui interagit avec le modèle. Les scénarios définissent les interactions d'acteurs légitimes qui communiquent avec le modèle, mais également le comportement d'éventuels attaquants. Les scénarios sont modélisés comme des graphes acycliques comprenant des opérations d'envois et de réception de messages vers et du modèle d'architecture à valider.

Supposons deux entités clients $clt1$ et $clt2$ interagissant avec une architecture composée de deux composants (cf figure 5.2). Les deux clients exécutent des requêtes d'écriture et de lecture sur des ressources res_1 et res_2 détenues par les composants $comp_1$ et $comp_2$.

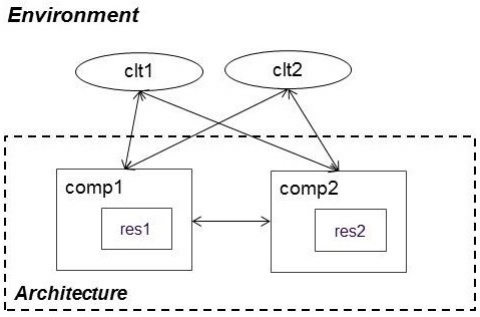

Figure 5.2: Modèle d'architecture à deux composants.

Nous définissons les interactions sous la forme d'événements CDL :

```
event evt_rd_clt1_comp1_res1 is {send cte_rd_clt1_comp1_res1 to comp1}
event evt_wr_clt2_comp2_res2 is {send cte_wr_clt2_comp2_res2 to comp2}
event ...
```

L'événement CDL $evt\_rd\_clt1\_comp1\_res1$ exprime une requête de lecture de la part de $clt1$ vers la ressource $res1$ détenue par $comp_1$. De même, $evt\_wr\_clt2\_comp2\_res2$ exprime une requête d'écriture de la part de $clt2$ vers la ressource $res2$ détenue par $comp_2$.

Les constantes $cte\_rd\_clt1\_comp1\_res1$ et $cte\_wr\_clt2\_comp2\_res2$ sont définies, par des variables structurées, dans le modèle Fiacre de la manière suivante :

```
const cte_rd_clt1_comp1_res1 :
T_REQ is {source = ID_CLT1, target = ID_COMP1, oper = RD, res = RES1}
const cte_wr_clt2_comp2_res2
T_REQ is {source = ID_CLT2, target = ID_COMP2, oper = WR, res = RES2}
```

Nous pouvons regrouper des ensembles de requêtes par des activités (*activity*) pour construire des séquences (avec l'opérateur ";") ou des alternatives (avec l'opérateur "[]") de requêtes. Des exemples d'activités act_clt1 et act_clt2 sont illustrés ci-dessous :

```
activity act_clt1 is { event evt_rd_clt1_comp1_res1;  ... ; event evt_clt1_end}
activity act_clt2 is { event evt_wr_clt2_comp2_res2;  ... ; event evt_clt2_end}
```

Les événements evt_clt1_end et evt_clt2_end spécifient les réceptions d'un marqueur de fin des activités respectives act_clt1 et act_clt2.

Enfin, nous regroupons, dans des scénarios CDL, les comportements (activités) des entités $clt1$ et $clt2$. Par exemple, le scénario $scenario1$, ci-dessous, décrit la concurrence des deux activités act_clt1 et act_clt2 respectivement pour les entités clientes de l'environnement $clt1$ et $clt2$.

```
cdl scenario1 is { main is { act_clt1 || act_clt2} }
```

Les scénarios mentionnés précédemment spécifient des demandes de traitements simples ($READ$ ou $WRITE$) qui impliquent un seul accès par requête à la ressource concernée. En cas de requêtes spécifiant des traitements nécessitant plusieurs accès, comme vu au paragraphe 3.2, le format des messages sera adapté.

5.1.3 Spécification des propriétés CDL

L'intégration des mécanismes de sécurité implique des opérations de vérification de propriétés formelles. Celles-ci concernent les propriétés initiales (*propriétés nominales*) du modèle de l'architecture (c'est-à-dire l'architecture non sécurisée) qui doivent être préservées lors de la transformation en un modèle de l'architecture sécurisée. Elles concernent aussi toutes les propriétés associées aux mécanismes eux-mêmes et qui doivent être vérifiées (*propriétés de sécurité*) .

5.1.3.1 Propriétés nominales

Un exemple de propriété nominale, à vérifier sur notre exemple précédent, concerne l'acquittement d'une opération de lecture d'une ressource détenue par un composant. Comme décrit en section 3.2.3, et à l'image de la propriété 3.1, la propriété $pty\_rd\_clt1\_comp1\_res1$ (5.1) exprime formellement que l'envoi (*send*), par l'entité $clt1$, d'une requête portant la constante $cte\_rd\_clt1\_comp1\_res1$, est suivi d'une réception *receive*, par la même l'entité

$clt1$ portant la constante $cte\_clt1\_comp1\_res1\_ack$. Pour cela, nous utilisons les deux événements evt_send et $evt\_receive$ définis précédemment en section 3.2.3.

$$\begin{aligned}&\textbf{pty\_rd\_clt1\_comp1\_res1 :}\\&\Box\ [evt\_send\ (clt1,\ cte\_rd\_clt1\_comp1\_res1)\\&\Rightarrow\ \Diamond\ evt\_receive\ (clt1,\ cte\_clt1\_comp1\_res1\_ack)]\end{aligned} \tag{5.1}$$

Comme vu en section, 4.1, les propriétés de vivacité sont exprimées avec la syntaxe CDL sous la forme suivante (5.2) :

$$\begin{aligned}&\textbf{propertyLTL}\ pty\_rd\_clt1\_comp1\_res1\ \textbf{is}\ \{\\&[\,]\ (\ |\ evt\_send\_clt1\_cte\_rd\_clt1\_comp1\_res1\ |\\&=>\ <>\ |\ evt\_receive\_clt1\_cte\_clt1\_comp1\_res1\_ack\ |)\\&\}\end{aligned} \tag{5.2}$$

Quand l'outil $Plug$ aura la capacité à vérifier des formules du type $SE - LTL$, la propriété pourra être vérifiée.

Actuellement, nous codons cette propriété sous la forme de l'observateur suivant :

```
property pty_rd_clt1_comp1_res1 is
{
   start / / evt_send_clt1_cte_rd_clt1_comp1_res1     /  -> wait;
   wait  / / evt_receive_clt1_cte_clt1_comp1_res1_ack /  -> start
}
```

OBP vérifie cet observateur avec les conditions que nous avons précisées au paragraphe 4.1.

5.1.3.2 Propriétés de sécurité

Les propriétés de sécurité décrites au chapitre 3 sont codées en langage CDL. Comme exemple, prenons la propriété générique $prt\_sap\_C\_1$ (3.12) vue précédemment, associée au comportement du mécanisme SAP. Si nous souhaitons exprimer que tout accès ($access$) en lecture à une ressource $res1$ du composant $comp_1$ accordé à une entité cliente $clt1$ est précédé d'un contrôle d'entrée ($check$) de la requête associée, nous l'exprimons par l'invariant suivant $prt\_sap\_c\_1\_comp1\_clt1\_rd\_res1$:

$$\begin{aligned}&\textbf{prt\_sap\_c\_1\_comp1\_clt1\_rd\_res1 :}\\&\Box\ [pre\_access\ (comp_1,\ clt1,\ (rd,\ res_1))\\&\Rightarrow\ pre\_check\ (comp_1,\ AccReq\ (clt1,\ (rd,\ res1)))]\end{aligned} \tag{5.3}$$

Nous que dans la propriété 3.12 vu prédédemment, l'événement evt_access était référencé. Ici, nous référençons pre_access pour pouvoir spécifier la propriété au format CDL avec le prédicat $comp1\_accessed\_clt1\_rd\_res1$. Ceci est équivalent car ce prédicat est vrai après l'occurence de l'événement et nous considérons ici que les événements sont uniques.

Cet invariant est exprimé par l'invariant CDL $inv\_rd\_clt1\_comp1\_res1$ de la manière suivante :

```
assert inv_rd_clt1_comp1_res1
```

avec :

```
predicate inv_rd_clt1_comp1_res1 is
{
    (not comp1_accessed_clt1_rd_res1) or comp_1_checked_clt1_rd_res_1
}
```

et les predicats CDL suivants :

- $comp1\_accessed\_clt1\_rd\_res1$: est vrai si la ressource $res1$ du composant $comp_1$ est accédée en lecture suite à une requête de l'entité $clt1$.
- $comp1\_checked\_clt1\_rd\_res1$: est vrai si le composant $comp_1$ a contrôlé la demande d'accès à la ressource $res1$ du composant $comp_1$ suite à une requête de l'entité $clt1$.

Une fois les scénarios et propriétés spécifiés en CDL, les contextes décrivant les comportements des entités de l'environnement vis à vis de l'architecture peuvent être spécifiés sous la forme de contextes CDL référençant toutes les propriétés à vérifier durant l'interprétation du modèle par OBP. Par exemple, le scénario $scenario1$ s'écrit de la manière suivante :

```
cdl scenario1 is
{
   properties
    pty_rd_clt1_comp1_res1,
    pty_wr_clt2_comp2_res2

assert inv_rd_clt1_comp1_res1;
assert inv_wr_clt2_comp2_res2

    main is { act_clt1 || act_clt2}
}
```

Durant l'exécution du modèle, soumis aux interactions de $clt1$ et $clt2$ mis en parallèle, OBP vérifie les quatre propriétés : deux propriétés de type vivacité, $pty\_rd\_clt1\_comp1\_res1$ et $pty\_wr\_clt2\_comp2\_res2$, et deux propriétés de type invariant $inv\_rd\_clt1\_comp1\_res1$ et $inv\_wr\_clt2\_comp2\_res2$.

5.2 Architecture simple

Nous mettons en œuvre cette technique d'exploration des modèles et de vérification des propriétés sur la première architecture simple (figure 5.3). Cette architecture est présentée sous la forme non sécurisée. Ensuite, nous modifions le modèle en vue de la sécuriser en intégrant des mécanismes de sécurité.

5.2.1 Description de l'architecture non sécurisée

Prenons l'exemple d'une architecture simple illustrée figure 5.3.

L'architecture est composée de 4 entités : Un contrôleur global GC, deux contrôleurs locaux LC_1 et LC_2 et un réseau de communication $Network$ qui fait le lien entre GC et les

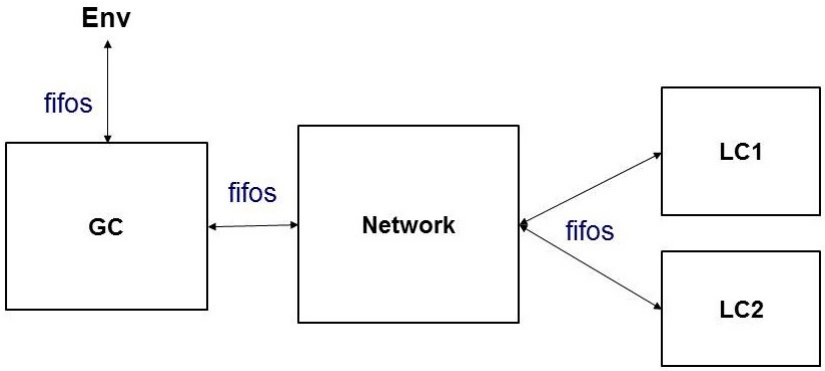

Figure 5.3: Architecture simple non sécurisée.

contrôleurs locaux. Les contrôleurs locaux détiennent respectivement une ressource res_1 et res_2.

Le fonctionnement de l'architecture est le suivant :

- Le processus GC reçoit les requêtes de l'environnement Env. GC traite la requête :
  - Si la requête est adressée à GC, GC traite la requête et retourne la réponse.
  - Si la requête est adressée à un des LC, GC transmet la requête via $Network$.
- Les processus LC_1 et LC_2 reçoivent les requêtes de $Network$, les traitent et retournent les réponses.
- Une entité de l'environnement, souhaitant accéder à l'architecture, envoie un message au GC indiquant la ressource cible (res_1 ou res_2) et l'opération ($READ$ ou $WRITE$) désirée sur cette ressource. Les types d'entités considérées ont des rôles différents : Administrateur ($ADMIN$) ayant tous les droits en lecture et écriture sur (res_1 ou res_2). GC_OWNER ayant les droits en lecture sur (res_1 ou res_2) mais pas en écriture, $LC1\_OWNER$ ayant les droits en lecture et écriture sur (res_1) mais pas sur res_2, $LC2\_OWNER$ ayant les droits en lecture et écriture sur (res_2) mais pas sur res_1.

  Nous considérons un type d'entité particulier $UNKNOWN$ qui caractérise un attaquant qui ne doit avoir aucun droit d'accès sur les ressources. Les règles sont résumées dans le tableau 5.1.

Table 5.1: Droits d'accès aux ressources.

Type d'entité	res_1 (sur LC1)		res_2 (sur LC2)	
	$READ$	$WRITE$	$READ$	$WRITE$
$ADMIN$	oui	oui	oui	oui
GC_OWNER	oui	non	oui	non
$LC1\_OWNER$	oui	oui	non	non
$LC2\_OWNER$	non	non	oui	oui
$UNKNOWN$	non	non	non	non

5.2.2 Scénarios de contexte CDL

Les contextes CDL, mis en oeuvre lors des simulations, décrivent les scénarios d'interaction avec des acteurs act_1, act_2, ..., act_n. Les explorations sont exécutées pour différents ensembles d'interactions avec l'environnement et de différentes complexités en nombre d'interactions. Ces interactions simulent, soit des interactions légitimes d'accès aux ressources de l'architecture, soit des accès non légitimes. Dans le cas de nos mesures pour cette section, nous appliquons des scénarios simples d'un acteur émettant des requêtes séquentiellement. Un exemple de requête est montrée ci-dessous :

```
activity requete is
{
event send_CLT1_PLC1_RED_RES1;
{    event recv_CLT1_PLC1_RED_RES1_ACK []
     event recv_CLT1_PLC1_RED_RES1_NAK
};
}
```

Cette requête spécifie la demande, par l'entité $clt1$, de la lecture de la ressource res_1 localisée sur le composant $comp1$. Suite à l'envoi de la requête, $clt1$ attend un retour de type ACK ou de type NAK.

Pour les mesures, consignées dans le tableau 5.2, nous faisons varier le nombre, de 4 à 122, de requêtes de ce type, en faisant varier le client (1 ou 2), l'opération (*read* ou *write*) et la ressource (res_1 ou res_2).

Nous spécifierons, dans le deuxième cas d'architecture décrite en section 5.4, des interactions plus complexes, ainsi que des scénarios d'attaques que nous ne décrivons pas pour ce cas simple.

Résultats des explorations pour l'architecture non sécurisée

Sur cet exemple simple, nous indiquons, dans le tableau 5.2, le résultat des explorations du modèle de l'architecture. Ils indiquent la complexité en nombre de configurations et de transitions du graphe (SdT) de comportement généré selon les différents scénarios impliqués. Ces ensembles d'interaction sont caractéristiques du trafic entre l'environnement et l'architecture.

Dans la colonne de droite, nous indiquons la longueur moyenne des traces d'exécution. Elle correspond au nombre moyen de transitions dans le graphe d'un ensemble de traces (les plus à gauche choisies arbitrairement dans le graphe) débutant à partir de l'état initial du modèle jusqu'à des configurations feuilles du SdT généré par OBP. Il est à noter, qu'en toute rigueur, il aurait fallu mesurer la longueur de toutes les traces du SdT, car elles sont de longueur différentes, pour pouvoir faire une analyse plus fine du rapport entre longueur des traces dans le cas non sécurisé et cas sécurisé. Mais ici, nous nous intéressons à un ordre de grandeur de ce rapport.

5.2.3 Formalisation des propriétés

La spécification des propriétés vérifiées dans le cadre de cette expérimentation suit les notations formelles décrites aux paragraphes 3.2 et 5.1.3.

Table 5.2: Explorations OBP sur l'exemple d'architecture simple non sécurisée.

Contextes Nb d'interactions	Nb de configurations	Nb de transitions	Longueur moyenne des exécutions
4	227	365	26
6	731	1 277	48
8	1 609	2 745	70
10	2 771	4 769	92
12	4 247	7 349	114
14	6 037	10 485	136
16	8 757	15 511	158
18	12 103	21 705	180
20	15 937	28 791	202
22	20 259	36 769	224
24	25 069	45 639	246
26	30 367	55 401	268
74	307951	568 809	796
122	874 687	161 049	1 324

Les propriétés concernant la disponibilité du traitement des requêtes expriment qu'une requête autorisée aboutit au traitement souhaité. La forme générale d'une propriété est comme celle décrite en section 3.2.3 et que nous rappelons ici.

$$\Box[\ envoi\ requete\ \Rightarrow\ \Diamond\ reponse] \tag{5.4}$$

Considérons une propriété qui spécifie que l'envoi ($send$), par l'entité $clt1$, d'une requête portant la constante $cte\_rd\_clt1\_lc1\_res1$, est acceptée ou refusée. Si la requête est acceptée, la réponse porte la constante $cte\_clt1\_lc1\_res1\_ack$. Si elle est refusée, elle porte la constante $cte\_clt1\_lc1\_res1\_nak$.

Comme décrit en section 3.2.3, et à l'image de la propriété 3.1, la propriété $pty\_rd\_clt1\_lc1\_res1$ (5.5) s'exprime formellement de la manière suivante :

$$\begin{aligned}&\textbf{pty\_rd\_clt1\_lc1\_res1 :}\\&\Box\ [evt\_send\ (clt1,\ cte\_rd\_clt1\_lc1\_res1)\\&\Rightarrow\ \Diamond\ ((evt\_receive\ (clt1,\ cte\_clt1\_lc1\_res1\_ack)\ \vee\\&(evt\_receive\ (clt1,\ cte\_clt1\_lc1\_res1\_nak))]\end{aligned} \tag{5.5}$$

Nous pouvons exprimer cette propriété sous la forme d'une propriété de vivacité qui

est spécifiée en CDL de la manière suivante (5.6) :

$$
\begin{array}{l}
\textbf{propertyLTL } pty\_rd\_clt1\_lc1\_res1 \textbf{ is} \\
\{ \\
[\,]\,(\,|\,evt\_send\_clt1\_cte\_rd\_clt1\_lc1\_res1\,| \\
=>\,<>\;|\,(evt\_receive\_clt1\_cte\_clt1\_lc1\_res1\_ack\;\vee \\
\quad evt\_receive\_clt1\_cte\_clt1\_lc1\_res1\_nak)\,|)) \\
\}
\end{array}
\tag{5.6}
$$

5.3 Sécurisation de l'architecture simple

Nous considérons maintenant une intégration des mécanismes de sécurité sur l'architecture précédente (cf figure 5.4). L'intégration des patrons suit les principes préconisés et décrits au chapitre 4. Conformément aux règles de sécurité proposées précédement, nous intégrons dans les entités GC et $Network$, les mécanismes SAP, CHP et $Firewall$. Dans les entités $LC1$ et $LC2$, nous intégrons les mécanismes SAP, CHP et $Auth$. Sur ce cas d'étude, nous devons vérifier un ensemble de propriétés dépendant des règles choisies pour la politique de sécurité. Elles concernent le contrôle d'accès (disponibilité) aux ressources de l'architecture, la disponibilité des traitements de requêtes (disponibilité) et le contrôle des signatures (Confidentialité, Intégrité). Les propriétés, concernant l'accès aux ressources de l'architecture, contrôlent les droits d'accès des ressources de LC1 et LC2 en lecture et écriture. Elles sont encodées par des invariants CDL.

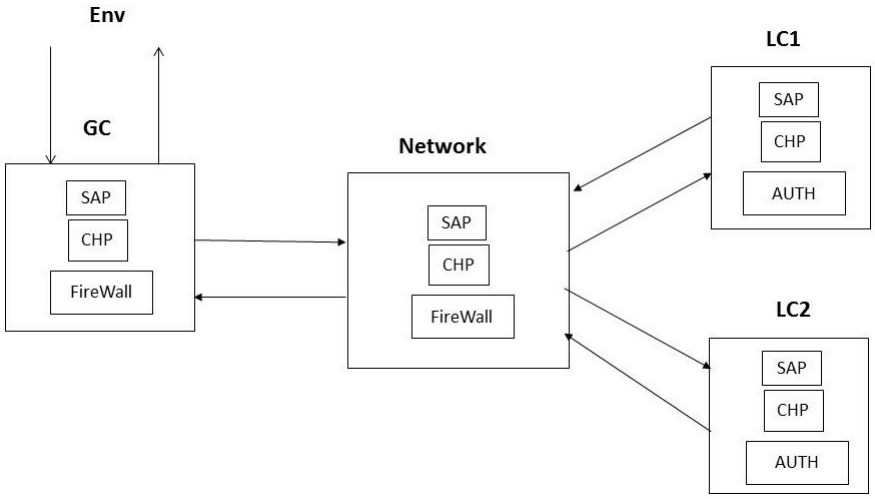

Figure 5.4: Architecture simple sécurisée.

D'autres propriétés concernent le contrôle des signatures. Chaque requête provenant de l'environnement doit être signée par les composants de type SAP. Chaque requête dispose d'un champ signature composé de 3 éléments correspondant aux signatures apposées aux différents niveaux de l'architecture : GC, $Network$ et LC. Une requête destinée au GC est signée par GC. Une requête à destination des $LC1$ ou $LC2$ est signée par GC et $Network$. Les propriétés à vérifier spécifient que les signatures ont bien été apposées avant tout traitement de la requête. Elles sont encodées par des invariants CDL.

Nous ne détaillons pas plus dans cette section les propriétés de sécurité pour cette architecture. Nous décrirons les propriétés de sécurité pour le deuxième cas d'architecture en section 5.4.

Résultats des explorations pour l'architecture sécurisée

Pour les explorations, nous appliquons les mêmes scénarios que pour le cas de l'architecture non sécurisée car nous voulons étudier l'impact de l'intégration des mécanismes de sécurité sur la complexité des explorations. Nous présentons les résultats des explorations dans le tableau 5.3. Nous indiquons également la longueur moyenne des chemins d'exécutions dans le graphe d'exploration qui, comme pour le cas précédent, dénote de la complexité d'éxécution pour l'ensemble d'interactions. Nous indiquons, dans la dernière colonne, le rapport entre les longueurs moyennes d'exécution du cas sécurisé et celles du cas non sécurisé.

Table 5.3: Explorations OBP sur l'exemple d'architecture simple sécurisée.

Contextes Nb d'interactions	Nb de configurations	Nb de transitions	Longueur moyenne des exécutions	Rapport sécurisé/ non sécurisé
4	497	827	44	1,692
6	1 859	3 227	84	1,750
8	3 754	6 570	118	1,686
10	6 263	10 997	152	1,652
12	9 959	17 597	192	1,684
14	13 921	24 633	226	1,662
16	19 359	34 534	260	1,646
18	27 445	49 485	300	1,667
20	35 401	64 035	334	1,653
22	45 813	83 251	374	1,670
24	55 843	101 580	408	1,659
26	68 581	125 061	448	1,672
74	717 109	1 322 757	1 336	1,678
122	2 047 477	3 783 909	2 224	1,680

Pour les scénarios choisis dans cette expérimentation, nous constatons que le rapport entre la longueur des traces d'exécution dans le cas sécurisé et non sécurisé est stable. Ceci s'explique ici dans la mesure où les traitements qui sont exécutés lors des requêtes sont impliquent les mêmes exécutions au sein des automates. Pour les expérimentations impiquant des ensembles de scénarios plus divers mélangeant scénarios nominaux et scénarios d'attaques de différents types, ce rapport évolue différemment.

5.4 Architecture Avancée

Nous considérons maintenant une architecture plus complexe, dans laquelle les mécanismes de sécurité ont été intégrés (figure 5.5). Cette architecture correspond par exemple à un

modèle abstrait d'une architecture vétronique d'un véhicule [Sti06].

5.4.1 Description de l'architecture et fonctionnement nominal

L'architecture (cf figure 5.5) est composée de 12 processus :

- 2 contrôleurs globaux GCS_1 et GCS_2. GCS_1 et GCS_2 disposent tous les deux de deux ressources res_1 et res_2.
- 2 réseaux de communication NET_1 et NET_2 qui font le lien entre les contrôleurs globaux et les contrôleurs locaux.
- 4 contrôleurs locaux PLC_1, PLC_2, PLC_3 et PLC_4, connectés au réseau NET_1.
- 4 devices DEV_1, DEV_2, DEV_3 et DEV_4, connectés au réseau NET_2. Les 4 devices disposent tous les quatre de deux ressources res_1 et res_2.

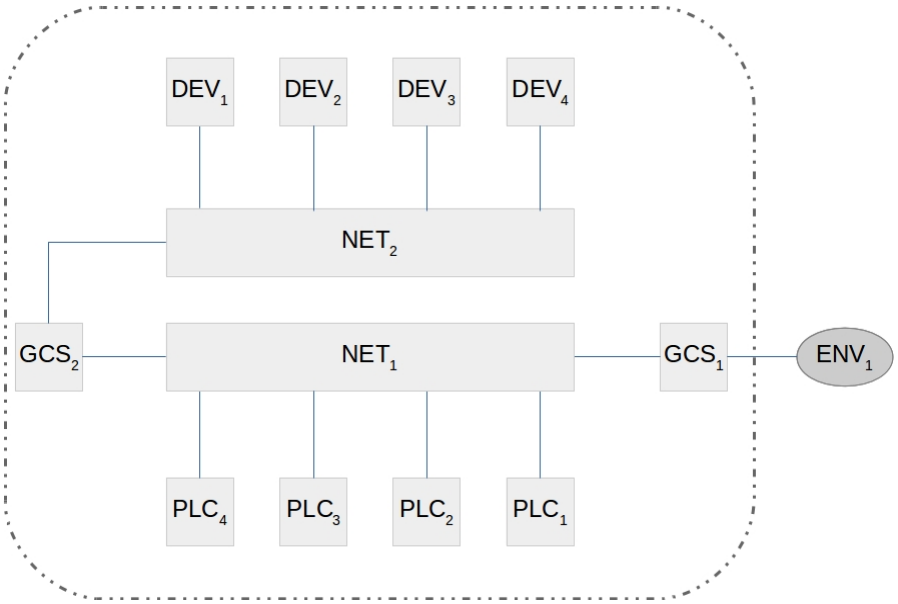

Figure 5.5: Architecture avancée.

Le fonctionnement de l'architecture en mode nominal (sans attaque) est le suivant:

- L'environnement, représentant la partie externe de l'architecture, contient des clients (ou acteurs). Ces clients (Clt) sont à l'origine des requêtes légitimes. Les clients reçoivent, en réponse à leurs requêtes, des réponses avec un atribut ACK ou NAK selon le résultat de la requête. Dans notre cas, les requêtes impliquent uniquement des opérations de lecture (RD) et d'écriture (WR) sur les ressources res_1 et res_2 détenues par les composants de type GCS et DEV. Mais nous pourrions étendre les capacités de l'architecture pour d'autres opérations, plus complexes, d'accès à des données ou d'exécution de traitements particuliers, sans pour cela modifier en profondeur les principes évoqués dans ce travail. Dans un tel cas, les mécanismes et les règles de sécurité devraient être adaptés aux types de requêtes appliquées à l'architecture.

- GCS_1 : un processus contrôleur central reçoit les requêtes provenant de l'environnement. Il relaye les messages vers les PLC et GCS_2 au travers du réseau simulé par le processus NET_1.

- GCS_2 : un processus contrôleur central relaye les messages vers les processus device DEV au travers du réseau simulé par le processus NET_2.

- NET_1 et NET_2 : 2 processus qui simulent les routeurs de communication. Ils ne contiennent pas de ressources accessibles.

- PLC_1 à PLC_4 : 4 processus qui simulent des contrôleurs locaux assurant des accès indirectes vers des ressources détenues par les DEV.

- DEV_1 à DEV_4 : 4 processus qui simulent des entités détenant chacune des ressources res_1 et res_2.

Une requête est envoyée par une entité cliente (un acteur) de l'environnement vers un device DEV. Cette requête transite par un PLC, qui envoit la requête à un DEV suivant la règle suivante : Une requête adressée à DEV_i ($\forall\ i \in \{1 \dots 4\}$) transite par PLC_i ($\forall\ i \in \{1 \dots 4\}$). La réponse de DEV_i transite également par PLC_i.

Par exemple, le client Clt_1 envoie une requête (Clt_1, DEV_1, RD, res_1) qui transite par PLC_1 et est envoyée à DEV_1. Elle spécifie une demande d'accès en lecture à la ressource res_1. DEV_1 renvoie une réponse ACK ou NAK en fonction de la disponibilité de la ressource res_1 concernée par la requête. Cette réponse est renvoyé à CLT_1 via PLC_1.

5.4.2 Scénarios nominaux

Nous présentons les résultats des explorations avec l'outil OBP dans le tableau 5.4. Comme pour la première architecture, les explorations sont exécutées pour différents ensembles d'interactions avec l'environnement et de différentes complexités en nombre d'interactions.

Pour ces mesures, nous avons considéré des scénarios nominaux où les acteurs Clt_i ($\forall i \in \{1 \dots 4\}$) émettent des requêtes vers les composants DEV_i ($\forall i \in \{1 \dots 4\}$).

Un exemple de requête CDL est montrée ci-dessous :

```
activity requete_1_1 is
{
event send_CLT1_DEV1_RD_RES1;
{    event recv_CLT1_DEV1_RD_RES1_ACK []
     event recv_CLT1_DEV1_RD_RES1_NAK
};
}
```

Cette requête spécifie que le client Clt_1 envoit une requête de lecture de la ressource res_1 détenues par DEV_1. DEV_1 est susceptible de renvoyer à Clt_1 une réponse ACK ou NAK en fonction de la disponibilité de la ressource res_1 concernée par la requête.

5.4.3 Politique de sécurité et sécurisation de l'architecture

Compte tenu du choix de cette architecture, nous considérons maintenant une politique de sécurité à lui appliquer et des choix d'intégration des mécanismes pour sécuriser l'architecture.

5.4.3.1 Choix d'une politique de sécurité

Nous considérons, pour notre cas d'étude, une politique de sécurité dont des exemples de règles sont énoncées ici :

- Toutes les ressources intégrées dans GCS_1, GCS_2, DEV_1, DEV_2, DEV_3 et DEV_4 sont protégées pour toutes les opérations d'accès RD et WR à leur ressources res_1 et res_2 .
- Les entités clientes de l'environnement Clt_i ($\forall\, i \in \{1 \dots 4\}$) ont les droits ($RD$, WR) sur les ressources intégrées dans les DEV_i ($\forall\, i \in \{1 \dots 4\}$).
- GCS_1 a le droit d'écriture (et non en lecture) sur les ressources gérées par GCS_2.
- GCS_2 a le droit de lecture (et non en écriture) sur les ressources gérées par GCS_1. GCS_2 contrôle les accès aux ressouces des PLC.
- NET_1 refuse tous les messages originaires d'entités non reconnues comme dans le cas d'attaques.
- NET_2 refuse toutes les messages qui n'ont pas comme source ni un PLC_i, ni un DEV_i ($\forall\, i \in \{1 \dots 4\}$).

5.4.3.2 Choix et intégration des mécanismes de sécurité

La sécurisation de l'architecture a pour but le respect des exigences d'intégrité, de confidentialité, de disponibilité. Comme pour le cas l'architecture simple vue précédemment, et conformément aux règles proposées et décrites chapitre 4, nous intégrons dans les entités de type GCS et PLC, les mécanismes SAP, CHP et $AUTH$. Dans les entités de type NET, nous intégrons les mécanismes SAP, CHP et $FireWall$.

Tous les composants vérifient que les signatures ont bien été apposées sur les requêtes reçues. Par exemple, si DEV_1 reçoit une requête, elle doit avoir été signée par GSC_1, NET_1, PLC_1, GSC_2. NET_2 vérifie les signatures de tous les messages reçus avant d'autoriser l'accès à DEV_1. Nous n'appliquons pas de patrons de sécurité sur les composants de type DEV.

A ce niveau, différents choix peuvent être faits pour l'implantation des mécanismes de sécurité. Nous rappelons que nous ne cherchons pas, dans ce travail, à valider leur pertinence. Nous cherchons à vérifier que, pour une politique de sécurité donnée et pour un choix d'implantation des mécanismes, les règles de sécurité sont satisfaites sur le modèle d'architecture.

5.4.4 Scénarios d'attaque CDL

Nous considérons maintenant les scénarios avec des interactions d'entités non légitimes de l'environnement (attaquants). Nous considérons 3 types d'attaquants : ATT, SPM et PEN qui sont susceptibles d'émettre des requêtes vers l'architecture :

- ATT : Les requêtes sources d'un ATT sont transmises à leur destinataire mais sont refusées au niveau du destinataire conformément aux règles de la politique de sécurité considérée. Un message de réponse (NAK) est renvoyé à l'environnement.
- SPM : Les requêtes sources d'un SPM sont considérés comme des spams et refusées par NET_1 qui renvoit un message de réponse (NAK) à l'environnement.

- PEN : Les requêtes sources d'un PEN sont générées par un attaquant supposé interne à l'architecture. Celles-ci sont donc détectées à l'aide du contrôle des signatures. Aucun message de réponse n'est renvoyé à l'environnement.

Les scénarios d'attaques sont spécifiées, comme pour les scénarios nominaux, en langage CDL. Un exemple de requêtes CDL de type ATT est spécifiée ci-dessous :

```
activity requete_att is
{
event send_ATT1_DEV1_RD_RES1;
event recv_ATT1_DEV1_RD_RES1_NAK
}
```

Elle spécifie une tentative d'accès en lecture à la ressource res_1 détenu par DEV_1. L'environnement est susceptible de recevoir un message d'erreur de type NAK.

Un exemple de requêtes CDL de type SPM est spécifiée ci-dessous :

```
activity requete_spm is
{
event send_SPM1_DEV1_RD_RES1;
event recv_SPM1_DEV1_RD_RES1_NAK
}
```

Elle spécifie une tentative d'accès en lecture à la ressource RES_1 détenu par DEV_1. L'environnement est susceptible de recevoir un message d'erreur de type NAK.

Un exemple de requêtes CDL de type PEN est spécifiée ci-dessous : Ce type d'attaque s'effectue directement dans les canaux de communication internes de l'architecture.

```
activity requete_pen is
{
event send_PEN1_DEV1_RD_RES1
}
```

Elle spécifie une tentative d'accès en lecture à la ressource res_1 détenu par DEV_1. L'environnement ne reçoit pas de message d'erreur.

5.4.5 Résultats des explorations pour l'architecture sécurisée

Nous présentons les résultats des explorations du modèle avec l'outil OBP dans les tableaux 5.4 et 5.5 respectivement pour des scénarios nominaux et des scénarios d'attaque de type ATT et SPM. Pour un gain de place, nous ne montrons pas ici de résultats pour les explorations avec les scénarios de type PEN. Ils indiqueraient des complexités plus faibles en nombre de configurations et de transitions puisque les messages de réponses ne sont pas renvoyés à l'environnement.

Table 5.4: Explorations pour l'architecture de avancée non sécurisée et sécurisée.

	Architecture non sécurisée			Architecture sécurisée		
Contextes	**nb. de confs**	**nb. de trans**	**Depth**	**nb. de confs**	**nb. de trans**	**Depth**
scn_1_1	49	48	48	63	62	62
scn_1_2	97	96	96	125	124	124
scn_1_3	145	144	144	187	186	186
scn_1_4	193	192	192	249	248	248
scn_2_1	2 344	4 415	96	3 686	6 977	124
scn_2_2	11 207	21 424	192	17 755	34 028	248
scn_2_3	26 590	51 027	288	42 208	81 153	372
scn_2_4	48 493	93 224	384	77 045	148 352	496
scn_3_1	110 137	299 913	144	204 621	560 051	186
scn_3_2	1 394 467	3 871 044	288	2 623 843	7 301 556	372
scn_3_3	5 396 320	15 039 747	432	10 183 726	28 435 959	558
scn_3_4	13 605 189	37 982 760	576	25 707 157	71 886 920	744
scn_4_1	5 114 765	17 939 618	192	10 953 437	38 624 884	248
scn_4_2	180 143 913	646 742 856	384	392 578 705	1 412 820 112	496
scn_4_3	1 150 191 955	4 145 224 977	576	2 515 317 972	9 082 222 251	744

5.4.5.1 Explorations avec un contexte en mode nominal

Pour ces mesures consignées dans le tableau 5.4, les scénarios utilisés décrivent des interactions de plusieurs clients Clt, chacun émettant plusieurs requêtes.

Les 15 scénarios du tableau, référencés avec la notation scn_i_i, correspondent aux scénarios tels que : scn_i_j ($\forall\ i,\ j \in \{1 \dots 4\}$) impliquent i acteurs et j messages par acteurs. Par exemple : scn_2_1 impliquent les acteurs Clt_1 et Clt_2. Chaque acteur envoie un seul message.

De même que pour notre premier cas d'étude, le rapport des longueurs moyennes des traces dans l'architecture sécurisé et non sécurisé est ici égal en moyenne à 1.3 pour tous les scénarios. Là aussi, la complexité dédiée à la sécurité de l'architecture ou le temps passé à la sécurité n'est pas proportionnelle au traffic.

La figure 5.6 illustre, à partir des mesures du tableau 5.4, les rapports en termes de nombre de configurations, de nombre de transitions et de profondeur entre les modèles sécurisés et non sécurisés. Nous illustrons ces rapports pour différents scénarios en faisant varier le nombre d'acteurs et de messages par acteur. Le rapport de profondeur est stable. Cela est dû au fait que le nombre de transitions ajoutées pour répondre à une demande est stable pour les demandes légitimes. Les rapports du nombre de configurations et de transitions sont stables dans le cas d'un acteur. Ces rapports augmentent d'environ 0.3 points à chaque fois que nous ajoutons un nouvel acteur. Cependant, leur augmentation dûe à l'augmentation des messages par acteur est beaucoup moins significative (presque non visible). Ces résultats montrent qu'en utilisant les modèles de sécurité, la complexité de notre modèle augmente mais de manière stable et prévisible. Notez que, dans les cas où des demandes légitimes et non-légitimes sont utilisées, la complexité peut diminuer car la plupart des requêtes non-légitimes sont traitées avec moins de transitions dans le cas du modèle sécurisé que dans celui non-sécurisé.

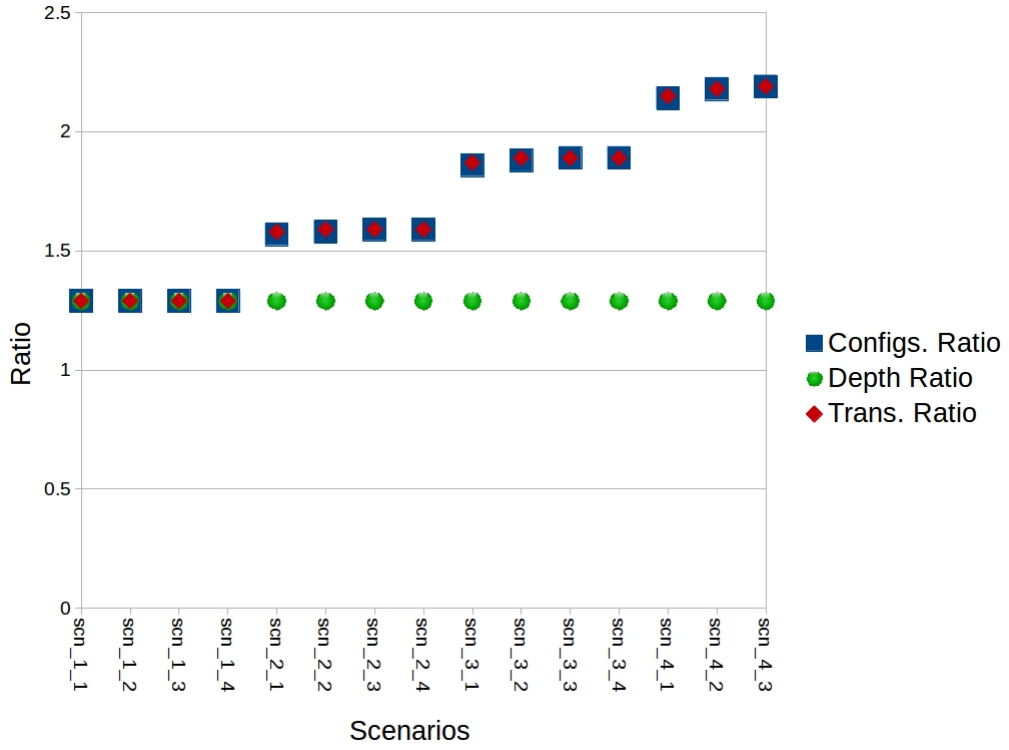

Figure 5.6: Comparaison entre modèle sécurisé et non sécurisé.

5.4.5.2 Explorations avec un contexte en mode attaquant

Pour les mesures consignées dans le tableau 5.5 (partie gauche), les scénarios utilisés décrivent des interactions d'un client Clt, émettant plusieurs requêtes conjointement à un attaquant (ATT) émettant plusieurs requêtes.

Les 12 scénarios de type ATT du tableau 5.5 (partie gauche), référencés avec la notation $sc\_att\_i\_i$, correspondent aux scénarios tels que : $sc\_att\_i\_j$ ($\forall\ i,\ j \in \{1 \dots 4\}$) impliquent un acteur légitime émettant i requêtes et un attaquant de type ATT émettant j requêtes.

5.4.5.3 Explorations avec un contexte en mode spam

Pour les mesures consignées dans le tableau 5.5 (partie droite), les scénarios utilisés décrivent des interactions d'un client Clt, émettant plusieurs requêtes conjointement à un attaquant de type (SPM) émettant plusieurs requêtes.

Les 12 scénarios de type SPM du tableau 5.5 (partie droite), référencés avec la notation $sc\_spm\_i\_i$, correspondent à aux scénarios tels que : $sc\_spm\_i\_j$ ($\forall\ i,\ j \in \{1 \dots 4\}$) impliquent un acteur légitime émettant i requêtes et un attaquant de type SPM émettant j requêtes.

5.4.6 Vérification des propriétés formelles

Les types de propriétés à vérifier sont, d'une part, des propriétés nominales, c'est à dire qui contrôlent l'exécution des fonctionnalités initiales de l'architecture avant la sécurisation. Ces propriétés doivent être préservées dans la nouvelle architecture sécurisée. D'autre part,

Table 5.5: Exploration avec les contextes en modes attaquant et spam.

	Architecture sécurisée				
Contextes	**nb. de confs**	**nb. de trans**	**Contextes**	**nb. de confs**	**nb. de trans**
sc_att_1_1	669	1 134	*sc_spm_1_1*	914	1 629
sc_att_1_2	2 143	3 938	*sc_spm_1_2*	1 765	3 196
sc_att_2_1	2 214	4 095	*sc_spm_2_1*	2 699	4 980
sc_att_2_2	6 023	11 314	*sc_spm_2_2*	5 273	9 836
sc_att_3_1	5 908	11 176	*sc_spm_3_1*	6 633	12 501
sc_att_3_2	14 201	27 030	*sc_spm_3_2*	13 079	24 816
sc_att_4_1	10 465	19 947	*sc_spm_4_1*	11 430	21 712
sc_att_4_2	24 105	46 126	*sc_spm_4_2*	22 611	43 176
sc_att_5_1	17 171	32 888	*sc_spm_5_1*	18 376	35 093
sc_att_5_2	38 307	73 562	*sc_spm_5_2*	36 441	69 876
sc_att_6_1	24 740	47 519	*sc_spm_6_1*	26 185	50 164
sc_att_6_2	54 235	104 378	*sc_spm_6_2*	51 997	99 956

les propriétés de sécurité contrôlent la bonne implantation des mécanismes de sécurité. Comme pour le cas d'étude précédent, ces propriétés de sécurité dépendent des règles choisies pour la politique de sécurité à implanter. Elles concernent le contrôle d'accès (disponibilité) aux ressources de l'architecture, la disponibilité des traitements de requêtes (disponibilité) et le contrôle des signatures (Confidentialité, Intégrité).

Nous passons en revue des exemples de propriétés vérifiées, nominales et de sécurité. Nous abordons également, en 5.4.6.3, la vérification des propriétés dans un mode que nous nommons *stand-alone*. Dans ce mode, les scénarios CDL sont remplacés par une modélisation sous la forme de processus complémentaires Fiacre.

5.4.6.1 Vérification des propriétés nominales

Les propriétés nominales sur cette architecture expriment la bonne prise en compte des requêtes. Suite à l'envoi d'une requête par un acteur de l'environnement, une réponse lui sera retournée comme vu en section 5.2.3.

Les propriétés vérifiées sur notre modèle, et concernant, par exemple, une entité cliente clt_1, sont listées tableau 5.6. Elles spécifient la prise en compte des demandes d'accès en lecture et écriture aux ressources res_1 et res_2 des entités GCS_1, GCS_2, DEV_1, DEV_2, DEV_3 et DEV_4. La réponse retournée à l'entité cliente de l'environnement est, soit de type ACK, soit de type NAK selon les droits d'accès implantés. Par exemple la propriété notée $pty\_rd\_clt1\_gcs1\_res1$ spécifie la demande d'accès, par clt_1, en lecture de la ressource res_1 détenue par GCS_1.

Comme décrit en section 5.2.3, les propriétés sont exprimées par des formules de vivacité LTL. Par exemple, la propriété $pty\_rd\_clt1\_gcs1\_res1$ s'exprime de la manière suivante

Table 5.6: Exemples de propriétés nominales vérifiées impliquant le client clt1.

Propriétés	Opération et ressource	Description
$pty\_op\_clt1\_gcs1\_res_R$	avec $op \in \{RD, WR\}$ et $R \in \{1, 2\}$	Accès par $clt1$ à res_R de GCS_1
$pty\_op\_clt1\_gcs2\_res_R$	avec $op \in \{RD, WR\}$ et $R \in \{1, 2\}$	Accès par $clt1$ à res_R de GCS_2
$pty\_op\_clt1\_dev1\_res_R$	avec $op \in \{RD, WR\}$ et $R \in \{1, 2\}$	Accès par $clt1$ à res_R de DEV_1
$pty\_op\_clt1\_dev2\_res_R$	avec $op \in \{RD, WR\}$ et $R \in \{1, 2\}$	Accès par $clt1$ à res_R de DEV_2
$pty\_op\_clt1\_dev3\_res_R$	avec $op \in \{RD, WR\}$ et $R \in \{1, 2\}$	Accès par $clt1$ à res_R de DEV_3
$pty\_op\_clt1\_dev4\_res_R$	avec $op \in \{RD, WR\}$ et $R \in \{1, 2\}$	Accès par $clt1$ à res_R de DEV_4

(5.7) :

$$\begin{aligned} &\textbf{pty\_rd\_clt1\_gcs1\_res1 :} \\ &\Box\ [evt\_send\ (clt1,\ cte\_rd\_clt1\_gcs1\_res1) \\ &\Rightarrow\ \Diamond\ ((evt\_receive\ (clt1,\ cte\_clt1\_gcs1\_res1\_ack)\ \vee \\ &(evt\_receive\ (clt1,\ cte\_clt1\_gcs1\_res1\_nak))] \end{aligned} \tag{5.7}$$

Dans notre langage CDL, elle est spécifiée de la manière suivante (5.8) :

$$\begin{aligned} &\textbf{propertyLTL}\ pty\_rd\_clt1\_gcs1\_res1\ \textbf{is} \\ &\{ \\ &[\ ]\ (\ |\ evt\_send\_clt1\_cte\_rd\_clt1\_gcs1\_res1\ | \\ &=>\ <>\ |\ (evt\_receive\_clt1\_cte\_clt1\_gcs1\_res1\_ack\ \vee \\ &\quad evt\_receive\_clt1\_cte\_clt1\_gcs1\_res1\_nak)\ |) \\ &\} \end{aligned} \tag{5.8}$$

5.4.6.2 Vérification des propriétés de sécurité

Les propriétés, concernant l'accès aux ressources de l'architecture, res_1 et res_2, contrôlent les droits d'accès (lecture ou écriture) selon les règles de la politique de sécurité choisie. D'autres propriétés concernent le contrôle des signatures devant être apposées par les composants par lesquels transitent les requêtes. Ces deux type de propriétés sont encodées par des invariants CDL.

Pour les scénarios incorporant des attaques (ATT, SPM, PEN), des propriétés de contrôles de signature sont détectées violées, ce qui nous assure que les mécanismes implantées dans les SAP jouent leur rôle. L'architecture sécurisée est capable de détecter les messages illégitimes (ATT, SPM) ou les tentatives de pénétration (PEN).

Propriétés de sécurité de type SAP

Le tableau 5.7 récapitule un sous ensemble des propriétés de sécurité, de type SAP, vérifiées. Ces propriétés ont été décrites en section 3.3.4 pour le mécanisme de sécurité SAP,

Table 5.7: Propriétés de sécurité vérifiées de type SAP.

Propriétés	Localisations	Types de propriétés
$prt\_sap\_1\_loc$	avec $loc \in \{gcs_i, net_i\ (i \in \{1,2\}),$ $plc_j\ (j \in \{1 \dots 4\})\}$	Disponibilité (vivacité)
$prt\_sap\_net\_1.a\_loc$	avec $loc \in \{gcs_i, net_i\ (i \in \{1,2\})\}$	Authenticité (invariant)
$prt\_sap\_net\_1.b\_loc$	avec $loc \in \{gcs_i, net_i\ (i \in \{1,2\})\}$	Authenticité (invariant)
$prt\_sap\_net\_2\_loc$	avec $loc \in \{gcs_i, net_i\ (i \in \{1,2\})\}$	Disponibilité (vivacité)
$prt\_sap\_net\_3\_loc$	avec $loc \in \{gcs_i, net_i\ (i \in \{1,2\})\}$	Disponibilité (vivacité)
$prt\_sap\_net\_4\_loc$	avec $loc \in \{gcs_i, net_i\ (i \in \{1,2\})\}$	Confidentialité (invariant)
$prt\_sap\_c\_1\_loc$	avec $loc \in \{gcs_i\ (i \in \{1,2\}),$ $plc_j\ (j \in \{1 \dots 4\})\}$	Authenticité (invariant)
$prt\_sap\_c\_2\_loc$	avec $loc \in \{gcs_i\ (i \in \{1,2\}),$ $plc_j\ (j \in \{1 \dots 4\})\}$	Disponibilité (vivacité)
$prt\_sap\_c\_3\_loc$	avec $loc \in \{gcs_i\ (i \in \{1,2\}),$ $plc_j\ (j \in \{1 \dots 4\})\}$	Disponibilité (vivacité)

Considérons, comme exemple, la propriété $prt\_sap\_net\_3\_net_2$. Elle s'applique au composant NET_2. Conformément à la spécification de propriété $prt\_sap\_net\_3$ (3.9) vue au paragraphe 3.3.4.2, la propriété est de type vivacité et est spécifiée comme suit :

$$\begin{aligned}&\textbf{prt\_sap\_net\_3\_net}_\mathbf{2}\ :\\ &\forall\ m\ \in \mathcal{M}ess,\\ &\Box\ [evt\_request\ (NET_2,\ FrwReq(m)) \Rightarrow\\ &\Diamond\ evt\_check\ (NET_2,\ FrwReq(m))]\end{aligned} \tag{5.9}$$

Elle signifie que toute requete de transfert de message par NET_2, doit être contrôlée.

Compte tenu des choix d'implantation, la propriété peut être reformulée en langage CDL comme suit :

$$\begin{aligned}&\textbf{propertyLTL}\ prt\_sap\_net\_3\_net\_2\ \textbf{is}\\ &\{\\ &[\,]\ (\ |\ pre\_net\_2\_at\_SAP\ \wedge\ pre\_net\_2\_available\ |\\ &=>\ <>\ |\ pre\_net\_2\_at\_CHP\ |)\\ &\}\end{aligned} \tag{5.10}$$

avec les prédicats :

- $pre\_net\_2\_at\_SAP$ vrai si NET_2 est dans l'état SAP et a reçu un message m,
- $pre\_net\_2\_available = NET_2.mess.source = DEV_i \vee NET_2.mess.target = DEV_i\ (\forall i \in \{1 \dots 4\})$,

- $pre\_net\_2\_at\_CHP$ vrai si NET_2 est dans l'état CHP, c'est à dire le message a été contrôlé par SAP.

La disponibilité du destinataire de message est basée, dans notre implantation, sur la valeur des champs *source* et *target* du message m. D'autres choix d'implantation pourraient être faits pour les automates des composants. Ceci impliquerait une redéfinition des prédicats référencés dans les propriétés.

Un autre exemple de propriété de sûreté est décrit ci-dessous. $prt\_sap\_c\_1\_plc_1$ est appliquée sur le composant PLC_1. Conformément à la spécification de propriété $prt\_sap\_c\_1$ (3.12) vue au paragraphe 3.3.4.3, la propriété, de type invariant, est spécifiée comme suit :

$$\begin{array}{l}\textbf{prt\_sap\_c\_1\_plc}_\textbf{1}\textbf{ :} \\ \forall\ e\ \in\ \mathcal{E}nt,\ \forall\ opRes\ \in\ \mathcal{O}pRes, \\ \Box\ [evt\_access\ (plc_1,\ e,\ opRes)\ \Rightarrow\ pre\_check\ (plc_1, AccReq(\ e,\ opRes))]\end{array} \tag{5.11}$$

Elle signifie que si PLC_1 accède à une de ses ressources, suite à la requête provenant de l'entité e avec l'opération spécifiée par $opRes$, la requête a précédemment été contrôlée par le mecanisme CHP de PLC_1 . Si nous instancions cette propriété pour une requête émise par l'entité clt_1 de l'environnement, et compte tenus des choix d'implantation, la propriété peut être reformulée en langage CDL comme suit :

$$\begin{array}{l}\textbf{assert}\ prt\_sap\_c\_1\_clt1\_plc\_1\ \textbf{is} \\ \{ \\ |\ pre\_access\_plc\_1\_clt1\_opRes\ |\ =>\ |\ pre\_check\_c\_plc\_1\_clt1\_opRes\ | \\ \}\end{array} \tag{5.12}$$

Propriétés de sécurité de type CHP

Le tableau 5.8 récapitule un sous ensemble des propriétés de sécurité, de type CHP, vérifiées. Ces propriétés ont été décrites en section 3.4.4 pour le mécanisme de sécurité CHP.

Table 5.8: Propriétés de sécurité vérifiées de type CHP.

Propriétés	Localisations	Types de propriétés
$prt\_chp\_1\_loc$	avec $loc \in \{gcs_i, net_i\ (i \in \{1,2\}),$ $plc_j\ (j \in \{1 \ldots 4\})\}$	Sécurité P3 (vivacité)
$prt\_chp\_2\_loc$	avec $loc \in \{gcs_i, net_i\ (i \in \{1,2\}),$ $plc_j\ (j \in \{1 \ldots 4\})\}$	Sécurité P4 (vivacité)
$prt\_chp\_3\_loc$	avec $loc \in \{gcs_i, net_i\ (i \in \{1,2\}),$ $plc_j\ (j \in \{1 \ldots 4\})\}$	Sécurité P5 (invariant)
$prt\_chp\_4\_loc$	avec $loc \in \{gcs_i, net_i\ (i \in \{1,2\}),$ $plc_j\ (j \in \{1 \ldots 4\})\}$	Sécurité (invariant)

Un exemple de propriété de type CHP est décrit ci-dessous. $prt\_chp\_2\_gcs_1$ est appliquée sur le composant GCS_1. Conformément à la spécification de propriété prt_chp_2

(3.16) vue au paragraphe 3.4.4, la propriété, de type vivacité, est spécifiée comme suit :

$$\begin{aligned}&\textbf{prt\_chp\_2\_gcs}_\mathbf{1}:\\ \forall\, m \in \mathcal{M}ess,&\\ &\Box\,[evt\_verify\ (GCS_1,\ FrwReq(m))\ \wedge\ (\neg\ policy\_conform\ (GCS_1,\ FrwReq(m)))\\ &\Rightarrow\ \Diamond\ (evt\_trigAct\ (GCS_1, cm) \wedge cm = counterOf\ (GCS_1,\ FrwReq(m)))]\end{aligned} \tag{5.13}$$

Elle signifie que si la requête de relayage reçue par GCS_1 est détectée, par le mécanisme CHP implanté sur GCS_1, non conforme à la politique de sécurité choisie , alors fatalement une action de contre mesure appropriée est déclenchée. Compte tenus des choix d'implantation, la propriété peut être reformulée en langage CDL comme suit :

$$\begin{aligned}&\textbf{propertyLTL}\ prt\_chp\_2\_gcs_1\ \ \textbf{is}\\ &\{\\ &[\,]\ (\ |\ pre\_gcs\_1\_at\_chp\ \wedge \neg pre\_gcs\_1\_conform\ |\\ &=><>\ |\ pre\_gcs\_1\_at\_TrigAct)\ |\ )\\ &\}\end{aligned} \tag{5.14}$$

avec les prédicats :

- $pre\_gcs\_1\_at\_chp$ vrai si GCS_1 est dans l'état CHP,
- $pre\_gcs_1\_conform$ vrai si la requête est conforme à la politique de sécurité choisie,
- $pre\_gcs\_1\_at\_TrigAct$ vrai si GCS_1 est dans l'état $TrigAct$, c'est à dire une action de contre mesure est déclenchée.

Propriétés de sécurité de type AUTH

Le tableau 5.9 récapitule un sous ensemble des propriétés de sécurité, de type $AUTH$, vérifiées. Ces propriétés ont été décrites en section 3.5.4 pour le mécanisme de sécurité $AUTH$,

Table 5.9: Propriétés de sécurité vérifiées de type AUTH.

Propriétés	Localisations	Types de propriétés
$prt\_auth\_1\_loc$	avec $loc \in \{gcs_i\ (i \in \{1,2\}),$ $plc_j\ (j \in \{1\ldots4\})\}$	Disponibilité (vivacité)
$prt\_auth\_2\_loc$	avec $loc \in \{gcs_i\ (i \in \{1,2\}),$ $plc_j\ (j \in \{1\ldots4\})\}$	Sûreté (invariant)

Propriétés de sécurité de type FWL

Le tableau 5.10 récapitule un sous ensemble des propriétés de sécurité, de type FWL, vérifiées. Ces propriétés ont été décrites en section 3.6.4 pour le mécanisme de sécurité FWL,

Table 5.10: Propriétés de sécurité vérifiées de type FWL.

Propriétés	Localisations	Types de propriétés
$prt\_firewall\_1\_loc$	avec $loc \in \{gcs_i, net_i \ (i \in \{1,2\})\}$	Disponibilité (vivacité)
$prt\_firewall\_2\_loc$	avec $loc \in \{gcs_i, net_i \ (i \in \{1,2\})\}$	Sûreté (invariant)

5.4.6.3 Vérification des propriétés en mode stand-alone

Une critique pouvant être formulée sur les vérifications précédentes concernent la complétude des scénarios appliqués à l'architecture. Ceux-ci sont non cycliques et imposent des limites en profondeur de l'arbre représentant le contexte interagissant avec l'architecture. Les travaux, portés sur l'outillage OBP, en l'occurrence la technique de *splitting* [DRB11, DBRL12] et plus récemment de $pastFreeze$ [CT16] ont permis une amélioration très efficace de la complexité en gérant au mieux les scénarios de l'environnement.

Mais il est parfois utile de conduire des explorations du modèle avec des comportements de l'environnement en boucles infinies. Ceci peut se justifier pour des cas de propriétés à vérifier qui sont indépendantes des données véhiculées par les scénarios.

Dans cette approche complémentaire, nous vérifions donc les propriétés dans un mode (que nous nommons stand-alone) où nous appliquons à l'architecture des scénarios cycliques. Nous ne réglons toujours pas le problème de complétude, mais nous implantons cette technique comme complémentaire à la précédente qui implique, elle, des scénarios non cycliques.

Pour cette implantation, nous intégrons des acteurs spécifiés en langage $FIACRE$, comme des processus intégrant le réseau de processus de l'architecture. Dans ce cas, CDL n'est utilisé que pour l'expression des propriétés (invariants, observateurs, formules LTL) devant être vérifiées. Chaque processus qui simule un acteur, boucle sur l'envoi de requêtes et l'attente de réponses. Ces acteurs simulent des clients légitimes ou des attaquants.

Dans les expérimentations, nous implantons des instances qui envoient, en boucle infinie, soit toujours la même requête (figure 5.7.a) ou soit des requêtes différentes choisies parmi une liste finie de requêtes (figure 5.7.b). Le point important dans nos expérimentations est de choisir les bons scénarios au regard des propriétés que l'on souhaite vérifier.

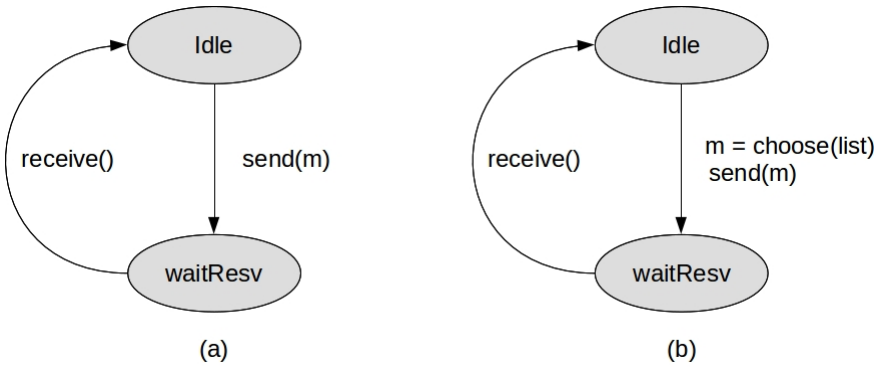

Figure 5.7: Automates d'un processus représentant un acteur de l'environnement.

Nous indiquons ici la complexité d'une exploration dans un cas spécifique. Considérons, que les acteurs de l'environnement considéré sont au nombre de 4 (1 client et 3 attaquants de type ATT, SPM et PEN). clt boucle sur l'envoi de 4 requêtes, att boucle sur l'envoi

de 4 requêtes, *spm* boucle sur l'envoi d'une requête, *pen* boucle sur l'envoi d'une requête. Le nombre de configurations est alors de 2 299 680 et le nombre de transitions est de 7 952 215.

5.5 Discussion sur les expérimentations

Ces expérimentations nous ont conforté dans la capacité à conduire, par cette méthode et technique, un processus d'intégration des mécanismes de sécurité et de validation formelle des modèles d'architectures générées. Mais compte tenu du nombre important de propriétés et de scénarios impliqués dans ce processus, et à prendre en compte dans la vérification, l'approche n'a de sens que si nous sommes capables de générer ces ensembles de propriétés et de scénarios.

Durant les expérimentations, nous nous sommes restreints au nombre limité de quatre patrons. Mais une démarche similaire (formalisation, composition, vérification) pourrait être entreprise pour l'intégration d'autres patrons décrits dans la littérature. Nous nous sommes également focalisés sur des ensembles de propriétés qui pourraient être étendus pour chaque patron.

Les expérimentations ont été conduites sans réflexion très poussée de la stratégie d'implantation des mécanismes de sécurité. C'est une des perspectives de ce travail qui doit être envisagée pour une optimisation de la stratégie au regard, par exemple, de critères de performances. L'idée, à poursuivre, est d'en déduire des préconisations de couplage des patrons qui influent sur la complexité et la robustesse face aux attaques. Pour une politique de sécurité donnée, la question est : Quelle méthode d'intégration des patrons préconisons nous ? Quelle est l'impact sur la complexité et sur la robustesse face aux attaques ? Quels sont les compromis à trouver ?

CHAPTER 6

Discussion et Perspectives

6.1 Discussion

Synthèse du travail

Dans ce travail, nous avons cherché à étudier une méthode et des concepts pour valider la robustesse de modèles d'architecture subissant des attaques potentielles. Nous avons considéré dans nos expérimentations des modèles abstraits d'architectures sous la forme de réseau d'entités communicantes. Même si nous n'avons pas pris en compte les spécificités des architectures SCADA, nos modèles doivent pouvoir s'adapter.

Notre objectif était de fournir des outils pour valider formellement un modèle d'architecture (*architecture sécurisée*) devant respecter une politique de sécurité donnée. Cette validation passe par la vérification des propriétés formelles du modèle de l'architecture, vérification basée sur une technique de *model-checking*. Les propriétés sont d'ordre fonctionnelle au sens où elle correspondent aux exigences initiales de l'architecture avant sécurisation. Ces propriétés doivent être préservées lors de la sécurisation. Mais aussi, elles correspondent aux exigences de sécurité qui découlent du choix de la politique de sécurité à mettre en œuvre sur l'architecture.

La sécurisation de l'architecture est basée sur une librairie de patrons de sécurité qui sont largement décrits dans la littérature. L'idée menée est d'automatiser la génération d'une architecture en composant le modèle de l'architecture initiale et un ensemble de patrons de sécurité répondant à la politique choisie.

Durant ce travail nous avons tenté de répondre aux enjeux décrits en introduction de ce rapport. Le premier est la formalisation des mécanismes de sécurité, basés sur les patrons, qui implantent les règles de sécurité souhaitées. Chaque patron de sécurité est décrit par une description formelle de sa structure et de son comportement, ainsi qu'une description formelle des propriétés de sécurité associées à ce patron.

Le deuxième enjeu est la capacité à intégrer ces mécanismes de sécurité au sein d'une architecture à sécuriser. Nous avons cherché à identifier des règles de composition qui permettent la transformation du modèle initial et la génération d'un modèle d'architecture sécurisée. Dans notre travail, nous avons opté pour des choix d'implantation qui peuvent se discuter. Un travail complémentaire doit être conduit pour optimiser ces implantations et identifier les bons critères de comparaison entre les différentes stratégies. La forme des automates choisie à un impact sur l'exploration des modèles par le nombre d'états et l'asynchronisme émergeant des implantations.

Le dernier enjeu consiste à valider formellement le résultat de cette génération aux regard des exigences, fonctionnelles et de sécurité.

Nos travaux sont illustrés dans des cas d'étude qui nous permettent d'expliquer la démarche et le processus à mettre en œuvre d'un point de vue ingénierie logicielle.

Bilan

Ce travail a montré qu'à partir d'une formalisation des règles de sécurité souhaitées et des mécanismes de sécurité à intégrer, nous sommes capables, d'une part, de générer, à partir d'un modèle initial, une nouveau modèle répondant aux exigences et, d'autre part, de valider formellement le modèle. La technique du vérification par *model-checking* est un bon support pour la validation formelle à condition de savoir gérer la complexité intrinsèque à cette technique. Elle nous permet de démontrer formellement le respect des propriétés sur le modèle de l'architecture.

Le bilan que nous tirons de ce travail laisse entrevoir plusieurs axes à poursuivre et qui est illustré figure 6.1.

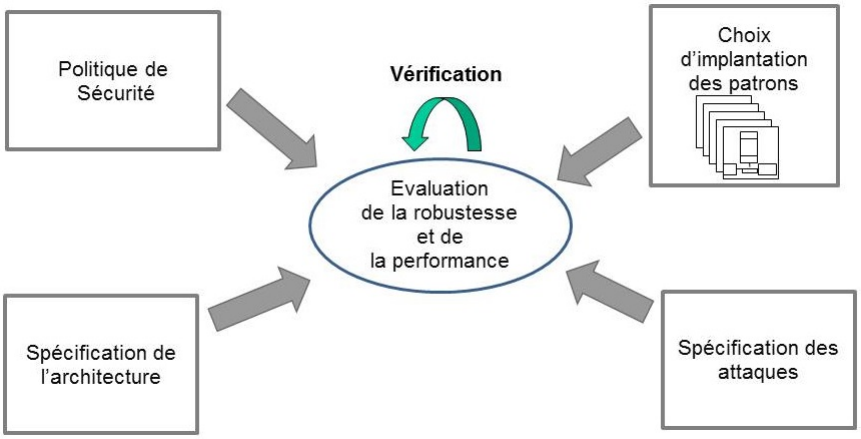

Figure 6.1: Paramètres du processus d'évaluation formelle.

Le but est d'évaluer la robustesse de modèles d'architectures subissant des attaques potentielles. En nous situant même dans le contexte restreint des architectures SCADA, nous constatons que beaucoup de paramètres rentre en ligne de compte dans ce processus d'évaluation. Tout d'abord, il importe que considérer le type d'architecture initialement prise en compte au démarrage du processus de sécurisation. Dans nos travaux, nous avons opté pour des modèles abstraits et simples. Mais d'autres formes d'architectures doivent pouvoir être considérées, avec des comportements, synchrones ou asynchrones, avec des dispositifs plus sujets aux attaques par canaux cachés, etc. Il serait utile d'étudier comment notre approche peut résister à de telles formes architecturales et comment adapter cette approche. Ensuite, la politique de sécurité choisie regroupe un ensemble de concepts et de règles qui peuvent générer des ensembles vastes de propriétés et de différentes formes. Cette politique impacte fortement les types de patrons à intégrer et la façon dont les patrons cohabitent au sein de l'architecture. Aussi, la manière d'implanter et d'intégrer les mécanismes de sécurité est à prendre en compte comme nous l'avons précédemment évoqué. Elle peut avoir des conséquences bénéfiques ou néfastes sur la qualité des architectures générées. Enfin, nous avons vu que la modélisation des attaques est fondamentale car ce sont elles qui dirigent le processus de sécurisation. La forme des scénarios d'attaques doit être envisagée de manière large au niveau de la performance des attaquants qui se renouvelle en permanence.

6.2 Perspectives

Durant notre travail, nous avons été confrontés à cette vaste problématique où beaucoup de paramètres sont à prendre en compte pour répondre au besoin de sécurisation des architectures.

Les perspectives de ce travail concernent plusieurs axes à prendre en compte dans différentes directions. Ils concernent la prise en compte des politiques de sécurité qui peuvent être complexes, être définies de manière dynamique, porter sur des niveaux différents des architectures (accès aux ressources, aux données, aux canaux de communication). Ces

politiques doivent reposer sur une formalisation indispensable à partir de laquelle, un processus de génération de modèle d'architecture peut être envisagé. Un choix plus vaste de mécanismes de sécurité va s'imposer ainsi que l'automatisation du processus d'intégration et de composition de patrons plus nombreux et différents. Aussi, les scénarios d'attaques plus complexes doivent être envisagés

Ceci touche à des aspects formalisation de la politique, des exigences, des mécanismes de sécurité candidats, mais aussi aux choix des implantations. Nous en avons testé certains mais il serait judicieux d'entamer une approche comparative plus structurée sur les bases de l'approche que nous proposons. Des critères seraient à identifier pour mener cette comparaison. Ils concerneraient, à première vue, la robustesse aux attaques et la performances des architectures obtenues.

Nous avons évoqué également la complexité en nombre de propriétés à vérifier. La génération automatique des propriétés doit être envisagée car elle est d'autant plus indispensable face à la gamme plus importante de patrons pouvant intégrer les architectures. De plus, l'ajout envisagé de types de patrons de sécurité, correspondant à de nouveaux cas d'attaques ou d'exigences de sécurité souhaitées augmentent le nombre et le type de propriétés à manipuler lors du processus d'évaluation. Il en va de même pour la génération automatique de scénarios, liés aux types d'attaques plus complexes à prendre en compte. Les travaux de modélisation d'attaquant doivent pouvoir être exploités pour consolider notre approche. Nous pouvons aussi vouloir répondre à des aspects de corruption partielle d'architectures avec des capacités de limiter les conséquences. C'est une caractéristique importante que doivent supporter les systèmes $SCADA$, connue comme limitation harmonieuse ($Graceful\ Degradation$).

Bibliography

[52085] Technical Report DoD 5200.28-STD. *Trusted Computer System Evaluation Criteria*. Department of Defense, 1985.

[AA+96] Ross J Anderson, British Medical Association, et al. *Security in clinical information systems*. British Medical Association London, 1996.

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Joaquim Romaguera i Ramió, Max Jacobson, and Ingrid Fiksdahl-King. *A pattern language*. Gustavo Gili, 1977.

[AL00] Ross Anderson and Jong-Hyeon Lee. Jikzi: A new framework for secure publishing. In *Security Protocols*, pages 21–36. Springer, 2000.

[Ale79] Christopher Alexander. *The timeless way of building*, volume 1. New York: Oxford University Press, 1979.

[Amo94] Edward G Amoroso. *Fundamentals of computer security technology*. Prentice-Hall, Inc., 1994.

[ASH03] Ehab S Al-Shaer and Hazem H Hamed. Firewall policy advisor for anomaly discovery and rule editing. In *Integrated Network Management, 2003. IFIP/IEEE Eighth International Symposium on*, pages 17–30. IEEE, 2003.

[ASL02] Ross Anderson, Frank Stajano, and Jong-Hyeon Lee. Security policies. *Advances in Computers*, 55:185–235, 2002.

[BC87] Kent Beck and Ward Cunningham. Using pattern languages for object-oriented programs. 1987.

[Bib77] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[BL73] D Elliott Bell and Leonard J LaPadula. Secure computer systems: Mathematical foundations. Technical report, DTIC Document, 1973.

[BLR03] Arosha K Bandara, Emil C Lupu, and Alessandra Russo. Using event calculus to formalise policy specification and analysis. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 26–39. IEEE, 2003.

[BN89] David FC Brewer and Michael J Nash. The chinese wall security policy. In *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, pages 206–214. IEEE, 1989.

[Boy09] Stuart A Boyer. *SCADA: supervisory control and data acquisition*. International Society of Automation, 2009.

[BS11] Michaela Bunke and Karsten Sohr. An architecture-centric approach to detecting security patterns in software. In *Engineering Secure Software and Systems*, pages 156–166. Springer, 2011.

[CCC08] Fredéric Cuppens Céline Coma, Nora Cuppens-Boulahia and Ana Rosa Cavalli. Context ontology for secure interoperability. In *3rd International Conference on Availability, Reliability and Security (ARES'08)*, March 2008.

[CCG+04] Sagar Chaki, Edmund Clarke, Orna Grumberg, Natasha Sharygina, Tayssir Touili, and Helmut Veith. An expressive verification framework for state/event systems. *Proceedings of the 5th international conference on integrated formal methods (IFM)*, 2004.

[CCO+05] Sagar Chaki, Edmund M. Clarke, Joel Ouaknine, Natasha Sharygina, and Nishant Sinha. Concurrent software verification with states, events, and deadlocks. *Proceedings of the 5th international conference on integrated formal methods (IFM)*, 17(4):461–483, 2005.

[CDF+07] Steven Cheung, Bruno Dutertre, Martin Fong, Ulf Lindqvist, Keith Skinner, and Alfonso Valdes. Using model-based intrusion detection for scada networks. In *Proceedings of the SCADA security scientific symposium*, volume 46, pages 1–12. Citeseer, 2007.

[Com09] Céline Coma. Interoperabilité et cohérence de politiques de sécurité pour les systèmes auto-organisants. In *Thèse de doctorat, Telecom Bretagne*, April 2009.

[CSD04] Harkeerat Bedi Chris Simmons, Sajjan Shiva and Dipankar Dasgupta. Avoidit: A cyber attack taxonomy. *9th Annual Symosium on Information Assurance (ASIA'14), Albany, NY*, 135, june 2004.

[CT16] Luka Le Roux Ciprian Teodorov, Philippe Dhaussy. Environment-driven reachability for timed systems : Safety verification of an aircraft landing gear system. *Int. Software Tools for Technology Transfer (STTT)*, 2016.

[CTD16] Zoe Drey Ciprian Teodorov, Luka Le Roux and Philippe Dhaussy. Past-free[ze] reachability analysis: reaching further with dag-directed exhaustive state-space analysis. *Software Testing, Verification and Reliability (STVR)*, August 2016.

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

[DBRL12] Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, and Luka Leroux. Improving model checking with context modelling. *Advances in Software Engineering*, 2012.

[DGFRLP04] Nelly Delessy-Gassant, Eduardo B Fernandez, Sajeed Rajput, and Maria M Larrondo-Petrie. Patterns for application firewalls. In *Proceedings of the Pattern Languages of Programs (PLoP) Conference*, 2004.

[DIR02] Nicole Dunlop, Jadwiga Indulska, and Kerry Raymond. Dynamic conflict detection in policy-based management systems. In *Enterprise Distributed Object Computing Conference, 2002. EDOC'02. Proceedings. Sixth International*, pages 15–26. IEEE, 2002.

[DPP+17] Jannik Dreier, Maxime Puys, Marie-Laure Potet, Pascal Lafourcade, and Jean-Louis Roch. Formally verifying flow properties in industrial systems. In *SECRYPT 2017, Madrid, Spain*, July 2017.

[DRB11] Philippe Dhaussy, Jean-Charles Roger, and Frederic Boniol. Reducing state explosion with context modeling for model-checking. In *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*, pages 130–137. IEEE, 2011.

[DT14] Philippe Dhaussy and Ciprian Teodorov. Context-aware verification of a landing gear system. In *Conference ABZ'14, Case Study Track, Toulouse, june2-6*, 2014.

[FB13] Eduardo Fernandez-Buglioni. *Security patterns in practice: designing secure architectures using software patterns*. John Wiley & Sons, 2013.

[FC08] Nora Cuppens-Boulahia Frédéric Cuppens. Modeling contextual security policies. *International Journal of Information Security (IJIS'08)*, 7(4), 2008.

[FCG07] Nora Cuppens-Boulahia Frédéric Cuppens and Meriam Ben Ghorbel. High level conflict management strategies in advanced access control models. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 186(1571-0661), July 2007.

[FCM10] Igor Nai Fovino, Alessio Coletta, and Marcelo Masera. Taxonomy of security solutions for the scada sector. *Project ESCORTS Deliverable*, 2, 2010.

[Fer00] Eduardo B Fernandez. Metadata and authorization patterns. *Departament of Computer Science and Eng., Florida Atlantic University TR-CSE-00-16*, 2000.

[FLP10] Eduardo B Fernandez and Maria M Larrondo-Petrie. Designing secure scada systems using security patterns. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1–8. IEEE, 2010.

[FLPS+03] Eduardo B Fernandez, Maria M Larrondo-Petrie, Naeem Seliya, Nelly Delessy-Gassant, and Markus Schumacher. A pattern language for firewalls. *M. Schumacher, et al*, 2003.

[FP01] Eduardo B Fernandez and Rouyi Pan. A pattern language for security models. In *proceedings of PLOP*, volume 1, 2001.

[Gam95] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[HAJ12] Munawar Hafiz, Paul Adamczyk, and Ralph E Johnson. Growing a pattern language (for security). In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 139–158. ACM, 2012.

[ILW06] Vinay M Igure, Sean A Laughter, and Ronald D Williams. Security issues in scada networks. *Computers & Security*, 25(7):498–506, 2006.

[Jus03] Krzysztof Juszczyszyn. Verifying enterprise's mandatory access control policies with coloured petri nets. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on*, pages 184–189. IEEE, 2003.

[Kru05] Ronald L Krutz. *Securing SCADA systems*. John Wiley & Sons, 2005.

[KS89] Robert Kowalski and Marek Sergot. A logic-based calculus of events. In *Foundations of knowledge base management*, pages 23–55. Springer, 1989.

[KV98] Ekkart Kindler and Tobias Vesper. Estl: A temporal logic for events and states. *Proceedings of the 19th International Conference on the Application and Theory of Petri Nets (ICATPN)*, 1420:365–383, 1998.

[LS99] Emil C Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *Software Engineering, IEEE Transactions on*, 25(6): 852–869, 1999.

[MP16] Pascal Lafourcade Maxime Puys, Marie-Laure Potet. Formal analysis of security properties on the opc-ua scada protocol. In *SAFECOMP'16,Trondheim, Norway*, September 2016.

[MXYJ10] Jianli Ma, Guoai Xu, Yixian Yang, and Yong Ji. Information system security function validating using model checking. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 1, pages V1–517. IEEE, 2010.

[PPK17] Maxime Puys, Marie-Laure Potet, and Abdelaziz Khaled. Generation of applicative attacks scenarios against industrial systems. In *Foundations and Practice of Security - 10th International Symposium, FPS 2017, Nancy, France*, October 2017.

[Puy18] Maxime Puys. *Sécurité des systèmes industriels: filtrage applicatif et recherche de scénarios d'attaques.* february 2018.

[RC10] Jonathan Rouzaud-Cornabas. *Formalisation de propriétés de sécurité pour la protection des systèmes d'exploitation.* PhD thesis, Thèse Université d'Orléans, 2010.

[RSK04] David Ferraiolo Ravi Sandhu and Rick Kuhn. Role-based access control. *American national standard for information technology : ANSI INCITS 359-2004*, February 2004.

[SA99] Frank Stajano and Ross Anderson. The resurrecting duckling: Security issues in ad-hoc wireless networks. 1999. In *Security Protocols, 7th International Workshop Proceedings, Lecture Notes in Computer Science, Springer-Verlag. http://www. cl. cam. ac. uk/fms27/duckling*, 1999.

[SFBH+13] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns: Integrating security and systems engineering.* John Wiley & Sons, 2013.

[Sti06] E Stipidis. Vetronics system integration. pages 401 – 415, April 2006.

[TCP14] Leroux L. Teodorov C. and Dhaussy P. Context-aware verification of a cruise-control system. In *4th International Conference on Model and Data Engineering (MEDI), Larnaca, Cyprus, September 24-26*, 2014.

[TLM08] Chee-Wooi Ten, Chen-Ching Liu, and Govindarasu Manimaran. Vulnerability assessment of cybersecurity for scada systems. *IEEE Transactions on Power Systems*, 23(4):1836–1846, 2008.

[WC03] Ronald Wassermann and Betty HC Cheng. Security patterns. In *Michigan State University, PLoP Conf.* Citeseer, 2003.

[WM08] Michael Weiss and Haralambos Mouratidis. Selecting security patterns that fulfill security requirements. In *International Requirements Engineering, 2008. RE'08. 16th IEEE*, pages 169–172. IEEE, 2008.

[YB97] Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. *Urbana*, 51:61801, 1997.

[YB98] Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. *Urbana*, 51:61801, 1998.

[YWM08] Nobukazu Yoshioka, Hironori Washizaki, and Katsuhisa Maruyama. A survey on security patterns. *Progress in informatics*, 5(5):35–47, 2008.

[ZJS11] Bonnie Zhu, Anthony Joseph, and Shankar Sastry. A taxonomy of cyber attacks on scada systems. In *Internet of things (iThings/CPSCom), 2011 international conference on and 4th international conference on cyber, physical and social computing*, pages 380–388. IEEE, 2011.

[ZS10] Bonnie Zhu and Shankar Sastry. Scada-specific intrusion detection/prevention systems: a survey and taxonomy. In *Proceedings of the 1st Workshop on Secure Control Systems (SCS)*, 2010.

Annexe : Publications

Les publications rédigées durant ce travail sont listées ci-dessous.

- Fadi Obeid and Philippe Dhaussy. Validation formelle d'implantation de patrons de sécurité: Application aux SCADA. Conf. Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'16), Besançon, juin, 2016.
- Fadi Obeid and Philippe Dhaussy. Model Checking of Security Patterns Combinations: Application to SCADA. Conf. RESSI'17, 2017, Grenoble.
- Fadi Obeid and Philippe Dhaussy. RITA Secure Communication Protocol: Application to SCADA, 8th International Conference on Network and Communications Security (NCS 2016). Editor(s): David Wyld et al, 2016.
- Fadi Obeid and Philippe Dhaussy. A Secured Data Communication Protocol, Conf. A Connected Ocean (ACO 2016), SeaTech Week, Brest, october, 2016.
- Fadi Obeid et Philippe Dhaussy. Validation formelle d'architecture logicielle basée sur des patrons de sécurité : Application aux SCADA. Conférence AFADL'18, Grenoble, 13-15 juin 2018.
- Fadi Obeid et Philippe Dhaussy. Model-checking for Secured Component Implementation. 17th International Conference on Security and Management (SAM'18), Las Vegas, USA, July 30 - August 2 2018.
- Fadi Obeid et Philippe Dhaussy. Secure Communication Protocol: Application to Large Number of Distributed Sensors. 17th International Conference on Security and Management (SAM'18), Las Vegas, USA, July 30 - August 2 2018.

Un publication est en attente de réponse :

- Fadi Obeid and Philippe Dhaussy. Formal Verification of Security Pattern Composition : Application to SCADA. Soumise à la revue : Computing and Informatics (CAI).

Un dépôt d'une enveloppe soleau: RITA : Algorithm de sécurité de communication (2016).