INSA
CENTRE VAL DE LOIRE

INSA Centre Val de Loire

ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES

Équipe Sécurité des Systèmes Distribués (SDS)

THÈSE présentée par :

Adrien JOUSSE

soutenue le : **13 Décembre 2022**

pour obtenir le grade de : **Docteur de l'INSA Centre Val de Loire**

Discipline/ Spécialité : **Informatique**

Protection obligatoire vérifiée au regard des objectifs safety du secteur automobile

Thèse dirigée par :

Christian TOINARD	Professeur des Universités, INSA Centre Val de Loire

Co-encadrants :

Vincent HUGOT	Maître de Conférences, INSA Centre Val de Loire
Benjamin VENELLE	Ingénieur cybersécurité, Valeo

RAPPORTEURS :

Frédéric MALLET	Professeur des Universités, Université Côte d'Azur
Damien SAUVERON	Maître de Conférences - HDR, Université de Limoges

---

JURY :

Fabrice BOUQUET	Professeur des Universités, Université de Bourgogne Franche Comté, Président du jury

LIFO

ANRT
ASSOCIATION NATIONALE RECHERCHE TECHNOLOGIE

Valeo

Remerciements

Je tiens à remercier mon directeur de thèse, Christian TOINARD, pour avoir accepté de diriger ma thèse, ainsi que pour le temps passé à relire, corriger et expliquer de nombreux éléments au cours de cette thèse.

Je remercie aussi mon encadrant industriel, Benjamin VENELLE, pour m'avoir donné la possibilité d'effectuer cette thèse et pour les échanges que nous avons eu.

Enfin, je remercie mon co-encadrant, Vincent HUGOT, pour le temps passé en réunions à expliquer de maintes fois les mêmes concepts, à discuter de choses et d'autres, et pour sa grande aide sur les sections comportant des notations et des définitions formelles.

J'adresse mes remerciements à Frédéric MALLET et Damien SAUVERON pour avoir accepté de relire cette thèse et d'en être rapporteurs. Leurs remarques ont été précieuses pour rédiger la version finale de ce manuscrit. Je tiens à remercier Fabrice BOUQUET d'avoir accepté d'être président du jury.

Je remercie aussi mes anciens collègues de Valeo : Frédéric CALVEZ, Antoine BOULANGER, Guillaume ARTUS, Célia BOUNOUAR, Aurel-Sorin SPORNIC et Corinne DELANGE, pour les moments de détente, pour m'avoir tout appris du secteur automobile et m'avoir aidé autant que vous le pouviez pour mener à bien cette thèse.

Enfin, ce manuscrit n'existerait probablement pas sans mes proches.

À mes amis : Killian, Maxime, Mélanie, Julien, Julia, Elodie, Pani, Geneviève, Marine et Martin merci pour les moments passés avec vous qui m'ont permis de décompresser, et qui, sans que vous le sachiez, ont joué une part importante dans la réalisation de ce travail.

Une pensée toute particulière pour la meilleure : Clara. Tu as su me supporter ces dernières années, qui étaient loin d'être les plus simples. Merci pour ton soutien indéfectible, ta patience et tout simplement ta présence. J'espère que je serai à même de te soutenir autant que tu m'as soutenu.

Enfin, un merci infini à mes parents et ma sœur, pour leur soutien, leur patience et leurs encouragements dans les moments les plus difficiles. Sans vous, je n'aurais pu mener ce travail à son terme.

Sommaire

Liste des figures

Listings

Glossaire

ABS	Anti-lock Braking System.
CAN	Controller Area Network.
CTL	Computation Tree Logic.
cybersecurité	Ensemble des mesures et processus visant à protéger le système d'attaques intentionnelles.
ECU	Electronic Control Unit.
IDS	Intrusion Detection System.
IPS	Intrusion Prevention System.
ISW	Intelligent Steering Wheel.
LTL	Linear Time Logic.
MAC	Mandatory Access Control.
MOST	Media Oriented Systems Transport.
OBD	On Board Diagnostic.
OEM	Original Equipement Manufacturer.
safety	Ensemble des mesures et processus visant à limiter les atteintes à la vie humaine en cas de défaillance du système.
sûreté de fonctionnement	Ensemble des mesures et processus visant à garantir le fonctionnement d'un système dans un environnement donné.

Chapitre 1

Introduction

1.1 Contexte

La demande des consommateurs pour des véhicules toujours plus intelligents, connectés et autonomes se traduit par une complexification et une informatisation croissante des systèmes automobiles. Pour offrir ces fonctionnalités, les véhicules se dotent de nouvelles interfaces de communication (Wi-Fi, 4G, USB, Bluetooth, interfaces de charge) qui sont autant de points d'entrée dans le système du véhicule. Ces différentes interfaces permettent d'accéder au système d'information du véhicule, composé d'Electronic Control Unit (ECU), interconnectées par différents réseaux (Controller Area Network (CAN), Media Oriented Systems Transport (MOST), Ethernet...).

Ainsi, la surface d'attaque des véhicules s'est grandement étendue, et n'a pas manqué d'être exploitée [Gre15]. Ces attaques peuvent avoir pour objectif d'activer des fonctionnalités supplémentaires, de voler des véhicules [Pol17], mais aussi d'organiser des attaques coordonnées sur les infrastructures routières [TG+15] ou de porter atteinte à l'intégrité physique des personnes [S+13].

Dans le dernier cas, la safety des passagers n'est pas garantie, ce qui n'est pas acceptable pour un véhicule homologué, devant préserver ses occupants. Ainsi, il est obligatoire [ISO21b, Par19] d'ajouter des mécanismes de sécurité aux systèmes automobiles afin de préserver la safety des occupants en cas d'attaque. Néanmoins, l'ajout de mécanismes de cybersécurité peut notamment compromettre la disponibilité avec pour conséquence un manque de safety. Il s'agit donc d'un équilibre à trouver entre sécurité et safety, sachant que ce dernier point est crucial.

1.2 Approche

Historiquement, les principales attaques sur les systèmes automobiles étaient liées au vol de véhicules. L'objectif de la sécurité était donc d'empêcher un attaquant d'exploiter

les systèmes exposés du véhicule. La sécurité offerte était donc périmétrique.

L'ouverture des systèmes automobiles modernes a rendu accessible le système interne du véhicule au monde extérieur, mettant ainsi en avant le besoin de mécanismes de sécurité plus avancés, dans une démarche de défense en profondeur. La mise en place de mécanismes de sécurité peut se faire au travers de procédés cryptographiques (authentification, chiffrement...) ou de contrôle d'accès.

Nous nous intéressons uniquement au contrôle d'accès, les approches cryptographiques sortent du cadre de notre étude. Plus particulièrement, nous portons notre attention sur le contrôle d'accès obligatoire, qui a déjà été exploré pour sécuriser la machine virtuelle JAVA dans [Ven15].

Le travail de [Ven15] n'offre pas de garanties de safety et n'est donc pas compatible tel quel avec le domaine automobile. En effet, le système doit préserver la safety dans toutes les situations possibles, y compris lorsque notre mécanisme de contrôle d'accès autorise, ou bloque, une action. Par exemple, lors d'un freinage automatique d'urgence, lorsqu'un obstacle est détecté devant le véhicule, un message est envoyé pour demander au véhicule de freiner. Si le mécanisme de contrôle d'accès bloque l'envoi de ce message, alors le véhicule risque d'aller droit sur l'obstacle.

On pourrait penser que pour se prémunir de ce type de situation, il suffit de ne pas bloquer ce message dans notre mécanisme. Or, un tel blocage est nécessaire en cas d'attaque. Prenons le même exemple, cette fois avec un attaquant pouvant envoyer des messages dans le système du véhicule. Cet attaquant pourrait demander au véhicule de freiner à n'importe quel moment, y compris lorsqu'il n'y a pas d'obstacle devant le véhicule avec le risque de provoquer des collisions arrières. Ce type de situation ne peut se résoudre avec un mécanisme de contrôle d'accès statique qui pourra uniquement autoriser ou interdire l'envoi du message demandant le freinage sur un cycle de vie du système.

Ainsi, l'approche de [Ven15] présente un contrôle d'accès obligatoire dont les autorisations et interdictions des actions évoluent en fonction de l'état du système. Les actions correspondent à des appels de fonctions ou des messages échangés. Il devient donc possible de prendre en compte l'état actuel du système pour moduler le contrôle. Reprenons notre exemple d'un freinage d'urgence. Avec un mécanisme de contrôle d'accès dynamique, l'envoi du message demandant le freinage doit être précédé de la réception d'un message indiquant un obstacle devant le véhicule. Ainsi, si un attaquant tente d'envoyer un message demandant le freinage du véhicule à n'importe quel moment, ce message ne sera pas pris en compte car aucun obstacle n'a été détecté.

C'est pourquoi nous proposons un contrôle d'accès basé sur des automates prenant en compte l'état du système. Ainsi, nous pensons pouvoir mettre en place une politique plus sûre que des politiques statiques de contrôle d'accès, permettant de bloquer des actions de façon relative à l'état du système. La politique en place devra aussi respecter

les contraintes du secteur automobile telle que la safety.

1.3 Prise en compte des contraintes du secteur automobile

L'approche proposée dans [Ven15] permet de bloquer des attaques avec les politiques adéquates, mais elle ne peut être utilisée telle quelle dans le secteur automobile. En effet, pour exploiter cette approche dans le secteur automobile, il est nécessaire de prendre en compte les contraintes de ce secteur, notamment :

— la safety ;

— l'embarquabilité de la solution ;

— le temps réel.

Les contraintes d'embarquabilité et de temps réel ne sont pas abordées dans notre étude, étant donné que nous n'avons pas fait d'expérimentations sur les systèmes de production. Ainsi, nos efforts se concentrent sur la prise en compte des contraintes safety du véhicule.

En utilisant des outils de vérification formelle, nous évaluons l'intégration du mécanisme de contrôle d'accès vis-à-vis des propriétés safety requises. De cette manière, nous pouvons affiner une politique de contrôle d'accès pour protéger le système tout en conservant ses propriétés safety. Pour effectuer la vérification, nous utilisons un cas d'usage réel qui présente donc un intérêt concret pour un équipementier.

1.4 Apports de la thèse

Cette thèse s'est déroulée au sein de l'entité GEEDS (Group Electronic Expertise & Development Services) de l'équipementier automobile Valeo. Ainsi, les problématiques et solutions abordées dans ce manuscrit sont centrées sur les missions d'un équipementier bien que certaines solutions peuvent être étendues à un Original Equipement Manufacturer (OEM).

Les travaux réalisés dans cette thèse proposent d'ajouter des moniteurs de référence vérifiés, garantissant une politique de contrôle d'accès basée sur des automates. En effet, les moniteurs existant sur des systèmes patrimoniaux ne prennent généralement pas en compte l'état du système, rendant le contrôle peu apte à préserver la safety exigée par le monde automobile. Ils sont déjà difficiles d'usage dans le monde de l'informatique décisionnel et sont tout à fait inopérants dans notre contexte. Par des phases itératives, nous montrons que l'on peut affiner la politique pour satisfaire au mieux la safety.

La démarche se décompose en deux grandes phases. Tout d'abord la modélisation et ensuite la vérification des propriétés de safety et de cybersécurité.

La modélisation se découpe en trois étapes :

(1) modélisation du système à sécuriser ;

(2) modélisation des capacités d'un attaquant ;

(3) modélisation de la politique de contrôle d'accès.

Pour l'étape 1, afin de faciliter le travail de modélisation, le système est découpé en sous-systèmes, plus simples à spécifier individuellement, qui sont ensuite synchronisés. Cette synchronisation consiste à faire évoluer plusieurs sous-systèmes en même temps sur certaines opérations, afin d'obtenir l'évolution du système global. Pour cette étape, nous utilisons un cas d'usage réel issu de l'expérience de Valeo. Afin d'observer l'adéquation de la politique de contrôle d'accès, un modèle d'attaque est défini (étape 2) et synchronisé avec le système global. Cette étape est cruciale pour observer les effets de la politique au regard d'une attaque, lors de la phase de vérification. Enfin, lors de l'étape 3 la politique est modélisée et synchronisée avec le reste du système. Si elle est adéquate, alors l'attaque sera évaluée comme inopérante lors de la phase de vérification.

Pour effectuer la vérification de notre modèle, nous avons besoin de règles pour l'évaluer. Ces règles sont écrites dans une logique temporelle, et permettent de spécifier des propriétés de disponibilité, de safety et de sécurité. Les différents modèles synchronisés sont ensuite vérifiés comme respectant les règles définies par le concepteur du système.

1.5 Plan du mémoire

Ce manuscrit s'articule autour de quatre chapitres. Le Chapitre 2 : "État de l'art" présente l'état de la littérature sur la safety dans l'automobile, l'architecture des systèmes automobiles et la cybersécurité de ces systèmes. Nous abordons aussi les modèles de contrôle d'accès et la vérification de systèmes. Le Chapitre 2 : "État de l'art" conclut sur le besoin de mettre en place un mécanisme de contrôle d'accès dynamique pour l'automobile, préservant la safety, et intégrable dans les systèmes existants. Ce mécanisme ne doit pas violer les propriétés de safety du système automobile.

Le Chapitre 3 : "Démarche générale" présente notre approche pour mettre en place un tel mécanisme dans un système automobile. Nous présentons aussi le cas d'usage réel qui est utilisé dans ce manuscrit, ainsi que la construction de la politique de contrôle d'accès. Le Chapitre 4 : "Modélisation formelle de notre approche" détaille les différentes étapes de la modélisation et de la vérification de notre cas d'usage réel avec deux outils.

Enfin, les expérimentations supplémentaires portant sur la synchronisation entre notre mécanisme de contrôle d'accès et le reste du système ainsi que l'impact des fautes sur notre mécanisme sont détaillées dans le Chapitre 5 : "Expérimentations supplémentaires".

Chapitre 2

État de l'art

La protection des systèmes automobiles est un domaine en plein essor suite aux découvertes de failles de sécurité importantes ces dernières années [Gre15, S+13, Lab18]. Certaines de ces failles ont porté atteinte à la safety du véhicule et ont démontré la sous estimation des risques liés à la sécurité des systèmes automobiles. Ainsi, les équipementiers et OEM travaillent à améliorer la sécurité des systèmes automobiles pour préserver la safety. Néanmoins, une sécurité excessive peut aussi contribuer à des défaillances safety. Dans un premier temps, ce chapitre présente les éléments nécessaires en terme de contrôle d'accès. Ensuite, nous présentons certains aspects du secteur automobile : la safety, les architectures, la sécurité de ces architectures et les analyses de risques. Enfin, nous terminons ce chapitre en abordant le model checking, la vérification en ligne et les logiques temporelles.

2.1 Contrôle d'accès

Afin d'améliorer la sécurité des systèmes automobiles, une approche utilisant du contrôle d'accès peut être utilisée. Étant donné la complexité des architectures automobiles, la configuration de la politique ne sera pas entièrement laissée à l'utilisateur final. Il serait cependant envisageable de laisser certains paramètres configurables dans une optique de gestion des données personnelles, bien que cet aspect ne soit pas abordé dans cette thèse. Nous nous concentrons sur l'usage du contrôle d'accès obligatoire (Mandatory Access Control (MAC)), mis en place par l'OEM ou l'équipementier. Dans le cas d'un équipementier, la politique sera appliquée à un seul produit, constitué d'un ou plusieurs ECU. En effet, l'équipementier ne sait pas avec quels autres ECU (et à fortiori quelles applications) son produit va communiquer. Il serait possible d'effectuer certaines déductions logiques (*e.g.* l'ECU du volant n'a apparemment pas de raison de communiquer avec l'ECU du moteur), mais la complexification des architectures rend cette déduction de moins en moins réalisable. Ainsi, il n'est pas possible de mettre en place un contrôle d'accès fin sur les interfaces externes du produit.

Le contrôle d'accès repose sur le principe de vérifier qu'un sujet a les permissions requises pour accéder à un objet. Nous représentons ces accès de la manière suivante : Sujet $\xrightarrow{\{\text{Permissions}\}}$ Objet . Afin de vérifier que le sujet peut effectuer cet accès, l'accès doit pouvoir être observé par un composant logiciel ou matériel. Dans le cas contraire, on parle de canaux cachés, qui ne sont donc pas pris en compte par notre approche. Pour avoir une vision cohérente du contexte du système, ce composant doit être notifié de chaque accès ou bien intercepter chaque accès. Dans le cas ou l'on souhaite bloquer un accès si les permissions sont insuffisantes, le composant doit être en mesure d'intercepter les accès.

Les approches MAC actuelles ont toutes comme inconvénient majeur la définition de leur politique qui doit refléter le fonctionnement nominal de l'intégralité du système. En effet, les approches actuelles reposent sur le principe de "tout ce qui n'est pas explicitement autorisé est implicitement interdit". Ainsi, si un accès (*i.e.* l'envoi d'un message) entre deux ECU n'est pas décrit par la politique, en cas d'oubli ou d'erreur, alors cet accès sera refusé ce qui peut avoir des conséquences dramatiques dans le cas d'un accès safety.

Imaginons que vous êtes dans votre voiture, sur l'autoroute, avec une fonction de conduite autonome activée. Le véhicule devant vous freine brusquement, et alors que votre voiture devrait freiner automatiquement, elle ne le fait pas, car une règle dans la politique statique de contrôle d'accès a été omise. Cette règle autorisait l'envoi de commandes de freinage depuis l'ECU en charge de la conduite autonome.

Afin de fonctionner correctement, la politique doit décrire toutes les interactions possibles, dans tous les contextes de fonctionnement possibles, afin que toutes les interactions possibles puissent avoir lieu. La spécification de ce type de politique est complexe, et différents travaux ont montré la nécessité d'une surcouche aux modèles MAC pour en faciliter leur spécification et leur vérification [EF20, ESF17]. De même, nous pensons qu'il est nécessaire d'utiliser des outils supplémentaires pour mettre en place des politiques de contrôle d'accès dynamiques respectant la safety.

De plus, les politiques statiques autorisent des interactions à se produire alors que le contexte du véhicule n'est pas le bon, ce qui permet à un attaquant d'effectuer des actions dans un mauvais contexte, avec le potentiel de violer la safety.

Imaginons que vous êtes en train de conduire votre voiture sur l'autoroute, mais cette fois la conduite autonome est désactivée. Votre véhicule freine brusquement, sans que vous ayez touché à la pédale de frein. Un attaquant a pris le contrôle de l'ECU en charge de la conduite autonome et a pu envoyer un message pour actionner les freins. Bien que ne devant pas avoir lieu dans ce contexte (car la conduite autonome est désactivée), l'envoi de ce message a été autorisé par la politique statique de contrôle d'accès. Bien que l'attaquant ne soit pas en mesure d'envoyer tous les messages possibles (la politique le restreint aux messages envoyés par l'ECU dont il a le contrôle), il peut envoyer des messages autorisés en dehors du contexte d'envoi attendu. Pour avoir une politique adéquate en toute circonstance (*i.e.* qui n'autorise pas les interactions qui ne sont pas

nécessaires dans le contexte courant), la politique doit pouvoir évoluer dynamiquement, en fonction du contexte du véhicule.

De plus, l'autorisation d'un accès avec une politique classique est coûteuse, la politique pouvant comporter plusieurs milliers de règles [Cla14], devant toutes êtres vérifiées lors de chaque demande d'accès. L'espace occupé par la politique de contrôle d'accès ainsi que la latence générée en cas de contrôle sont des points bloquants pour intégrer ce type de solution dans des systèmes temps réels avec des performances limitées. Enfin, le degré de finesse des solutions actuelles ne correspond pas à celui dont nous avons besoin pour ce travail de thèse. Les mécanismes historiques tels que SELinux contrôlent les appels systèmes sur une machine, et n'ont de fait pas la trace des messages envoyés entre deux applications sur des ECU différents. Les approches de contrôle d'accès niveau réseau telles que les firewall permettent de contrôler les échanges entre deux machines, mais ne nous permettent pas de savoir précisément quelle est l'application source et l'application destination, ces informations se trouvant dans les couches hautes du modèle OSI. Afin de contrôler les échanges entre applications, il est nécessaire de se placer à la frontière entre le réseau et le système, afin d'observer les messages échangés, tout en ayant la connaissance de l'application effectuant l'envoi ou la réception.

Une politique de contrôle d'accès classique n'est pas adaptée pour tenir compte du contexte de fonctionnement des différents ECU d'un système automobile. De plus, la force du contrôle d'accès obligatoire représente ici un risque : bloquer des interactions safety. Afin d'assurer que la politique de contrôle d'accès mise en place ne pourra pas bloquer des interactions safety, nous proposons d'utiliser des outils de vérification formelle permettant d'évaluer l'impact de notre politique sur un système automobile en présence ou non d'une attaque.

2.2 Spécificités du secteur automobile

2.2.1 Safety

La safety des véhicules est un enjeu majeur des équipementiers et des OEM. Il est en effet interdit de mettre sur le marché un nouveau véhicule qui n'a pas un niveau de safety satisfaisant au regard des standards [ISO18, Par19]. Ainsi, les mesures safety mises en place ont évolué au fil du temps et de la réglementation, passant de systèmes passifs, centralisés, à des systèmes actifs, décentralisés sur l'architecture du véhicule. La safety est un sous ensemble de la sûreté de fonctionnement [UAcLR01]. La sûreté de fonctionnement englobe la disponibilité, la maintenabilité, la fiabilité, la safety et la sécurité du système considéré. Globalement, la sûreté de fonctionnement vise à garantir le fonctionnement d'un système dans un environnement donné. Plus spécifiquement, la safety s'intéresse à la protection des personnes en cas de défaillances non intentionnelles du système, ce qui la rend vulnérable aux attaques informatiques. Enfin, la cybersecurité vise à protéger un système d'acteurs mal intentionnés visant à compromettre les

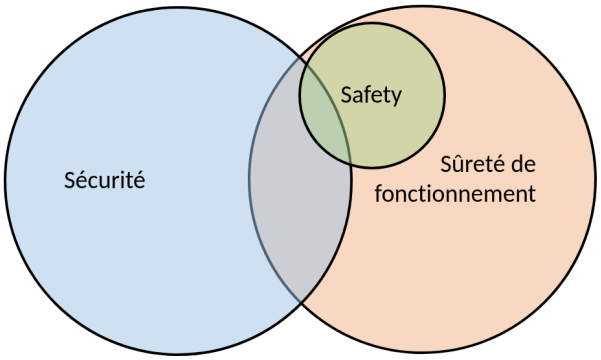


FIGURE 2.1 – Association safety, sûreté de fonctionnement et sécurité

différentes composantes de la sûreté de fonctionnement.

Le système idéal se trouverait à l'intersection de ces trois domaines comme le montre la Figure 2.1 : "Association safety, sûreté de fonctionnement et sécurité". Or, les systèmes automobiles offrent peu de sécurité et se trouvent donc du côté de la safety et de la sûreté de fonctionnement. Notre objectif est de ramener les systèmes automobiles vers le centre de cette figure, en implémentant des mécanismes de sécurité tout en préservant les aspects safety et sûreté de fonctionnement du domaine automobile.

Les architectures automobiles évoluent en fonction des besoins exprimés par les consommateurs, des contraintes réglementaires et des contraintes techniques. Ces contraintes techniques sont liées à la safety, à la masse du véhicule, à l'espace disponible, au prix et ont donc un impact sur les mécanismes de sécurité qui peuvent être mis en place dans un système automobile.

Historiquement, les véhicules n'embarquaient pas d'ECU, le contrôle du véhicule était accompli par des systèmes mécaniques. La safety des systèmes automobiles était assurée par la conception du véhicule, *e.g.* en cas de choc frontal, le capot doit se déformer pour éviter de rentrer dans l'habitacle. Les mesures de safety étaient des mesures passives, elles ne prenaient pas de décisions sur le contrôle du véhicule.

Dans les années 1970, les premiers ECU sont introduits dans les systèmes automobiles, principalement pour gérer l'alimentation en essence du moteur. L'objectif ici était d'améliorer les performances du moteur. D'année en année, le nombre d'ECU a augmenté, pour proposer des fonctions d'aide à la conduite plus avancées (avertisseur de franchissement de ligne blanche), et des fonctions de safety plus avancées telles que l'Anti-lock Braking System (ABS) [Gmb04]. L'introduction de fonctions safety telles que l'ABS a changé le paradigme dans lequel évoluait la safety, passant de mécanismes passifs, à des mécanismes actifs. Dans le cas de l'ABS, un capteur surveille la vitesse de rotation de la roue lors du freinage. Si cette vitesse devient nulle, alors l'ECU de l'ABS commande le desserrage de la mâchoire des plaquettes de frein pour permettre à la roue de tourner, maximisant ainsi son adhérence. L'ECU alterne ensuite entre serrage et

desserrage pour ralentir au maximum la roue, sans arrêter sa rotation. Pour influencer le fonctionnement de l'ABS, il faut agir sur l'environnement du véhicule (présence de verglas sur la route), ou bien le désactiver complètement via le réseau du véhicule (normalement impossible sur les véhicules récents car l'ABS est devenu obligatoire [S+13]).

L'envoi des commandes de serrage/desserrage par l'ABS est conditionné par la vitesse de rotation de la roue, qui représente son contexte de fonctionnement. Ainsi, si l'ABS se met en route en dehors de son contexte de fonctionnement légitime, c'est-à-dire lorsque la roue tourne toujours, le véhicule peut être déstabilisé et mettre en danger ses occupants. Notre objectif est de garantir que l'envoi d'une commande de l'ABS pour serrer ou desserrer la mâchoire des freins se produit bien lorsqu'il est dans le bon contexte de fonctionnement. On constate que les mécanismes safety actuels dépendent du contexte de fonctionnement du véhicule afin d'enclencher les bonnes décisions aux bons moments.

Pour mettre en place des fonctions safety plus avancées, les équipementiers et OEM ont eu besoin d'informations sur le contexte de fonctionnement de différents ECU du véhicule. Pour des fonctions safety telles que l'ABS, un unique ECU peut embarquer le capteur et l'actionneur afin de contrôler la vitesse d'une roue du véhicule et interagir avec le système de freinage ; le contexte de fonctionnement est directement connu par l'ECU. C'est un système autonome et isolé.

Des systèmes plus avancés tels que la conduite autonome (qui est une fonction safety étant donné qu'une fois activée elle doit préserver la vie des passagers) peuvent quant à eux requérir des informations provenant de différents capteurs, mais aussi envoyer des commandes à différents actionneurs. Les commandes envoyées peuvent permettre de contrôler la vitesse du véhicule, le freinage, mais aussi la direction. L'ECU en charge de la conduite autonome du véhicule doit donc être en mesure de commander d'autres ECU du véhicule, mais seulement s'il est allumé. On peut donc traduire le contexte de fonctionnement de la conduite autonome par : "Si la conduite autonome est activée, alors elle peut envoyer des messages à différents ECU du véhicule". De même "Si la conduite autonome est désactivée, alors elle ne peut pas envoyer des messages à différents ECU du véhicule". En effet, si la conduite autonome n'est pas activée, alors l'ECU n'a aucune légitimité à envoyer des commandes pour conduire le véhicule, étant donné que c'est le conducteur qui est en charge de la conduite.

De plus, l'interface vous permettant d'activer la conduite autonome n'est pas directement reliée à l'ECU en charge de la conduite autonome (de même que vous n'avez pas directement accès à l'ECU qui contrôle l'injection d'essence dans votre moteur). Par exemple, si le bouton d'activation se trouve sur votre écran d'info-divertissement, et que l'ECU en charge de la conduite autonome est dans le domaine "Cœur" comme le montre la Figure 2.2 : "Exemple simplifié d'une architecture d'un véhicule avec conduite autonome", l'activation de la conduite autonome provoquera l'envoi d'un message sur

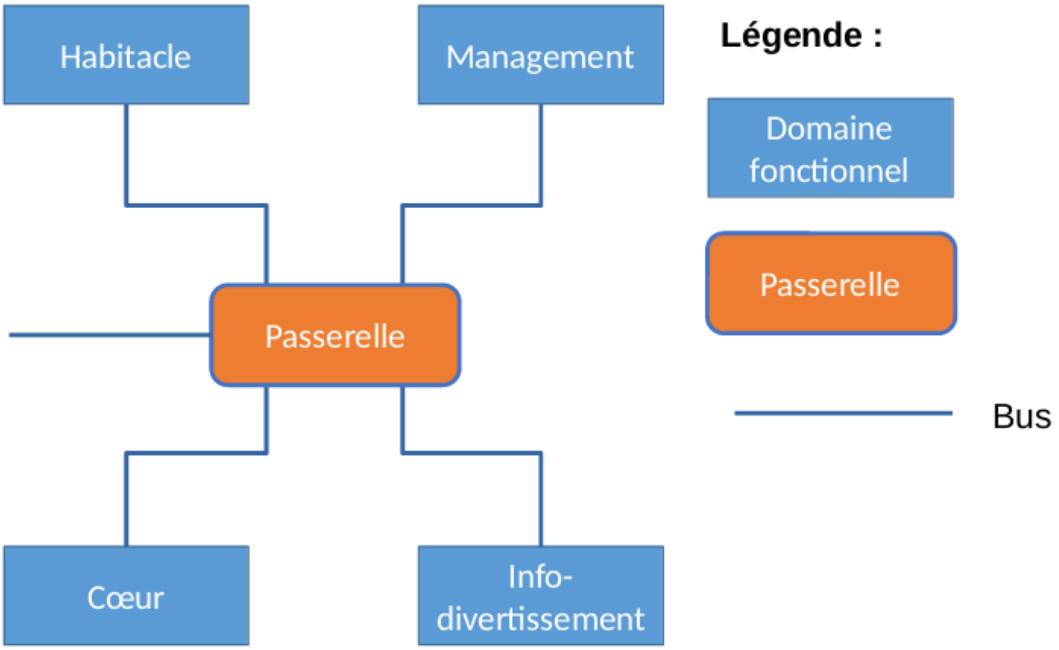


FIGURE 2.2 – Exemple simplifié d'une architecture d'un véhicule avec conduite autonome

le réseau du véhicule, qui traversera un ou plusieurs domaines fonctionnels (composés d'ECU) pour activer ou désactiver la conduite autonome. Ce message est observable par les ECU sur lesquels il transite (la passerelle de la Figure 2.2 ainsi qu'un ou plusieurs ECU du domaine habitacle), et représente le début du contexte de fonctionnement de la conduite autonome. Une fois ce message observé, l'ECU en charge de la conduite autonome va recevoir des données provenant de différents capteurs, et envoyer des messages à différents ECU pour avoir un effet sur le comportement du véhicule (ralentir, freiner, tourner...). Dans ce travail de thèse, nous ne nous intéressons pas à la cohérence entre les messages envoyées et les données reçues des capteurs (*e.g.* la conduite autonome tourne à gauche alors que les données reçues indiquent un virage à droite). Nous nous focalisons sur l'autorisation de l'envoi des messages (*e.g.* la conduite autonome est activée, donc l'envoi d'un message pour tourner à gauche est autorisée).

Ainsi, nous constatons que les fonctions de safety actuelles agissent sur différents éléments du système d'information du véhicule afin d'accomplir leurs tâches. Afin de garantir un fonctionnement correct, les décisions doivent être traitées en cohérence avec le contexte du véhicule (*i.e.* les actions demandées par la conduite autonome sont uniquement appliquées si elle est activée). La réception de ces informations est soumise aux contraintes de l'architecture du système d'information du véhicule.

2.2.2 Architectures des systèmes d'information automobiles

Pour permettre aux ECU de communiquer entre eux, les OEM ont introduits des bus tels que le CAN [ISO15], le FlexRay [ISO13], le MOST [ISO20] et plus récemment Automotive

Ethernet [IEE18] dans les véhicules. Dans un premier temps, l'ensemble des ECU du véhicule étaient interconnectés au travers d'un unique bus (*cf.* Figure 2.3a). Sur des architectures complexes, le taux d'occupation du bus créait une latence trop importante pour certaines fonctions, dégradant l'expérience des passagers. Afin de résoudre ce problème, des architectures avec plusieurs bus ont été introduites, avec pour objectifs de :

(1) préserver les propriétés safety du véhicule ;

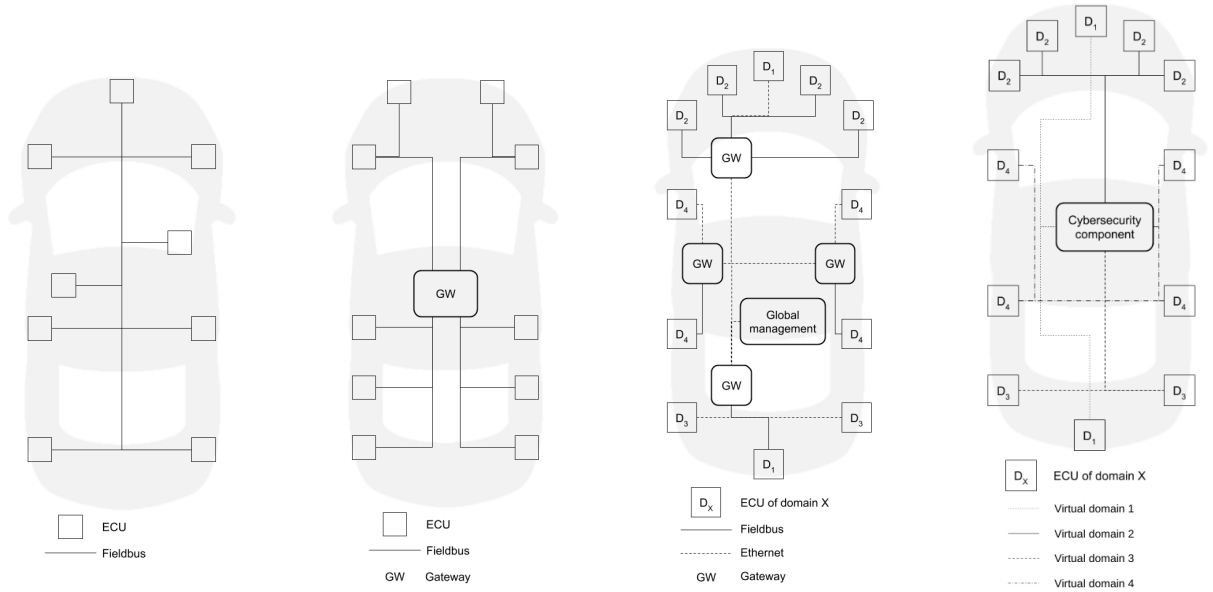
(2) proposer de nouvelles fonctionnalités ;

(3) limiter la latence des communications entre les ECU.

Dans ce type d'architecture (*cf.* Figure 2.3b : "Architecture multibus"), les ECU sont interconnectés en fonction de leur emplacement physique afin de limiter la quantité de câble nécessaire, et donc le poids du véhicule. De fait, des ECU qui ne sont pas dépendants les uns des autres peuvent être directement connectés les uns aux autres, sans souci d'isolation au sens de la sécurité. Ainsi, lorsqu'un ECU envoie un message sur le bus, le message est reçu par tous les ECU présents sur le même bus, sans possibilité de filtrage. On constate que pour sécuriser les communications entre les différents ECU du véhicule, une solution centralisée n'est donc pas adaptée. En l'absence d'un point central où toutes les communications transitent, il est nécessaire de distribuer les mécanismes de sécurité sur les différents bus du véhicule ou sur l'ensemble des ECU de l'architecture du véhicule.

De plus, avec certains bus utilisés dans le secteur automobile, il n'est pas possible de distinguer deux applications d'un même ECU envoyant un message. Avec Ethernet il sera possible d'identifier l'ECU source et la destination (via les adresses MAC), mais les informations d'identification de l'application se trouvent dans les couches hautes (5 à 7) du modèle OSI [ISO94]. Ainsi, il n'est pas pertinent de mettre en place des mécanismes de contrôle d'accès au niveau des couches basses tels que des firewall ou du filtrage par adresse MAC. Les mécanismes de sécurité fonctionnant dans les couches hautes du modèle OSI (Next Generation Firewall, Depp Packet Inspection...) ne sont pas non plus appropriés. Ils ne sont pas assez rapides pour fonctionner dans un système temps réel distribué. Enfin, en tant qu'équipementier, Valeo n'a pas les informations nécessaires sur l'architecture du véhicule pour mettre en place un mécanisme de contrôle centralisé. En effet, nous ne possédons qu'une vue partielle du système, qui se limite aux ECU que nous produisons, pour des questions de secret industriel. Ainsi, pour sécuriser les communications, il est nécessaire de placer les mécanismes de sécurité sur les ECU, à un niveau où les applications sont identifiables. Ce faisant, nous serons en mesure de sécuriser, au minimum, les ECU produits par Valeo.

L'utilisation des bus a permis de faire passer les véhicules de systèmes mécaniques isolés à des systèmes d'information. Ces architectures présentaient peu d'interfaces vers l'extérieur, et par conséquent une surface d'attaque peu importante. L'interface la plus présente était l'On Board Diagnostic (OBD) [ISO21a]) qui nécessitait un accès



(a) Architecture monobus (b) Architecture multibus (c) Architecture Ethernet physique (d) Architecture Ethernet logique

FIGURE 2.3 – Évolution des architectures automobiles

physique au véhicule afin d'être exploitée. Néanmoins, l'évolution des besoins et des technologies a peu à peu ouvert ces systèmes vers l'extérieur et les a transformés en systèmes d'information coopératifs.

En effet, ces dernières années les systèmes automobiles se sont dotés de nouvelles interfaces permettant d'étendre leurs interactions avec les véhicules (V2V), les infrastructures routières (V2X) ou les périphériques mobiles des passagers (Bluetooth, Wi-Fi, USB, 4G).

Ces nouvelles interfaces ont étendu la surface d'attaque des systèmes automobiles, permettant ainsi aux attaquants de pénétrer ces systèmes [Gre15] sans contact physique préalable, et de mettre les véhicules dans des situations où la safety des passagers n'était plus assurée. Ainsi, le système d'information des véhicules, qui était avant isolé, devient connecté et accessible depuis l'extérieur, ouvrant la porte à de nouvelles menaces, avec le pouvoir de violer la safety. Il apparaît donc que pour préserver la safety du véhicule dans les systèmes actuels, des mécanismes de sécurité doivent être utilisés.

2.2.3 Sécurité des architectures actuelles

Les bus automobiles ont été développés afin de répondre à des problématiques de coût et de poids (réduire la longueur du câblage dans le véhicule). Étant donné que la seule interface qui permettait d'accéder au bus du véhicule était la prise OBD, accessible uniquement depuis l'intérieur du véhicule, et que la priorité du secteur automobile est la safety, les risques de sécurité ont été sous évalués. Dès lors, des attaques sur la prise OBD ont montré que la sécurité des systèmes automobiles n'était pas suffisante pour protéger

le véhicule [Sec17]. En effet, par définition, lorsqu'un message est émis sur un bus, il est consommé par tous les nœuds qui y sont connectés. De fait, un attaquant connecté à la prise OBD peut effectuer des attaques par rejeu, ou forger des messages dans le but d'altérer le fonctionnement du véhicule. Étant donné que les messages transmis sur certains bus (tels que le CAN) ne comportent pas d'informations permettant d'identifier la source ou la destination du message, aucun ECU connecté au bus n'est capable de déterminer s'il s'agit d'un message légitime ou non.

Afin de contrer les situations de rejeu ou de forgeage de message, les industriels ont mis en place le SecOC [Aut17] qui permet d'authentifier ou de garantir l'intégrité d'un message. Ainsi, lors de la réception d'un message, l'ECU est capable de vérifier si ce dernier provient d'un ECU autorisé (*i.e.* appartenant à l'architecture) et si les données sont intègres. Néanmoins, l'intégrité des données ne garantit pas l'intégrité de la décision d'envoi du message. Un ECU compromis peut envoyer des messages avec des données intègres, mais dont la décision d'envoi n'est pas intègre car l'ECU est contrôlé par un acteur extérieur.

Ce type de situation où l'intégrité du système est perdue s'apparente à une faute safety : une condition anormale pouvant causer la défaillance d'un système. Si la défaillance se produit, alors il y a un risque de blessure des passagers selon le cas d'usage (la défaillance d'un système non utilisé ne représente pas un évènement dangereux). Cependant, la safety se base sur des modèles de fautes involontaires et ne prend pas en compte les acteurs mal intentionnés. De fait, la safety est dans l'impossibilité de se protéger contre des attaques informatiques. Ainsi, la sécurité est obligatoire pour préserver la safety en cas d'attaque.

Afin de se protéger des attaques où l'intégrité de la décision d'envoi est mise en jeu, il faut déterminer à quel moment un ECU a la légitimité d'envoyer ou de recevoir un message précis. Pour cela, nous devons observer les messages reçus et envoyés par un ECU, afin de pouvoir déterminer son contexte de fonctionnement, et donc quels sont les messages autorisés.

Comme nous l'avons vu sur les architectures comportant un seul bus, la seule solution est de placer un mécanisme de contrôle d'accès sur chacun des ECU connecté au bus. Sur les architectures comportant plusieurs bus, une passerelle est utilisée pour interconnecter les différents bus. Cette passerelle observe les messages échangés entre les différents bus auxquels elle est connectée. Ainsi, il est possible de déterminer le "contexte du bus", tel que la somme des contextes des ECU auxquels le bus est connecté. Cette approche est néanmoins imprécise, particulièrement si deux ECU remplissant une même fonction (pour des raisons de redondance), sont connectés au même bus et que l'un d'eux est défaillant. Du point de vue du contexte du bus, la fonction sera défaillante, alors qu'un des deux ECU est encore fonctionnel. Ainsi, la passerelle est en mesure de déterminer un contexte gros grain (*e.g.* au niveau du bus) pour y appliquer certaines règles. Les passerelles peuvent avoir des fonctionnalités de filtrage des messages permettant ainsi

de restreindre le domaine de diffusion d'un message. À partir des messages échangés, il serait donc possible d'adapter dynamiquement les règles de filtrage pour suivre le contexte des bus connectés à la passerelle. On retrouve ici la notion de domaine de sécurité, où chaque domaine a sa propre politique de contrôle d'accès, avec la passerelle correspondant à l'interface entre chacun des bus. Une politique est appliquée à chaque interface créant ainsi un domaine de sécurité. Néanmoins, sur un même bus, nous retrouvons les mêmes problématiques que précédemment.

Sur les architectures utilisant Ethernet, on trouve des fonctionnalités supplémentaires permettant de réduire le domaine de diffusion. L'utilisation des VLAN (Virtual Local Area Network), permet de segmenter l'architecture physique du véhicule en de multiples réseaux logiques (*cf.* Figure 2.3d : "Architecture Ethernet logique") permettant de restreindre un message à un domaine logique. Bien que les attaques soient plus complexes à mettre en œuvre sur les architectures Ethernet, dû à la présence d'une adresse source et destination dans les messages, il n'y a toujours aucun moyen de s'assurer de l'intégrité de la décision d'envoi d'un message. Dans cette situation, en l'absence d'authentification, un attaquant est toujours en mesure de forger des messages qui seront consommés par l'ECU cible. Si l'attaquant prend le contrôle d'un ECU, alors il est en mesure d'envoyer des messages en apparence légitime, mais qui risquent de violer des propriétés safety selon le contexte de fonctionnement du véhicule.

Comme nous l'avons vu, la sécurité des architectures automobiles n'était pas une préoccupation majeure des acteurs industriels. Avec l'augmentation du nombre d'interfaces accessibles depuis l'extérieur, et l'utilisation de technologies déjà en place dans le domaine informatique, l'ajout de nouveaux mécanismes de sécurité est obligatoire pour améliorer la sécurité des architectures automobiles afin de préserver la safety du véhicule. Afin de mettre en place des mécanismes pertinents en face des attaques possibles, les experts du secteur automobile effectuent des analyses de risques permettant de déterminer les points les plus urgents.

2.2.4 Analyse de risques

Étant donné la complexification des systèmes automobiles et leur interconnexion avec une diversité de systèmes toujours plus importante, des attaques [Gre15, S+13, TG+15, Sec17, BD22] ont prouvé la nécessité de mettre en place des mécanismes de cybersécurité. Néanmoins, tous les mécanismes de cybersécurité ne protègent pas des mêmes menaces : le contrôle d'accès protège des accès illégitimes, le chiffrement protège de la divulgation des informations. De plus, il est nécessaire de mettre des mécanismes adéquats au regard des vulnérabilités du système : une porte à serrure ne sert à rien si vous ne pouvez pas fermer vos fenêtres. Pour proposer des mécanismes de cybersécurité adéquats, les experts du secteur automobile effectuent des analyses de risques sur les systèmes à protéger.

Afin d'uniformiser le processus d'analyse de risques entre les différents acteurs du

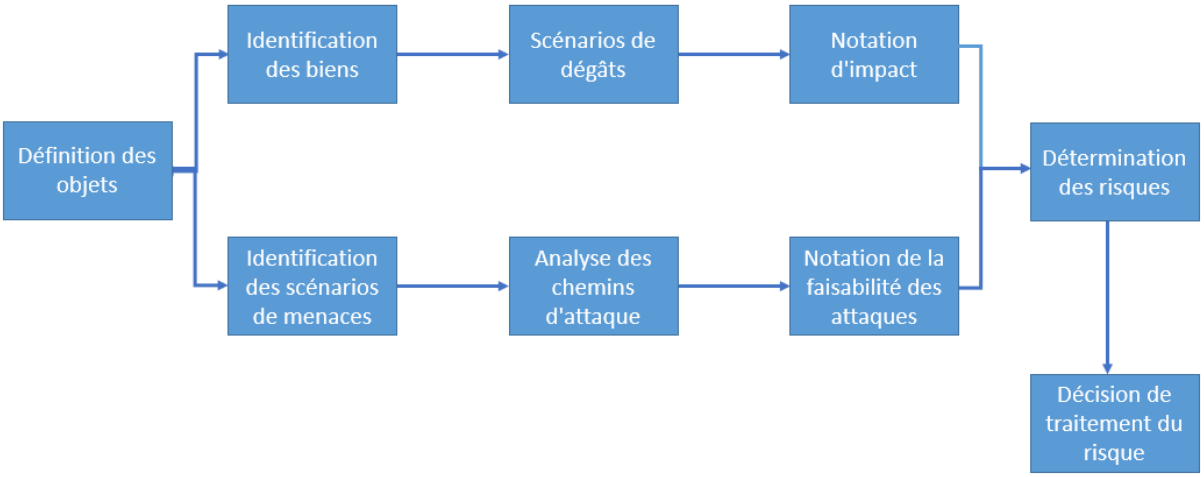


FIGURE 2.4 – Étapes à suivre d'après [PRD21] lors d'une analyse de risques

secteur automobile, l'ISO/SAE 21434 [ISO21b] définit les étapes obligatoires qui doivent être couvertes par l'analyse de risques. Ces différentes étapes sont rappelées par la Figure 2.4 d'après [PRD21]. L'ISO/SAE 21434 ne définit pas de méthode d'analyse, mais les différentes étapes qui doivent être suivies par la méthode utilisée. Une description détaillée des tâches à effectuer dans chaque étape est donnée dans [DNR20].

À l'issue de l'analyse de risque d'un produit, les experts du secteur automobile disposent d'informations permettant de proposer des mécanismes de cybersécurité adéquats. Ils savent notamment :

— les biens du systèmes (*i.e.* les composants avec des propriétés de cybersécurité) ;
— les dégâts causés au système si ces propriétés sont perdues ;
— les menaces pouvant causer les pertes de ces propriétés ;
— les chemins d'attaques qui vont causer la menace ;
— le risque d'une menace en considérant les impacts de ses dégâts et la faisabilité de ses chemins d'attaque.

Ces éléments ne sont pas fixes et évoluent en même temps que la conception du produit avance. Le changement d'un composant peut introduire de nouvelles interfaces permettant d'accéder au cœur du système créant de nouveaux chemins d'attaque. Ainsi, la conduite d'une analyse de risque n'est pas une étape unique, mais un processus qui suit le développement d'un produit. Les éléments produits par l'analyse de risques permettent de cibler en priorité certains composants du système pour mettre en place des contre-mesures visant à réduire les risques, supprimer totalement la source du risque, ou bien conserver le risque. En effet, si les impacts sur le système sont faibles et que l'attaque est difficile à mettre en œuvre, ne pas mettre de contre-mesures en place est intéressant.

À partir des résultats de l'analyse de risque, des mécanismes de cybersécurité vont être mis en place afin de répondre aux risques les plus importants. Ces mécanismes vont

donc avoir eux même un impact sur le fonctionnement du système, spécialement en cas d'attaque, afin de préserver le fonctionnement attendu. Cependant, comment s'assurer que les mécanismes déployés n'ont pas eux aussi des effets indésirables ? Pour cela, des approches de vérification formelle peuvent être utilisées.

2.3 Vérification

Afin de garantir, autant que faire se peut, qu'un système réel a le fonctionnement attendu, des approches de vérification formelle peuvent être utilisées. Pour notre part, nous distinguons deux grands types de vérification formelle : la vérification hors ligne et la vérification en ligne [HSZ14, Ing19].

2.3.1 Model checking

Généralités

La vérification formelle hors ligne peut prendre différentes formes [Eme08] et s'effectue sur une abstraction ou une sous partie du système réel.

Historiquement, la vérification hors-ligne se faisait manuellement en utilisant la logique de Hoare [Hoa69]. La logique de Hoare nécessite l'usage d'axiomes et de règles d'inférence afin de prouver manuellement, ligne par ligne, qu'un programme séquentiel respecte certaines conditions. Cette approche est fastidieuse et sujette aux erreurs car toutes les lignes du programme doivent être vérifiées par un mathématicien par rapport aux axiomes définis. Ainsi, des programmes de grande taille sont difficiles voir impossibles à vérifier et les systèmes distribués ne sont pas vérifiables à cause des multiples entrelacements possibles. Des extensions à la logique de Hoare ont été proposées, afin de l'étendre à des systèmes concurrents [Lam80] ou temps réel [Hoo94]. Ces extensions ne gèrent pas le problème d'automatisation qu'il est nécessaire de résoudre pour avoir une solution applicable dans un domaine industriel. En effet, dans l'industrie automobile, l'emploi de plusieurs mathématiciens uniquement pour vérifier des programmes n'est pas envisageable. De fait, nous avons exploré une approche plus utilisée et en partie automatisée : le model checking.

Le model checking repose sur l'utilisation d'une représentation abstraite d'un système réel pour vérifier si cette représentation satisfait un ensemble de propriétés. Ces propriétés constituent la spécification que le système abstrait doit vérifier. Si la représentation abstraite satisfait la spécification, la représentation est un modèle de la spécification (d'où le terme "model-checking"). Les propriétés à vérifier doivent toujours être manuellement définies, mais la phase de vérification est automatique, contrairement à la vérification exploitant la logique de Hoare, qui requiert l'intervention d'un mathématicien. Dans certains cas [BK09, HP00], la représentation abstraite est le système réel et ne nécessite donc pas d'intervention manuelle pour être vérifiée. De plus, le model checking peut être

appliqué à des systèmes concurrents et/ou réactifs, où une exécution est potentiellement infinie.

Un outil est ensuite utilisé pour effectuer automatiquement la vérification de la correction du modèle par rapport aux propriétés définies. Le modèle et les propriétés doivent suivre le formalisme dicté par l'outil de vérification choisi. Différents outils, CPAchecker [BK09], SPIN [Hol97], SPOT [DLLF+16], JAVA Pathfinder [HP00] sont disponibles pour effectuer la vérification, venant chacun avec leurs contraintes et spécificités. Par exemple, CPAchecker [BK09], est dédié à la vérification de programmes écrits en C. Le model checking a l'avantage de vérifier que toutes les configurations possibles du système abstrait respectent la spécification. Pour un système avec peu d'acteurs et d'interactions, le model checking exhaustif – générant tous les états du système abstrait – est utilisable.

Si le nombre d'états générés devient trop important, d'autres types de model checking peuvent être utilisés, comme le model checking symbolique [BCM+92] ou le model checking on-the-fly [GPVW96]. Le modèle checking on-the-fly permet de détecter la violation d'une propriété sans avoir à explorer la totalités des états du système. Dans le cas où le système abstrait possède un nombre infini d'états, des approches telles que le model checking régulier [BJNT00] permettent d'évaluer des propriétés du système par rapport à une sur-approximation de ses états.

Cependant, l'utilité du model checking est conditionnée par la qualité de la représentation abstraite du système réel. Si certains aspects du système réel sont simplifiés (pour réduire le temps de développement ou la taille du système abstrait), leur omission ne doit pas avoir d'impact sur les propriétés à vérifier, au risque de rendre la vérification inutile. En effet, si la représentation vérifie certaines propriétés, mais que des aspects du système réel influent sur ces propriétés et ne sont pas modélisés, alors il est possible que le système réel ne vérifie pas ces mêmes propriétés. Il y aurait donc une incohérence entre la modélisation du système et le système réel.

Si la phase de vérification détecte des erreurs, le système réel et/ou sa représentation comportent des erreurs. Dans le cas où la représentation est erronée, il convient de la modifier, afin qu'elle reflète plus fidèlement le comportement du système et de vérifier à nouveau si les propriétés sont vérifiées. Si les erreurs proviennent du système en lui même, deux options sont possibles :

(1) Modifier le système si cela est encore possible, mettre à jour sa modélisation en conséquence et vérifier à nouveau les propriétés ;

(2) Conserver le système tel quel et mettre en place des mécanismes pour éviter les erreurs trouvées (cf. Section 2.3.2 : "Vérification en ligne").

Notations formelles

Traditionnellement, la représentation abstraite du système est traduite sous forme de structures de Kripke [Kri63, CGP01]. Une structure de Kripke est un 4-uplet $M =$

$\langle S, I, R, L\rangle$ avec :

- S : ensemble fini d'états
- $I \subseteq S$: états initiaux
- $R \subseteq S \times S$: relation de transition vérifiant : $\forall s \in S, \exists s' \in S$ tel que $(s, s') \in R$
- $L : S \longrightarrow 2^{AP}$: fonction d'étiquetage avec AP un ensemble de propositions atomiques et 2^{AP} l'ensemble des parties de AP

Les structures de Kripke sont des automates simplifiés, s'affranchissant des états finaux ainsi que des labels de transitions. Nous faisons le choix de parler d'automates d'états finis plutôt que de structures de Kripke car les automates sont plus généraux. De plus, les automates nous permettent de parler de chemins – en utilisant les labels de transitions – ce qui nous sera utile par la suite.

La définition d'un automate fini non déterministe est un 5-uplet $A = \langle \Sigma, Q, I, F, \Delta\rangle$ avec :

- Q : ensemble fini d'états
- Σ : alphabet fini
- $I \subseteq Q$: états initiaux
- $F \subseteq Q$: états terminaux
- $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$: relation de transition

Un automate est déterministe si :

- $|I| \leqslant 1$
- La relation de transition Δ est une fonction partielle $Q \times \Sigma \longrightarrow Q$

Pour simplifier la phase de modélisation, la plupart des outils permettent de décomposer le système réel sous forme de composants qui vont être synchronisés entre eux [BK08] pour obtenir un modèle représentant l'évolution globale du système. On obtient ainsi plusieurs modèles, décrivant chacun un composant du système qui sont ensuite synchronisés pour obtenir le modèle du système global.

La définition du produit synchronisé est donnée ci-dessous. Prenons un nombre $n \in \mathbb{N}$ d'automates finis non déterministes. Chaque automate est noté $A_k = \langle \Sigma_k, Q_k, I_k, F_k, \Delta_k\rangle$, pour $0 \leqslant k \leqslant n$. De plus, nous définissons un symbole spécial _, que nous appelons "stay", non utilisé par les automates : $\_ \notin \bigcup_k \Sigma_k$. Intuitivement, ce symbole sert à indiquer qu'un automate ne change pas d'état dans le produit synchronisé.

Prenons notre ensemble de synchronisation

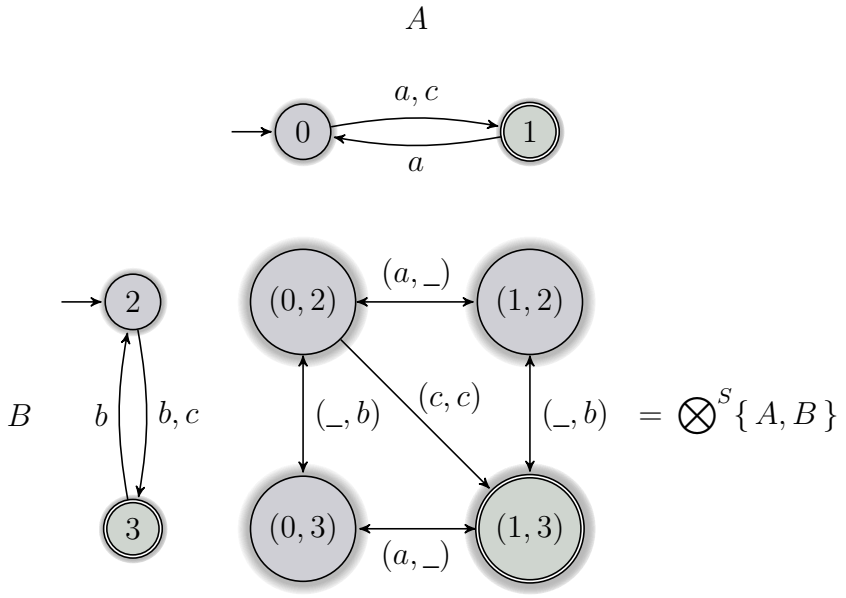
$$S \subseteq \prod_k \Big[\Sigma_k \cup \{\_\}\Big] \tag{2.1}$$

Le produit synchronisé de vecteurs de A_k par rapport à S est donné par

$$\bigotimes_k^S A_k = \left[\begin{array}{lcl} \Sigma & = & S \\ Q & = & \prod_k Q_k \\ I & = & \prod_k I_k \\ F & = & \prod_k F_k \\ \Delta & = & \left\{ \vec{p} \xrightarrow{\vec{v}} \vec{q} \;\middle|\; \begin{array}{l} \vec{v} \in S \\ \forall k, \vee \begin{cases} p_k \xrightarrow{v_k} q_k \in \Delta_k \\ v_k = \_ \wedge p_k = q_k \in Q_k \end{cases} \end{array} \right\} \end{array}\right] \tag{2.2}$$

Σ correspond à notre ensemble de vecteurs de synchronisation. Q est un sous ensemble du produit des états des composants. Les états initiaux correspondent au produit des états initiaux des composants. Les états finaux correspondent au produit des états finaux des composants. Pour toute transition labellisée $\vec{v}$ de $\vec{p}$ vers $\vec{q}$, $\vec{v}$ est un vecteur de synchronisation. Pour chaque élément k du vecteur, deux cas sont possibles. Soit il existe une transition $p_k \xrightarrow{v_k} q_k \in \Delta_k$. Soit les états p_k et q_k sont identiques et v_k indique l'absence de transition.

Prenons un exemple avec l'ensemble de synchronisation suivant $S = \{ (a, \_), (\_, b), (c, c) \}$ et les automates A et B ci-dessous :



(2.3)

Les transitions des automates A et B comportement des transitions avec deux lettres séparées par une virgule. Il s'agit d'une notation compacte pour regrouper deux transitions, chacune labélisée par une lettre. Les deux automates se synchronisent sur la lettre c. Les deux automates peuvent effectuer les transitions a et b indépendemment l'un de l'autre. Ainsi, nous obtenons l'automate produit $\bigotimes^S \{ A, B \}$. Depuis l'état initial de

l'automate produit, les deux automates peuvent avancer via une transition synchronisée (c, c) ou bien des transitions non synchronisées $(a, \_)$ et $(\_, b)$. En empruntant la transition c, chacun des automates arrive dans un état final, menant donc à un état final dans l'automate synchronisé via la transition (c, c).

L'approche utilisant les vecteurs de synchronisation est la définition standard, mais complexe à représenter lorsque l'on travaille avec un grand nombre d'automates (la taille du vecteur est égale au nombre d'automates). Si on ne souhaite synchroniser que deux automates, il est nécessaire de définir la totalité du vecteur en indiquant que tous les autres automates ne progressent pas. De fait, pour faciliter l'implémentation de ce produit, nous utilisons une autre structure d'association, similaire à un dictionnaire en Python qui permet d'omettre les éléments ne nécessitant pas de synchronisation. La définition de ce produit synchronisé nommé est donnée ci-dessous.

Prenons $\mathcal{C} = \{ A_1, \ldots, A_n \}$ un ensemble d'automates finis non déterministes, représentant les composants de notre système. Nous définissons

$$\mathbb{D}(X) \;=\; \mathcal{C} \to X \quad \text{et} \quad \mathbb{D}_{\mathrm{p}}(X) = \mathcal{C} \nrightarrow X \tag{2.4}$$

les types de fonctions (totales, resp. partielle) associant à des composants des valeur dans X. Un élément de $\mathbb{D}(X)$ est donc une association d'un élément de l'ensemble X à chaque composant. Par exemple,

$$\{ A_1 : 1, \ldots, A_n : n \} \;\in\; \mathbb{D}(\mathbb{N})\,. \tag{2.5}$$

Cependant, nous avons besoin d'associations plus spécifiques, qui expriment le concept d'associer à chaque automate A un élément de $A.\Sigma$ ou $A.Q$ par exemple. Cela correspondrait à $\mathbb{D}(.X)$ avec $X = \Sigma$ ou $X = Q$. Formellement, nous définissons

$$\mathbb{D}(.X) \;=\; \left\{ d \in \mathbb{D}\Big(\bigcup_{c \in \mathcal{C}} c.X\Big) \;\middle|\; \forall c \in \mathcal{C},\ d(c) \in c.X \right\}. \tag{2.6}$$

Dans la définition, nous filtrons les fonctions d'association afin de ne garder que celles qui associent à un automate A une valeur de sa propre composante $A.X$, par opposition aux composantes $.X$ d'autres automates de la collection $\mathcal{C}$.

La version partielle est définie de façon similaire :

$$\mathbb{D}_{\mathrm{p}}(.X) \;=\; \left\{ d \in \mathbb{D}_{\mathrm{p}}\Big(\bigcup_{c \in \mathcal{C}} c.X\Big) \;\middle|\; \forall c \in \mathcal{C},\ d(c) \in c.X \right\} \tag{2.7}$$

Définissons

$$S \;\subseteq\; \mathbb{D}_{\mathrm{p}}(.\Sigma) \tag{2.8}$$

notre ensemble de dictionnaires de synchronisation, à application partielle dans Σ. La

définition du produit synchronisé nommé de $\mathcal{C}$ par rapport à S est donnée par :

$$\bigotimes^{S} \mathcal{C} = \left[\begin{array}{lcl} \Sigma & = & S \\ Q & = & \mathbb{D}(.Q) \\ I & = & \mathbb{D}(.I) \\ F & = & \mathbb{D}(.F) \\ \Delta & = & \left\{ p \xrightarrow{d} q \;\middle|\; \begin{array}{l} d \in S,\ \forall c \in \mathcal{C}, \\ \bigvee \begin{cases} p(c) \xrightarrow{d(c)} q(c) \in c.\Delta \\ c \notin \mathrm{dom}(d),\ p(c) = q(c) \in c.Q \end{cases} \end{array} \right\} \end{array} \right] \tag{2.9}$$

L'alphabet Σ du produit correspond à notre ensemble de dictionnaires de synchronisation. Q est un ensemble d'ensemble d'états des composants. Les états initiaux correspondent à l'ensemble des ensembles d'états initiaux des composants du produit. Les états finaux correspondent à l'ensemble des ensembles d'états finaux des composants du produit. Pour toute transition labellisée d d'un ensemble d'états p vers un ensemble d'états q, d est un dictionnaire appartenant à notre ensemble de synchronisation. Pour chaque élément du dictionnaire, deux cas sont possibles. Soit il existe une transition d'un état de c dans p vers un état de c dans q, appartenant à $c.\Delta$. Soit le composant c n'appartient pas au domaine du dictionnaire d et par conséquent l'état de c dans p et q est identique (il n'y a pas de transition).

L'approche utilisant le produit synchronisé est particulièrement intéressante dans le cas des systèmes concurrents étant donné que chaque sous système peut être représenté au travers d'un ou plusieurs composants. Les différents composants se synchronisent sur des opérations complémentaires (*e.g.* le composant A envoie le message M, le composant B reçoit le message M). Nous utilisons le formalisme exposé dans [Mou11] (initialement proposé dans [dAH01a, dAH05, dAH01b]) pour décrire les opérations sur lesquelles se synchronisent les différents composants constituant le modèle. Ce formalisme est aussi utilisé par un outil développé par le Dr. Hugot que nous utilisons lors de la phase de vérification. Enfin, les propriétés utilisées pour vérifier le système abstrait doivent refléter des propriétés du système réel et dépendent là encore de l'outil utilisé.

Néanmoins, le model checking peut se heurter à une explosion du nombre d'états à vérifier (le nombre d'états du système produit augmente exponentiellement avec le nombre d'acteurs). Nous nous concentrons sur le model checking d'états finis, de manière exhaustive ou symbolique si un outil le propose. Dans le cas où un modèle comporte un nombre infini d'états, d'autres approches de vérification, telles que la vérification en ligne, permettent d'apporter une certaine confiance dans le système sans passer par la phase de modélisation.

2.3.2 Vérification en ligne

L'approche en ligne permet quant à elle de s'assurer que l'exécution courante du système satisfait un ensemble de propriétés. Pour cela, un moniteur décrivant une ou plusieurs propriétés du système est intégré au système réel. Ce moniteur est conçu pour réagir à différents événements au sein du système permettant ainsi de savoir si l'exécution courante respecte les propriétés décrites par le moniteur. Ainsi, seule une exécution du système est vérifiée, et non l'intégralité du système comme avec la vérification hors ligne.

L'approche en ligne requiert uniquement l'ajout d'un moniteur – un automate fini – dans le système réel, réagissant à certains évènements. Le moniteur doit être en mesure d'observer les évènements qu'il consomme afin de déclencher des transitions. En fonction des transitions empruntées par le moniteur, certaines actions peuvent être déclenchées [FMFR11] (alerte, blocage, autorisation) permettant ainsi de repérer ou d'éviter des déviations du système par rapport aux propriétés désirées. Un moniteur avec des capacités de blocage sur le système réel ne doit pas être en mesure de perturber des propriétés safety ou de disponibilité du système.

Cette approche a déjà été explorée dans [Ven15], dont le but était de sécuriser une machine virtuelle JAVA. Nous donnons un exemple d'un des automates en notation Grafcet utilisé dans [Ven15] en Figure 2.5b : "Exemple d'un Grafcet utilisé dans [Ven15]". Par souci de simplicité, nous appelons "automate" le système d'états-transitions de [Ven15] bien qu'il ne corresponde pas strictement à la définition que nous avons donné en Section 2.3.1 : "Model checking", mais peut s'y réduire. L'approche globale décrite dans [Ven15] utilise plusieurs automates avançant en parallèle pour faire évoluer au fur et à mesure de l'exécution une politique de contrôle d'accès globale appliquée aux entités de la machine virtuelle JAVA.

Prenons en exemple l'automate de la Figure 2.5b : "Exemple d'un Grafcet utilisé dans [Ven15]" qui autorise l'interaction *Maçon* $\Longrightarrow$ $Fille$ $\Longrightarrow$ *Réfrigérateur*. L'automate de la Figure 2.5b est paramétré par les valeurs présentée en Figure 2.5a. Les différentes flèches utilisées ($\longrightarrow$, $\Longrightarrow$, $\xrightarrow{invoke}$, $\xRightarrow{bière}$) caractérisent le type de relation entre les objets (respectivement : toute relation, flux d'information, interaction d'appel, flux de données). L'état q_1 concerne uniquement le Maçon, q_2 la Fille et q_3 le Réfrigérateur. Dans le premier état (labélisé q_1), le Maçon est autorisé à effectuer l'action décrite par P_1, c'est-à-dire qu'il est autorisé à interagir avec la Fille. Les autres interactions (P_2 à P_6) sont refusées au Maçon dans l'état q_1. Pour passer dans l'état q_2 l'automate doit observer l'interaction *Maçon* $\Longrightarrow$ $Fille$ qui est autorisée par la politique. Toute autre entité n'aura pas la capacité de faire avancer cet automate (car le premier état concerne uniquement q_1, c'est-à-dire le Maçon). Le deuxième état concerne un nouveau sujet : q_2 (la Fille) qui a son propre ensemble d'opérations autorisées et interdites. Ainsi, à partir du deuxième état, seule la Fille peut faire progresser l'automate. Elle peut uniquement accéder au Réfrigérateur (P_2). Ainsi, ces automates remplissent le rôle d'un moniteur, en garantissant

que l'exécution courante du système respecte certaines propriétés. Dans le cas de la Figure 2.5b : "Exemple d'un Grafcet utilisé dans [Ven15]", le Maçon n'est pas en mesure d'accéder directement au Réfrigérateur, il doit pour cela passer par une entité médiatrice : la Fille.

Néanmoins, l'approche de [Ven15] ne s'intéresse pas au respect des propriétés de safety ou de disponibilité (le blocage d'un des malwares présenté causait un crash de la machine virtuelle JAVA). Dans le cas d'un système avec des propriétés safety tel qu'un système automobile, un tel comportement n'est pas acceptable. Perdre le contrôle (accélération, direction, freinage...) de votre véhicule sur l'autoroute à cause d'un mécanisme de cybersécurité bloquant une attaque n'est pas une situation acceptable.

Les automates de [Ven15] sont générés semi-automatiquement à partir d'une politique de contrôle d'accès calculée par JAAS (Java Authentication and Authorization Service) et permettent de contraindre l'exécution de la machine virtuelle JAVA. D'autres travaux [RCB08, FMFR11, PPS06, BLS11] ont exploré la génération de moniteurs depuis des formules de logique temporelle. Des recherches sont toujours en cours pour générer des moniteurs [FS15]. Ainsi, ces moniteurs générés automatiquement garantissent la détection de déviations du système par rapport aux formules de logique à partir desquelles ils ont été générés. Dans notre cas, il n'existe pas de politique de contrôle d'accès pré-établie qui nous permettrait de générer nos moniteurs. Néanmoins, certaines propriétés de safety ou de cybersécurité peuvent être exprimées en logique temporelle, nous permettant d'utiliser la synthèse de moniteurs pour générer un moniteur à partir d'une propriété exprimée en logique temporelle. La spécification des moniteurs pourrait utiliser des patterns [DAC99, MdS15] permettant de faciliter cette étape. Ces deux approches n'ont pas été explorées dans le cadre de cette thèse par manque de temps. Nos moniteurs sont donc implémentés manuellement. Comparativement à l'approche de [Ven15], nous devons garantir la préservation de propriétés de safety et de disponibilité même lorsqu'un moniteur de sécurité bloque un accès. Ainsi, la vérification hors-ligne garde tout son sens même si les propriétés de cybersécurité sont préservées grâce à un moniteur.

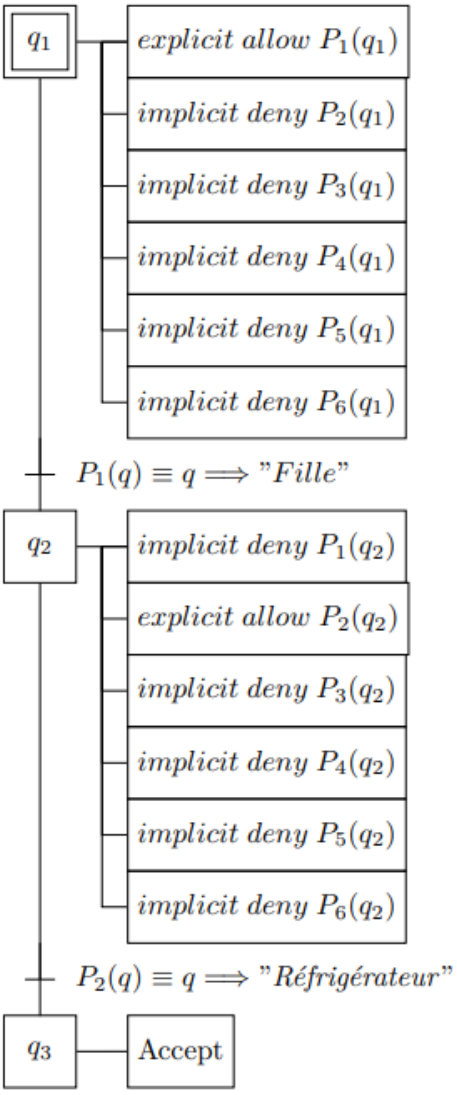
L'effort de modélisation est moindre dans l'approche en ligne — mais pas totalement absent, car il faut maîtriser la sémantique des messages observés pour être capable d'interpréter l'évolution du système par rapport à nos propriétés cibles. Ces propriétés peuvent être écrites dans différents langages, en fonction de l'outil de vérification utilisé, ou de la génération des moniteurs. Les outils que nous utilisons exploitent les logiques Linear Time Logic (LTL) et Computation Tree Logic (CTL) dont nous allons détailler le formalisme dans la section suivante.

2.3.3 Logiques temporelles

Afin de donner la syntaxe et la sémantique de Computation Tree Logic (CTL) et de Linear Time Logic (LTL), nous commençons par décrire CTL*, une généralisation de

$$
\begin{aligned}
Q \equiv & \{ \\
& q_1 = \text{"}Maçon\text{"}, q_2 = \text{"}Fille\text{"}, \\
& q_3 = \text{"}Réfrigérateur\text{"}, q_4 = \text{"}Apprenti\text{"} \\
& \} \\
P \equiv & \{ \\
& P_1(q_1) = q_1 \Longrightarrow q_2 = \text{"}Maçon\text{"} \Longrightarrow \text{"}Fille\text{"}, \\
& P_2(q_2) = q_2 \Longrightarrow q_3 = \text{"}Fille\text{"} \Longrightarrow \text{"}Réfrigérateur\text{"}, \\
& P_3(q_3) = q_3 \overset{bière}{\Longrightarrow} q_2 = \text{"}Réfrigérateur\text{"} \overset{bière}{\Longrightarrow} \text{"}Fille\text{"}, \\
& P_4(q_2) = q_2 \overset{bière}{\Longrightarrow} q_1 = \text{"}Fille\text{"} \overset{bière}{\Longrightarrow} \text{"}Maçon\text{"}, \\
& P_5(q_4) = q_4 \xrightarrow{invoke} q_1 = \text{"}Apprenti\text{"} \xrightarrow{invoke} \text{"}Maçon\text{"}, \\
& P_6(q_4) = q_4 \longrightarrow q_3 = \text{"}Apprenti\text{"} \longrightarrow \text{"}Réfrigérateur\text{"} \\
& \}
\end{aligned}
$$

(a) Paramètres du Grafcet présenté dans la Figure 2.5b



$$allow\ \text{"}Maçon\text{"} \Longrightarrow \text{"}Fille\text{"} \Longrightarrow \text{"}Réfrigérateur\text{"}$$

(b) Exemple d'un Grafcet utilisé dans [Ven15]

FIGURE 2.5 – Exemple des automates utilisés dans [Ven15]

CTL et LTL. Les logiques temporelles CTL [CE82], LTL [Pnu77], et CTL* [EH83], sont des logiques modales conçues spécifiquement dans deux buts :

(1) avoir des algorithmes de vérification efficaces ;

(2) avoir une syntaxe et une sémantique relativement faciles d'utilisation pour spécifier les propriétés.

CTL* : Computation Tree et Linear Time Logic (CTL+LTL+$\cdots$)

CTL* évalue si la séquence d'états parcourus par un automate lors de son exécution satisfait certaines propriétés.

Pour commencer, prenons A une structure de Kripke avec les notations définies en Section 2.3.1 : "Notations formelles". Nous définissons $\mathbb{A}$ un ensemble de propriétés atomiques sur les états qui seront utilisées pour définir des propriétés de logique. Par exemple : "les freins ne sont jamais bloqués".

Ces propriétés sont associées aux états du système via une fonction de labellisation :

$$\ell : Q \to \wp(\mathbb{A}) \,, \tag{2.10}$$

associant à chaque état du système un ensemble de propriétés atomiques, satisfaites dans cet état. $\wp(\mathbb{A})$ désigne l'ensemble des sous-ensembles de $\mathbb{A}$.

Nous notons $Q^\omega = \mathbb{N} \to Q$ l'ensemble des mots infinis sur Q. Un chemin (ou une *exécution*, une *trace*) est un mot $\pi \in Q^\omega$, tel que pour tout $k \in \mathbb{N}$, $\pi[k] \to \pi[k+1] \in \Delta$. Nous utilisons la notation par index et par slicing du langage Python, que nous utiliserons par la suite lors de la phase de vérification. Nous définissons

$$\Pi(q) \;=\; \left\{ \pi \in Q^\omega \;\middle|\; \pi[0] = q \;\wedge\; \forall k \in \mathbb{N},\; \pi[k] \to \pi[k+1] \in \Delta \right\} \tag{2.11}$$

l'ensemble des chemins démarrant dans l'état q.

Syntaxe et sémantique de CTL*

En CTL*, il est possible d'exprimer 2 types de formules : les formules d'états qui sont les points d'entrée et les formules de chemins.

La syntaxe est telle que :

$$\varphi \in \text{CTL* (état)} \quad ::= \quad p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists\psi \qquad\qquad p \in \mathbb{A} \tag{2.12}$$

$$\psi \in \text{CTL* (chemin)} \quad ::= \quad \varphi \mid \neg\psi \mid \psi \wedge \psi \mid \circ\psi \mid \psi \,\mathbf{U}\, \psi \tag{2.13}$$

Ici, $\exists$ signifie "il existe un chemin tel que" ; l'opérateur temporel $\circ$ est prononcé "à l'état suivant". L'opérateur temporel **U** signifie "jusqu'à".

Nous donnons maintenant la sémantique des opérateurs précédemment définis. Un état $q \in Q$ satisfait une formule d'état φ si :

$$q \models p \quad \Leftrightarrow \quad p \in \ell(q) \tag{2.14}$$
$$q \models \neg\varphi \quad \Leftrightarrow \quad q \not\models \varphi \tag{2.15}$$
$$q \models \varphi \wedge \varphi' \quad \Leftrightarrow \quad q \models \varphi \ \wedge \ q \models \varphi' \tag{2.16}$$
$$q \models \exists\psi \quad \Leftrightarrow \quad \exists \pi \in \Pi(q) : \ \pi \models \psi \tag{2.17}$$

Un chemin $\pi \in Q^\omega$ satisfait une formule de chemin ψ si :

$$\pi \models \varphi \quad \Leftrightarrow \quad \pi[0] \models \varphi \tag{2.18}$$
$$\pi \models \neg\psi \quad \Leftrightarrow \quad \pi \not\models \psi \tag{2.19}$$
$$\pi \models \psi \wedge \psi' \quad \Leftrightarrow \quad \pi \models \psi \ \wedge \ \pi \models \psi' \tag{2.20}$$
$$\pi \models \circ\psi \quad \Leftrightarrow \quad \pi[1:] \models \psi \tag{2.21}$$
$$\pi \models \psi \ \mathbf{U} \ \psi' \quad \Leftrightarrow \quad \exists k \in \mathbb{N} : (\forall i \in [\![0, k[\![, \ \pi[i:] \models \psi) \ \wedge \ \pi[k:] \models \psi' \tag{2.22}$$

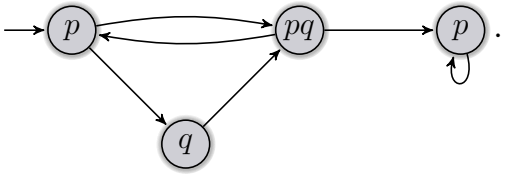
Nous ajoutons maintenant les opérateurs booléens classiques ($\top, \bot, \vee, \Rightarrow, \Longleftrightarrow$ ...), pour les formules d'état et de chemin. De plus, pour les formules de chemin :

$$\Diamond\psi \ \equiv \ \top \ \mathbf{U} \ \psi \quad \text{et} \quad \Box\psi \ \equiv \ \neg\Diamond\neg\psi \ . \tag{2.23}$$

L'opérateur $\Diamond$ signifie "fatalement". $\Box$ se lit "toujours". Pour les formules de chemin, nous ajoutons un quantificateur universel sur les chemins :

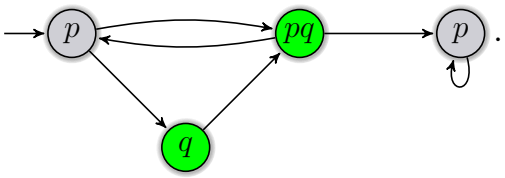
$$\forall\psi \ \equiv \ \neg\exists\neg\psi \ . \tag{2.24}$$

Prenons un exemple pour détailler le principe de CTL* (repris de [BK08]). Nous prenons $\mathbb{A} = \{ p, q \}$ et la structure de Kripke suivante :



. (2.25)

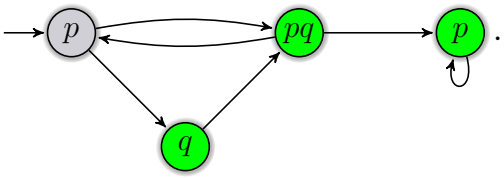
La vérification de la formule CTL* q sur la structure de Kripke 2.25 produit la Figure 2.26. Les états verts sont ceux où la formule q est satisfaite, c'est-à-dire les états possédant la propriété atomique q.



. (2.26)

La vérification de la formule CTL* $\Box\circ p$ sur la structure de Kripke 2.25 produit la Figure 2.27. Les états satisfaits sont ceux dont tous les chemins, à l'état suivant, possèdent la

propriété p. Ainsi, l'état initial ne vérifie pas la propriété car il possède un chemin allant vers un état comportant uniquement le label q.



(2.27)

Maintenant que la syntaxe et la sémantique de CTL* sont définies, nous allons voir à quel fragment de CTL* correspond LTL. Nous utiliserons LTL lors de la vérification avec SPIN, dans le Chapitre 4 : "Modélisation formelle de notre approche".

LTL : Linear Time Logic

Les formules LTL expriment des propriétés de la forme "Pour toutes les exécutions d'un système partant de l'état initial, telle et telle propriétés sont satisfaites". De fait, LTL correspond au fragment suivant de CTL* :

$$\varphi \ ::= \ \forall\psi \qquad \psi \ ::= \ p \mid \neg\psi \mid \psi \wedge \psi \mid \circ\psi \mid \psi \, \mathbf{U} \, \psi \qquad\qquad p \in \mathbb{A} \tag{2.28}$$

Cependant, le $\forall$ initial n'est traditionnellement pas écrit en LTL. De fait, les formules LTL sont des propriétés de chemin de CTL*, quantifiées universellement, sans quantificateur de chemin explicite. Il nous reste maintenant à détailler CTL, qui sera utilisée avec l'outil du Dr. Hugot dans le Chapitre 4 : "Modélisation formelle de notre approche".

CTL : Computation Tree Logic

CTL nécessite de lier chaque opérateur temporel avec un quantificateur de chemin, et inversement. Ainsi, CTL correspond à la restriction de CTL* suivante :

$$\varphi \in \text{CTL (état)} \quad ::= \quad p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists\psi \qquad\qquad p \in \mathbb{A} \tag{2.29}$$

$$\psi \in \text{CTL (chemin)} \quad ::= \quad \neg\psi \mid \circ\varphi \mid \varphi \, \mathbf{U} \, \varphi \tag{2.30}$$

Dans une syntaxe plus directe :

$$\varphi \in \text{CTL} \ ::= \ p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists\circ\varphi \mid \exists[\varphi \, \mathbf{U} \, \varphi] \mid \forall[\varphi \, \mathbf{U} \, \varphi] \tag{2.31}$$

Ces couples d'opérateurs sont considérés comme des opérateurs uniques. Dans la syntaxe réduite, nous avons :

— $\exists\circ$: "pour certains chemins, à l'état suivant" ;

— $\exists\mathbf{U}$: "pour certains chemins, jusqu'à" ;

— $\forall\mathbf{U}$: "pour tous les chemins, jusqu'à".

D'autres couples d'opérateurs peuvent être ajoutés :

$$\exists\Diamond\varphi \equiv \exists[\top \mathbf{U} \varphi] \quad \text{"il existe un chemin où fatalement"} \tag{2.32}$$
$$\forall\Diamond\varphi \equiv \forall[\top \mathbf{U} \varphi] \quad \text{"est inévitable"} \tag{2.33}$$
$$\exists\Box\varphi \equiv \neg\forall\Diamond\neg\varphi \quad \text{"il existe un chemin où toujours"} \tag{2.34}$$
$$\forall\Box\varphi \equiv \neg\exists\Diamond\neg\varphi \quad \text{"invariablement"} \tag{2.35}$$
$$\forall\circ\varphi \equiv \neg\exists\circ\neg\varphi \quad \text{"pour tous les chemins, à l'état suivant"} \tag{2.36}$$

Bien qu'étant chacune un sous ensemble de CTL*, CTL et LTL sont incomparables [HR04] (Fig 3.23) et [BK08] (Fig 6.27), chacune pouvant exprimer des propriétés ne pouvant l'être par l'autre. Ainsi, comme nous le verrons lors de la vérification, certaines propriétés exprimées avec un outil, ne le seront pas avec l'autre. Ces différentes logiques sont donc complémentaires pour formaliser différentes propriétés.

De même, l'approche hors ligne et l'approche en ligne ne sont pas incompatibles et ont toutes les deux leurs avantages, résumés dans la Table 2.1.

TABLE 2.1 – Comparaison entre les approche hors ligne et en ligne

	Hors ligne	**En ligne**
Investissement en temps	Substantiel	Moindre
Altération du système réel	Aucune	Substantielle
Détection d'erreurs de spécification	Selon les propriétés	Selon le moniteur
Détection d'attaques	Selon les propriétés	Selon le moniteur
Blocage d'attaques	Possible [a]	Possible
Vérification de l'intégralité du système	Oui	Non

a. Si l'attaque est modélisée, et que le système réel est modifié pour atténuer les effets de l'attaque

En modélisant notre politique de contrôle d'accès obligatoire dynamique (c'est-à-dire un moniteur) dans un modèle hors ligne, nous serons en mesure de vérifier que le système modélisé préserve ses propriétés safety, de disponibilité et de sécurité. Cette phase de vérification permet d'affiner la politique et le modèle afin d'obtenir un moniteur intégrable dans le système représenté par la modélisation. Une fois intégré dans le système réel, le moniteur doit le protéger des attaques modélisées tout en préservant les propriétés safety, de disponibilité et de sécurité du système.

2.4 Conclusion

L'état de l'art montre que la safety est l'enjeu principal de l'industrie automobile. D'autres aspects, tels que la sécurité, ont été sous estimés face à la safety, pour des raisons légitimes (peu d'interfaces exposées, risque de compromettre la safety). Ces dernières années, les

mécanismes safety se sont complexifiés, et sont en mesures d'agir sur différents aspects du véhicule. L'activation des mécanismes safety dépend du contexte de fonctionnement du véhicule. S'ils sont activés en dehors des contextes prédéfinis, ils peuvent représenter un risque safety pour les passagers. Les mécanismes safety sont maintenant exposés via des interfaces plus nombreuses, ne nécessitant pas d'accès physique et sans prendre en compte le contexte de fonctionnement du véhicule pour en contrôler l'accès.

Malgré les défenses périmétriques en place, ces interfaces ont été exploitées par des attaquants leur permettant d'accéder aux fonctions du véhicule, en dehors des contextes prévus, mettant ainsi en péril les passagers. Ainsi, les différents acteurs se retrouvent face à des problématiques de sécurité autrefois réservées aux systèmes d'information classiques qu'ils se doivent de résoudre dans le but de préserver la safety des systèmes automobiles. L'état de l'art montre que les mécanismes de sécurité existant au niveau réseau ne permettent pas d'identifier précisément l'application qui a envoyé un message sur le réseau du véhicule. De plus, ils ne prennent pas en compte la chronologie des messages envoyés pour déterminer si la décision d'envoi ou de réception est légitime. Ainsi, ils ne sont pas appropriés pour contrôler efficacement les échanges entre les différentes applications du système automobile. Les mécanismes de contrôle d'accès offrent des propriétés de sécurité fortes, au prix d'une spécification complexe qui nécessite des outils supplémentaires pour faciliter l'écriture de propriétés de sécurité. De plus, ils ne permettent pas de prendre en compte le contexte de fonctionnement du véhicule, risquant ainsi sa safety. Afin de déployer les mécanismes de cybersécurité sur les composants les plus critiques, un processus d'analyse de risques est effectué pendant le développement d'un composant pour identifier les biens à protéger. Pour vérifier que les mécanismes de contrôle d'accès ne violent pas les propriétés safety du véhicule, le model checking ou la vérification en ligne peuvent être utilisés.

Chapitre 3

Démarche générale

Comme nous l'avons vu dans le chapitre précédent, le secteur automobile a besoin de nouveaux mécanismes de sécurité pour se protéger des attaques, sans entraver les propriétés safety des systèmes. Les mécanismes existant ne permettent pas d'identifier finement les différentes applications du système automobile qui communiquent entre elles. Ainsi, il est nécessaire de mettre en place un nouveau mécanisme de contrôle d'accès obligatoire dynamique, appliqué aux messages échangés entre les applications du système automobile et vérifié pour assurer sa compatibilité avec la safety.

Un système de contrôle d'accès centralisé n'est pas adéquat pour un système automobile étant donné que tous les messages échangés ne passent pas par une entité unique (*i.e.* une application ou un ECU) pour aller d'une application à une autre. De plus, un système centralisé introduit un point de défaillance unique et de la latence pour recevoir un message et renvoyer une autorisation ou une interdiction. Enfin, en tant qu'équipementier, nous n'avons pas la vision complète du système automobile, mais seulement de certains composants, ce qui ne nous permet pas de mettre en place un mécanisme centralisé pour toute l'architecture. Pour observer les messages échangés entre les applications, le contrôle peut être distribué sur chaque domaine fonctionnel, voir même sur chacun des ECU d'un même domaine.

Dans ce chapitre, nous allons voir quelles sont les possibilités pour ajouter un mécanisme de contrôle d'accès à un domaine fonctionnel ou un ECU. Pour cela, nous commençons par voir quelles formes peut prendre le contrôle d'accès, puis nous détaillons la création de nouveaux opérateurs pour l'outil du Dr. Hugot permettant d'abstraire un mécanisme de contrôle d'accès en coupure. Ensuite, nous présentons le cas d'usage réel qui sera utilisé dans le reste du manuscrit pour tester la mise en place du contrôle d'accès. Ce cas d'usage est un produit concret pouvant être vendu par un équipementier tel que Valeo. Pour des raisons de confidentialité, nous ne donnons pas de détails sur la commercialisation de ce produit. Ensuite nous détaillons le profil de l'attaquant voulant compromettre le système, et donnons une description de la politique de contrôle d'accès visant à protéger le système de l'attaquant décrit. Enfin, nous décrivons les différents

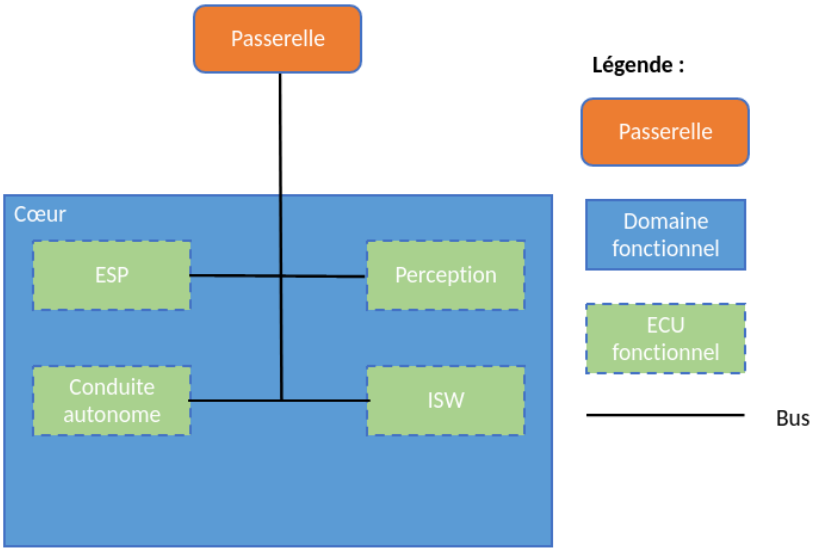


FIGURE 3.1 – Architecture simplifié du domaine fonctionnel *Cœur*

types de propriétés que nous voulons vérifier sur le système pour nous assurer de son bon fonctionnement.

3.1 Implantation du mécanisme

Comme nous l'avons vu sur la Figure 2.2 : "Exemple simplifié d'une architecture d'un véhicule avec conduite autonome" les véhicules automobiles sont organisés en différents domaines, responsables de certaines fonctions du véhicule. Chaque domaine correspond à un (ou dans certains cas, plusieurs) bus, sur lequel les messages sont diffusés entre les différents ECU, et par conséquent les différentes applications du domaine.

Par exemple, au sein du domaine fonctionnel *Cœur* (*cf.* Figure 3.1 : "Architecture simplifié du domaine fonctionnel *Cœur*"), les messages échangés entre une application de l'ECU *Perception* et une application de l'ECU *Conduite autonome* transitent uniquement sur le bus du domaine. Ainsi, une application responsable du contrôle d'accès placée dans un autre domaine fonctionnel ne serait pas en mesure de recevoir les messages échangés entre ces deux applications. De plus, un tel contrôle ajouterait de la latence si les messages devaient être bloqués. De fait, pour observer les messages échangés entre les applications d'un domaine il faut soit :

— ajouter un ECU dédié sur le bus désiré, qui consommerait tous les messages émis sur le bus en question ;

— ajouter un composant logiciel à un ou plusieurs ECU du bus.

Dans une première section, nous présentons quelle forme peut prendre le contrôle d'accès s'il correspond à un nouvel ECU. La seconde section s'intéresse à l'ajout d'un composant logiciel dédié au contrôle d'accès sur un ou plusieurs ECU du domaine.

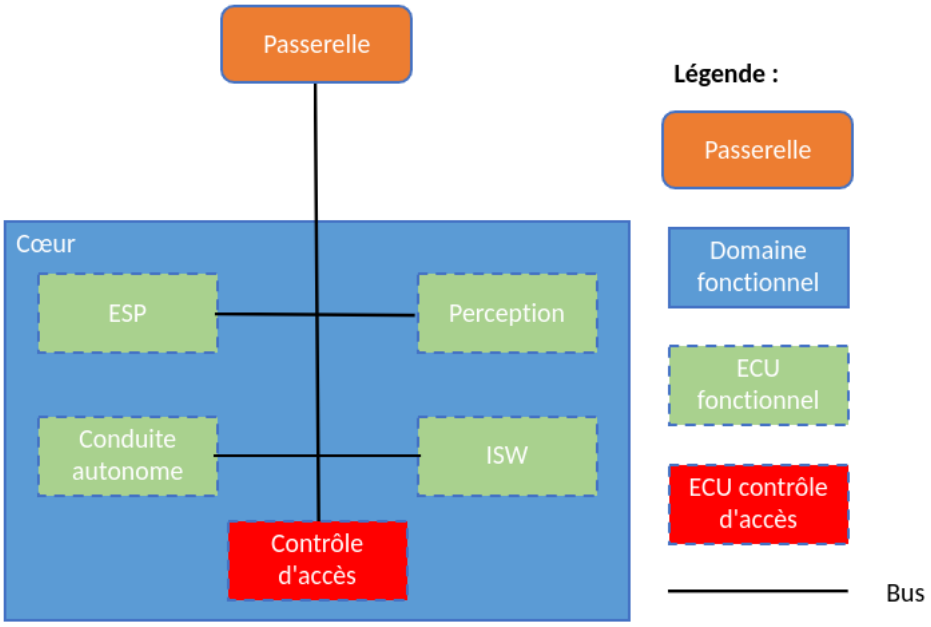


FIGURE 3.2 – Architecture au sein d'un domaine fonctionnel avec un ECU dédié au contrôle d'accès

Enfin, nous détaillons la modélisation de la diffusion des messages qui est nécessaire au sein d'un système automobile.

3.1.1 Ajout d'un ECU supplémentaire sur le bus

La première solution est de connecter un nouvel ECU hébergeant une application dédiée au contrôle d'accès au bus du domaine *Cœur*. L'architecture de cette solution est décrite en Figure 3.2 : "Architecture au sein d'un domaine fonctionnel avec un ECU dédié au contrôle d'accès". Dans un système automobile utilisant cette approche, le contrôle d'accès peut avoir deux modes de fonctionnement :

— celui d'un Intrusion Detection System (IDS) ;

— celui d'un Intrusion Prevention System (IPS).

Dans les deux cas, nous considérons que l'application de contrôle d'accès sur le nouvel ECU peut consommer tout ou partie des messages émis sur le bus par les autres ECU du domaine. De cette manière, le contrôle d'accès peut permettre de détecter les attaques visant n'importe quel ECU du domaine, ou bien visant un ECU particulier qui aurait été identifié comme plus attractif lors de l'analyse de risque.

Fonctionnement en mode IDS

Si le mécanisme de contrôle d'accès adopte le fonctionnement d'un IDS, il n'est pas en mesure de bloquer des messages étant donné qu'ils sont diffusés simultanément à tous les ECU connectés au bus. Ainsi, il est uniquement possible d'alerter sur l'occurrence

d'un message violant la politique de contrôle d'accès. Afin de répondre à une violation de la politique, le mécanisme peut avoir le fonctionnement d'un IPS.

Fonctionnement en mode IPS

En mode IPS, en plus de consommer tous les messages émis sur le bus, notre mécanisme de contrôle d'accès est capable de prendre des mesures en réponse à la détection d'un message violant la politique. En mode IPS, il serait possible de redémarrer l'application/l'ECU attaqué afin de rétablir un contexte de fonctionnement stable. Cette approche a cependant un inconvénient majeur : la capacité de demander le redémarrage d'une application à distance. L'ECU hébergeant l'IPS aurait la capacité de redémarrer toutes les applications connectées au même bus et pourrait être exploité par un attaquant afin de perturber la disponibilité du système.

Conclusion sur l'approche matérielle

Dans les deux cas présentés ci-dessus, un ECU dédié au contrôle d'accès est ajouté sur un bus du système automobile. Étant donné qu'il s'agit d'un mécanisme de détection et de réponse, sur un système utilisant de la diffusion, cette approche n'ajoute pas de latence dans le système : la détection peut être effectuée en parallèle de la réception par d'autres ECU. L'approche IDS/IPS a besoin de la diffusion pour observer ce qui transite sur le bus en parallèle du reste du fonctionnement du système, de la même manière qu'un antivirus observe le comportement des programmes d'un ordinateur sans en empêcher le fonctionnement. Effectuer le contrôle en parallèle grâce à la diffusion permet de ne pas introduire de latence, et de préserver la safety du système. Pour sécuriser un véhicule entier, composé de plusieurs bus, il faut donc plusieurs ECU dédiés au contrôle d'accès (un par bus). Or, l'industrie automobile doit limiter le poids du véhicule, sa consommation d'énergie, et son prix, ce qui limite l'applicabilité de l'approche décrite ici. De plus, comme nous l'avons vu, nous ne pouvons pas bloquer une attaque avant qu'elle se produise, mais seulement la détecter et y répondre. Ainsi, nous proposons d'explorer dans la section suivante une approche passant par l'ajout d'un composant logiciel sur les ECU d'un domaine permettant un mode de fonctionnement supplémentaire.

3.1.2 Ajout d'un composant logiciel sur les ECU

La seconde solution est d'ajouter un composant logiciel dédié au contrôle d'accès sur un ou plusieurs ECU existants connectés au domaine *Cœur*. Par exemple, sur la Figure 3.3 : "Implantation de notre mécanisme sur tous les ECU du domaine *Cœur*", tous les ECU du domaine *Cœur* embarquent un composant logiciel supplémentaire dédié au contrôle d'accès. Le contrôle d'accès ainsi ajouté peut avoir trois rôles :

— celui d'un IDS ;
— celui d'un IPS ;
— celui d'un mécanisme d'autorisation.

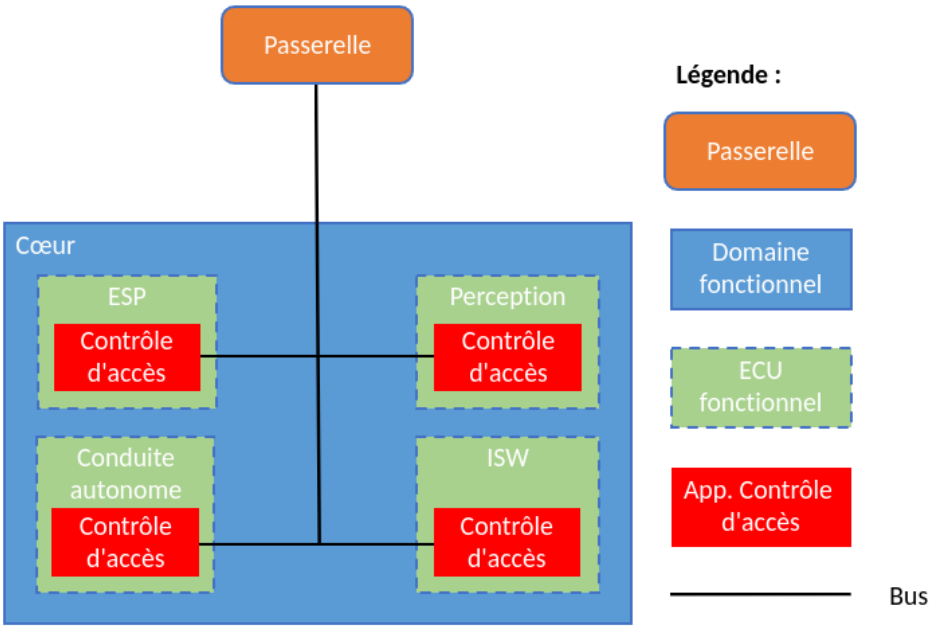


FIGURE 3.3 – Implantation de notre mécanisme sur tous les ECU du domaine *Cœur*

Fonctionnement en mode IDS ou IPS

En mode IDS ou IPS, comme dans la section précédente, le composant logiciel consomme par défaut tous les messages émis sur le bus. De fait, les ECU embarquant ce composant doivent avoir les ressources nécessaires pour analyser tous les messages émis sur le bus en plus de remplir leurs fonctions habituelles. Ainsi, cette solution occasionne une surcharge importante si elle est déployée sur tous les ECU telle quelle. Pour y remédier, l'ECU peut uniquement consommer les messages qui sont destinés à des applications qu'il héberge, permettant ainsi de limiter la puissance nécessaire pour analyser les messages reçus. Dans ce cas, chaque ECU peut détecter et répondre aux attaques le concernant (l'ECU peut se redémarrer lui-même mais n'a pas cette capacité sur les autres ECU, limitant ainsi son attractivité pour un attaquant). Si un seul ECU du bus embarque le composant dédié au contrôle d'accès (comme sur la Figure 3.4 : "Architecture au sein d'un domaine fonctionnel avec une application dédiée au contrôle d'accès sur un ECU existant"), alors il doit pouvoir consommer tous les messages émis sur le bus comme nous l'avons vu précédemment. Ainsi, on retrouve le besoin de performances pour analyser tous les messages sur un seul ECU, en plus des autres fonctions qu'il doit remplir. L'approche logicielle est intéressante, car elle s'affranchit des contraintes de poids, limite la consommation d'énergie et le coût associé en reposant sur du matériel existant. Néanmoins, elle possède certaines contraintes : les ECU en place doivent pouvoir héberger un mécanisme supplémentaire qui peut s'avérer consommateur de ressources (dans le cas où tous les messages émis sur le bus sont consommés), et le mécanisme n'est pas capable de bloquer une attaque. L'approche logicielle nous permet aussi de placer notre composant où nous le souhaitons sur le chemin de réception des messages pour en faire un mécanisme d'autorisation.

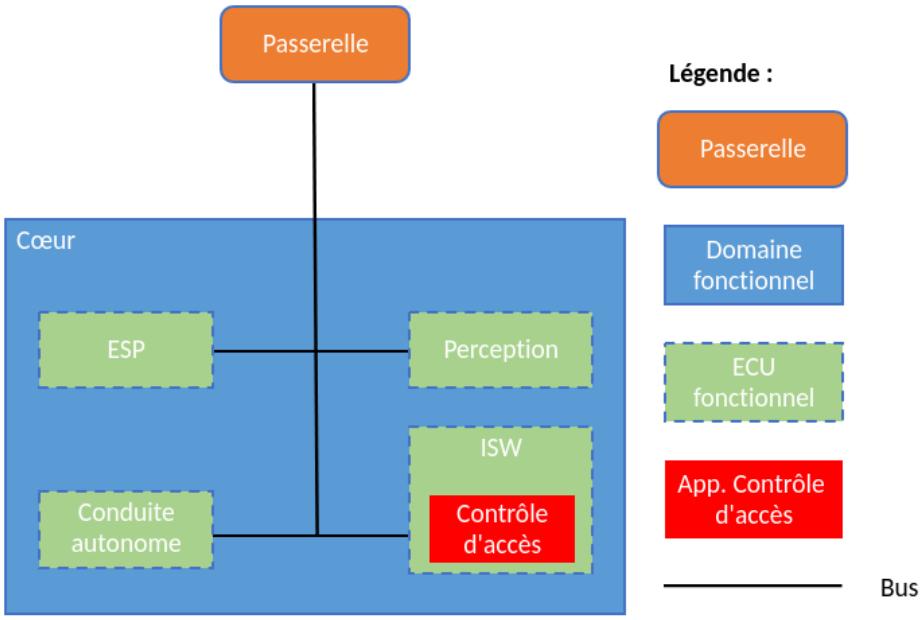


FIGURE 3.4 – Architecture au sein d'un domaine fonctionnel avec une application dédiée au contrôle d'accès sur un ECU existant

Fonctionnement en mécanisme d'autorisation

Notre mécanisme de contrôle d'accès peut aussi prendre la forme d'un mécanisme d'autorisation s'il est implémenté sous la forme d'un composant logiciel embarqué sur des ECU existants. Cette approche n'était pas compatible avec l'ajout d'un composant physique dédié pour des raisons de latence (chaque message doit être envoyé au composant de contrôle, analysé, et une autorisation doit être renvoyée). Reprenons la Figure 3.3 : "Implantation de notre mécanisme sur tous les ECU du domaine *Cœur*", mais cette fois, le composant logiciel est un mécanisme de contrôle d'accès placé en coupure de la réception des messages sur chaque ECU. Le composant ajouté consomme uniquement les messages à destination d'une application hébergée sur l'ECU considéré. Ainsi, il devient possible de bloquer un message interdit par la politique de contrôle d'accès avant même qu'il soit reçu par l'application cible et donc de protéger les applications des attaques. Le déploiement sur tous les ECU d'un même domaine est cependant difficilement atteignable. Premièrement, chaque ECU doit avoir les ressources nécessaires pour accueillir le composant de contrôle d'accès. De plus, nous devons savoir avec quelles autres applications notre ECU va communiquer afin de définir la politique de contrôle d'accès. Or, cette connaissance est entre les mains de l'OEM. Ainsi, une approche utilisant du contrôle d'accès sous la forme d'un mécanisme d'autorisation n'est pas déployable telle que présentée sur la Figure 3.3 : "Implantation de notre mécanisme sur tous les ECU du domaine *Cœur*" par un équipementier. Elle peut néanmoins être mise en place sur un unique ECU telle que présentée sur la Figure 3.4 : "Architecture au sein d'un domaine fonctionnel avec une application dédiée au contrôle d'accès sur un

ECU existant".

Conclusion sur l'approche logicielle

L'utilisation d'un composant logiciel nous permet de mettre en place un mécanisme d'autorisation qui n'était pas envisageable avec l'ajout d'un ECU pour des raisons de latence. De cette manière, les attaques peuvent être bloquées, si l'ECU a la capacité d'accueillir ce nouveau composant. De plus, l'approche logicielle permet aussi d'implémenter notre mécanisme sous la forme d'un IDS/IPS. Ces différents mécanismes peuvent être combinés grâce à un système fonctionnant principalement en diffusion. De plus, les approches IDS/IPS requièrent la diffusion des messages afin de fonctionner correctement. Or, le formalisme de vérification utilisé par l'outil du Dr. Hugot, décrit dans le Chapitre 2 : "État de l'art", n'offre pas d'opérateurs de diffusion. Nous allons donc détailler dans la section suivante les opérateurs que nous avons ajouté pour modéliser la diffusion.

3.1.3 Ajouts des nouveaux opérateurs

Dans l'état de l'art, nous avons vu que l'outil du Dr. Hugot exploite le produit synchronisé afin d'obtenir l'exécution du système global. Pour faciliter l'écriture des systèmes, nous utiliserons les notations classiques des automates d'interfaces, tels que définis par exemple dans [Mou11]. La convention est d'utiliser le suffixe ! pour un envoi de message et ? pour sa réception. Ceci s'exprime trivialement en fonction du produit synchronisé d'automates. Cependant, cette convention présente une limitation pour nos applications : lorsqu'un message est émis depuis un composant, il doit être consommé par un unique autre composant. Ainsi, dans le cas où notre mécanisme de contrôle d'accès fonctionne comme un IDS, nous ne sommes pas en mesure de modéliser la réception des messages par l'IDS et par l'ECU cible. De plus, nous avons vu que la diffusion est très présente dans les systèmes automobiles dû à la présence de bus de communication interconnectant les différents ECU. Pour modéliser fidèlement un système automobile, l'ajout d'un opérateur de diffusion est donc essentielle. De fait, il était nécessaire d'étendre le fonctionnement de l'outil du Dr. Hugot pour qu'un message puisse être diffusé à plusieurs composants.

Pour comprendre le fonctionnement des nouveaux opérateurs, nous allons observer leur fonctionnement avec un exemple. Nous commençons par rappeler le fonctionnement des opérateurs de [Mou11] puis nous détaillons celui des nouveaux opérateurs.

Rappels sur le fonctionnement

La Figure 3.5 représente un composant `Emetteur` possédant trois états : 0, 1 et 2. Dans l'état 0, l'automate emprunte une transition correspondant à l'envoi d'un premier message `message!` pour arriver dans l'état 1. Dans l'état 1, l'automate emprunte une transition correspondant à l'envoi d'un second message `autre_message!` et arrive dans l'état 2. Un composant `Destinataire` représenté sur la Figure 3.6 réceptionne le premier message

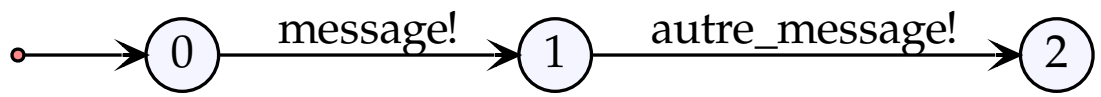


FIGURE 3.5 – Automate du composant `Emetteur`

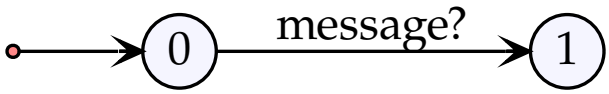


FIGURE 3.6 – Automate du composant `Destinataire`

`message?` en passant de l'état 0 à l'état 1. La synchronisation de ces deux composants produit l'automate de la Figure 3.7. Chaque état de cet automate indique dans quels états se trouvent les automates sous-jacents. Ainsi, dans le premier état, les automates `Emetteur` et `Destinataire` sont tous les deux dans leur état 0. Sur la Figure 3.7, on constate que les deux automates sont passés de leur état 0 à leur état 1 simultanément grâce à la synchronisation des transitions `message!` et `message?`. L'automate `Emetteur` effectue sa dernière transition sans se synchroniser car aucun automate ne se synchronise sur celle-ci. L'automate `Destinataire` ne change pas d'état.

Ajoutons maintenant un automate supplémentaire se synchronisant sur les mêmes transitions, utilisant les opérateurs nouvellement définis.

Fonctionnement des opérateurs synchrones bloquants

Pour cela, nous introduisons deux nouveaux opérateurs `!+, ?+`. Ces deux opérateurs correspondent à une diffusion synchrone bloquante, où le message est garanti d'arriver à tous les destinataires, en même temps. Nous allons maintenant donner une définition mathématique de ces opérateurs synchrones.

Reprenons les définitions pour $\mathcal{C}$, $\mathbb{D}(X)$ et $\mathbb{D}_{\mathrm{p}}(\Sigma)$ définies en Chapitre 2 : "État de l'art". Nous définissons $\mathcal{M}$ l'ensemble des messages synchrones, $\mathcal{D}_m$ l'ensemble des destinataires d'un message m et $\mathcal{E}_m$ l'ensemble des émetteurs pour ce même message.

$$\mathcal{M} = \{m \mid m?+ \in \bigcup_{c \in C} c.\Sigma\} \cup \{m \mid m!+ \in \bigcup_{c \in C} c.\Sigma\} \tag{3.1}$$

$$\mathcal{D}_m = \{A \mid A \in C, m?+ \in A.\Sigma\} \tag{3.2}$$

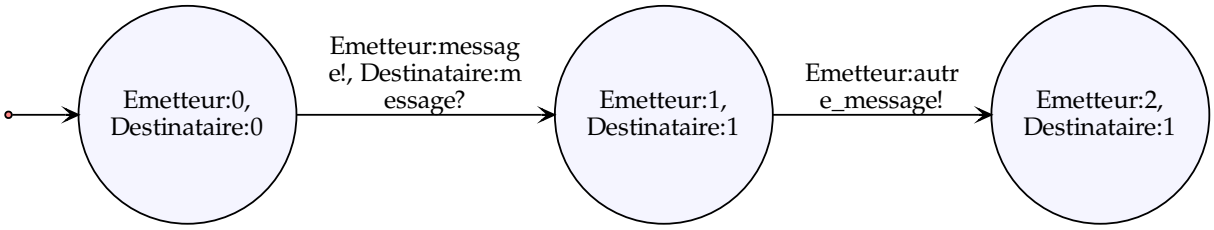


FIGURE 3.7 – Produit synchronisé des automates des composants `Emetteur` et `Destinataire`

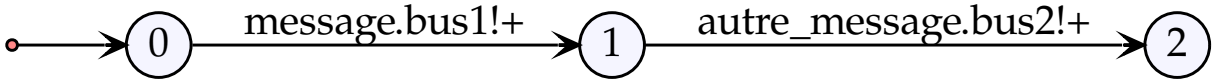


FIGURE 3.8 – Automate du composant `Emetteur` avec les opérateurs synchrones

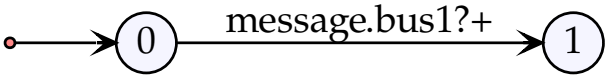


FIGURE 3.9 – Automate du composant `Destinataire` avec les opérateurs synchrones

$$\mathcal{E}_m = \{A \mid A \in C, m!+ \in A.\Sigma\} \tag{3.3}$$

$$\mathcal{E}_m \cap D_m = \varnothing \tag{3.4}$$

L'ensemble des messages synchrones $\mathcal{M}$ correspond à l'union de tous les messages réceptionnés (?+) ou envoyés (!+) de manière synchrone, de tous les composants. L'ensemble des destinataires $\mathcal{D}_m$ d'un message synchrone m est défini tel que l'ensemble des composants comportant une réception de message synchrone dans leur alphabet. L'ensemble des émetteurs $\mathcal{E}_m$ d'un message synchrone m est défini tel que l'ensemble des composants comportant un envoi de message synchrone dans leur alphabet. Ces deux ensembles doivent être distincts pour un message donné (un composant ne peut pas se synchroniser avec lui-même car il devrait emprunter deux transitions différentes, en même temps). Notre ensemble de synchronisation de dictionnaires de synchronisation pour les opérateurs synchrones est donné par

$$S = \Big\{ \{ E : m!+ \} \cup \{ D : m?+ \mid D \in D_m \} \;\Big|\; m \in \mathcal{M},\ E \in E_m \Big\} \tag{3.5}$$

$\mathcal{S}$ est un ensemble de dictionnaires, tel que chaque dictionnaire est composé d'un émetteur d'un message, et de tous les destinataires de ce même message, pour tous les messages synchrones.

Ces deux opérateurs nous permettent donc de modéliser un réseau à diffusion, sans pertes. Reprenons nos automates `Emetteur` et `Destinataire` de la section précédente. Nous modifions leurs transitions afin d'ajouter nos nouveaux opérateurs et un domaine de diffusion. En effet, si nous souhaitons modéliser un système automobile complet, avec plusieurs bus (qui sont des domaines de diffusion), nous devons pouvoir définir sur quel bus un message peut se propager. La synchronisation de ces deux composants avec l'ensemble de dictionnaires de synchronisation 3.6 produit l'automate de la Figure 3.10. Cet automate est identique à l'automate de la Figure 3.7 : "Produit synchronisé des automates des composants `Emetteur` et `Destinataire`". Ce résultat est attendu car ces opérateurs permettent de synchroniser 2 ou plusieurs automates entre eux.

$$S = \Big\{ \{ Emetteur : message.bus1!+,\ Destinataire : message.bus1?+ \}, \\ \{ Emetteur : autre\_message.bus2!+ \} \Big\} \tag{3.6}$$

Si nous ajoutons un troisième automate, décrit par la Figure 3.11 : "Automate du composant `Contrôle d'accès` avec les opérateurs synchrones" se synchronisant sur la

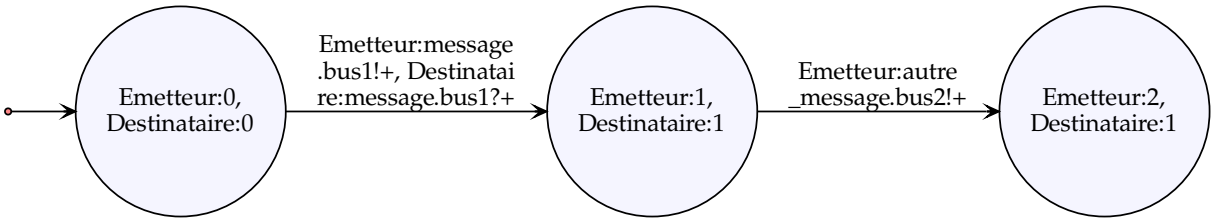


FIGURE 3.10 – Produit synchronisé des automates des composants `Emetteur` et `Destinataire` avec les opérateurs synchrones

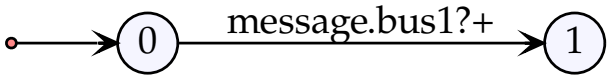


FIGURE 3.11 – Automate du composant `Contrôle d'accès` avec les opérateurs synchrones

même transition que l'automate `Destinataire`, nous obtenons l'automate produit de la Figure 3.12 : "Produit synchronisé des automates des composants `Emetteur`, `Destinataire` et `Contrôle d'accès` avec les opérateurs synchrones". L'ensemble de dictionnaires de synchronisation utilisé est le suivant :

$$S = \Big\{\{Emetteur : message.bus1!+,\ Destinataire : message.bus1?+, \\ Contrôle\ d'accès : message.bus1?+\}, \\ \{Emetteur : autre\_message.bus2!+\}\Big\} \tag{3.7}$$

Sur cet automate, nous constatons que les automates `Emetteur`, `Destinataire` et `Contrôle d'accès` évoluent tous de leur état 0 à leur état 1 sur la première transition. Comme précédemment, l'automate `Emetteur` effectue sa dernière transition sans se synchroniser.

Les opérateurs synchrones que nous avons défini ne permettent pas les pertes. De fait, si un automate ne peut pas se synchroniser sur la transition requise, alors cette transition est bloquante pour tous les automates concernés. Considérons par exemple l'automate de `Contrôle d'accès` erroné de la Figure 3.13. La première transition de cet automate se synchronise avec la seconde transition de l'automate `Emetteur`. La seconde transition de l'automate `Contrôle d'accès` se synchronise avec la première transition de l'automate

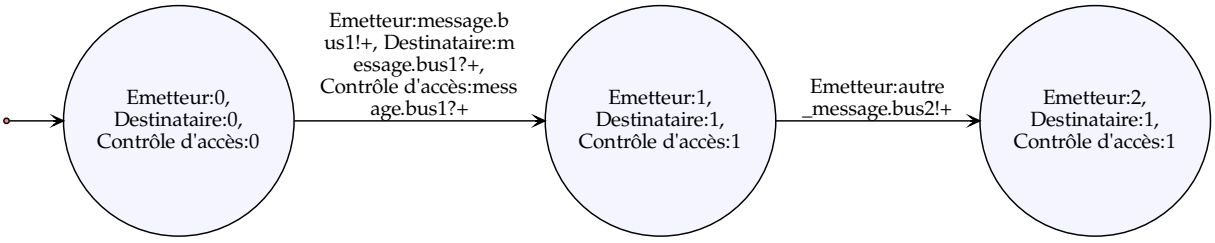


FIGURE 3.12 – Produit synchronisé des automates des composants `Emetteur`, `Destinataire` et `Contrôle d'accès` avec les opérateurs synchrones

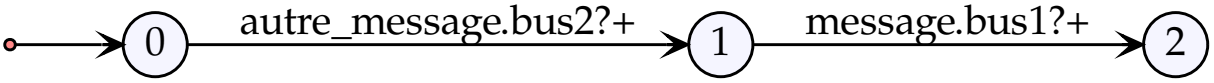


FIGURE 3.13 – Automate du composant `Contrôle d'accès` erroné avec les opérateurs synchrones

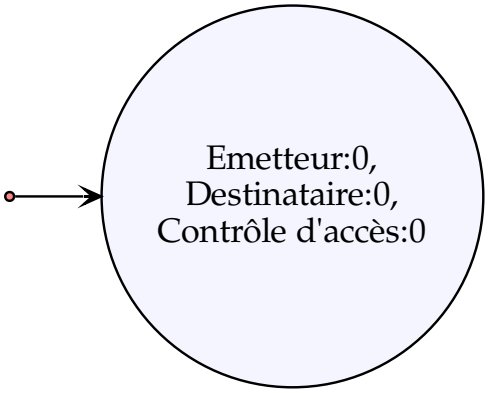


FIGURE 3.14 – Produit synchronisé des automates des composants `Emetteur`, `Destinataire` et `Contrôle d'accès` erroné avec les opérateurs synchrones

`Emetteur`. L'ensemble de dictionnaires de synchronisation est donc le suivant :

$$S = \Big\{\{Emetteur : message.bus1!+,\ Destinataire : message.bus1?+, \\ Contrôle\ d'accès : message.bus1?+\}, \\ \{Emetteur : autre\_message.bus2!+,\ Contrôle\ d'accès : autre\_message.bus2?+\}\Big\} \tag{3.8}$$

Ainsi, aucun des deux automates ne peut progresser, ce qui produit l'automate synchronisé de la Figure 3.14 où l'on constate que tous les automates sont bloqués dans leur état initial. On peut penser que l'automate `Destinataire` aurait effectué sa première transition de manière synchronisée avec l'automate `Emetteur`. Or, comme l'automate `Contrôle d'accès` requiert aussi la synchronisation sur cette transition, mais ne peut l'effectuer initialement, la transition est bloquante pour tous les états.

Ainsi, en créant un opérateur de diffusion fortement synchronisé, nous avons créé un mécanisme de coupure. Si l'automate `Contrôle d'accès` ne permet pas de se synchroniser sur une transition à un moment donné, tous les autres automates se synchronisant sur cette transition sont bloqués. Cet aspect des opérateurs synchrones nous sera très utile dans la phase de modélisation avec l'outil du Dr. Hugot de manière à abstraire simplement un mécanisme de contrôle d'accès en coupure. Il s'agit d'une abstraction pessimiste, tous les automates devant se synchroniser sont bloqués si un seul d'entre eux ne peut se synchroniser. De plus, ces opérateurs nous permettent de modéliser un bus de communication dans une situation idéale (*i.e.* sans pertes). Cet ajout nous permet de répondre à un besoin de réalisme par rapport à un système automobile (les bus fonctionnent en diffusion), mais aussi à des besoins de modélisation. En revanche, cet opérateur est trop fort pour un mécanisme à base d'IDS/IPS. En effet, si l'IDS/IPS

n'est pas en mesure de réceptionner un message, il ne doit pas bloquer l'intégralité du système.

Nos opérateurs actuels n'autorisent pas les pertes (absence de synchronisation avec certains composants), et ne sont donc pas représentatifs d'un réseau. De fait, nous introduisons deux autres opérateurs, afin de simuler des pertes dans un réseau à diffusion, qui sont détaillés dans la section suivante.

Fonctionnement des opérateurs asynchrones non bloquants

Afin de modéliser une diffusion avec pertes, nous introduisons les opérateurs asynchrones non bloquants `!*`, `?*`, qui s'apparentent à un multicast comme celui présent sur Ethernet ou UDP.

La définition mathématique de cette synchronisation est donnée ci-dessous.

Nous reprenons les définitions pour $\mathcal{C}$, $\mathbb{D}(X)$ et $\mathbb{D}_{\mathrm{p}}(\Sigma)$ définies en Chapitre 2 : "État de l'art". Nous définissons $\mathcal{M}$ l'ensemble des messages asynchrones, $\mathcal{D}_m$ l'ensemble des destinataires d'un message m et $\mathcal{E}_m$ l'ensemble des émetteurs pour ce même message.

$$\mathcal{M} = \{m \mid m?* \in \bigcup_{c \in C} c.\Sigma\} \cup \{m \mid m!* \in \bigcup_{c \in C} c.\Sigma\} \tag{3.9}$$

$$\mathcal{D}_m = \{A \mid A \in C, m?* \in A.\Sigma\} \tag{3.10}$$

$$\mathcal{E}_m = \{A \mid A \in C, m!* \in A.\Sigma\} \tag{3.11}$$

$$\mathcal{E}_m \cap D_m = \varnothing \tag{3.12}$$

L'ensemble des messages asynchrones $\mathcal{M}$ correspond à l'union de tous les messages réceptionnés (?*) ou envoyés (!*) de manière asynchrone, de tous les composants. L'ensemble des destinataires $\mathcal{D}_m$ d'un message asynchrone m est défini tel que l'ensemble des composants comportant une réception de message asynchrone dans leur alphabet. L'ensemble des émetteurs $\mathcal{E}_m$ d'un message asynchrone m est défini tel que l'ensemble des composants comportant un envoi de message asynchrone dans leur alphabet. Ces deux ensembles doivent être distincts pour un message donné (un composant ne peut pas se synchroniser avec lui-même car il devrait emprunter deux transitions différentes, en même temps). Notre ensemble de synchronisation de dictionnaires de synchronisation pour les opérateurs asynchrones est donné par

$$S \;=\; \Big\{ \{ E : m!* \} \cup \{ D : m?* \mid D \in X \} \;\Big|\; m \in \mathcal{M},\ E \in E_m,\ X \subseteq D_m \Big\} \tag{3.13}$$

$\mathcal{S}$ est un ensemble de dictionnaires, tel que chaque dictionnaire est composé d'un émetteur d'un message, et d'un sous ensemble des destinataires de ce même message, pour tous les messages asynchrones.

Cette section décrit uniquement le fonctionnement de ces opérateurs. L'étude des pertes sera abordée dans la section Section 5.3 : "Pertes de messages".

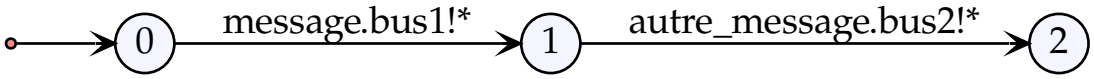


FIGURE 3.15 – Automate du composant `Emetteur` avec les opérateurs asynchrones

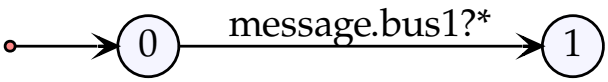


FIGURE 3.16 – Automate du composant `Destinataire` avec les opérateurs asynchrones

Reprenons l'exemple utilisé précédemment pour décrire le fonctionnement des opérateurs asynchrones non bloquants. L'automate `Emetteur` est représenté sur la Figure 3.15 et celui du `Destinataire` sur la Figure 3.16. Les deux automates décrivent les mêmes transitions que précédemment, seul l'opérateur de synchronisation sur les transitions change.

La synchronisation de ces deux automates avec l'ensemble de dictionnaires de synchronisation 3.14 produit l'automate de la Figure 3.17.

$$\begin{aligned} S = \Big\{ & \{Emetteur : message.bus1!*\}, \\ & \{Emetteur : message.bus1!*,\ Destinataire : message.bus1?*\}, \\ & \{Emetteur : autre\_message.bus2!*\} \Big\} \end{aligned} \tag{3.14}$$

Depuis l'état initial, on constate que l'automate `Emetteur` peut emprunter la transition correspondant à l'envoi du message `message.bus1!*` sans se synchroniser avec l'automate `Destinataire`. Ainsi, on arrive dans un état où `Destinataire` est toujours dans son état initial (0) et `Emetteur` est dans l'état 1, ce qui n'était pas possible avec les opérateurs précédents. Le message a été envoyé, et perdu, sans bloquer l'automate `Emetteur`.

Si nous ajoutons l'automate `Contrôle d'accès`, identique à celui de la Figure 3.11 : "Automate du composant `Contrôle d'accès` avec les opérateurs synchrones", avec les opérateurs asynchrones, nous obtenons l'automate synchronisé de la Figure 3.18. Cet automate utilise l'ensemble de dictionnaires de synchronisation 3.15.

$$\begin{aligned} S = \Big\{ & \{Emetteur : message.bus1!*\}, \\ & \{Emetteur : message.bus1!*,\ Destinataire : message.bus1?*\}, \\ & \{Emetteur : message.bus1!*,\ Contrôle\ d'accès : message.bus1?*\}, \\ & \{Emetteur : message.bus1!*,\ Destinataire : message.bus1?*, \\ & \ Contrôle\ d'accès : message.bus1?*\}, \\ & \{Emetteur : autre\_message.bus2!*\} \Big\} \end{aligned} \tag{3.15}$$

Pour passer de l'état initial à un des états suivants, 4 transitions sont possibles, indiquant avec quels autres automates la synchronisation a lieu. Par exemple, pour passer de l'état `Emetteur:0, Destinataire:0, Contrôle d'accès:0` à l'état `Emetteur:1,`

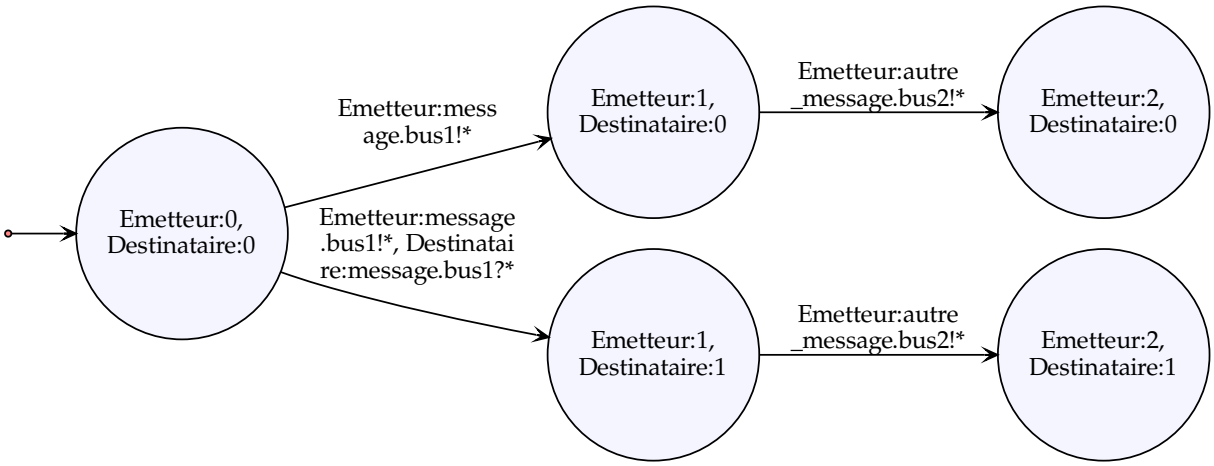


FIGURE 3.17 – Produit synchronisé des automates des composants `Emetteur`, `Destinataire` avec les opérateurs asynchrones

`Destinataire:1, Contrôle d'accès:1`, la synchronisation a eu lieu entre tous les automates. Pour aller de l'état `Emetteur:0, Destinataire:0, Contrôle d'accès:0` à l'état `Emetteur:1, Destinataire:1, Contrôle d'accès:0`, seuls les automates `Emetteur` et `Destinataire` se sont synchronisés. Les problématiques de resynchronisation seront abordées dans le Chapitre 5 : "Expérimentations supplémentaires". L'automate `Contrôle d'accès` est donc désynchronisé. Enfin, reprenons l'automate `Contrôle d'accès` erroné de la section précédente (*cf.* Figure 3.13 : "Automate du composant `Contrôle d'accès` erroné avec les opérateurs synchrones") en utilisant les opérateurs asynchrones pour observer la différence lors de la synchronisation. Nous obtenons l'automate synchronisé de la Figure 3.19 utilisant l'ensemble de dictionnaires de synchronisation 3.16. Le système peut progresser alors que l'automate `Contrôle d'accès` empêchait la synchronisation avec les opérateurs synchrones. En effet, en autorisant une perte, on autorise aussi le fait que les automates ne soient pas en mesure de se synchroniser.

$$
\begin{aligned}
S = \Big\{ & \{Emetteur : message.bus1!*\}, \\
& \{Emetteur : message.bus1!*,\ Destinataire : message.bus1?*\}, \\
& \{Emetteur : message.bus1!*,\ Contrôle\ d'accès : message.bus1?*\}, \\
& \{Emetteur : message.bus1!*,\ Destinataire : message.bus1?*, \\
& \ Contrôle\ d'accès : message.bus1?*\}, \\
& \{Emetteur : autre\_message.bus2!*\}, \\
& \{Emetteur : autre\_message.bus2!*,\ Contrôle\ d'accès : autre\_message.bus2?*\}\Big\}
\end{aligned}
\tag{3.16}
$$

Avec ces opérateurs supplémentaires, nous avons maintenant tous les éléments nécessaires pour modéliser un système automobile. Dans un premier temps, nous concentrons nos efforts sur l'ajout d'un unique mécanisme de contrôle d'accès sur un seul ECU afin de prouver l'efficacité de notre approche. Nous choisissons de protéger l'ECU Intelligent Steering Wheel (ISW), tel que présenté sur la Figure 3.4 : "Architecture au sein

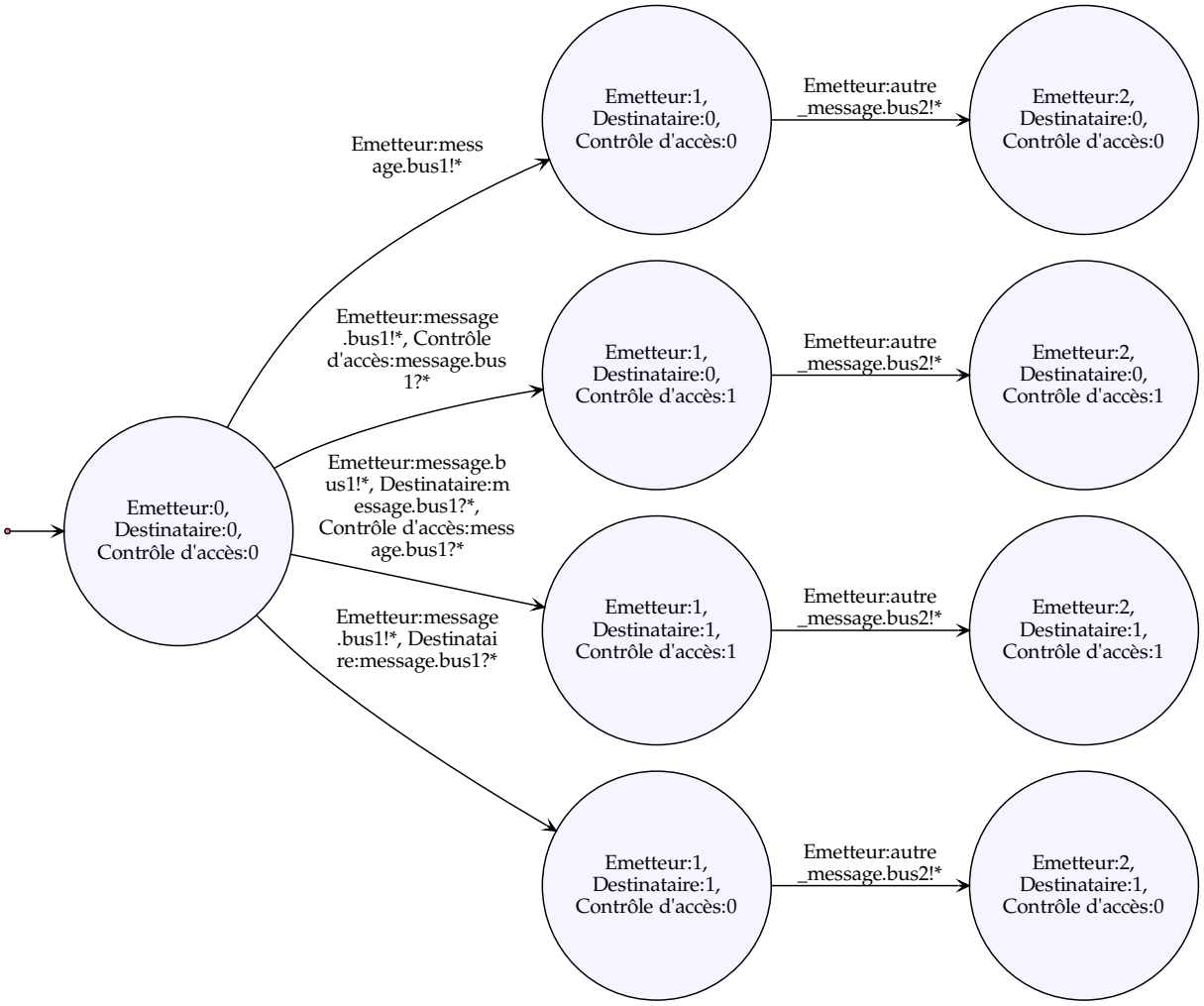


FIGURE 3.18 – Produit synchronisé des automates des composants `Emetteur`, `Destinataire` et `Contrôle d'accès` avec les opérateurs asynchrones

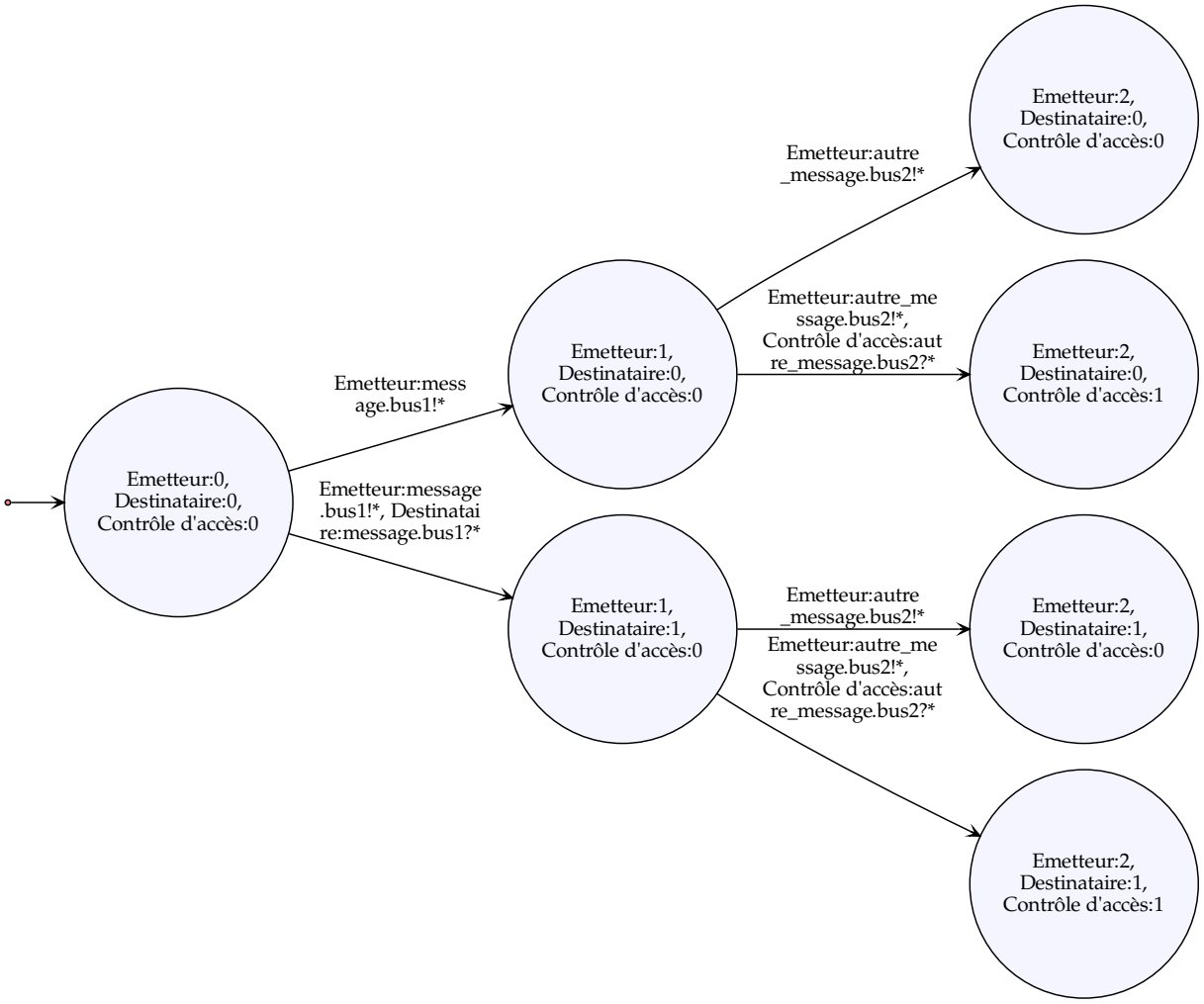


FIGURE 3.19 – Produit synchronisé des automates des composants `Emetteur`, `Destinataire` et `Contrôle d'accès` erroné avec les opérateurs asynchrones

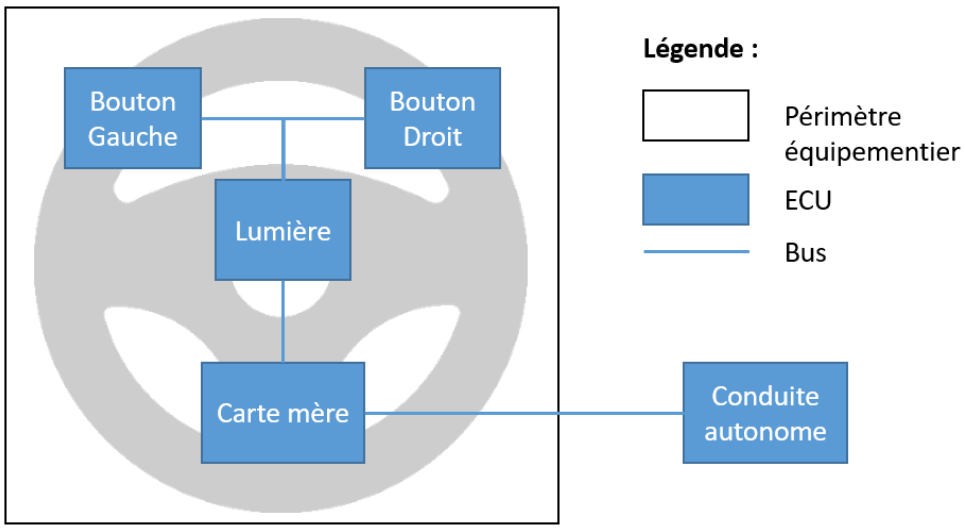


FIGURE 3.20 – Architecture physique de l'ISW

d'un domaine fonctionnel avec une application dédiée au contrôle d'accès sur un ECU existant". Dans la section suivante, nous présentons en détail l'ISW : son architecture physique, système et son comportement attendu.

3.2 Présentation du cas d'usage réel

Afin d'apporter des exemples concrets aux sections suivantes, nous allons décrire ici le cas d'usage qui est utilisé dans le reste de ce manuscrit. Ce cas d'usage est représentatif de la situation d'un équipementier dans l'industrie automobile. Comme nous l'avons détaillé dans le Chapitre 2 : "État de l'art", les équipementiers n'ont pas accès à l'intégralité des informations qui sont nécessaires pour mettre en place du contrôle d'accès fin. De fait, nous avons concentré nos efforts sur la protection de composants individuels en prenant en compte l'impossibilité pour un équipementier de maîtriser et connaître des composants externes. Pour simplifier, nous considérons un équipementier fournissant un unique ECU dans l'architecture d'un véhicule.

Notre cas d'usage est un volant intelligent, Intelligent Steering Wheel (ISW), composé de deux boutons, une lumière d'indication d'état de la conduite autonome et une carte mère. Dans un premier temps, nous allons détailler son architecture physique puis son architecture système, qui apportent des contraintes à notre approche. Ensuite, nous abordons le comportement des différents composants de l'ISW, ce qui nous permettra de comprendre les perturbations produites par l'attaquant.

3.2.1 Architecture physique de l'ISW

L'architecture physique de l'ISW est présentée sur la Figure 3.20. Chacun des éléments de l'ISW (les boutons gauche et droit, la lumière d'indication de la conduite autonome

et la carte mère) sont des composants électroniques capables de communiquer entre eux via des bus. L'ISW est composé de trois bus différents. Tous les messages envoyés sur un bus sont reçus par tous les nœuds connectés à ce même bus, comme nous l'avons vu précédemment. De fait, si un autre nœud est connecté à un des bus existants, il sera en mesure de voir les messages émis sur ce bus, mais aussi d'en émettre. Nous verrons que ce point est crucial dans la Section 3.3 : "Présentation de l'attaquant". Deux des bus de l'ISW sont internes, c'est-à-dire qu'ils connectent des composants internes à l'ISW. Le premier connecte les deux boutons à la lumière. Le second connecte la lumière à la carte mère. Le troisième bus est externe, c'est-à-dire qu'il permet de communiquer avec un ou plusieurs ECU en dehors de l'ISW. Dans notre cas, ce bus connecte la carte mère à l'ECU en charge de la conduite autonome. Cette connexion directe entre ces deux ECU est une simplification d'une architecture en production. En effet, comme nous l'avons vu sur la Figure 2.2 : "Exemple simplifié d'une architecture d'un véhicule avec conduite autonome", les architectures automobiles sont découpées en domaines – composés de multiples ECU – interconnectés par une passerelle. Dans un système réel, l'ISW ne serait probablement pas directement connecté à l'ECU en charge de la conduite autonome, ce qui ne change rien quant au besoin de communication. Cette simplification est sans conséquence et elle facilite la phase de modélisation abordée dans le Chapitre 4 : "Modélisation formelle de notre approche". Bien que présenté dans l'architecture de l'ISW de la Figure 3.20 : "Architecture physique de l'ISW", l'ECU en charge de la conduite autonome est en dehors de notre champ d'action en tant qu'équipementier et n'est donc pas instrumentable avec notre solution.

Comme nous l'avons vu sur la Figure 3.4 : "Architecture au sein d'un domaine fonctionnel avec une application dédiée au contrôle d'accès sur un ECU existant", nous voulons implanter notre mécanisme sur l'ISW. Or, nous constatons ici que l'ISW est composé de plusieurs composants physiques. Nous devons donc choisir sur quel(s) composant(s) implémenter notre solution en prenant en compte certaines contraintes. L'ECU en charge de la conduite autonome est en dehors de notre champ d'action. De plus, les cartes électroniques permettant la communication des boutons avec la lumière sont très simples et ne permettent pas l'hébergement d'une application complexe telle que du contrôle d'accès. Ainsi, seules la lumière d'indication et la carte mère peuvent accueillir notre mécanisme de contrôle d'accès.

Nous allons maintenant présenter l'architecture système de l'ISW afin de présenter le rôle de chacun des composants.

3.2.2 Architecture système de l'ISW

Les deux boutons de l'ISW sont directement accessibles au conducteur depuis son volant pour qu'il puisse activer ou désactiver la conduite autonome. Bien que directement connectés à la lumière via un bus (*cf.* Figure 3.20 et Figure 3.21), les boutons ne contrôlent pas directement son allumage. En effet, la lumière ne fait que transmettre

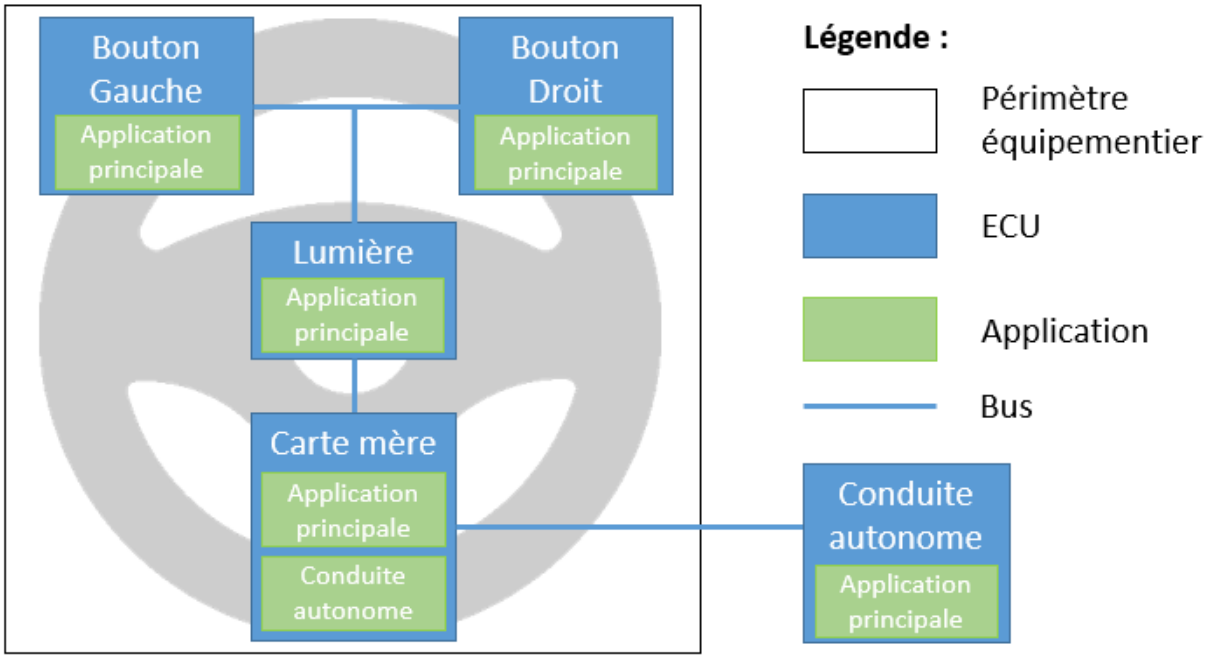


FIGURE 3.21 – Architecture système de l'ISW

l'état (*i.e.* pressé ou relâché) de chacun des boutons à la carte mère. La carte mère est responsable de la logique d'allumage et d'extinction de la lumière ainsi que de l'envoi des messages d'activation/désactivation à l'ECU en charge de la conduite autonome. Chaque composant de l'ISW comporte au moins une application, comme présenté en Figure 3.21 : "Architecture système de l'ISW". En réalité, certains composants – notamment la carte mère – de l'ISW peuvent comporter plusieurs applications permettant de remplir plus de fonctions. Nous choisissons d'ajouter une application dédiée aux communications entre la carte mère et la conduite autonome. Lors du développement d'un tel produit, cela permet de se concentrer sur les communications internes à l'ISW sans avoir besoin des spécifications de communication avec l'ECU conduite autonome. La carte mère pourrait héberger plus d'applications, mais pour des raisons de confidentialité, de simplicité et de pertinence, nous n'avons pas inclus ces applications.

Chaque application est responsable de certaines tâches afin d'implémenter le comportement de l'ISW qui est décrit dans la section suivante.

3.2.3 Comportement de l'ISW

Le comportement de l'ISW est comme suit : dans l'état initial, la conduite autonome est désactivée, la lumière d'indication est éteinte et les deux boutons sont relâchés. Lorsque les deux boutons sont pressés en même temps, la carte mère de l'ISW envoie un message pour activer la conduite autonome. Lorsque la conduite autonome est activée, la carte mère envoie un message pour allumer la lumière d'indication de conduite autonome. Lorsque les deux boutons sont à nouveau pressés, la carte mère éteint d'abord la lumière,

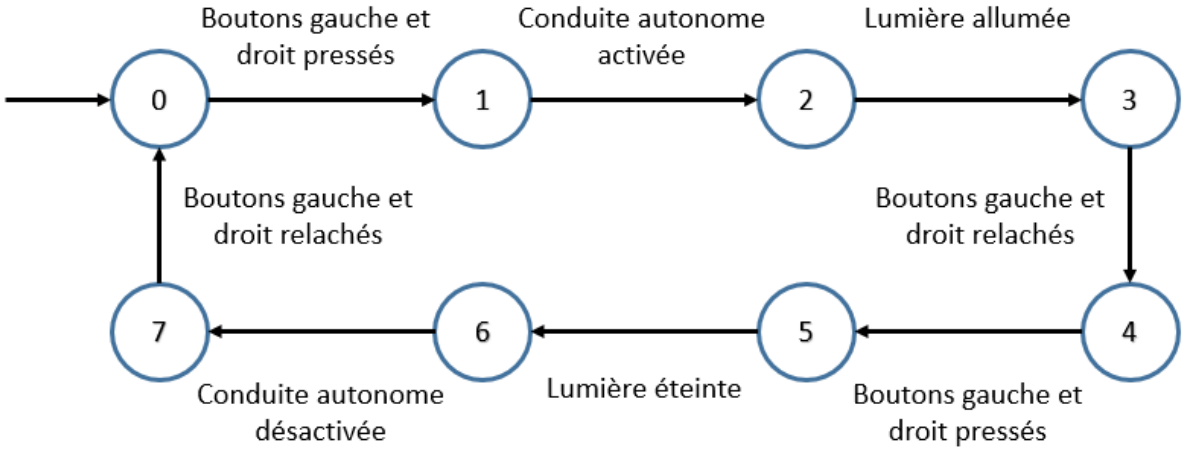


FIGURE 3.22 – Représentation sous forme d'automate du fonctionnement global de l'ISW

puis désactive la conduite autonome. Ce comportement peut être représenté sous la forme d'un automate, tel que décrit par la Figure 3.22 : "Représentation sous forme d'automate du fonctionnement global de l'ISW".

Comme nous l'avons vu dans les deux sections précédentes, les messages sont envoyés au travers de différents bus, interconnectant les composants de l'ISW. L'architecture physique et système de l'ISW apportent des contraintes supplémentaires qui modifient le comportement initialement décrit. En effet, nous avons vu que la carte mère n'est pas directement connectée aux deux boutons, elle ne connaît donc pas leur état individuel. De fait, la lumière doit transférer l'état de chacun des boutons à la carte mère pour que cette dernière puisse décider d'envoyer un message d'activation ou de désactivation de la conduite autonome et un message d'allumage ou d'extinction à la lumière.

Ainsi, le comportement de l'ISW peut être raffiné par rapport à la Figure 3.22 en y ajoutant le transfert de l'état des boutons de la lumière vers la carte mère. Cet automate raffiné, présenté en Figure 3.23 : "Représentation sous forme d'automate du fonctionnement raffiné de l'ISW" indique le transfert de l'état des boutons à chaque changement d'état. Néanmoins, cet automate n'indique pas si l'état du bouton gauche est reçu et transféré avant celui du bouton droit (et inversement). Nous constatons dans le Chapitre 4 : "Modélisation formelle de notre approche" que de nombreux entrelacements sont possibles et vont complexifier les automates représentant nos différents systèmes.

Nous allons maintenant détailler le comportement individuel de chaque composant. Pour certains composants, tout ou partie du comportement décrit est traduit en formule de logique temporelle dans le chapitre suivant.

Comportement des boutons

Les boutons gauche et droit peuvent être dans deux états : pressé et relâché. À chaque changement d'état, le bouton se synchronise avec la lumière et lui envoie un message

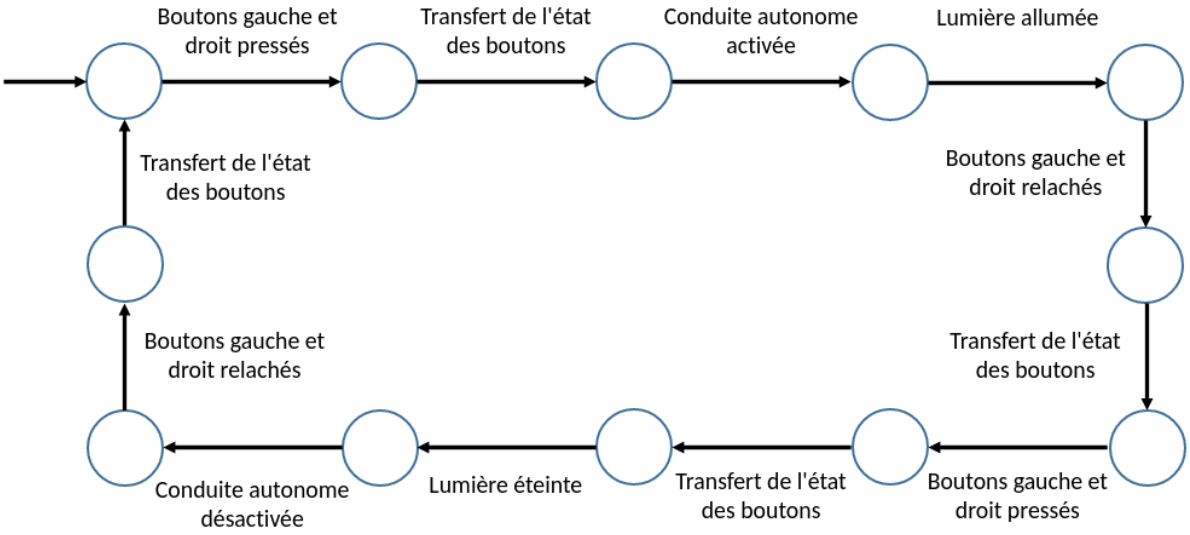


FIGURE 3.23 – Représentation sous forme d'automate du fonctionnement raffiné de l'ISW

indiquant son état actuel. Un bouton ne peut pas envoyer deux messages de suite avec un même état.

Comportement de la lumière

La lumière se synchronise avec les boutons et reçoit l'état actuel de chacun d'eux. Lorsqu'elle reçoit l'état d'un bouton (*i.e.* pressé ou relâché), elle doit envoyer un message à la carte mère indiquant quel bouton a changé d'état (*i.e.* gauche ou droit), et l'état correspondant. Elle attend aussi les messages de la carte mère lui demandant de s'allumer ou de s'éteindre.

Comportement de la carte mère

La carte mère contient toute la logique de l'ISW. Elle contrôle l'état de la lumière et de la conduite autonome en fonction des deux boutons.

Lorsque les deux boutons sont pressés, la carte mère envoie un message d'activation à la conduite autonome, suivi d'un message d'allumage de la lumière d'indication de la conduite autonome. La carte mère attend ensuite un relâchement des deux boutons pour noter en interne que l'activation est terminée.

Lorsque les deux boutons sont à nouveau pressés, la carte mère envoie un message d'extinction de la lumière d'indication, suivi d'un message de désactivation de la conduite autonome. La carte mère attend ensuite un relâchement des deux boutons pour noter en interne que la désactivation est terminée. Elle attend ensuite une prochaine pression des deux boutons pour redémarrer un cycle d'activation.

Ainsi, après chaque demande d'activation de la conduite autonome, la carte mère demande l'allumage de la lumière. De même, avant chaque désactivation de la conduite

autonome, la carte mère demande l'extinction de la lumière. De cette manière, dès lors que la lumière est allumée, le conducteur sait que la conduite autonome est activée.

Comportement de la conduite autonome

Comme nous l'avons déjà précisé, l'ECU en charge de la conduite autonome est hors de notre portée en tant qu'équipementier. Ainsi, la comportement attendu vise uniquement à s'interfacer avec le reste du système. La conduite autonome comporte deux états : activée et désactivée. Le passage d'un état à un autre dépend des messages reçus de la carte mère.

3.2.4 Conclusion sur le comportement de l'ISW

Nous avons détaillé le comportement général de l'ISW ainsi que de chacun de ses composants plus en détails. Avec le détail de ces comportements, l'ISW met en avant l'importance d'une politique de contrôle d'accès sûre pour l'automobile. Étant donné que la lumière indique au conducteur s'il doit faire attention à la route, il est critique qu'elle reflète de manière fiable l'état de la conduite autonome. Si la lumière est allumée alors que la conduite autonome est désactivée, les passagers du véhicule se trouvent dans une situation non sûre où il y a violation de la safety, et où leur absence d'attention sur la route pourrait être fatale. De fait, une politique de contrôle d'accès pour l'ISW doit empêcher l'allumage de la lumière lorsque la conduite autonome est désactivée, afin d'éviter de tromper le conducteur. De plus, la politique doit préserver le comportement attendu de l'ISW, et doit donc toujours permettre d'activer la conduite autonome et d'allumer la lumière.

Ainsi, nous devons avoir la garantie que notre politique de contrôle d'accès préserve ces deux propriétés de safety, c'est-à-dire être sûre vis-à-vis des propriétés exprimées et doit préserver la disponibilité du système. Pour vérifier que la politique de sécurité mise en place est efficace nous allons présenter dans la section suivante le modèle de notre attaquant qui va perturber le fonctionnement du système. Il s'agit d'offrir un modèle d'attaque qui soit le plus large possible afin de pouvoir contrer un grand nombre de scénarios d'attaques.

3.3 Présentation de l'attaquant

Afin de tester notre politique de contrôle d'accès sur le système, nous devons savoir si elle permet de nous protéger d'une attaque. Dans le cas contraire, mettre en place la politique est inutile, elle ajoute des contraintes (temps de contrôle), sans apporter de bénéfices (sécurité supplémentaire).

Pour l'ISW, nous souhaitons nous protéger d'un allumage intempestif de la lumière. L'envoi d'un message pour allumer la lumière depuis une entité extérieure ou depuis

un ECU contrôlé par un attaquant peut mener à deux situations :

(1) la lumière s'allume alors que la conduite autonome est désactivée (*i.e.* menant à une situation anormale, en dehors du comportement nominal du système : les deux boutons n'ont pas été pressés) ;

(2) la lumière s'allume alors que la conduite autonome est activée (*i.e.* menant à une situation normale, dans le comportement nominal du système, mais pouvant contourner certaines contraintes du système).

La situation 1 est plus grave que la 2 étant donné qu'elle trompe le conducteur, l'amenant à penser qu'il peut relâcher son attention sur la route, alors que la conduite autonome n'est pas activée. On se trouve dans une situation où la safety des passagers n'est pas garantie. La situation 2 ne met pas en danger les passagers du véhicule, la lumière est allumée et la conduite autonome aussi. En revanche, elle contourne certaines obligations et ne permet pas au conducteur de maîtriser la conduite autonome. Ce faisant, il est possible que le système se retrouve dans un état non prévu où certains messages ne peuvent plus être traités, créant ainsi une situation de déni de service (Denial of Service, DoS). Nous concentrons nos efforts sur la situation 1, étant donné que l'impact de la situation 2 dépend de l'implémentation du système.

Notre attaquant a un objectif simple : il veut commander la lumière. Pour prendre en compte différents scénarios d'attaque, nous considérons qu'il peut envoyer son message à n'importe quel moment, directement à la lumière, autant de fois qu'il le souhaite. Cet aspect de l'attaquant est modélisé avec une formule de logique temporelle dans le Chapitre 4 : "Modélisation formelle de notre approche".

Notre politique de contrôle d'accès doit donc permettre de protéger le système de l'attaquant présenté. La politique empêche l'allumage de la lumière si la conduite autonome n'a pas été activée par le conducteur. La section suivante détaille les différentes étapes de construction de cette politique.

3.4 Construction de la politique de contrôle d'accès dynamique

Avec la définition de notre cas d'usage, du lieu d'implantation de notre mécanisme et du profil de notre attaquant, nous pouvons maintenant définir la politique de contrôle d'accès que nous allons appliquer au système.

Dans un premier temps, nous donnons le principe de fonctionnement de notre politique ainsi que les notations que nous allons utiliser. Nous définissons ensuite la politique de contrôle d'accès pour l'ISW.

3.4.1 Principe général

Contrairement aux mécanismes historiques, notre mécanisme de contrôle d'accès dynamique basé sur les automates ne doit pas nécessiter la spécification de l'intégralité du fonctionnement nominal du système. Ainsi, plutôt que de fonctionner sur le principe "Tout ce qui n'est pas explicitement autorisé est implicitement interdit", notre politique fonctionne sur le principe suivant : "Tout ce qui est explicitement autorisé à certaines applications, est implicitement refusé aux autres" présenté par [Ven15]. Ce principe a deux avantages :

(1) les règles implicitement interdites sont automatiquement calculées ;

(2) tout ce qui n'est pas dans une règle d'autorisation ou d'interdiction est implicitement autorisé pour tous.

De fait, seules les fonctions pouvant avoir un impact safety ou sur la sécurité en cas de mauvaise utilisation, doivent être manuellement autorisées ou interdites. Les autres fonctions sont considérées comme du bruit fonctionnel et seront implicitement autorisées. L'écriture de la politique requiert donc une expertise safety, fonctionnelle et en sécurité, prenant ainsi en compte les spécificités et exigences de ces différents métiers. L'expertise fonctionnelle est requise pour déterminer par quels flux fonctionnels sont traduites les différentes fonctions du système. Ces flux correspondent à l'échange d'un ou plusieurs messages ordonnés, entre des applications du système, permettant de remplir une fonction du système. Par exemple, avec notre cas d'usage de l'ISW, la fonction "Allumer la lumière", se fait par l'échange d'un message depuis une application de la carte mère, vers une application de la lumière. Il existe donc un flux

$$F_1 : Carte\ mère : conduite\_autonome \xrightarrow{\{Allumage\}} Lumière : application\_principale$$

observable entre ces deux applications. Ainsi nous sommes en mesure de représenter les échanges entre deux applications présentes sur un même ECU ou non avec la structure de flux suivante :

$$ECU_A : application_A \xrightarrow{\{Permission\}} ECU_B : application_B$$

avec :

$$\begin{aligned} ECU_A &= \text{Un ECU du système considéré} \\ ECU_B &= \text{Un ECU du système considéré différent ou non de l'} ECU_A \\ application_A &= \text{Une application hébergée par l'} ECU_A \\ application_B &= \text{Une application hébergée par l'} ECU_B \end{aligned}$$

3.4.2 Politique applicable à l'ISW

Prenons l'exemple de notre ISW. Sans politique de contrôle d'accès, un ECU connecté sur le bus entre la lumière et la carte mère est capable d'envoyer des messages à la

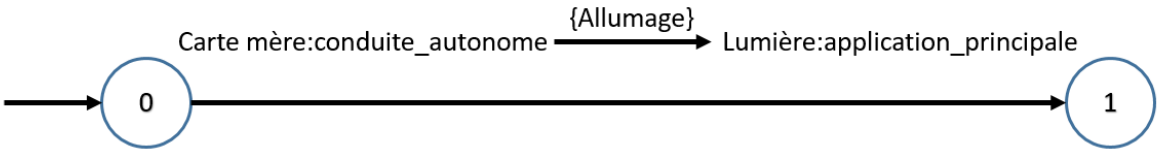


FIGURE 3.24 – Politique de contrôle d'accès basée sur les automates

lumière pour l'allumer alors que la conduite autonome est désactivée. De même, si une application (différente de celle allumant la lumière) présente sur la carte mère est compromise, alors elle pourra envoyer des messages pour allumer la lumière alors que la conduite autonome est désactivée. Dans ces deux situations, la safety des passagers n'est plus garantie car l'intégrité du système a été compromise : la lumière ne reflète plus l'état de la conduite autonome. La mise en place d'une politique de contrôle d'accès dynamique doit empêcher l'allumage de la lumière tant que la conduite autonome n'est pas activée afin de ne pas mettre en danger les passagers.

Première politique de contrôle d'accès pour l'ISW

Avec le flux F_1, nous pouvons mettre en place une politique de contrôle d'accès dynamique telle que présentée sur la Figure 3.24. Sur cette figure, notre politique de contrôle d'accès est représentée sous la forme d'un automate. Cet automate indique que dans l'état 0, un message d'allumage peut être échangé entre l'application `Carte mère:conduite_autonome` vers l'application `Lumière:application_principale`. Lorsque ce message est observé, l'automate arrive dans l'état 1, et l'échange de ce message n'est plus possible.

Cette politique est pertinente car elle permet uniquement à l'application `conduite_autonome` de l'ECU `Carte mère` d'envoyer le message `Allumage`. Elle permet aussi à l' `application_principale` de l'ECU `Lumière` de réceptionner le message `Allumage` envoyé par l'application `Carte mère:conduite_autonome`. Ainsi, elle permet de se protéger du cas où un acteur extérieur au système envoie ce message. Cette attaque, décrite en Section 3.3 : "Présentation de l'attaquant", met en danger les occupants du véhicule et doit donc être évitée par la mise en place de la politique de contrôle d'accès. Avec cette politique, on constate qu'une des attaques présentée en Section 3.3 : "Présentation de l'attaquant" peut toujours avoir lieu. En effet, l'`application_principale` de la `Lumière` va refuser tous les messages d'allumage qui ne proviennent pas de l'application `conduite_autonome` de la `Carte mère`. De fait, nous bloquons toutes les attaques provenant d'une application différente (hébergée sur un ECU légitime ou non), qui enverrait un message d'allumage. Néanmoins, en considérant que notre attaquant est en mesure de prendre le contrôle de l'application `conduite_autonome` de la `Carte mère`, l'envoi du message d'allumage sera autorisé qu'il soit légitime ou non. Pour savoir quand l'envoi d'un message est légitime, il est nécessaire de prendre en compte le contexte de fonctionnement du système.

Deuxième politique de contrôle d'accès pour l'ISW

Pour protéger l'allumage de la lumière en cas de compromission de la carte mère, il faut définir dans quel contexte le message d'allumage peut être envoyé – c'est-à-dire quels sont les pré-requis sur l'état du système pour permettre l'envoi de ce message. Pour l'ISW, afin de ne pas tromper le conducteur et de ne pas mettre en danger les passagers, la lumière doit être allumée uniquement lorsque la conduite autonome est activée. Ainsi, le système doit remplir les deux objectifs d'intégrité suivant, et ce même en cas d'attaque :

(1) la lumière est allumée après avoir activé la conduite autonome ;

(2) la lumière est éteinte avant de désactiver la conduite autonome.

Dans un état normal du système, ces deux objectifs sont remplis par la carte mère sans nécessiter la mise en place d'une politique de contrôle d'accès. C'est la présence d'un attaquant qui rend nécessaire la mise en place de la politique pour préserver le fonctionnement du système.

Pour remplir les deux objectifs, nous devons observer 4 flux :

$$F_1 : Carte\ mère : conduite\_autonome \xrightarrow{\{Allumage\}} Lumière : application\_principale$$

$$F_2 : Carte\ mère : conduite\_autonome \xrightarrow{\{Extinction\}} Lumière : application\_principale$$

$$F_3 : Carte\ mère : conduite\_autonome \xrightarrow{\{Activation\}} Conduite\ autonome : application\_principale$$

$$F_4 : Carte\ mère : conduite\_autonome \xrightarrow{\{Désactivation\}} Conduite\ autonome : application\_principale$$

À partir des flux F_1 à F_4, chaque objectif peut être traduit en un automate, tel que présenté sur la Figure 3.25 et la Figure 3.26, représentant notre politique de contrôle d'accès dynamique. L'ajout du pré-requis à l'envoi d'un message d'allumage de la lumière ou de la désactivation de la conduite autonome correspond à l'ajout d'une transition dans les automates.

Cette politique, composée de deux automates, tient compte de l'état réel du système puisqu'elle prend en considération l'état des composants et la chronologie. Dans l'état 0 de la Figure 3.25, seul l'échange d'un message d'activation depuis l'application `Carte mère:conduite_autonome` vers l'application `Conduite autonome:application_principale` est autorisé. L'observation de ce message amène l'automate dans l'état 1 où seul l'échange d'un message d'allumage depuis l'application `Carte mère:conduite_autonome` vers l'application `Lumière:application_principale` est autorisé. L'observation de ce message ramène l'automate dans l'état 0, permettant ainsi un nouveau cycle d'activation

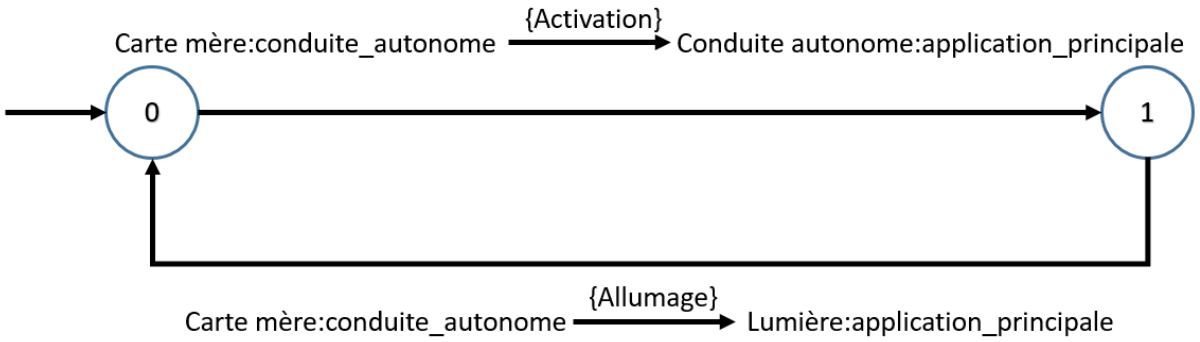


FIGURE 3.25 – Automate de contrôle d'accès pour l'objectif 1

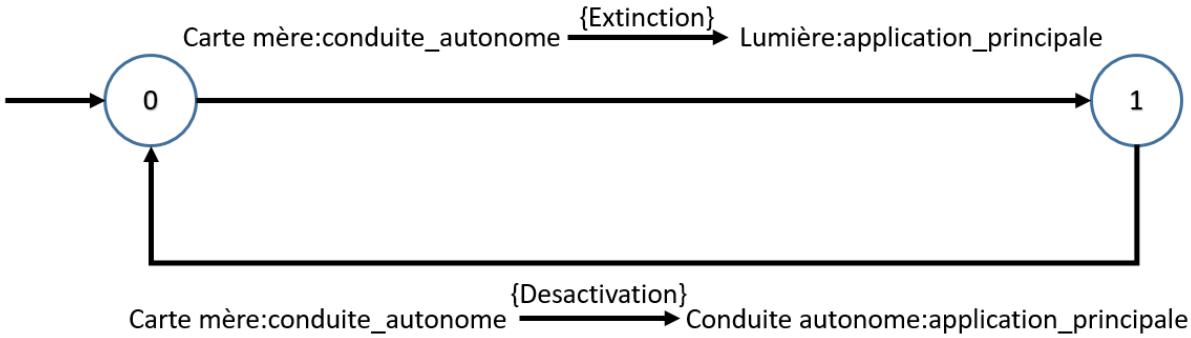


FIGURE 3.26 – Automate de contrôle d'accès pour l'objectif 2

qui correspond au comportement d'un système réactif. À partir de ces deux objectifs, nous pouvons déduire plusieurs éléments. Tout d'abord, cette politique a pour but de préserver l'intégrité fonctionnelle du système. Nous cherchons à garantir que le système aura le comportement pour lequel il a été conçu et qu'il évoluera d'une manière qui n'est pas dommageable aux passagers.

Cette politique décrit un aspect du comportement fonctionnel nominal du système, qui est connu des équipes en charge du développement. Sans cette connaissance métier, il n'est pas possible de mettre en place une politique de contrôle d'accès. En effet, comment pouvons-nous savoir quelles applications peuvent communiquer entre elles dans une situation nominale ? Dans une situation exceptionnelle (accident) ? Les équipes de développement ont la connaissance, parfois informelle, des communications entre les applications d'un même ECU. Ainsi, elles sont en mesure d'indiquer quels sont les échanges possibles dans une situation nominale ou une situation exceptionnelle. Du côté de la sécurité, les analyses de risques permettent d'identifier des chemins d'attaque ainsi que l'impact des attaques sur le système. Grâce aux informations des équipes de développement, nous pouvons mettre en place une politique d'autorisation des échanges fonctionnels qui sont censés se dérouler (*e.g.* l'application `conduite_autonome` de la `Carte mère` peut envoyer un message pour allumer la lumière). De cette manière, nous garantissons que ces échanges pourront se produire et ne pourront pas être effectués par d'autres (*e.g.* l'`application_principale` de la `Carte mère` ne peut pas allumer la lumière).

Ainsi, seule l'application `conduite_autonome` de la carte mère peut allumer la lumière, et ce uniquement après avoir activé la conduite autonome (*cf.* Figure 3.25). En cas de compromission de l'application `conduite_autonome`, les envois de messages pour allumer la lumière ne sont plus inconditionnellement autorisés.

Néanmoins, cette politique n'est pas encore idéale car elle applique indépendemment les deux objectifs ce qui peut produire des résultats inattendus. Par exemple, si l'application `conduite_autonome` de la `Carte mère` est compromise, il est possible d'échanger des messages afin d'obtenir la séquence de flux suivante :

$$F_3 : Carte\ mère : conduite\_autonome \xrightarrow{\{Activation\}} Conduite\ autonome : application\_principale$$

$$F_2 : Carte\ mère : conduite\_autonome \xrightarrow{\{Extinction\}} Lumière : application\_principale$$

$$F_4 : Carte\ mère : conduite\_autonome \xrightarrow{\{Desactivation\}} Conduite\ autonome : application\_principale$$

$$F_1 : Carte\ mère : conduite\_autonome \xrightarrow{\{Allumage\}} Lumière : application\_principale$$

Cette séquence aboutit à l'allumage de la lumière alors que la conduite autonome est désactivée. Bien qu'un pré-requis soit présent pour contraindre l'envoi des messages relatifs à la lumière, on constate qu'un risque safety subsiste si les automates de contrôle d'accès ne sont pas correctement spécifiés.

Troisième politique de contrôle d'accès pour l'ISW

Afin de limiter ce risque safety, il est nécessaire de lier l'automate de la Figure 3.25 et celui de la Figure 3.26 tel que présenté en Figure 3.27. Avec l'automate de la Figure 3.27 l'état de la lumière est conditionné par celui de la conduite autonome. De fait, il n'est pas possible d'allumer la lumière sans avoir préalablement activé la conduite autonome. Il n'est pas non plus possible de désactiver la conduite autonome sans éteindre la lumière auparavant. Étant donné que nous travaillons dans un système réactif, nous devons pouvoir procéder à plusieurs activations/désactivations. De fait, l'automate de la Figure 3.27 boucle sur lui même.

3.4.3 Placement de la politique de contrôle d'accès

Jusqu'ici, nous avons observé le fonctionnement de notre politique sans évoquer sur quel composant elle peut être placée. Nous avons vu dans la Section 3.2 : "Présentation du cas d'usage réel" que l'ISW est un ECU composé de plusieurs composants. Certains composants ne peuvent pas héberger notre mécanisme de contrôle d'accès car ils sont trop peu performants (*e.g.* les boutons) ou bien en dehors de notre champ d'action (*e.g.*

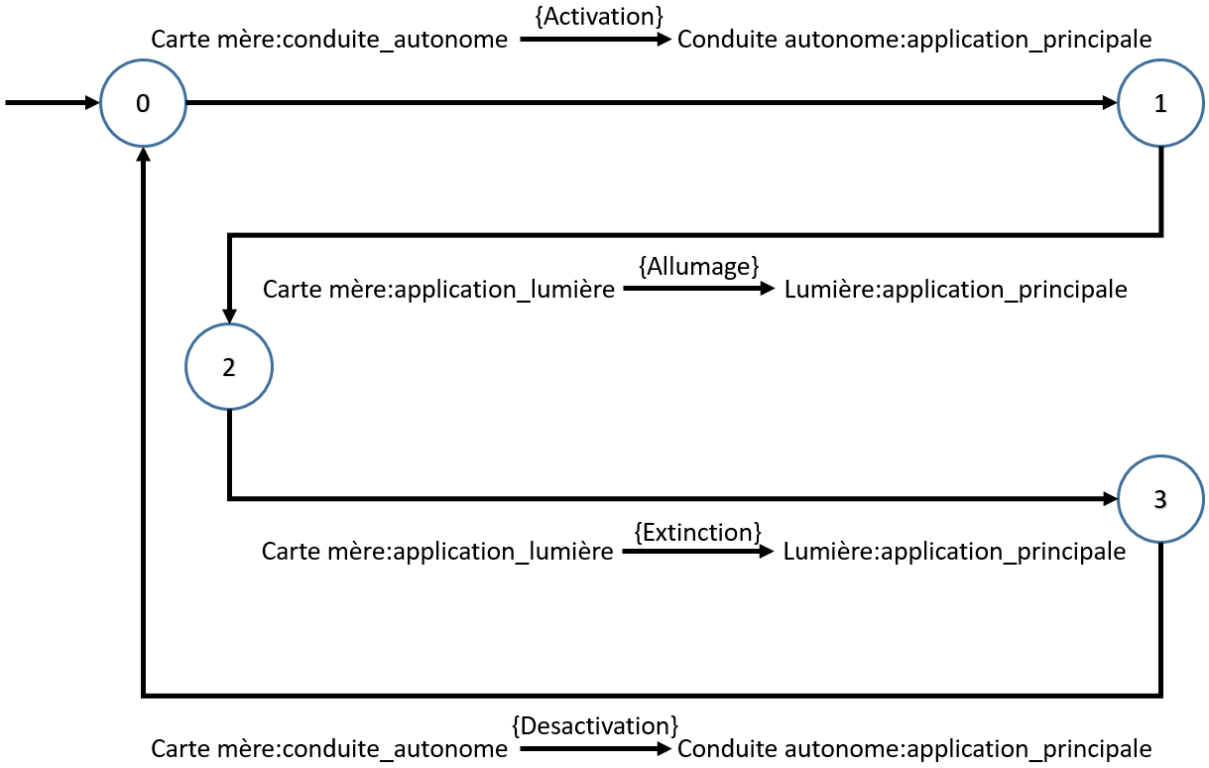


FIGURE 3.27 – Unification des objectifs 1 et 2

la conduite autonome). De fait, seules la carte mère et la lumière pourraient héberger notre mécanisme. Pour rappel, la politique de contrôle d'accès que nous souhaitons appliquer est décrite par l'automate de la Figure 3.27 : "Unification des objectifs 1 et 2".

Chaque transition de l'automate de la politique indique l'échange d'un message entre une application source et une application destination. Ainsi, si la politique est placée sur l'ECU hébergeant l'application source, la politique autorise (ou non) l'envoi du message de l'application source vers l'application destination. Si la politique est placée sur l'ECU hébergeant l'application destination, la politique autorise (ou non) la réception du message envoyé par l'application source vers l'application destination. La même politique peut donc être utilisée en envoi et en réception.

On constate que tous les messages contrôlés par la politique de la Figure 3.27 sont échangés entre la carte mère et un autre composant du système (*e.g.* la lumière ou la conduite autonome). De fait, la carte mère semble être l'emplacement idéal pour placer notre mécanisme de contrôle d'accès étant donné qu'elle accède à tous les messages nécessaires à l'évolution de la politique. La carte mère est la source de tous les messages échangés dans cette politique. De fait, appliquer la politique à la carte mère revient à autoriser ou non l'envoi de ces messages depuis la carte mère. Ainsi, un attaquant prenant le contrôle d'une application de la carte mère sera contraint par la politique de contrôle d'accès en place pour envoyer des messages sur le bus. Néanmoins, en plaçant notre mécanisme de contrôle d'accès sur la carte mère, un attaquant connecté

directement sur le bus entre la carte mère et la lumière peut envoyer des messages à la lumière sans subir les contraintes de la politique.

Pour protéger la lumière en toutes circonstances, il faut appliquer la politique de contrôle d'accès directement sur la lumière. Appliquer la politique à la lumière revient à autoriser ou non la réception de certains messages. Étant donné que la lumière contrôle tous les messages qui lui sont destinés, les messages envoyés par un attaquant connecté au bus ne seront pas autorisés. De même, les messages provenant d'une application compromise de la carte mère seront contraints par la politique, ils devront respecter l'enchaînement qu'elle impose afin d'être reçus par la lumière.

Néanmoins, la lumière n'est pas connectée au même bus que la conduite autonome (*cf.* Figure 3.20 : "Architecture physique de l'ISW"). Les messages échangés entre la carte mère et la conduite autonome sont donc inaccessibles à la lumière, empêchant l'automate de la politique de contrôle d'accès d'avancer. Pour que le contrôle d'accès soit placé sur la lumière, il est donc nécessaire d'ajouter une étape de synchronisation pour que la lumière ait connaissance de l'état de la conduite autonome.

3.4.4 Conclusion sur la politique de contrôle d'accès

Les règles de notre politique sont principalement des autorisations données à certaines applications, leur permettant d'envoyer ou de recevoir des messages. Si une autorisation est explicitement donnée à une application, alors elle est implicitement refusée aux autres.

Il est important de noter que notre politique permet à un attaquant de simuler les actions des boutons pour activer la conduite autonome sans interactions du conducteur. Elle ne nous protège pas de toutes les attaques qui existent, mais uniquement de celle décrite dans la Section 3.3 : "Présentation de l'attaquant". Pour se prémunir d'autres attaques, d'autres politiques sont nécessaires pour préserver la **safety** du système.

Avec notre politique définie nous pouvons maintenant passer à la définition des différentes propriétés que nous allons utiliser pour vérifier la modélisation du système.

3.5 Types de propriétés à vérifier

Afin de nous assurer que la modélisation du système a bien le comportement décrit en Section 3.2.3 : "Comportement de l'ISW", nous définissons certaines propriétés que la modélisation va devoir vérifier. Nous nous intéressons aux types de propriétés suivants :

(1) **safety**, les plus pertinentes pour notre cas d'usage ;

(2) disponibilité ;

(3) intégrité fonctionnelle ;

(4) cybersécurité.

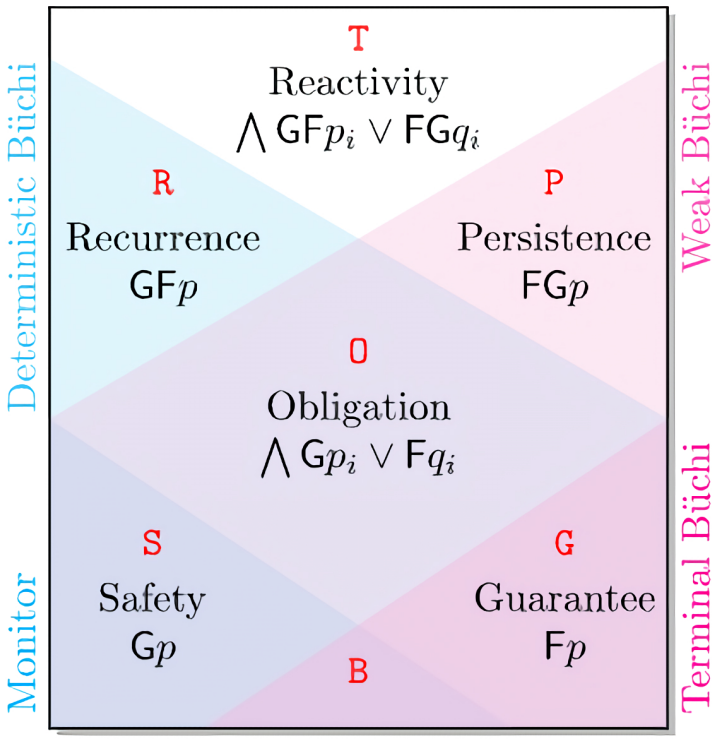


FIGURE 3.28 – Classification des propriétés de logique temporelle d'après [MP90] (figure reprise de [(a)])

Ces types de propriétés sont à distinguer des types de propriétés définis en logique temporelle. Les différentes propriétés exprimables en logique temporelle peuvent être classifiées comme le montre la Figure 3.28. Sur la Figure 3.28, la classification dépend du type d'automate permettant d'implémenter la propriété considérée. Par exemple, les propriétés de récurrence (de la forme "infiniment souvent p") sont implémentables par des automates de Büchi déterministes (des ω-automates dont la condition d'acceptation est de passer infiniment souvent par un état final). Les propriétés de safety ("toujours p") sont implémentables par des moniteurs (des automates finis dont tous les états sont finaux). Cela est lié au fait qu'à partir d'une trace finie d'exécution, on peut savoir si la propriété de safety a été violée (l'évènement redouté s'est produit), ou si elle est toujours vérifiée (l'évènement redouté ne s'est pas encore produit). Les moniteurs n'étant pas nécessairement complets, certains mots peuvent être rejetés, résultant en une violation de la propriété implémentée. En revanche, les moniteurs ne sont pas en mesure de conclure sur une trace finie pour une propriété de la forme "infiniment souvent p", étant donné que la reconnaissance d'une trace infinie est nécessaire pour ce type de propriété. Ces moniteurs ressemblent aux moniteurs de cybersécurité que nous utilisons dans ce manuscrit, bien que les notres soient moins précis que ceux utilisés en logique temporelle. Sur la Figure 3.28, on retrouve certains termes utilisés aussi dans d'autres domaines, et n'ayant pas le même sens (par exemple le terme **safety**). Toute propriété de

(a). https://spot.lrde.epita.fr/hierarchy.html

la forme $\Box p$ ($\Box$ et **G** sont des notations équivalentes) est une propriété de safety au sens de [MP90]. Or, dans le secteur automobile, une propriété de **safety** vise uniquement à préserver la vie humaine. Ainsi, une propriété telle que $\Box freins\_fonctionnels$ est une propriété safety au sens de classification de [MP90], mais aussi au sens automobile. En revanche, $\Box gps\_fonctionnel$ est une propriété safety au sens de [MP90], mais pas au sens automobile – l'absence de GPS ne risque pas de causer des blessures ou la mort des occupants du véhicule.

Ainsi, les propriétés **safety** (au sens automobile) nous permettent de vérifier que le système se trouve dans une configuration où la vie des passagers n'est pas en danger [ISO18, Syn]. Par exemple : "lorsque la lumière d'indication de la conduite autonome est allumée, alors la conduite autonome est activée". Autrement dit, il n'existe pas de configurations où la lumière est allumée et la conduite autonome est désactivée – ce qui serait trompeur pour le conducteur. Ce sont les propriétés les plus importantes pour un système automobile. Elles doivent être vérifiées dans toutes les configurations, même lorsque les autres propriétés ne sont pas vérifiées.

Les propriétés de disponibilité sont là pour vérifier que le système peut toujours remplir les fonctions pour lesquels il a été conçu [oST18]. Plus particulièrement, nous nous intéressons au fait que le système reste utilisable sur une certaine période de temps, correspondant à son exécution. Par exemple, la propriété : "la lumière d'indication de la conduite autonome peut toujours s'allumer" est une propriété de disponibilité. Ces propriétés peuvent être plus ou moins étendues en fonction du nombre d'actions prises en compte pour décrire une partie du comportement du système. Par exemple, "une fois que la conduite autonome est activée, alors un jour la lumière s'allumera et une fois que la lumière s'éteint alors un jour la conduite autonome se désactivera". Si on prend en compte les pressions et les relâchements des boutons, ces propriétés peuvent devenir assez complexes : "si les deux boutons ne sont pas infiniment souvent pressés/relâchés, alors fatalement la lumière s'allumera".

Les propriétés d'intégrité fonctionnelles visent à garantir que le système remplit bien son rôle, c'est-à-dire qu'il est intègre [Hol15, Mar22]. Par exemple : "l'allumage de la lumière implique l'activation de la conduite autonome". Cette propriété garantit que si la lumière est allumée, alors il n'existe pas de configurations où la conduite autonome n'est pas activée. En revanche, cette propriété ne garantit pas que la lumière s'allumera un jour. De plus, il est aussi possible que la conduite autonome s'active alors même que la lumière est éteinte.

Enfin les propriétés de cybersécurité correspondent à un comportement du système que l'on va chercher à garantir car une attaque visant ce comportement a été identifiée. Par exemple : "la lumière ne peut être allumée qu'après l'activation de la conduite autonome". Cette propriété peut là encore être enrichie en ajoutant par quel composant peut être effectuée l'action : "la lumière ne peut être allumée par la carte mère qu'après l'activation de la conduite autonome".

Lors des différentes étapes de vérification, nous vérifierons des propriétés de ces différents types, traduites dans un langage compris par l'outil utilisé. Par conséquent, certaines propriétés ne pourront être vérifiées uniformément avec ces deux outils. Notre attention se porte particulièrement sur les propriétés de **safety** qui sont les plus importantes dans un système automobile. Les différents types de propriétés ne sont pas mutuellement exclusifs. Ainsi, une propriété de type disponibilité peut faire partie d'une propriété fonctionnelle. Par exemple : "Si la conduite autonome est activée, et que les boutons ne sont pas infiniment souvent pressés/relâchés, alors fatalement la lumière s'allumera". Toutes les propriétés ne sont pas exprimables dans les deux formalismes utilisés dans le chapitre suivant. Par conséquent, certaines propriétés ne sont pas vérifiables par les deux outils.

3.6 Conclusion

Dans ce chapitre, nous avons détaillé les différents éléments nécessaires à la mise en place et au test de notre mécanisme de contrôle d'accès. Pour cela, nous avons commencé par présenter les différents modes de fonctionnement qu'un mécanisme de contrôle d'accès pourrait adopter (IDS, IPS, autorisation) en fonction de son emplacement dans un système automobile. Nous avons ensuite défini de nouveaux opérateurs de modélisation afin de représenter les différents modes de communication présents dans un système automobile. Puis, nous avons présenté le cas d'usage réel de l'ISW que nous allons utiliser dans la suite du manuscrit pour tester notre approche. Une attaque a aussi été définie, visant à perturber le fonctionnement de l'ISW. Il est important de noter que d'autres attaques sont possibles sur l'ISW avec différents impacts. Nous avons ensuite exposé la construction de la politique de contrôle d'accès permettant de contrer l'attaque précédemment décrite. Intuitivement, on peut penser que notre politique permettra toujours d'allumer la lumière lorsque cela est nécessaire. Cependant, comment garantir que la politique a bien les effets désirés ? De plus, comment s'assurer que la politique n'interfère pas avec d'autres fonctionnalités du système ? Par exemple en bloquant totalement l'allumage de la lumière d'indication, ou l'activation de la conduite autonome. Pour s'assurer que notre politique dynamique de contrôle d'accès ne perturbe pas le fonctionnement du système, nous allons la vérifier formellement au regard d'un certain nombre de propriétés. Cette vérification nécessite une modélisation du système à vérifier, ainsi qu'un ensemble de propriétés que le système doit vérifier. Ces propriétés peuvent être de différents types (**safety**, disponibilité, fonctionnelle, cybersécurité), que nous avons présentés dans la dernière section de ce chapitre. Dans le prochain chapitre, nous passons à la vérification des propriétés requises sur le système modélisé.

Chapitre 4

Modélisation formelle de notre approche

Notre mécanisme de contrôle d'accès doit être en mesure de bloquer l'envoi ou la réception de messages qui vont à l'encontre de la politique en place. Ainsi, si l'accès n'est pas autorisé, le message ne sera jamais réceptionné par l'application destination, ce qui peut avoir un impact safety sur le véhicule. L'intégration de notre politique de contrôle d'accès dynamique peut donc avoir un impact safety sur le véhicule si sa spécification n'est pas correcte.

Bien que notre mécanisme soit plus adapté qu'une politique de contrôle d'accès statique par son nombre de règles plus réduit, et la prise en compte de l'état du véhicule, il est aussi plus complexe car évoluant dynamiquement en fonction des messages observés. De fait, il est difficile de garantir l'innocuité de la politique.

Par conséquent, la vérification de la compatibilité de la politique est une étape charnière pour envisager sa mise en place dans un système réel. Cette vérification doit garantir que la politique de sécurité n'a pas d'impacts sur la safety, la disponibilité et l'intégrité au regard des capacités de l'attaquant.

Pour vérifier la politique, nous utilisons deux outils de vérification formelle qui vont nous permettre d'explorer de manière systématique l'ensemble des états du système modélisé. Le premier outil a été développé par le Dr. V. Hugot [Hug20] et propose une représentation graphique de l'ensemble des états. Cette représentation est très utile pour se familiariser avec les concepts de la vérification formelle et facilite l'interprétation. Le second, SPIN [Hol97, Hol11] est un outil industriellement reconnu [Kar96] offrant des abstractions supplémentaires pour spécifier des systèmes plus complexes au prix d'une plus grande expertise en matière de vérification. Ce chapitre aborde la modélisation et la vérification de notre système avec l'outil du Dr. V. Hugot puis avec SPIN. Pour chaque outil, nous enrichissons de manière itérative la modélisation afin d'observer les impacts sur les propriétés à vérifier.

4.1 Modélisation avec le framework du Dr. Hugot

La première étape de notre approche est de modéliser le fonctionnement de notre système avec la framework du Dr. Hugot qui est intuitive à prendre en main. Les sous-systèmes sont modélisés directement en automates d'états finis (classiques ou synchronisés [dAH01c, Mou11]), et l'outil se charge de calculer le *produit synchronisé* des sous-systèmes pour obtenir l'automate décrivant le comportement global du système.

4.1.1 Modélisation de l'ISW

Pour notre ISW, le fonctionnement global peut être assez simplement représenté sous forme d'un automate (simplifié) comme nous l'avons vu sur la Figure 3.22 : "Représentation sous forme d'automate du fonctionnement global de l'ISW".

Cet automate représente bien le comportement de l'ISW tel que décrit en Section 3.2 : "Présentation du cas d'usage réel". Cependant, cet automate ne permet pas au conducteur de relâcher un des deux boutons avant que la lumière soit allumée. De plus, on ne sait pas dans quel ordre les deux boutons sont pressés ou relâchés. Enfin, sur la Figure 3.20 : "Architecture physique de l'ISW" on constate que les deux boutons sont reliés à la lumière et non directement à la carte mère, la lumière doit transmettre le statut des boutons à la carte mère, ce qui produit l'automate de la Figure 3.23 : "Représentation sous forme d'automate du fonctionnement raffiné de l'ISW".

On comprend que les nombreuses actions possibles, provenant de l'utilisateur du système ou de l'extérieur, complexifient la construction manuelle de l'automate de l'ISW. La solution est donc de décomposer le système global en plusieurs automates, chacun décrivant un aspect du système global, puis de les synchroniser sur certaines transitions. L'ISW est composé de plusieurs sous-systèmes (les boutons, la lumière d'indication de la conduite autonome et la carte mère) qui communiquent au travers d'un réseau (*cf.* Figure 3.20 : "Architecture physique de l'ISW") ce qui en fait un système distribué. Ainsi, chaque sous-système va être représenté par son propre automate, ce qui devrait rendre la spécification des sous-systèmes plus simple, voir triviale pour certains d'entre eux. Nous pourrons ensuite synchroniser les automates représentant chaque sous-système afin d'obtenir l'évolution globale dans toute sa complexité.

Afin de savoir quels sont les sous-systèmes à modéliser, il convient de définir quelles sont les frontières du système global. Toute modélisation est par nature un exercice d'abstraction. L'ISW fait partie du système d'information du véhicule, qui évolue dans un environnement non contrôlé, où des évènements inattendus peuvent se produire. Par exemple, une bascule de bit liée à un rayon cosmique sur la zone mémoire stockant la politique est un évènement possible de l'environnement opérationnel. Cette bascule peut transformer une autorisation en interdiction, et avoir un impact safety sur le véhicule. Néanmoins, ce type d'évènement étant peu probable, il est peu pertinent à modéliser. Ainsi, notre modélisation ne prend pas en compte certains éléments environnementaux

externes au véhicule. Cependant, nous introduisons la notion d'attaquant qui est l'élément externe qui nous intéresse. Ses capacités sont modélisées et nous donnons les preuves d'efficacité de notre politique au regard du profil de l'attaquant. Par ailleurs, nous choisissons de simplifier ou d'exclure de la modélisation certains éléments logiciels afin de conserver une spécification exploitable. Par exemple, les aspects réseaux sont simplifiés (pas de tampon d'envoi ou de réception, pas de temps de transport), et d'autres applications présentes sur les ECU ne sont pas modélisées (diagnostic, chien de garde [(a)]). Ces simplifications ne sont pas sans risques, certains comportements du système sont omis, certains évènements non modélisés, et cela peut avoir un impact sur le système final. Néanmoins, cette première approche est nécessaire avant d'introduire plus de raffinements afin de mieux prendre en considération tout l'environnement. Finalement, nous reviendrons sur ce point car la simplification du monde réel étant impossible, on introduit généralement des modes d'urgence.

Nous allons maintenant décrire la modélisation de chacun des sous-systèmes de l'ISW tels qu'ils ont été modélisés avec l'outil du Dr. Hugot.

Modélisation des boutons

Pour modéliser les boutons qui permettent à l'utilisateur d'activer ou désactiver la conduite autonome, nous utilisons le code suivant :

```
1  button_left = NFA.spec("""
2      0 # Etats initiaux
3      __ # Etats finaux (ici aucun)
4      0 L_pressed.button! 1 # Transition de l'etat 0 (initial) à l'état 1
5      1 L_released.button! 0""").named("Button Left") # Transition de l'état 1 (initial) à l'état 0
        et nommage de l'automate
6
7  button_right = NFA.spec("""
8      0
9      __
10     0 R_pressed.button! 1
11     1 R_released.button! 0""").named("Button Right")
```

Listing 4.1 – Code de génération des automates des boutons

Les parties qui nous intéressent dans le Listing 4.1 sont en vert. La première ligne définit la liste des états initiaux – ici 0. La seconde ligne définit les états finaux (nous n'en définissons pas car notre système global – l'ISW – n'a pas d'état final, il réagit indéfiniment aux actions du conducteur). Les lignes suivantes décrivent les transitions du système. Par exemple, la ligne `0 L_pressed.button! 1` décrit une transition allant de l'état 0 vers l'état 1 avec comme label `L_pressed.button!`. Ainsi, nous pouvons décrire intuitivement nos automates correspondants aux Figure 4.1 : "Automate de spécification du bouton gauche" pour le bouton gauche et Figure 4.2 : "Automate de spécification du bouton droit" pour le bouton droit.

(a). https://fr.wikipedia.org/wiki/Chien_de_garde_(informatique)

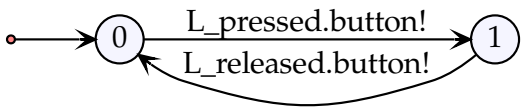


FIGURE 4.1 – Automate de spécification du bouton gauche

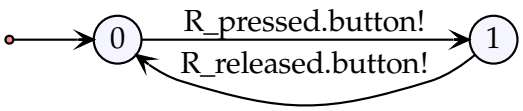


FIGURE 4.2 – Automate de spécification du bouton droit

Nous pouvons maintenant passer à la modélisation de la lumière.

Modélisation de la lumière

Pour rappel, la lumière reçoit l'état de chacun des boutons et transfert cet état à la carte mère, qui sera en charge d'effectuer les actions nécessaires en fonction de l'état du système. L'automate de la lumière est décrit par le code du Listing 4.2 : "Code de génération de l'automate de la lumière". Là où les boutons doivent uniquement connaître leur état (*i.e.* pressé ou relâché), la lumière doit connaître son état (*i.e.* allumée ou éteinte) ainsi que si elle doit transmettre à la carte mère des informations sur l'un des boutons (*i.e.* le bouton gauche a été pressé, le bouton droit a été relâché...). Ainsi, chaque état de la lumière est représenté par 5 tuples. L'état initial de la lumière est présenté sur la ligne 2 du Listing 4.2. Le premier tuple `("Light", 0)` indique que la lumière est éteinte dans son état initial. Les quatres tuples suivants indiquent si la lumière a des informations à transmettre à propos de l'état de chaque bouton à la carte mère. Par exemple, le second tuple `("LeftPFwd", 0)` indique que la lumière n'a pas d'informations à envoyer concernant une pression du bouton gauche. Si le tuple avait été `("LeftPFwd", 1)`, cela aurait indiqué que la lumière doit envoyer un message à la carte mère pour avertir de la pression du bouton gauche. On retrouve la même construction pour le bouton droit, pour la pression et le relâchement. Ici encore, nous ne donnons pas d'états finaux (ligne 3). Nous ne donnons pas non plus de transitions car celles-ci sont décrites programmatiquement par la fonction `extend(q)`.

Dans cette fonction, chaque état est traité comme un dictionnaire (*cf.* ligne 14), chaque tuple correspond donc à un couple clef :valeur dans le dictionnaire. En fonction de ces couples (*cf.* lignes 16, 13, 28...), un nouveau dictionnaire est construit (qui représente donc un nouvel état), dont on change une ou plusieurs valeurs selon le comportement voulu, puis on ajoute une transition entre l'état courant, le nouveau dictionnaire créé et un label pour la transition. En tant qu'exemple, prenons les lignes 16 à 22 du Listing 4.2 qui décrivent l'ajout d'une transition pour recevoir un message d'allumage de la lumière. Si la lumière est éteinte dans l'état courant (`if qdict["Light"] == 0:`), alors on crée un nouveau dictionnaire, où l'on passe la valeur de la clef `"Light"` à 1 (*cf.* ligne 20). Ensuite, on ajoute une transition à l'automate, entre l'état courant `q` et le nouvel état `tuple(newd.items())` avec le label souhaité, ici `"light_on.can?+"` (*cf.* ligne 22). Nous appelons ensuite la fonction `growtofixpoint()` avec les paramètres nécessaires. Cette fonction permet de jouer chacune des transitions décrite dans `extend()` sur les états de l'automate. Tant que de nouveaux états sont générés, on continue de jouer les transitions sur les nouveaux

états créés, permettant ainsi de construire la totalité de l'automate.

Autre particularité de la lumière, les lignes 28, 37, 46 et 55 qui conditionnent la réception d'une pression ou bien d'un relâchement d'un des boutons. La réception ou le relâchement d'un bouton est uniquement autorisé s'il n'y a pas déjà un évènement de ce type à transmettre à la carte mère. Cela permet d'éviter les états où la lumière doit envoyer deux messages à la carte mère : un message indiquant la pression d'un bouton et un autre indiquant le relâchement de ce même bouton.

```
1  isw_light = NFA(
2      {(("Light", 0), ("LeftPFwd", 0), ("RightPFwd", 0), ("LeftRFwd", 0), ("RightRFwd", 0))}, #
       Etats initiaux
3      set(), # Etats finaux
4      set(), # Transitions
5      name="Light ECU", # Nom de l'automate
6      worder=tuple
7  )
8
9  def grow(A):
10     has = False
11
12     def extend(q):
13         nonlocal has
14         qdict = dict(q)
15         # Si dans l'état courant la lumière est éteinte
16         if qdict["Light"] == 0:
17             # Un nouvel état est créé
18             newd = qdict.copy()
19             # Dans lequel la lumière est allumée
20             newd["Light"] = 1
21             # Et une transition est ajoutée entre ces deux états
22             has = A.try_rule(q, "light_on.can?+", tuple(newd.items())) or has
23         if qdict["Light"] == 1:
24             newd = qdict.copy()
25             newd["Light"] = 0
26             has = A.try_rule(q, "light_off.can?+", tuple(newd.items())) or has
27
28         if qdict["LeftPFwd"] == 0 and qdict["LeftRFwd"] == 0:
29             newd = qdict.copy()
30             newd["LeftPFwd"] = 1
31             has = A.try_rule(q, "L_pressed.button?", tuple(newd.items())) or has
32         if qdict["LeftPFwd"] == 1:
33             newd = qdict.copy()
34             newd["LeftPFwd"] = 0
35             has = A.try_rule(q, "L_pressed_forward.can!", tuple(newd.items())) or has
36
37         if qdict["RightPFwd"] == 0 and qdict["RightRFwd"] == 0:
38             newd = qdict.copy()
39             newd["RightPFwd"] = 1
40             has = A.try_rule(q, "R_pressed.button?", tuple(newd.items())) or has
41         if qdict["RightPFwd"] == 1:
```

```
            newd = qdict.copy()
            newd["RightPFwd"] = 0
            has = A.try_rule(q, "R_pressed_forward.can!", tuple(newd.items())) or has

        if qdict["LeftRFwd"] == 0 and qdict["LeftPFwd"] == 0:
            newd = qdict.copy()
            newd["LeftRFwd"] = 1
            has = A.try_rule(q, "L_released.button?", tuple(newd.items())) or has
        if qdict["LeftRFwd"] == 1:
            newd = qdict.copy()
            newd["LeftRFwd"] = 0
            has = A.try_rule(q, "L_released_forward.can!", tuple(newd.items())) or has

        if qdict["RightRFwd"] == 0 and qdict["RightPFwd"] == 0:
            newd = qdict.copy()
            newd["RightRFwd"] = 1
            has = A.try_rule(q, "R_released.button?", tuple(newd.items())) or has
        if qdict["RightRFwd"] == 1:
            newd = qdict.copy()
            newd["RightRFwd"] = 0
            has = A.try_rule(q, "R_released_forward.can!", tuple(newd.items())) or has

    for q in A.Q.copy():
        extend(q)
    return has

isw_light.growtofixpoint(grow, record_steps=True)
```

Listing 4.2 – Code de génération de l'automate de la lumière

Cette modélisation produit l'automate des Figures 4.3 et 4.4. Cet automate comprend 18 états et 66 transitions. Nous évitons ainsi la spécification manuelle complexe d'un tel automate. De plus, une description sous forme de programme pour les transitions est très adaptée lorsque l'on décrit un comportement plus complexe, tel que celui de la carte mère, comme nous allons le voir dans la section suivante.

Modélisation de la carte mère

La carte mère de l'ISW est responsable de la gestion de la lumière (envoi des messages d'allumage/extinction) et de la conduite autonome (envoi des messages d'activation/-désactivation). Ces actions sont effectuées en fonction de l'état de chacun des deux boutons, qui est communiqué par la lumière.

La modélisation de la carte mère est donnée par le Listing 4.3 : "Code de génération de l'automate de la carte mère". L'état interne de la carte mère est composé de cinq tuples. Le premier et le deuxième tuple indiquent respectivement si les boutons gauche et droit sont pressés ou non. Le troisième indique si la conduite autonome est activée, le quatrième si la lumière est activée. Le dernier tuple nous permet de forcer un relâchement

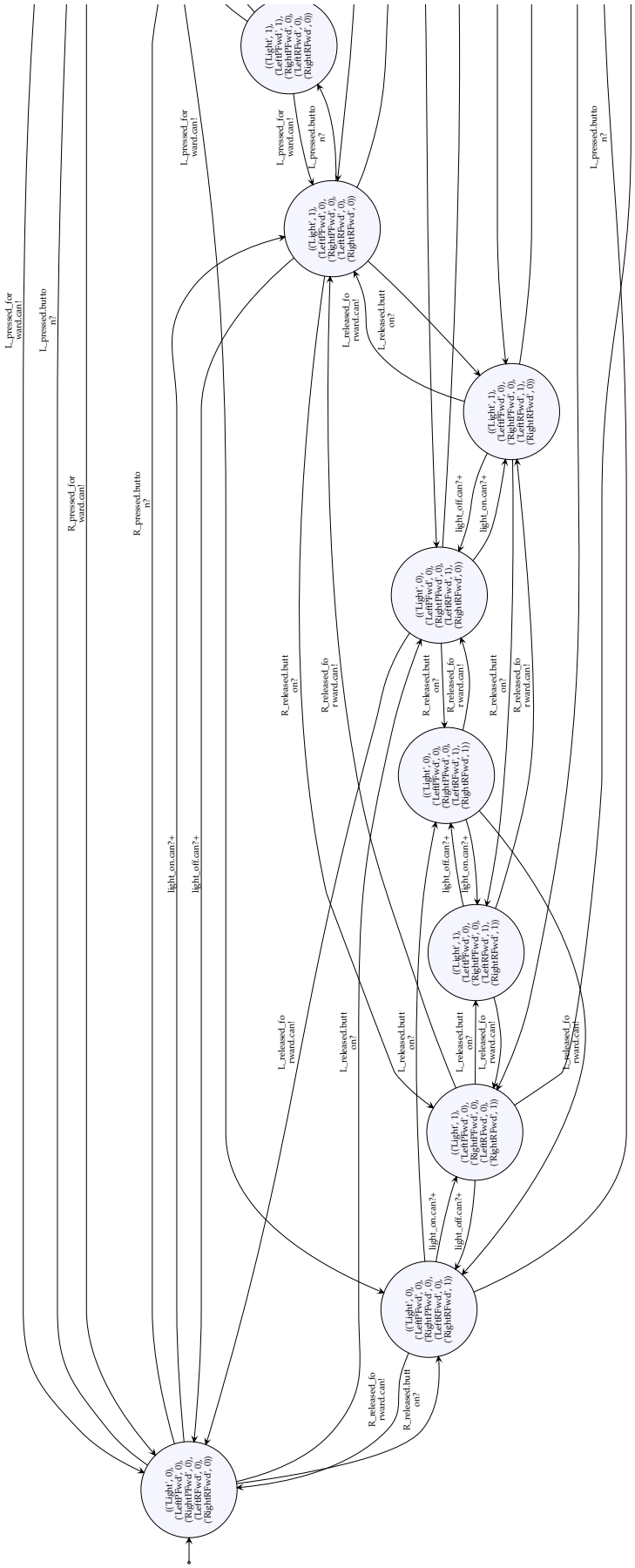


FIGURE 4.3 – Automate de spécification de la lumière (première partie)

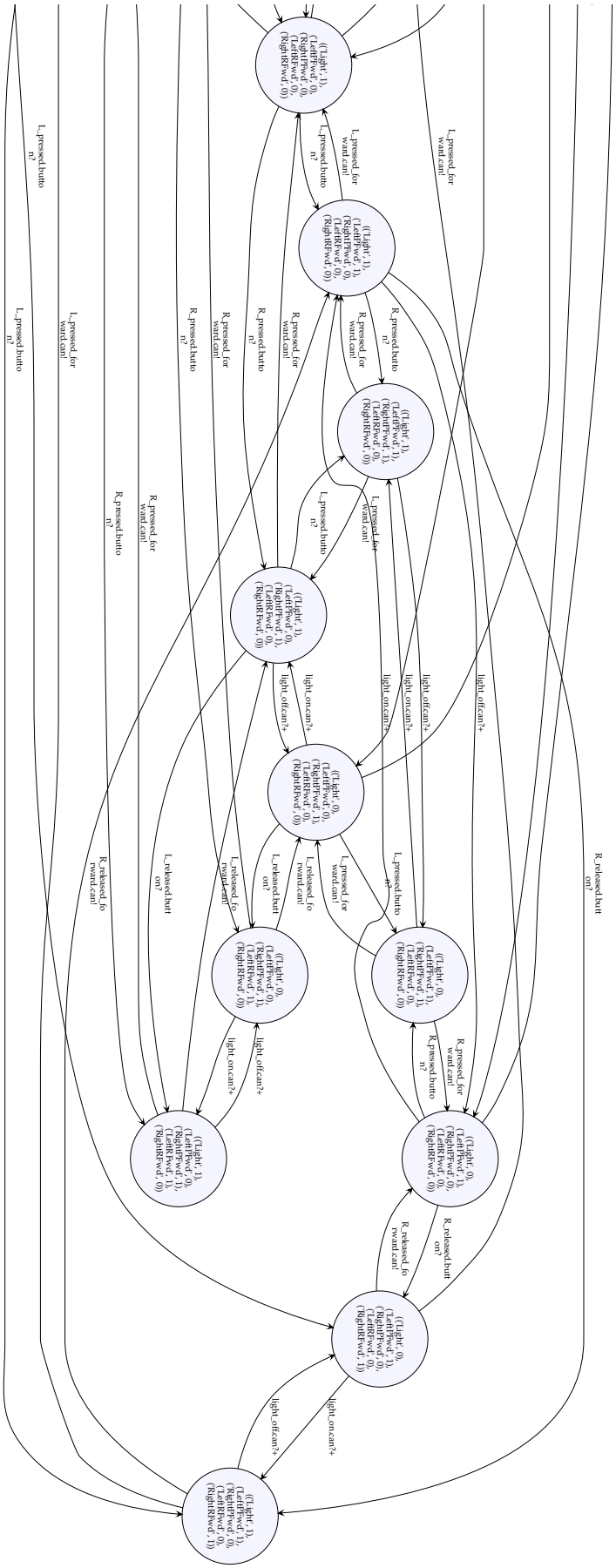


FIGURE 4.4 – Automate de spécification de la lumière (deuxième partie)

des boutons entre l'activation et la désactivation de la conduite autonome (sinon un appui prolongé pourrait provoquer une désactivation).

Comme pour la lumière, nous allons maintenant décrire les transitions de l'automate de spécification de la carte mère, qui se trouvent dans les lignes 16 à 73 du Listing 4.3 : "Code de génération de l'automate de la carte mère". Les lignes 16 à 34 décrivent les transitions liées à la réception d'un changement d'état d'un des boutons. Par exemple, les lignes 16 à 22 indiquent que si le bouton gauche n'est pas pressé (`if qdict["LPressed"] == 0:`), alors un message indiquant que le bouton gauche a été pressé peut être reçu. Les lignes 23 à 26 décrivent la gestion du relâchement (si le bouton est déjà pressé, un message indiquant un relâchement peut être reçu).

Les lignes 36 à 47 nous permettent de forcer un relâchement des boutons pour terminer une activation/désactivation de la conduite autonome. Si `ActState` est à 0, alors une transition interne (`"actState.int;"`, sans envoi de message) est autorisée pour passer `ActState` à 1 uniquement si les deux boutons sont relâchés, la lumière allumée et la conduite autonome activée. Si `ActState` est à 1, alors une transition interne (`"actState.int;"` sans envoi de message) est autorisée pour passer `ActState` à 0 uniquement si les deux boutons sont relâchés, la lumière éteinte et la conduite autonome désactivée. De cette manière, même si le conducteur garde les deux boutons pressés indéfiniment, il ne peut pas activer puis désactiver indéfiniment la conduite autonome car un relâchement est obligatoire. On obtient un comportement du type "presser pour activer, relâcher, puis presser pour désactiver".

Nous allons maintenant détailler les transitions permettant d'envoyer un message d'allumage de la lumière d'indication de la conduite autonome (lignes 49 à 54). Pour envoyer un message d'allumage (`"light_on.can!+"`), la lumière doit être éteinte, les boutons gauche et droit pressés, la conduite autonome déjà activée (pour préserver la safety) et `ActState` doit être à 0. Pour envoyer un message d'extinction, la lumière doit être allumée, les deux boutons doivent être pressés, la conduite autonome doit être activée (là encore pour préserver la safety) et `ActState` doit être à 1. On constate que si la variable `ActState` n'existait pas, il serait possible d'envoyer un message d'allumage, immédiatement suivi d'un message d'extinction, sans qu'une action du conducteur puisse conditionner l'envoi du message d'extinction. L'ajout de la variable `ActState`, qui change de valeur lors d'un relâchement des boutons dans certaines conditions (comme nous l'avons vu précédemment), permet de conditionner cet envoi.

Pour les transitions concernant la conduite autonome, le raisonnement est analogue à celui de la lumière d'indication. Il convient de noter que l'envoi des messages d'activation et de désactivation a lieu lorsque la lumière est éteinte pour des raisons de safety (*i.e.* la lumière n'est allumée que lorsque la conduite autonome est activée).

```
1 mainboard = NFA(
2     {(("LPressed", 0), ("RPressed", 0), ("AutoDriveActivation", 0), ("LightActivation", 0), ("
      ActState", 0))}, # Etats initiaux
3     set(), # Etats finaux
```

```
    set(), # Transitions
    name="mainboard", # Nom de l'automate
    worder=tuple
)

def grow(A):
    has = False

    def extend(q):
        nonlocal has
        qdict = dict(q)
        # Si dans l'état courant le bouton gauche n'est pas pressé
        if qdict["LPressed"] == 0:
            # Un nouvel état est créé
            newd = qdict.copy()
            # Dans lequel le bouton gauche est pressé
            newd["LPressed"] = 1
            # Et une transition est ajoutée entre ces deux états
            has = A.try_rule(q, "L_pressed_forward.can?", tuple(newd.items())) or has
        if qdict["LPressed"] == 1:
            newd = qdict.copy()
            newd["LPressed"] = 0
            has = A.try_rule(q, "L_released_forward.can?", tuple(newd.items())) or has
        if qdict["RPressed"] == 0:
            newd = qdict.copy()
            newd["RPressed"] = 1
            has = A.try_rule(q, "R_pressed_forward.can?", tuple(newd.items())) or has
        if qdict["RPressed"] == 1:
            newd = qdict.copy()
            newd["RPressed"] = 0
            has = A.try_rule(q, "R_released_forward.can?", tuple(newd.items())) or has

        if qdict["ActState"] == 0:
            newd = qdict.copy()
            newd["ActState"] = 1
            if qdict["LPressed"] == 0 and qdict["RPressed"] == 0 and qdict["LightActivation"] ==
    1 and qdict[
                "AutoDriveActivation"] == 1:
                has = A.try_rule(q, "actState.int;", tuple(newd.items())) or has
        if qdict["ActState"] == 1:
            newd = qdict.copy()
            newd["ActState"] = 0
            if qdict["LPressed"] == 0 and qdict["RPressed"] == 0 and qdict["LightActivation"] ==
    0 and qdict[
                "AutoDriveActivation"] == 0:
                has = A.try_rule(q, "actState.int;", tuple(newd.items())) or has

        if qdict["LightActivation"] == 0:
            newd = qdict.copy()
            newd["LightActivation"] = 1
            if qdict["LPressed"] == 1 and qdict["RPressed"] == 1 and qdict["AutoDriveActivation"]
```

```
    == 1 and qdict[
53                 "ActState"] == 0:
54                 has = A.try_rule(q, "light_on.can!+", tuple(newd.items())) or has
55         if qdict["LightActivation"] == 1:
56             newd = qdict.copy()
57             newd["LightActivation"] = 0
58             if qdict["LPressed"] == 1 and qdict["RPressed"] == 1 and qdict["AutoDriveActivation"]
    == 1 and qdict[
59                 "ActState"] == 1:
60                 has = A.try_rule(q, "light_off.can!+", tuple(newd.items())) or has
61
62         if qdict["AutoDriveActivation"] == 0:
63             newd = qdict.copy()
64             newd["AutoDriveActivation"] = 1
65             if qdict["LPressed"] == 1 and qdict["RPressed"] == 1 and qdict["LightActivation"] ==
    0 and qdict[
66                 "ActState"] == 0:
67                 has = A.try_rule(q, "auto_drive_enabled.can_ext!+", tuple(newd.items())) or has
68         if qdict["AutoDriveActivation"] == 1:
69             newd = qdict.copy()
70             newd["AutoDriveActivation"] = 0
71             if qdict["LPressed"] == 1 and qdict["RPressed"] == 1 and qdict["LightActivation"] ==
    0 and qdict[
72                 "ActState"] == 1:
73                 has = A.try_rule(q, "auto_drive_disabled.can_ext!+", tuple(newd.items())) or has
74
75     for q in A.Q.copy():
76         extend(q)
77     return has
78
79 mainboard.growtofixpoint(grow, record_steps=True)
```

Listing 4.3 – Code de génération de l'automate de la carte mère

La modélisation de la carte mère produit l'automate de spécification des Figures 4.5 et 4.6. On constate que cet automate est constitué de 6 cycles orientés correspondant à de multiples pressions et relâchements des boutons. En effet, nous avons choisi de prendre en compte le fait que l'utilisateur puisse presser et relâcher de manière erratique les boutons. De fait, il est possible de boucler indéfiniment dans un cycle si l'utilisateur ne maintient pas les deux boutons pressés ou relâchés. Chaque transition entre deux de ces cycles correspond à un envoi de message (`"light_on.can!+"`, `"light_off.can!+"`, `"auto_drive_enabled.can_ext!+"`, `"auto_drive_disabled.can_ext!+"`) ou une action interne (`ActState`).

Maintenant que la modélisation de l'ISW en lui-même est complète, nous pouvons passer à la modélisation de l'attaquant qui va chercher à perturber le fonctionnement du système.

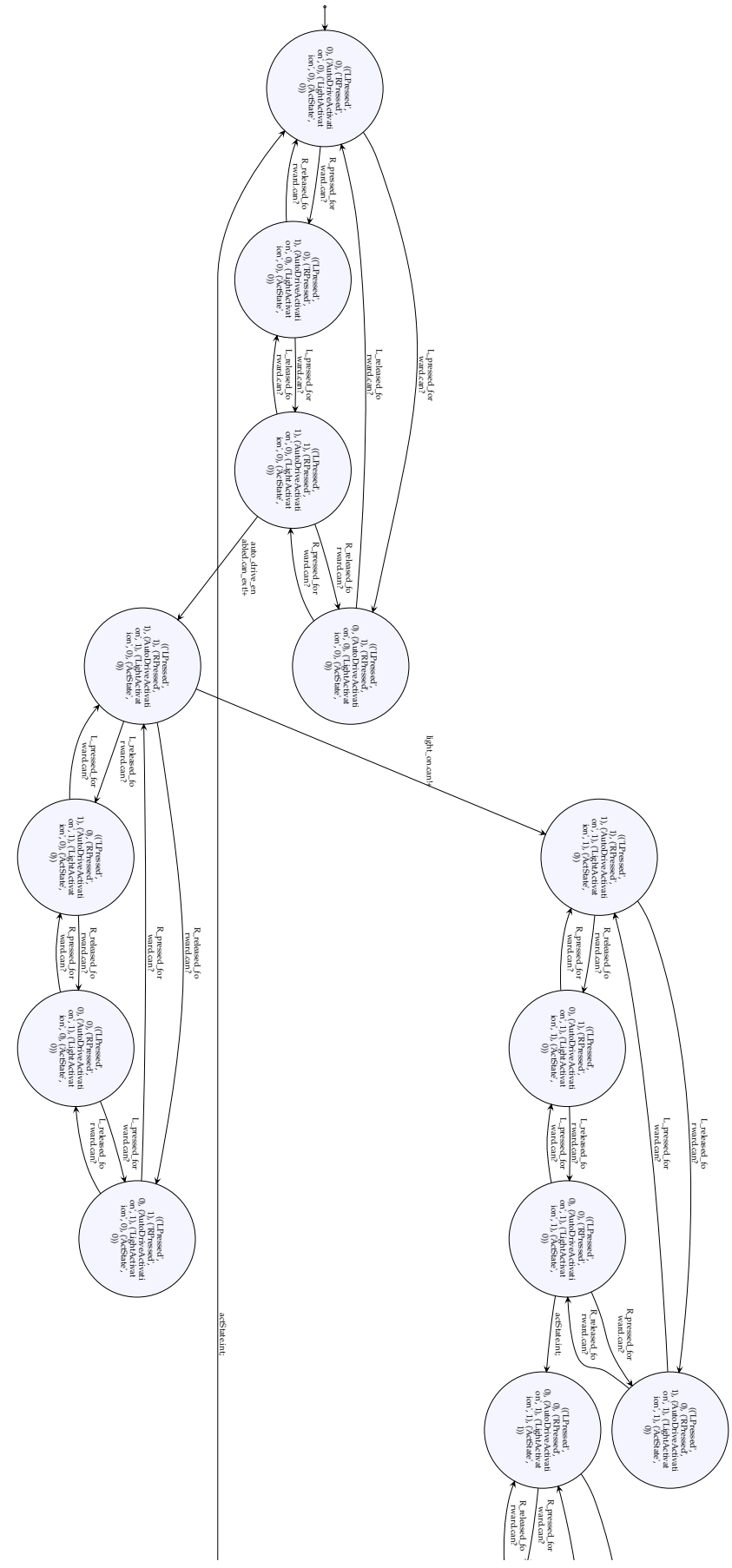


FIGURE 4.5 – Automate de spécification de la carte mère (première partie)

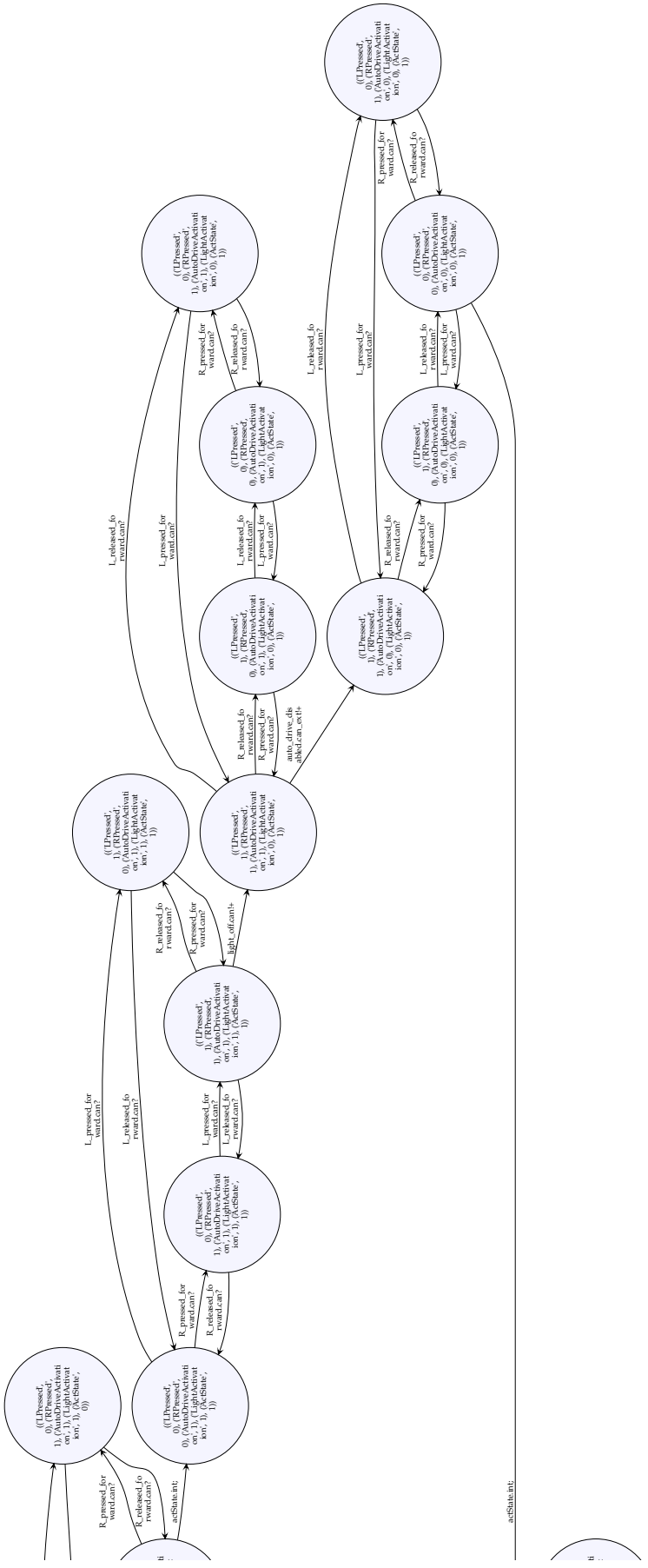


FIGURE 4.6 – Automate de spécification de la carte mère (deuxième partie)

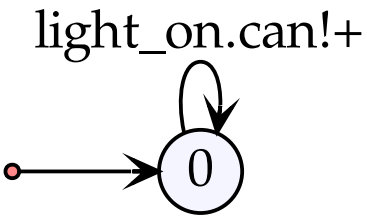


FIGURE 4.7 – Automate de spécification du comportement de l'attaquant

4.1.2 Modélisation de l'attaquant

Le comportement de l'attaquant, décrit en Section 3.3 : "Présentation de l'attaquant" est plus simple que celui de l'ISW : il ne s'agit que d'un seul système, effectuant une seule et unique action (*i.e.* envoyer un message pour allumer la lumière), pouvant être répétée à l'infini, indépendamment de l'état des autres sous-systèmes. Ainsi, sa modélisation peut être faite très facilement avec l'outil du Dr. Hugot, comme le montre le Listing 4.4. On retrouve un unique état initial, aucun état final et une unique transition bouclant sur l'état initial qui correspond à l'envoi d'un message demandant l'allumage de la lumière. Ainsi, l'attaquant est en mesure d'envoyer un nombre illimité de messages d'allumage de la lumière et ce à n'importe quel moment.

```
1 attacker = NFA.spec("""
2     0
3     --
4     0 light_on.can!+ 0""").named("Attacker")
```

Listing 4.4 – Code de génération de l'automate de l'attaquant

Cette modélisation produit l'automate de la Figure 4.7. La dernière étape avant d'effectuer une première vérification de l'ISW est de modéliser le système de la conduite autonome afin d'observer les éventuels impacts de l'attaquant (*e.g.* la lumière d'indication de la conduite autonome est allumée alors que la conduite autonome est désactivée).

4.1.3 Modélisation de la conduite autonome

L'application en charge de la conduite autonome est en dehors de notre champ d'action en tant qu'équipementier. Sa modélisation est pourtant essentielle car son état participe à déterminer si le système est dans une configuration sûre ou non. Si la lumière d'indication est allumée et que la conduite autonome est activée, alors la situation est sûre. Si la conduite autonome est désactivée, alors la situation n'est pas sûre, car le conducteur peut penser qu'il peut relâcher son attention sur la route, alors que non.

De fait, nous effectuons une modélisation simple, où la conduite autonome est uniquement activée ou désactivée, ce qui se traduit par le code du Listing 4.5. Notre automate se compose d'un unique état initial, pas d'état final, et deux transitions. La première transition va de l'état initial `0` à l'état `1` et correspond à l'activation de la conduite autonome (*cf.* ligne 4 du Listing 4.5). La seconde transition part de l'état `1`, revient à l'état

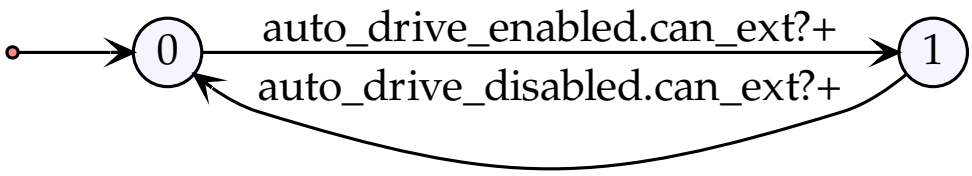


FIGURE 4.8 – Automate de spécification du comportement de la conduite autonome

initial et correspond à une désactivation de la conduite autonome (*cf.* ligne 5 du Listing 4.5).

```
1 auto_drive = NFA.spec("""
2     0
3     --
4     0 auto_drive_enabled.can_ext?+ 1
5     1 auto_drive_disabled.can_ext?+ 0""").named("Auto Drive")
```

Listing 4.5 – Code de génération de l'automate de la conduite autonome

Cette modélisation produit l'automate de la Figure 4.8 : "Automate de spécification du comportement de la conduite autonome".

Avec tous les sous-systèmes de l'ISW de modélisés ainsi que la conduite autonome, nous pouvons effectuer une première étape de vérification, permettant de voir si le système, dans son comportement nominal, possède bien les propriétés voulues, surtout les propriétés **safety** et si une violation de ces propriétés est bien observée en présence de l'attaquant.

4.2 Première vérification avec l'outil du Dr. Hugot

Pour illustrer que notre attaquant perturbe bien le système modélisé et que notre politique permet bien de contrer l'attaquant, 5 vérifications différentes sont effectuées :

(1) vérification du système seul ;

(2) vérification du système attaqué ;

(3) vérification du système avec une politique erronée ;

(4) vérification du système attaqué avec une politique correcte ;

(5) vérification du système avec une politique correcte.

Dans cette section, nous allons nous intéresser aux vérifications 1 et 2. Avant de pouvoir vérifier notre système, nous devons définir les propriétés de **safety**, de disponibilité, d'intégrité et de cybersécurité qui doivent être vérifiées afin d'estimer la qualité du système. Les propriétés peuvent s'appliquer au système global, ou bien à un seul composant. Ainsi, nous allons commencer par vérifier des propriétés sur les composants, afin de montrer que la vérification peut être utile avant même d'avoir la modélisation complète du système. En effet, cela permet de garantir que les composants ont individuellement

le comportement souhaité. Lors de la vérification, si le système, sans être attaqué, ne respecte pas les propriétés définies, la modélisation est à revoir. L'outil du Dr. Hugot utilise la logique CTL pour définir les propriétés de vérification que nous allons décrire en détails dans la section suivante.

4.2.1 Vérification individuelle des composants de l'ISW

Pour vérifier une modélisation, l'outil du Dr. Hugot nécessite de labelliser les états possédant des attributs intéressants pour exploiter ces labels dans la définition de propriétés à vérifier. Ainsi, pour chaque vérification nous donnons les propriétés à vérifier ainsi que la fonction de labellisation. Nous commençons par vérifier la carte mère car elle possède le comportement le plus complexe. Nous passons ensuite à la vérification de la lumière et terminons par les autres composants de l'ISW.

Vérification de la carte mère

Pour rappel, l'automate de spécification de la carte mère est donné en Figure 4.5 : "Automate de spécification de la carte mère (première partie)" et Figure 4.6 : "Automate de spécification de la carte mère (deuxième partie)". Bien que comportant peu d'états (24) il est difficile d'en vérifier le comportement étant donné le nombre important de transitions (54). Ainsi, pour nous assurer que le comportement de la carte mère est en adéquation avec la description faite en Section 3.2.3 : "Comportement de l'ISW" nous pouvons définir des formules de logique temporelle reprenant une partie de la description faite en Section 3.2.3. Les propriétés que nous souhaitons vérifier sont les suivantes : après chaque demande d'activation de la conduite autonome, la carte mère demande l'allumage de la lumière et avant chaque désactivation de la conduite autonome, la carte mère demande l'extinction de la lumière. Elles peuvent être notées en CTL de la manière suivante :

Propriété d'intégrité fonctionnelle

$$(AutoDriveActivated \ \wedge \ \neg ActStateON) \Rightarrow \forall\Diamond(LightActivated) \tag{4.1}$$

Propriété d'intégrité fonctionnelle

$$(\neg LightActivated \ \wedge \ ActStateON) \Rightarrow \forall\Diamond(\neg AutoDriveActivated) \tag{4.2}$$

La première formule peut se traduire par : si la conduite autonome est activée, et que les boutons n'ont pas été relâchés, alors pour tous les chemins, la lumière s'allumera à un moment. La seconde formule peut se traduire par : si la lumière est éteinte, et si les boutons ont été relâchés, alors pour tous les chemins, la conduite autonome sera désactivée à un moment. Ces deux propriétés sont des propriétés **d'intégrité fonctionnelle** d'après la description faite en Section 3.5 : "Types de propriétés à vérifier". Il s'agit bien de propriétés d'intégrité que l'attaquant peut chercher à compromettre.

Afin de vérifier les propriétés sur la carte mère, nous utilisons la fonction de labellisation suivante :

```
def mainboard_labeling(mainboard):
    # Chaque état est associé à des labels
    labels = dict()
    # Pour chaque état de l'automate
    for q in mainboard.Q:
        # On définit un ensemble de labels associés à cet état
        labels[q] = set()
        # Un état est un tuple de tuples, on crée un dictionnaire avec une paire clef : valeur pour chaque tuple
        dictq = dict(q)
        # Si la carte mère a demandé l'activation de la conduite autonome
        if dictq["AutoDriveActivation"] == 1:
            # On ajoute un label à l'état
            labels[q] |= {"AutoDriveActivated"}
        if dictq["LightActivation"] == 1:
            labels[q] |= {"LightActivated"}
        if dictq["ActState"] == 1:
            labels[q] |= {"ActStateON"}
```

Listing 4.6 – Labellisation des états pour effectuer la vérification de la carte mère

Cette fonction permet d'associer à un état un ensemble de labels. Ces labels sont ensuite utilisés dans les propriétés à vérifier.

Les propriétés 4.1 et 4.2 sont définies dans une syntaxe comprise par l'outil (*cf.* lignes 3 et 5 du Listing 4.7). Les lignes 4 et 6 du Listing 4.7 correspondent à la vérification du modèle par rapport à la propriété voulue.

```
def labeling(A):
    [...]
    # AutoDriveActivated && ¬ActStateON ⇒ ∀◇(LightActivated)
    fo = (IMPLIES, (AND, "AutoDriveActivated", (NOT, "ActStateON")), (AF, "LightActivated"))
    checkvisu(A, labels, fo, visu=("simple"))
    # ¬LightActivated && ActStateON ⇒ ∀◇(¬AutoDriveActivated)
    fo = (IMPLIES, (AND, (NOT, "LightActivated"), "ActStateON"), (AF, (NOT, "AutoDriveActivated")))
    checkvisu(A, labels, fo, visu=("simple"))
```

Listing 4.7 – Définition des propriétés 4.1 et 4.2 avec l'outil du Dr. Hugot

De façon simple, nous pouvons vérifier la conjonction de ces deux propriétés, qui produit l'automate de la Figure 4.9 : "Automate de vérification de la carte mère pour les formules 4.1 et 4.2". On constate sur cet automate que 8 états apparaissent en rouge, c'est-à-dire que 8 états ne vérifient pas la conjonction des deux propriétés. En zoomant sur ces états rouges (*cf.* Figure 4.10 : "Zoom sur les états rouges de l'automate de la Figure 4.9"), on constate qu'il y a un cycle après l'activation de la conduite autonome et un autre avant l'allumage de la lumière. Ces cycles correspondent à des pressions/relâchements des boutons, qui peuvent donc bloquer indéfiniment l'allumage de la lumière, violant ainsi la première propriété (le raisonnement est analogue pour la deuxième propriété). C'est un choix de modélisation que nous avons fait : l'utilisateur peut actionner les

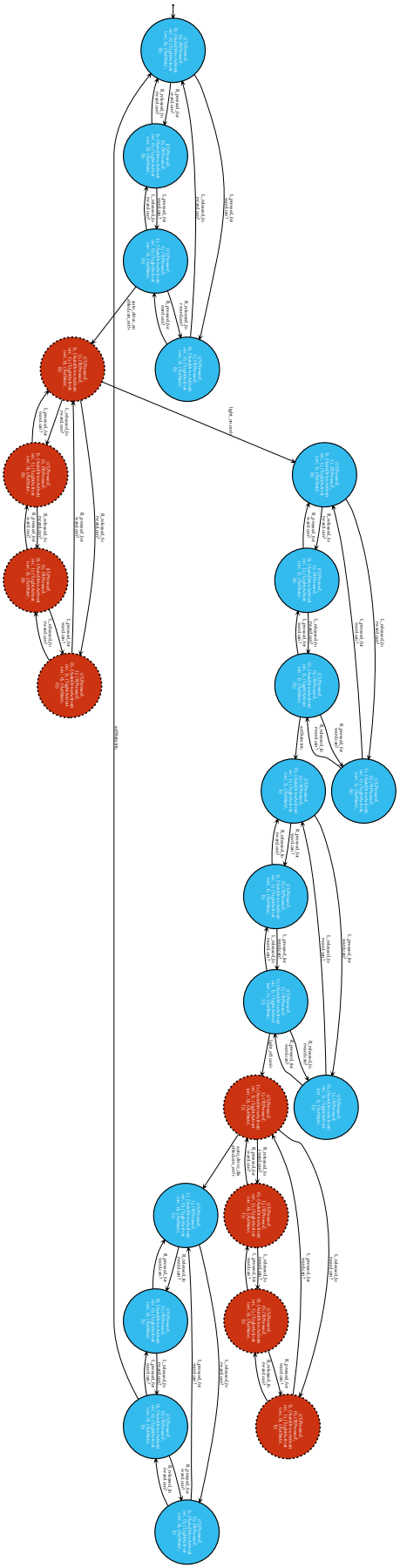


FIGURE 4.9 – Automate de vérification de la carte mère pour les formules 4.1 et 4.2

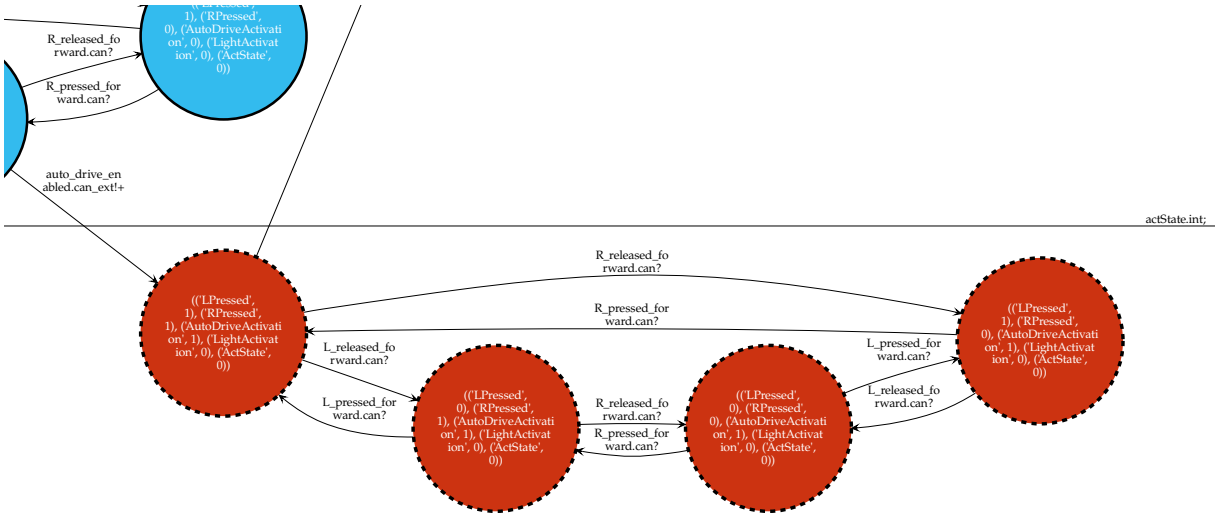


FIGURE 4.10 – Zoom sur les états rouges de l'automate de la Figure 4.9

boutons autant qu'il le souhaite, et le système gère autant que possible ces actions. Étant donné que la modélisation ne prend pas en compte le délai d'une action, les actions d'un potentiel conducteur (pression et relâchement des boutons) sont aussi rapides que le traitement des messages échangés entre les différents composants, ce qui semble peu réaliste par rapport à un système réel. De fait, nous considérons que bien qu'il existe des états violant ces propriétés, la spécification de la carte mère reste correcte par rapport au comportement que nous avons défini. Pour que les propriétés 4.1 et 4.2 soient vérifiées, il faudrait ajouter une condition indiquant que les boutons ne sont pas infiniment souvent pressés et relâchés. Or, ce type de propriété n'est pas adapté à la logique CTL et sera donc plus facilement vérifiable avec la logique LTL utilisée par SPIN.

Pour nous assurer que la modélisation de la carte mère respecte au moins partiellement le comportement décrit en Section 3.2.3, nous pouvons assouplir les formules à vérifier. De plus, nous allons ajouter une propriété de safety pour voir que la modélisation de la carte mère ne met pas en danger les occupants du véhicule.

Propriété d'intégrité fonctionnelle

$$(AutoDriveActivated \ \wedge \ \neg ActStateON) \Rightarrow \exists\Diamond(LightActivated) \tag{4.3}$$

Propriété d'intégrité fonctionnelle

$$(\neg LightActivated \ \wedge \ ActStateON) \Rightarrow \exists\Diamond(\neg AutoDriveActivated) \tag{4.4}$$

Propriété safety

$$LightActivated \Rightarrow AutoDriveActivated \tag{4.5}$$

La première formule peut se traduire par : si la conduite autonome est activée, et que les boutons n'ont pas été relâchés, alors il existe un chemin où la lumière s'allumera à un moment. La seconde formule peut se traduire par : si la lumière est éteinte, et si les boutons ont été relâchés, alors il existe un chemin où la conduite autonome sera désactivée à un moment. Les propriétés 4.3 et 4.4 sont des propriétés d'intégrité que

l'attaquant peut aussi tenter de compromettre. Enfin, la dernière formule indique que si la lumière est allumée, alors la conduite autonome est activée. Cette propriété est uniquement violée lorsque la lumière est allumée et que la conduite autonome est désactivée, ce qui représente une situation non sûre. Il s'agit donc d'une propriété de safety.

Chacune de ces formules est ensuite définie dans la syntaxe de l'outil, puis vérifiée :

```
def labeling(A):
    [...]
    # AutoDriveActivated && ¬ActStateON ⇒ ∃◇(LightActivated)
    fo = (IMPLIES, (AND, "AutoDriveActivated", (NOT, "ActStateON")), (EF, "LightActivated"))
    checkvisu(A, labels, fo, visu=("simple"))
    # ¬LightActivated && ActStateON ⇒ ∃◇(¬AutoDriveActivated)
    fo = (IMPLIES, (AND, (NOT, "LightActivated"), "ActStateON"), (EF, (NOT, "AutoDriveActivated")
    ))
    checkvisu(A, labels, fo, visu=("simple"))
    # LightActivated ⇒ AutoDriveActivated
    fo = (IMPLIES, "LightActivated", "AutoDriveActivated")
    checkvisu(A, labels, fo, visu=("simple"))
```

Listing 4.8 – Définition des propriétés 4.3, 4.4 et 4.4 avec l'outil du Dr. Hugot

La vérification des formules 4.3, 4.4 et 4.5 produit l'automate de vérification de la Figure 4.11. On constate que tous les états sont bleus indiquant que les trois propriétés sont vérifiées. Ainsi, il existe donc au moins un chemin respectant le comportement défini en Section 3.2.3 et notre spécification préserve la safety des occupants ce qui est un point crucial.

Nous avons vu ici que les erreurs trouvées lors de la vérification nécessitent d'être interprétées par rapport à la modélisation et à sa représentativité du système réel. Ici, notre modélisation nous semble cohérente par rapport au comportement attendu du système réel car certains chemins correspondent au comportement décrit et aucun chemin ne viole la propriété de **safety**. Lorsque nous vérifierons l'ensemble du système, nous utiliserons une formule similaire à la conjonction de 4.3 et 4.4. Les formules 4.1 et 4.2 produiront toujours certains états rouges de par la construction de notre modélisation. Avant de passer à la vérification du système global, nous allons maintenant détailler la vérification de la lumière.

Vérification de la lumière

Étant donné que la lumière n'a pas d'intelligence – elle transmet les états des boutons et attend les commandes de la carte mère – les propriétés intéressantes à vérifier sont peu nombreuses. Nous choisissons de vérifier que la lumière alterne entre l'état allumé et l'état éteint. Cette propriété fonctionnelle peut être traduite de la manière suivante :

Propriété d'intégrité fonctionnelle

$$(LightOFF \Rightarrow \forall\Diamond(LightON)) \wedge (LightOn \Rightarrow \forall\Diamond(LightOFF)) \tag{4.6}$$

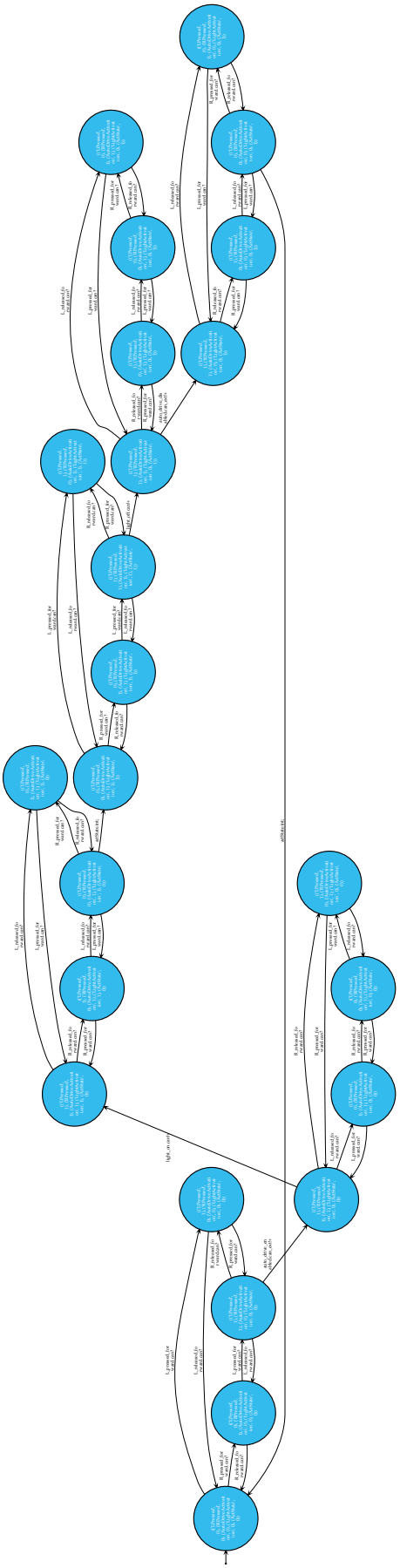


FIGURE 4.11 – Automate de vérification de la carte mère pour les formules 4.3, 4.4 et 4.5

Cette propriété signifie que lorsque la lumière est éteinte (*resp.* allumée), alors pour tous les chemins suivants, la lumière sera un jour allumée (*resp.* éteinte). La vérification de cette propriété produit l'automate de la Figure 4.12 où tous les états sont rouges, pour les mêmes raisons que la carte mère : on peut boucler indéfiniment sur des pressions et relâchements de boutons. Là encore, la propriété à exprimer pour prendre en compte ces multiples pressions et relâchements n'est pas exprimable en CTL.

La vérification de la propriété affaiblie 4.7 produit un automate de vérification où tous les états sont bleus. Cette propriété permet de vérifier qu'il existe au moins un chemin ou la lumière est fatalement allumée/éteinte.

Propriété d'intégrité fonctionnelle

$$(LightOFF \Rightarrow \exists\Diamond(LightON)) \wedge (LightOn \Rightarrow \exists\Diamond(LightOFF)) \tag{4.7}$$

N'ayant pas d'autres propriétés intéressantes à vérifier uniquement sur la lumière, nous considérons que la spécification est correcte, car il existe toujours au moins un chemin conforme au comportement défini en Section 3.2.3 : "Comportement de l'ISW". Ainsi, nous allons maintenant passer à la vérification des derniers composants de l'ISW.

Vérification des autres composants de l'ISW

Les spécifications des autres composants – les boutons et la conduite autonome – de l'ISW sont assez simples (1 à 2 états, autant de transitions). Ainsi, nous ne définissons pas de formule de logique pour les vérifier. Notons que bien que des propriétés ne seraient pas utiles pour vérifier uniquement ces composants étant donné leur simplicité, ces mêmes propriétés pourraient être utilisées pour vérifier que les comportements de ces composants sont préservés dans le système global. Dans notre cas, nous utilisons d'autres propriétés prenant en compte le système dans sa globalité. Pour l'attaquant, nous formalisons ses capacités d'attaque au travers de la propriété suivante :

$$\forall\Box(send\_light\_on) \tag{4.8}$$

Cette propriété signifie que pour tous les chemins, dans chaque état, l'attaquant peut envoyer une commande d'allumage de la lumière. Si nous reprenons l'automate de spécification de l'attaquant (*cf.* Figure 4.7), on constate que la seule transition boucle sur l'état initial, autorisant l'envoi illimité de cette commande. L'attaquant vérifie donc la propriété 4.8.

Maintenant que les composants individuels ont été vérifiés, et qu'ils sont conformes au comportement que nous avons décrit en Section 3.2.3 : "Comportement de l'ISW", nous pouvons passer à la définition des propriétés pour le système global.

4.2.2 Définition des propriétés à vérifier sur le système global

Les propriétés que nous allons définir permettent de garantir la **safety** et la sécurité du système. Nous avons vu que pour préserver la safety, la lumière ne doit pas être

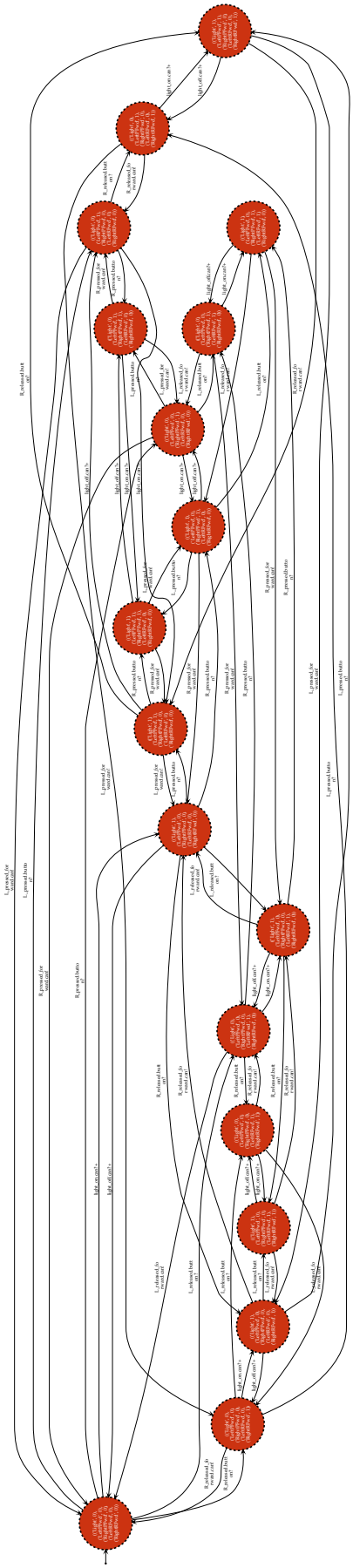


FIGURE 4.12 – Automate de vérification de la lumière pour la propriété 4.6

allumée si la conduite autonome est désactivée. De fait, nous définissons une propriété de safety, ϕ : "Si la lumière est allumée, alors la conduite autonome est activée". Pour que cette propriété soit violée, il faut que la lumière soit allumée et la conduite autonome désactivée, ce qui peut pousser le conducteur à relâcher son attention sur la route et donc mettre en danger les occupants. Il s'agit donc bien d'une propriété visant à vérifier la préservation de la safety.

Bien que de même nature – safety – que la propriété 4.5 ($LightActivated \Rightarrow AutoDriveActivated$), ϕ est différente car elle fonctionne à une autre échelle. Là où la propriété 4.5 s'intéresse à un composant individuel (la carte mère), ϕ s'intéresse aux différents composants du système.

Notre politique de contrôle d'accès ne doit pas rendre le système inopérant pour préserver sa sécurité. Par exemple, un moniteur empêchant en permanence d'allumer la lumière sera correct par rapport à ϕ mais au prix d'une violation de l'intégrité. Ainsi, la vérification des propriétés de disponibilité, telles que "Il est toujours possible d'allumer la lumière", que nous nommons ψ, est aussi un point d'importance. Enfin, nous ajoutons aussi une propriété d'intégrité fonctionnelle ξ pour vérifier que le système a toujours le comportement attendu. ξ est une propriété similaire à la conjonction de 4.3 et 4.4, appliquée aux composants du système (la carte mère, la conduite autonome et la lumière).

Ainsi, les trois propriétés que nous allons vérifier avec l'outil du Dr. Hugot sont les suivantes :

Propriété de safety

$$\phi = Light \Rightarrow AutoDrive$$

Propriété de disponibilité

$$\psi = \exists\Diamond\, Light$$

Propriété d'intégrité fonctionnelle

$$\xi = (AutoDrive \wedge \neg ActStateON) \Rightarrow \exists\Diamond\, Light$$
$$\wedge\, (\neg Light \wedge ActStateON) \Rightarrow \exists\Diamond\, \neg AutoDrive$$

ϕ est une propriété de safety, que notre attaquant veut enfreindre. ψ est une propriété de disponibilité nous permettant d'observer les effets de la politique sur le système. Et ξ est une propriété d'intégrité.

Afin de vérifier nos propriétés, nous devons labeliser les états correspondants aux différents atomes de nos propriétés. Avec le code du Listing 4.9 nous labellisons les états où la lumière est allumée, ceux où la conduite autonome est activée, ainsi que ceux où la carte mère possède les propriétés désirées.

```
1 def labeling(A):
2     labels = dict()
```

```
3     for q in A.Q:
4         labels[q] = set()
5         dictq = dict(q)
6         if dictq["Auto Drive"]:
7             labels[q] |= {"AutoPilot"}
8         if dict(dictq["Light ECU"])["Light"]:
9             labels[q] |= {"Light"}
10        if dict(dictq["mainboard"])["ActState"]:
11            labels[q] |= {"ActStateON"}
```

Listing 4.9 – Labellisation des états pour effectuer la vérification

À partir des labels des états, nous pouvons ensuite définir ϕ, ψ et ξ dans une syntaxe comprise par l'outil (*cf.* lignes 3, 6, 9 et 10 du Listing 4.10). Les lignes 5, 8 et 12 du Listing 4.10 correspondent à la vérification du modèle par rapport à la propriété voulue.

```
1  def labeling(A):
2      [...]
3      # ϕ = Light ⇒ AutoDrive
4      phi = (IMPLIES, "Light", "AutoPilot")
5      checkvisu(A, labels,     , visu=("simple"))
6      # ψ = ∃◇Light
7      psi = (EF, ("Light"))
8      checkvisu(A, labels, psi, visu=("simple"))
9      # ξ = ((AutoDrive ∧ ¬ActStateON) ⇒ ∃◇(Light))
10     #       ∧ ((¬Light ∧ ActStateON) ⇒ ∃◇(¬AutoDrive))
11     xi = (AND, (IMPLIES, (AND, "AutoPilot", (NOT, "ActStateON")), (EF, "Light")),
12          (IMPLIES, (AND, (NOT, "Light"), "ActStateON"), (EF, (NOT, "AutoPilot"))))
13     checkvisu(A, labels, xi, visu=("simple"))
```

Listing 4.10 – Définition des propriétés ψ, ϕ et ξ avec l'outil du Dr. Hugot

Nous pouvons maintenant effectuer une première vérification du système.

4.2.3 Résultats de la vérification

La phase de vérification génère des automates où chaque état est coloré en rouge s'il viole la propriété voulue, en bleu sinon. Ainsi, l'automate des Figures 4.13 et 4.14, qui vérifie la modélisation de l'ISW (sans attaque ou politique de contrôle d'accès) pour les propriétés ϕ, ψ et ξ indique que notre modèle vérifie toutes les propriétés. Ainsi, notre modélisation de l'ISW respecte la safety, sa spécification fonctionnelle, et la disponibilité (par rapport aux propriétés définies).

La révélation d'états rouges lors de cette première vérification aurait pu indiquer :

(1) une erreur dans la modélisation (le système modélisé ne correspond pas au système réel) ;

(2) un défaut du système réel (la modélisation est correcte et la même violation peut être trouvée dans le système réel) ;

FIGURE 4.13 – Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ (première partie)

FIGURE 4.14 – Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ (deuxième partie)

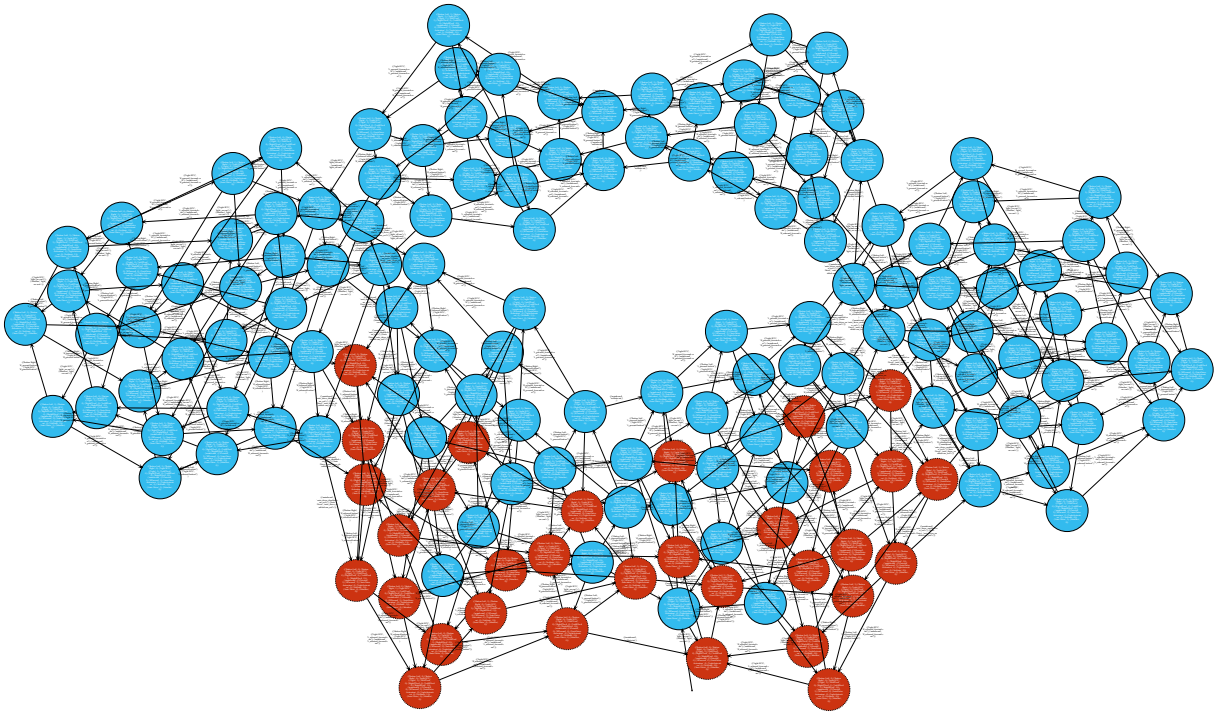


FIGURE 4.15 – Automate de vérification de l'ISW pour les propriétés ϕ ψ et ξ en cas d'attaque

(3) une erreur dans l'écriture de la propriété à vérifier (la modélisation est correcte, mais pas la formule à vérifier).

Si maintenant nous vérifions ϕ, ψ et ξ avec notre attaquant ajouté à la modélisation, nous obtenons la Figure 4.15 : "Automate de vérification de l'ISW pour les propriétés ϕ ψ et ξ en cas d'attaque". On constate qu'il y a des états rouges sur cette figure.

En vérifiant les propriétés une à une, on constate que la propriété violée est ϕ. Ainsi, notre attaquant perturbe la safety du système. Si on zoome sur un des états rouges (*cf.* Figure 4.16 : "Zoom sur un des états violant la propriété ϕ"), on constate que la lumière est allumée : `[...] ('Light ECU', (('Light', 1), [...]`, alors que la conduite autonome est désactivée `[...]('Auto Drive', 0)[...]`. Cela correspond bien à une violation de ϕ, l'attaquant a réussi à allumer la lumière alors que la conduite autonome est désactivée, ce qui peut induire le conducteur en erreur.

En revanche, si l'on observe l'automate de vérification de ψ et ξ (*cf.* Figure 4.17 : "Automate de vérification de l'ISW pour les propriétés ψ et ξ en cas d'attaque"), on constate que notre attaquant ne perturbe pas le fonctionnement du système au sens de ψ et ξ, c'est-à-dire que la lumière peut toujours être allumée, et le comportement fonctionnel est toujours valide : tous les états sont bleus.

Étant donné que notre attaquant a bien un effet négatif sur le système : il viole une

(b). Sur certains automates comportant un grand nombre d'états, il est difficile de localiser l'état initial. Il est ciblé par une flèche n'ayant pas d'état source et n'étant pas forcément visible

FIGURE 4.16 – Zoom sur un des états violant la propriété ϕ

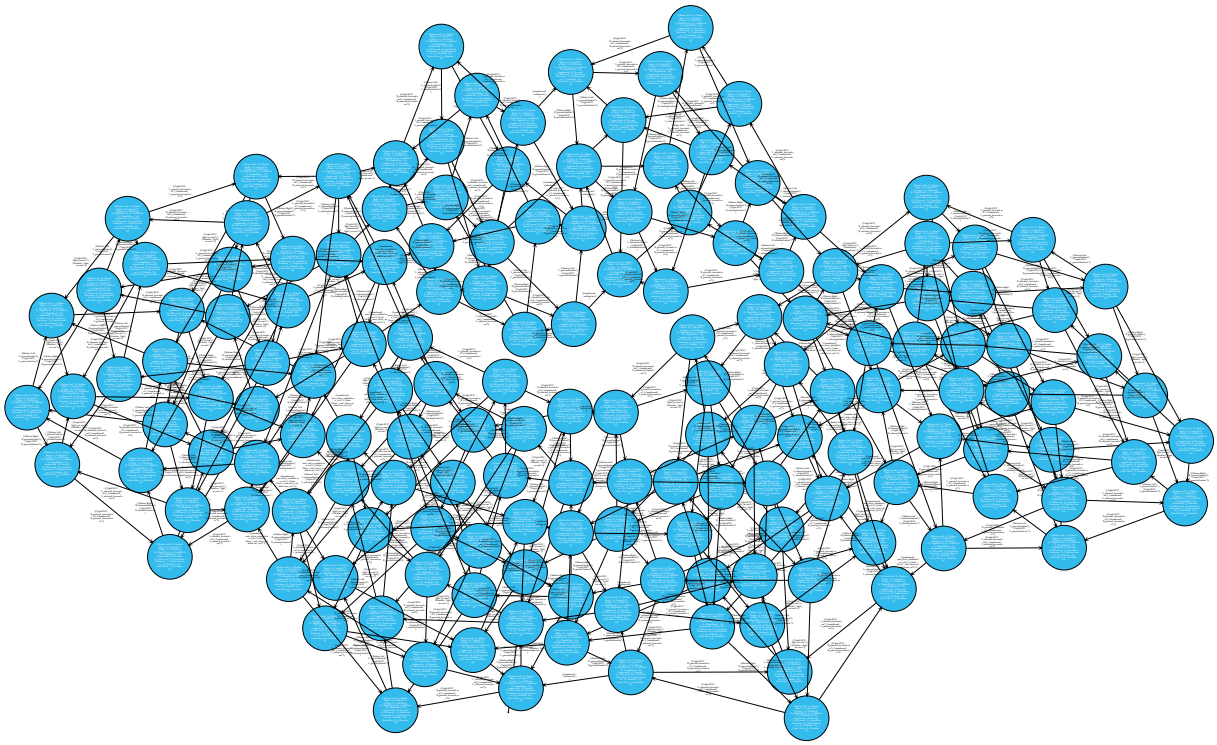


FIGURE 4.17 – Automate de vérification de l'ISW pour les propriétés ψ et ξ en cas d'attaque [(b)]

propriété de safety dans certains états. Nous allons intégrer la politique définie en Section 3.4 : “Construction de la politique de contrôle d’accès dynamique” afin de préserver la safety du système avec ce profil d’attaquant.

4.3 Enrichissement de la modélisation

Nous avons constaté que l’attaque modélisée perturbe le fonctionnement de l’ISW d’une manière qui n’est pas acceptable : une **violation de la safety** du système. De fait, il est nécessaire de mettre en place une contre mesure, sous la forme d’une politique de contrôle d’accès. Cette politique peut avoir des effets négatifs sur le système, qui sera donc à nouveau vérifié avec la politique pour s’assurer qu’il respecte les propriétés nécessaires. Dans la section précédente, nous avons effectué les vérifications 1 et 2 :

(1) vérification du système seul ;

(2) vérification du système attaqué ;

(3) vérification du système avec une politique erronée ;

(4) vérification du système attaqué avec une politique correcte ;

(5) vérification du système avec une politique correcte.

Dans cette section, nous commençons par vérifier le système avec une politique erronée (vérification 3) puis nous passons à la vérification d’un système protégé par notre politique en cas d’attaque (vérification 4). Enfin, nous terminons avec la vérification du système lorsqu’il est protégé par notre politique sans être attaqué (vérification 5).

4.3.1 Modélisation de la politique de contrôle d’accès

Comme nous l’avons vu, l’attaque modélisée allume la lumière d’indication de la conduite autonome alors que la conduite autonome est désactivée, causant une violation d’une propriété safety. Afin de contrer cette attaque, il faut s’assurer que la lumière ne puisse être allumée que lorsque la conduite autonome est activée, et qu’elle est éteinte avant la désactivation de la conduite autonome.

Les envois des messages d’allumage et d’extinction doivent donc être encadrés par les envois des messages d’activation et de désactivation de la conduite autonome, générant ainsi la politique représentée par l’automate de la Figure 4.18 : “Automate de spécification de la politique de contrôle d’accès”. On constate que cet automate impose une séquence précise pour l’envoi des messages, permettant ainsi de bloquer :

— l’allumage de la lumière si la conduite autonome n’est pas activée ;

— la désactivation de la conduite autonome si la lumière n’est pas éteinte (une autre attaque potentielle).

La génération de cet automate est faite avec le code du Listing 4.11.

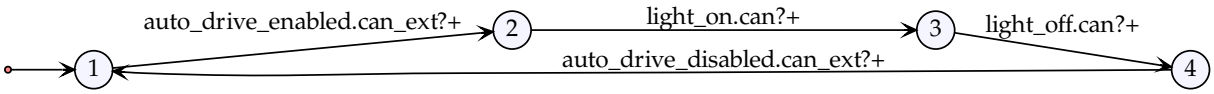


FIGURE 4.18 – Automate de spécification de la politique de contrôle d'accès

```
access_control = NFA.spec("""
    1
    __
    1 auto_drive_enabled.can_ext?+ 2
    2 light_on.can?+ 3
    3 light_off.can?+ 4
    4 auto_drive_disabled.can_ext?+ 1""").named("Access control")
```

Listing 4.11 – Code de génération de l'automate de la politique

Sur la Figure 4.18 et le Listing 4.11, on notera l'usage des opérateurs synchrones `"?+"` dont nous avons détaillé le fonctionnement dans la Section 3.1.3 : "Ajouts des nouveaux opérateurs". L'introduction des opérateurs de diffusion synchrones peut sembler surprenante étant donné que notre cas d'usage de l'ISW ne nécessite pas de diffusion dans sa forme de base. Dans le cas présent, il permet de considérer l'automate comme un contrôle en coupure. En effet, avec les opérateurs synchrones, si un message ne peut être reçu, alors il n'est pas envoyé.

Ainsi, la politique est considérée comme un composant à part entière et agit bien en coupure : si elle ne peut pas se synchroniser, le message ne sera pas envoyé, et donc reçu par aucun automate.

4.3.2 Vérification avec une politique de contrôle d'accès erronée

Pour montrer qu'une politique de contrôle d'accès mal spécifiée peut avoir des effets négatifs sur le système, nous allons détailler dans cette section une vérification du système avec une politique erronée. L'objectif est d'observer une violation de certaines propriétés du système, liée à l'ajout de la politique. La politique utilisée est modélisée par l'automate de la Figure 4.19, généré à partir du code du Listing 4.12. Cette politique contraint le système à :

(1) envoyer le message d'activation de la conduite autonome ;

(2) envoyer le message de désactivation de la conduite autonome ;

(3) envoyer le message d'allumage de la lumière ;

(4) envoyer le message d'extinction de la lumière.

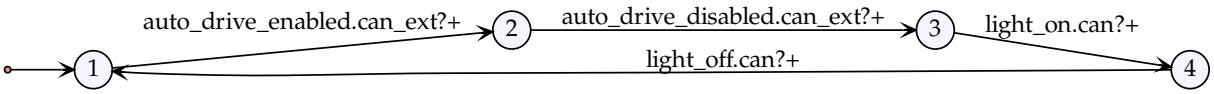


FIGURE 4.19 – Automate de spécification d'une mauvaise politique de contrôle d'accès

```
bad_access_control = NFA.spec("""
    1
    --
    1 auto_drive_enabled.can_ext?+ 2
    2 auto_drive_disabled.can_ext?+ 3
    3 light_on.can?+ 4
    4 light_off.can?+ 1""").named("Bad access control")
```

Listing 4.12 – Code de génération d'un mauvais automate de politique

La spécification de cette politique ne correspond pas au comportement attendu du système car la lumière d'indication de la conduite autonome ne reflète pas l'état de la conduite autonome. Il s'agit d'une erreur grossière ayant pour seul but de mettre en avant les conséquences que peut avoir une politique incorrecte, comme nous allons le voir lors de la vérification.

D'après le comportement de l'ISW que nous avons précédemment décrit (*cf.* Section 3.2.3 : "Comportement de l'ISW"), on comprend que cette politique va empêcher le bon fonctionnement du système. Le message d'allumage de la lumière ne pourra jamais être envoyé. En effet, la carte mère envoie le message d'allumage après avoir envoyé le message d'activation de la conduite autonome, ce qui est interdit par la politique de la Figure 4.19. Ainsi, même si la safety du système est préservée (*i.e.* la lumière n'est jamais allumée, donc ϕ est vérifiée), cela est accompli au détriment de la disponibilité (ψ est donc violée) comme le montre la Figure 4.20 : "Automate de vérification de l'ISW pour les propriétés ψ et ξ avec une mauvaise politique de contrôle d'accès". ξ est aussi violée car une fonction du système – l'allumage de la lumière – ne peut être remplie.

Ainsi, la vérification de propriétés de disponibilité et d'intégrité est essentielle pour assurer que la politique est correctement spécifiée pour le système. Dans le cas présent, la propriété de safety – ϕ – est toujours vérifiée, mais on pourrait imaginer une erreur de spécification dans la politique qui pourrait provoquer une violation de la safety.

4.3.3 Vérification avec la politique de contrôle d'accès en cas d'attaque

Après l'ajout de la politique de contrôle d'accès, nous vérifions à nouveau le système global, toujours par rapport aux propriétés ϕ, ψ et ξ. Dans un premier temps, la vérification est effectuée lorsque le système est attaqué, l'objectif étant de s'assurer que la politique empêche l'attaque d'aboutir. Cette vérification génère l'automate de la Figure 4.21 : "Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ en cas d'attaque, avec une politique de contrôle d'accès". On constate que tous les états sont bleus, indiquant que ϕ, ψ et ξ sont vérifiées dans chacun des états : l'attaque est bloquée

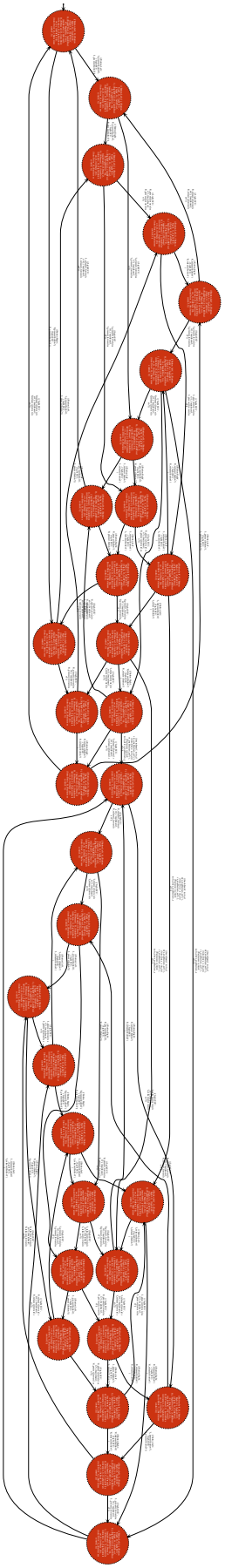


FIGURE 4.20 – Automate de vérification de l'ISW pour les propriétés ψ et ξ avec une mauvaise politique de contrôle d'accès

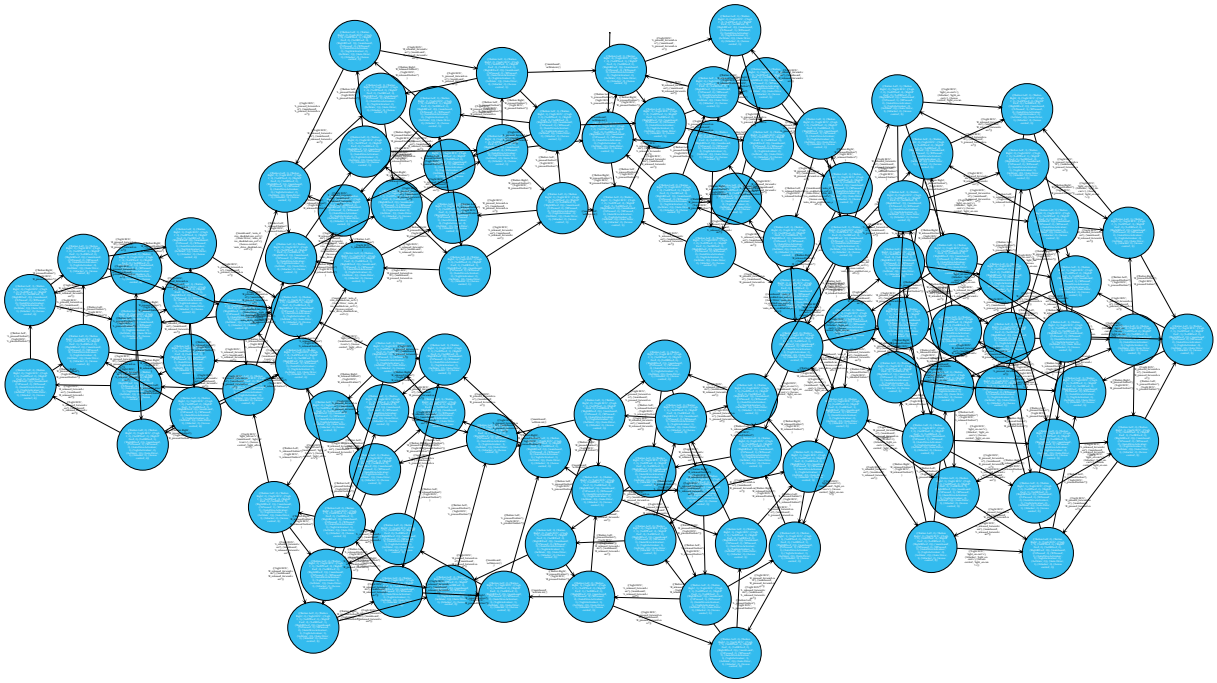


FIGURE 4.21 – Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ en cas d'attaque, avec une politique de contrôle d'accès

et la **safety** est préservée. Ainsi, nous sommes capables de garantir des propriétés de **safety** face aux capacités d'un attaquant, formalisées dans le même formalisme que les propriétés à vérifier. De plus, il est toujours possible d'allumer la lumière : la politique n'empêche donc pas le système de fonctionner correctement pour bloquer l'attaque.

Néanmoins, l'attaquant a pu allumer la lumière alors que la conduite autonome est activée, une tâche qui revient normalement à la carte mère. Cette action constitue une violation du point de vue de la cybersécurité, bien que la **safety** du système soit préservée. De plus, cette action peut causer une désynchronisation entre la carte mère, le contrôle d'accès et le reste du système. En effet, le contrôle d'accès n'autorise qu'un seul allumage de la lumière, qui a donc été effectué par l'attaquant. De fait, lorsque la lumière enverra son message demandant l'allumage de la lumière, ce dernier sera refusé. Nous abordons ce point plus en détails dans le Chapitre 5 : "Expérimentations supplémentaires". Cette action est autorisée par notre politique étant donné qu'elle ne contrôle que les méthodes, et pas la source ou la destination des messages envoyés. Ainsi, l'attaquant peut allumer la lumière, sans perturber le fonctionnement du système d'une manière qui est observable avec nos propriétés. Il est nécessaire de mettre en place un contrôle de l'application source ayant envoyé le message pour contrer cette situation. Contrôler la source ajoute des contraintes supplémentaires pour l'attaquant qui doit maintenant usurper l'identité de l'application qui a la légitimité d'envoyer le bon message, au bon moment (*i.e.* quand l'automate de contrôle d'accès est dans le bon état). Cette usurpation peut être complexe à mettre en œuvre si les messages sont authentifiés. Cependant, comme nous l'avons dit dans l'état de l'art, les procédés cryptographiques

ne sont pas abordés ici. De plus, nous n'avons pas eu le temps d'explorer cette approche avec l'outil du Dr. Hugot qui nécessitait des modifications supplémentaires pour offrir les abstractions nécessaires à la mise en place du contrôle de la source d'un message. Nous abordons ce point plus en détails lorsque nous passerons à SPIN.

Afin de nous assurer qu'en plus de préserver la safety, la politique ne perturbe pas le fonctionnement du système en temps normal, nous allons effectuer une ultime vérification de notre système avec la politique, lorsqu'il n'est pas attaqué.

4.3.4 Vérification avec la politique de contrôle d'accès, sans attaque

Même si la politique permet au système de fonctionner normalement en cas d'attaque, et de préserver sa propriété de **safety**, nous devons aussi nous assurer que le système fonctionne nominalement lorsqu'il n'est pas attaqué mais que la politique est active. Pour cela, nous effectuons une dernière vérification de l'ISW, sans attaque mais avec la politique afin de garantir que l'intégration de la politique ne bloque pas des aspects safety, fonctionnels ou de disponibilité. Cette vérification génère l'automate des Figures 4.22 et 4.23, où tous les états sont bleus, indiquant que ϕ, ψ et ξ sont préservées dans chacun des états malgré la présence de la politique.

Notre politique de contrôle d'accès ne perturbe pas le fonctionnement de notre système même dans un cas nominal. Elle permet donc de préserver la **safety**, la disponibilité et l'intégrité fonctionnelle du système.

4.4 Conclusion sur la modélisation avec l'outil du Dr. Hugot

L'outil du Dr. Hugot est très adapté pour vérifier rapidement le respect des propriétés de safety au regard d'un profil d'attaquant. Il permet de voir au premier regard si le système global respecte les propriétés voulues et aide à analyser les chemins erronés. Ces deux points sont des atouts importants pour le domaine automobile, peu familier avec la vérification formelle. Cette approche a produit des résultats encourageants en termes de **safety**. L'ajout de notre politique de contrôle d'accès permet de préserver la safety sans affecter la disponibilité ou l'intégrité, même en présence de l'attaquant. Celui-ci ne peut mettre en danger les occupants du véhicule, la **safety** est donc préservée. Enfin, l'outil du Dr. Hugot permet de vérifier rapidement si la politique comporte une erreur, comme nous l'avons vu avec la Figure 4.20 : "Automate de vérification de l'ISW pour les propriétés ψ et ξ avec une mauvaise politique de contrôle d'accès".

Néanmoins, nous voulions expérimenter d'autres pistes nécessitant des abstractions supplémentaires, et une logique différente – LTL – qui n'étaient pas nativement offertes par l'outil du Dr. Hugot. De plus, nous voulions confronter ces premiers résultats à ceux obtenus avec un autre outil afin de les mettre en perspective. De fait, nous allons

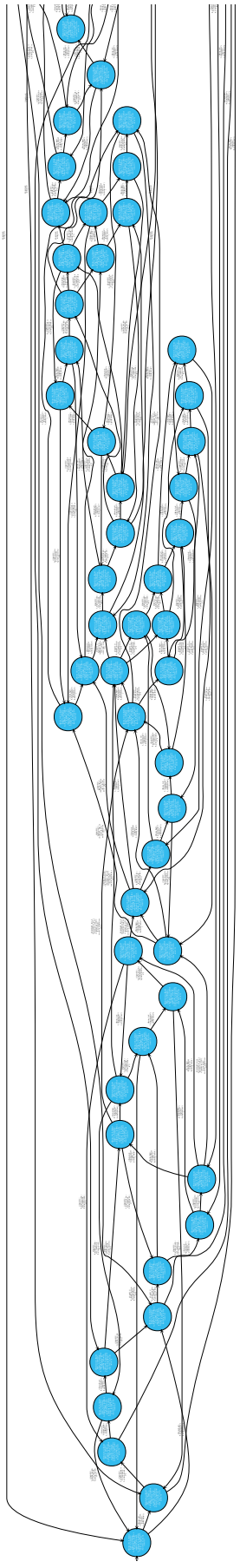


FIGURE 4.22 – Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ avec une politique de contrôle d'accès (première partie)

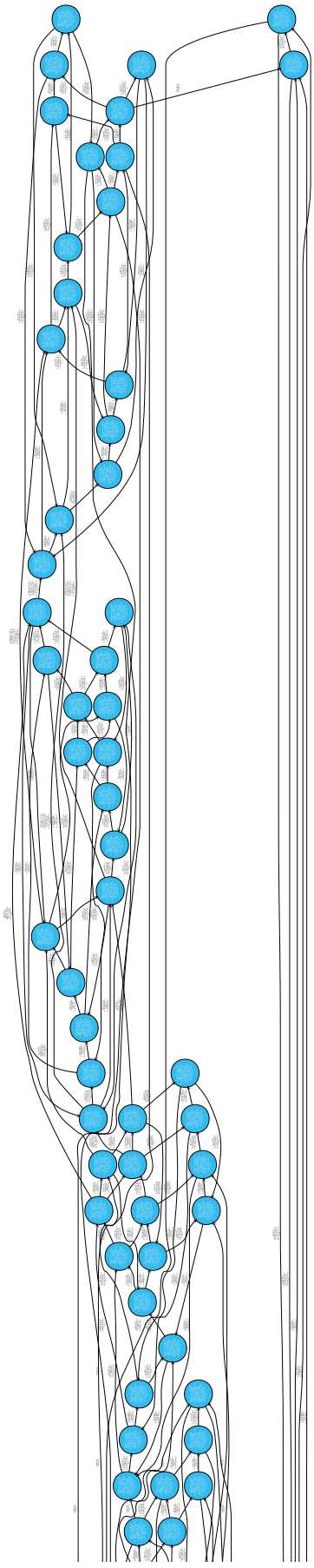


FIGURE 4.23 – Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ avec une politique de contrôle d'accès (deuxième partie)

à nouveau modéliser et vérifier l'ISW avec un outil connu de la communauté de la vérification formelle : SPIN/PROMELA.

4.5 Modélisation avec SPIN

Nous passons maintenant à la modélisation et à la vérification de l'ISW en utilisant SPIN. Comme avec l'outil du Dr. Hugot, la première étape est de modéliser à nouveau les différents composants de l'ISW. Le format de spécification utilisé par SPIN est plus bas niveau que la spécification sous forme d'automates utilisée avec l'outil du Dr. Hugot. En effet, SPIN nécessite de spécifier des éléments supplémentaires (canaux de communication, boucles, conditions) rendant la spécification des différents composants plus complexe. La spécification s'effectue grâce à un langage nommé PROMELA [Hol91], permettant de décrire la structure de l'automate sous-jacent pour chaque composant. SPIN n'offre pas de représentation graphique pour afficher le résultat du produit synchronisé ou de la vérification. Bien que proposant des traces en tant que contre exemple lors de la vérification, cela reste moins convivial que la visualisation colorée proposée par l'outil du Dr. Hugot.

Avec SPIN et les abstractions supplémentaires offertes par PROMELA, nous allons modéliser à nouveau nos différents composants, en restant le plus proche possible des modélisations faites avec l'outil du Dr. Hugot. L'objectif est d'observer, lors de la phase de vérification, si notre politique de contrôle d'accès a les mêmes effets.

4.5.1 Prélude du modèle PROMELA

La modélisation de l'ISW avec le langage PROMELA utilisé par SPIN nécessite de définir certains éléments en dehors des spécifications individuelles des composants. Ces différents éléments sont donnés dans le Listing 4.13 : "Prélude du code PROMELA".

Nous commençons par définir trois macros (lignes 2, 4 et 7) pour activer notre contrôle d'accès, et paramétrer notre attaquant facilement lors de la phase de vérification. Nous importons deux fichiers (lignes 10 et 11) définissant des fonctions utilitaires et les formules LTL qui seront vérifiées sur la modélisation. Nous définissons ensuite des identifiants pour chacun de nos composants (lignes 14-20), la structure d'un message échangé entre 2 composants (lignes 23-28) et un ensemble de canaux de communication pour échanger des messages entre 2 composants (lignes 31-38). Dans notre cas, les canaux ne peuvent pas stocker de messages (`[0] of`). Pour échanger un message entre deux composants, le message est envoyé et reçu en une seule opération, de manière synchronisée [R.12]. Enfin, nous déclarons certaines variables utilisées dans les formules LTL et nous démarrons les processus souhaités dans le bloc `init`.

```
1 /* 1 active le contrôle d'accès, 0 désactive */
2 #define AC 1
3 /* 1 active l'attaquant, 0 désactive */
```

```
#define ATTACKER 1
/* 1 l'attaquant peut envoyer un nombre illimité de message,
0 l'attaquant peut envoyer un seul message */
#define ATTACKER_LOOP 1

/* Importation de fichiers */
#include "utility.pml"
#include "ltl.pml"

/* Identifiants des composants */
byte attacker_id = 5
byte mainboard_id = 3
byte light_id = 2
byte auto_drive_id = 4
byte left_id = 11
byte right_id = 12
byte light_ac_id = 1

/* Structure d'un message */
typedef message {
    byte src;
    byte dst;
    bool cmd;
    byte button_pressed;
};

/* Canaux de communication pour la modélisation */
chan buttons[2] = [0] of { message };

chan light_to_mainboard = [0] of { message };
chan light_to_light_AC = [0] of { message };

chan mainboard_to_auto_drive = [0] of { message };
chan mainboard_to_light = [0] of { message };
chan mainboard_to_light_AC = [0] of { message };

/* Type personnalisé pour envoyer un message à
deux canaux */
typedef mcast_group_of_2 {
    chan group[2];
};

/* Variables requises pour la modélisation */
bool auto_drive_status = 0;
bool light_status = 0;
bool mainboard_L_pressed = 0, mainboard_R_pressed = 0;
bool act_state = 0, mainboard_auto_drive_status = 0, mainboard_light_status = 0;
bool attack_succeed = 0;

init
{
```

```
55      /* Démarrage des proctypes */
56      atomic
57      {
58          run button(left_id)
59          run button(right_id)
60          #if AC == 1
61          run light_AC()
62          #endif
63          run light();
64          run mainboard();
65          run auto_drive();
66          #if ATTACKER == 1
67          run attacker();
68          #endif
69      }
70  }
```

Listing 4.13 – Prélude du code PROMELA

Nous allons maintenant passer à la déclaration des différents composants. Avec PROMELA, le comportement de chaque composant est spécifié dans un proctype.

4.5.2 Modélisation de l'ISW

Comme précédemment, nous commençons par modéliser les différents composants de l'ISW. L'objectif est là encore de spécifier le comportement des différents composants en accord avec la description faite en Section 3.2 : "Présentation du cas d'usage réel". Nous débutons par une description des boutons, de la lumière et de la carte mère. Puis nous passons à la modélisation de l'attaquant et de la conduite autonome.

Modélisation des boutons

Les boutons de l'ISW sont modélisés par un unique proctype et sont identifiés par un ID défini dans le prélude comme le montre le Listing 4.14 : "Code du proctype des boutons".

```
1  proctype button(byte id)
2  {
3      /* Initialisation de l'état du bouton et des paramètres du message */
4      bool pressed = 0;
5      message sent;
6      sent.src = id;
7      sent.dst = light_id;
8      /* Boucle infinie d'envoi avec changement d'état du bouton */
9      do
10     ::  pressed = !pressed -> sent.cmd = pressed -> buttons[(id % 2)]!sent;
11     od
12 }
```

Listing 4.14 – Code du proctype des boutons

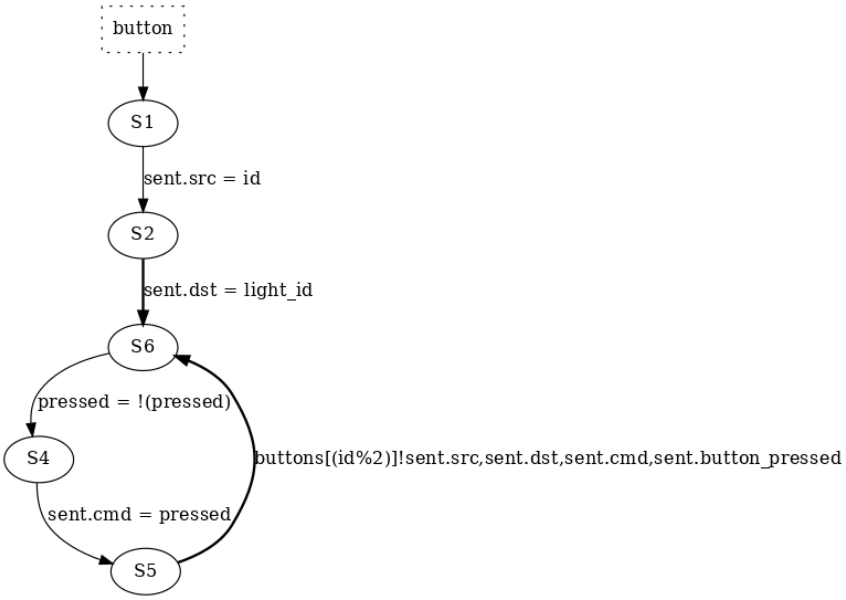


FIGURE 4.24 – Automate sous-jacent au proctype des boutons

Par défaut, les boutons sont relâchés (`bool pressed = 0;`). À chaque tour de boucle, les boutons changent leur état interne (`pressed = !pressed`) et envoient leur nouvel état à la lumière avec un message (`buttons[(id % 2)]!sent`). Ce proctype décrit l'automate sous-jacent présenté par la Figure 4.24. On retrouve bien la boucle représentant le changement d'état du bouton et l'envoi du message, comme sur les Figure 4.1 : "Automate de spécification du bouton gauche" et Figure 4.2 : "Automate de spécification du bouton droit". On constate que chaque transition de l'automate de la Figure 4.24 correspond à une instruction du Listing 4.14. Les automates sont donc difficilement comparables avec ceux de l'outil du Dr. Hugot, même si certaines structures – des boucles – peuvent se retrouver, bien que nous implémentons les mêmes comportements. Pour cette raison, ainsi que pour l'absence de retours visuels lors de la phase de vérification, nous ne présenterons pas les automates de SPIN et ne les comparerons pas avec ceux de l'outil du Dr. Hugot.

Nous allons maintenant passer à la modélisation de la lumière avec SPIN.

Modélisation de la lumière

La lumière reçoit les messages envoyés par les deux boutons, et doit transmettre l'état reçu à la carte mère. La lumière peut se trouver dans 4 situations différentes :

(1) elle n'a aucun état de bouton à transmettre ;

(2) elle doit transmettre l'état du bouton gauche à la carte mère ;

(3) elle doit transmettre l'état du bouton droit à la carte mère ;

(4) elle doit transmettre l'état des deux boutons à la carte mère.

Ces 4 situations sont représentées dans le Listing 4.15 : "Code du proctype de la lumière" par 4 blocs de code : `start` (situation 1), `forward_left` (situation 2), `forward_right` (situation 3) et `forward_both` (situation 4), définis par des labels (*cf.* lignes 13, 25, 42 et 61).

Comme lors de la modélisation avec l'outil du Dr. Hugot, la lumière transfère à la carte mère le premier état reçu, et ne peut pas recevoir le changement d'état d'un bouton tant que l'état actuel n'a pas été transféré à la carte mère. Ceci constitue une limitation de notre modélisation : que se passe-t-il si le conducteur appuie frénétiquement sur un bouton ? Comme avec la modélisation faite avec l'outil du Dr. Hugot, il n'y a pas de notion de temps, ainsi le traitement électronique de la pression d'un bouton est aussi rapide que la pression en elle même. Nous considérons que cela constitue une des limites de la modélisation.

Chacune des 4 situations est constituée d'une boucle dans laquelle sont effectuées les différentes opérations liées à la situation. À titre d'exemple, nous allons détailler la situation 2 (lignes 25 à 35 du Listing 4.15). Nous commençons par définir le message qui peut être envoyé à la carte mère (lignes 26 à 29). Ce message provient de la lumière, est envoyé à la carte mère, indique l'état du bouton (pressé ou relâché) ainsi que le bouton concerné. Ensuite, nous décrivons les opérations que peut effectuer la lumière dans cet état :

— envoyer l'état du bouton gauche à la carte mère (ligne 32) ;

— recevoir l'état du bouton droit (ligne 33) ;

— recevoir une commande de la carte mère (impactant l'état de la lumière, ligne 34).

Ces différentes actions sont placées dans une boucle, ainsi la lumière peut recevoir autant de messages de la carte mère que possible. En revanche, les opérations sur les boutons déclenchent un saut vers une autre situation et ne seront exécutées qu'une fois. Une particularité intéressante de SPIN réside dans sa gestion des différentes options d'une boucle ou d'une conditionnelle. Dans un bloc conditionnel ou une boucle, SPIN sélectionne de manière non-déterministe une des options exécutables. Si aucune option n'est exécutable, la boucle est bloquée jusqu'à ce qu'une option devienne exécutable.

Dans les cas où la lumière envoie l'état du bouton gauche ou bien reçoit l'état du bouton droit, alors on change de situation (*cf.* `goto`), c'est-à-dire que les opérations à effectuer par la lumière changent. Par exemple, si la lumière reçoit l'état du bouton droit (*cf.* ligne 33), alors on passe dans le bloc `forward_both` où il n'est plus possible de recevoir l'état d'un bouton (la lumière doit transmettre l'état d'un des deux boutons à la carte mère).

Si la lumière reçoit un message de la carte mère, alors l'état de la lumière est mis à jour en même temps que le message est reçu (*cf.* ligne 34).

```
1 proctype light()
2 {
3     /* Initialisation des variables de la lumière et des paramètres du message*/
4     bool light_L_pressed = 0, light_R_pressed = 0;
5     message received;
```

```
    message sent;
    message sent_2;

    /* Etat start, la lumière attend une commande d'un des deux boutons. En fonction
    de l'ID du bouton, la lumière passe dans le bloc forward_left ou forward_right
    Si un message est reçu dans le canal de communication avec la carte mère est reçu,
    l'état de la lumière est mis à jour */
start:
    do
    ::  buttons[(left_id % 2)]?received -> light_L_pressed = received.cmd -> goto forward_left;
    ::  buttons[(right_id % 2)]?received -> light_R_pressed = received.cmd -> goto forward_right;
    ::  atomic{mainboard_to_light?received; light_status = received.cmd;}
    od

    /* Un message du bouton gauche a été reçu et doit être transmis à la carte mère
    Si l'envoi est effectué, alors la lumière retourne dans le bloc start, il n'y a
    plus d'état à transmettre à la carte mère
    La lumière peut toujours recevoir l'état du bouton droit et passer dans le bloc
    forward_both, ou bien une commande de la carte mère impactant l'état de la lumière */
forward_left:
    sent.src = light_id;
    sent.dst = mainboard_id;
    sent.cmd = light_L_pressed;
    sent.button_pressed = left_id;

    do
    ::  light_to_mainboard!sent -> goto start;
    ::  buttons[(right_id % 2)]?received -> light_R_pressed = received.cmd -> goto forward_both;
    ::  atomic{mainboard_to_light?received; light_status = received.cmd;}
    od

    /* Un message du bouton droit a été reçu et doit être transmis à la carte mère
    Si l'envoi est effectué, alors la lumière retourne dans le bloc start, il n'y a
    plus d'état à transmettre à la carte mère
    La lumière peut toujours recevoir l'état du bouton droit et passer dans le bloc
    forward_both, ou bien une commande de la carte mère impactant l'état de la lumière */
forward_right:
    sent.src = light_id;
    sent.dst = mainboard_id;
    sent.cmd = light_R_pressed;
    sent.button_pressed = right_id;

    do
    ::  buttons[(left_id % 2)]?received -> light_L_pressed = received.cmd -> goto forward_both;
    ::  light_to_mainboard!sent -> goto start;
    ::  atomic{mainboard_to_light?received; light_status = received.cmd;}
    od

    /* Des messages du bouton gauche et droit ont été reçus et doivent être transmis
    à la carte mère
    Si l'un des deux est transmis, la lumière retourne dans le bloc forward_left
```

```
57      (resp. forward_right)
58      si l'état du bouton droit (resp. gauche) a été transmis
59      La lumière peut toujours recevoir une commande de la carte mère impactant
60      l'état de la lumière */
61  forward_both:
62      sent.src = light_id;
63      sent.dst = mainboard_id;
64      sent.cmd = light_L_pressed;
65      sent.button_pressed = left_id;
66
67      sent_2.src = light_id;
68      sent_2.dst = mainboard_id;
69      sent_2.cmd = light_R_pressed;
70      sent_2.button_pressed = right_id;
71
72      do
73      ::  light_to_mainboard!sent -> goto forward_right;
74      ::  light_to_mainboard!sent_2 -> goto forward_left;
75      ::  atomic{mainboard_to_light?received; light_status = received.cmd;}
76      od
77  }
```

Listing 4.15 – Code du proctype de la lumière

Ainsi, nous obtenons un comportement similaire à celui spécifié avec l'outil du Dr. Hugot (*cf.* Section 4.1 : "Modélisation avec le framework du Dr. Hugot"). Nous pouvons maintenant passer à la spécification de la carte mère.

Modélisation de la carte mère

Comme pour la modélisation faite avec l'outil du Dr. Hugot, la carte mère doit être en mesure de gérer au maximum les messages transmettant l'état des boutons. Ainsi, la spécification de la carte mère est constituée d'une unique boucle infinie définissant les différentes actions possibles.

La carte mère doit :

(1) réceptionner les états des deux boutons envoyés par la lumière d'indication ;

(2) allumer la lumière d'indication ;

(3) activer la conduite autonome ;

(4) désactiver la conduite autonome ;

(5) éteindre la lumière d'indication ;

(6) détecter le relâchement des boutons à la fin de l'activation de la conduite autonome ;

(7) détecter le relâchement des boutons à la fin de la désactivation de la conduite autonome.

Ces 7 situations sont chacune décrites par une option dans la boucle infinie du `proctype` de la carte mère.

À titre d'exemple, nous allons détailler l'activation de la conduite autonome (*cf.* lignes 22 à 26 du Listing 4.16). La première ligne de chaque situation définit quelles sont les conditions à remplir pour que cette situation soit exécutable. Pour l'activation de la conduite autonome, il faut que la carte mère ait reçu des messages indiquant que les deux boutons sont pressés, que la conduite autonome soit désactivée, que la lumière soit éteinte et que les deux boutons n'aient pas encore été relâchés (*i.e.* il s'agit d'une activation). Dès lors, la carte mère construit un message (*cf.* lignes 23 à 25) et l'envoie à la conduite autonome (ligne 26). La carte mère change son état interne en indiquant que la conduite autonome est maintenant activée (ligne 26) et peut exécuter la situation permettant l'allumage de la lumière, ou bien recevoir un changement d'état des boutons.

```
1  proctype mainboard()
2  {
3      /* Déclaration des variables de la carte mère */
4      message received;
5      message sent;
6      bool light_off = 1;
7
8      /* La carte mère effectue différentes actions dans une boucle infinie */
9      do
10     /* Elle peut recevoir des messages de la lumière (état des boutons) */
11     ::  light_to_mainboard?received ->
12         /* En fonction du bouton pressé, la carte mère change l'état connu du
13         bouton concerné */
14         if
15         ::  received.button_pressed == left_id -> mainboard_L_pressed = received.cmd;
16         ::  received.button_pressed == right_id -> mainboard_R_pressed = received.cmd;
17         fi
18
19     /* Si les deux boutons sont pressés, la conduite autonome désactivée,
20     la lumière éteinte, et que l'on attend pas un relâchement des boutons,
21     alors la carte mère envoie un message d'activation à la conduite autonome */
22     ::  mainboard_L_pressed && mainboard_R_pressed && !mainboard_auto_drive_status && !
       mainboard_light_status && !act_state->
23         sent.src = mainboard_id;
24         sent.dst = auto_drive_id;
25         sent.cmd = 1;
26         mainboard_to_auto_drive!sent -> mainboard_auto_drive_status = 1;
27
28     /* Si les deux boutons sont pressés, la conduite autonome activée,
29     la lumière éteinte, et que l'on attend pas un relâchement des boutons,
30     alors la carte mère envoie un message d'allumage à la lumière */
31     ::  mainboard_L_pressed && mainboard_R_pressed && mainboard_auto_drive_status && !
       mainboard_light_status && !act_state->
32         sent.dst = light_id;
33         mainboard_to_light!sent -> mainboard_light_status = 1;
34
35     /* Si les deux boutons sont pressés, la conduite autonome activée,
36     la lumière allumée, et que les boutons ont été relâchés,
```

```
37      alors la carte mère envoie un message d'extinction de la lumière */
38      ::  mainboard_L_pressed && mainboard_R_pressed &&  mainboard_auto_drive_status &&
        mainboard_light_status && act_state ->
39          sent.src = mainboard_id;
40          sent.dst = light_id;
41          sent.cmd = 0;
42          mainboard_to_light!sent -> mainboard_light_status = 0; /* on demande à la lumière de s'é
        teindre */
43
44      /* Si les deux boutons sont pressés, la conduite autonome activée,
45      la lumière éteinte, et que les boutons ont été relâchés,
46      alors la carte mère envoie un message de désactivation de la conduite autonome */
47      ::  mainboard_L_pressed && mainboard_R_pressed &&  mainboard_auto_drive_status && !
        mainboard_light_status && act_state ->
48          sent.dst = auto_drive_id;
49          mainboard_to_auto_drive!sent -> mainboard_auto_drive_status = 0; /* on désactive la
        conduite autonome */
50
51      /* Si les deux boutons sont relâchés, la conduite autonome activée,
52      la lumière allumée, et que les boutons n'ont pas encore été relâchés
53      alors que l'activation de la conduite autonome est terminée,
54      alors la carte mère note que les deux boutons ont été relâchés
55      à la fin de l'activation */
56      :: !mainboard_L_pressed && !mainboard_R_pressed && mainboard_auto_drive_status &&
        mainboard_light_status && !act_state ->
57          act_state = 1;
58
59      /* Si les deux boutons sont relâchés, la conduite autonome désactivée,
60      la lumière éteinte, et que l'on attend un relâchement des boutons,
61      alors la carte mère note que les boutons ont été relâchés à la
62      fin de la désactivation*/
63      :: !mainboard_L_pressed && !mainboard_R_pressed && !mainboard_auto_drive_status && !
        mainboard_light_status && act_state ->
64          act_state = 0;
65
66      od
67  }
```

Listing 4.16 – Code du proctype de la carte mère

Maintenant que les éléments au cœur de l'ISW sont spécifiés, nous pouvons passer à la spécification de l'attaquant.

4.5.3 Modélisation de l'attaquant

Notre attaquant conserve les mêmes capacités que lors de la modélisation avec l'outil du Dr. Hugot. Nous permettons à notre attaquant d'envoyer un nombre de messages illimité, ou un seul, nous permettant de distinguer une violation de propriété par déni de service, ou par violation d'intégrité.

Sa modélisation en PROMELA est donnée par le Listing 4.17.

```
proctype attacker()
{
    /* Initialisation des variables de l'attaquant et des paramètres du message */
    message sent;
    sent.src = attacker_id;
    sent.dst = light_id;
    sent.cmd = 1;
    sent.button_pressed = attacker_id;

    /* Boucle infini d'envoie de message de l'attaquant*/
    do
    ::  mainboard_to_light!sent;
    od
}
```

Listing 4.17 – Code du proctype de l'attaquant

Comme avec l'outil du Dr. Hugot, nous pouvons formaliser les capacités de notre attaquant avec une formule de logique temporelle. Lorsqu'il envoie un nombre illimité de messages, ses capacités peuvent être formalisées par :

$$\Box\Diamond send\_message$$

Cette formule peut être traduite par "Infiniment souvent, l'attaquant envoie un message". S'il ne peut envoyer qu'un unique message :

$$\Diamond send\_message$$

Celle-ci se traduit par "Fatalement, l'attaquant envoie un message".

Afin que notre modélisation de l'ISW soit exploitable, il ne manque plus que la modélisation de la conduite autonome qui est détaillée dans la section suivante.

4.5.4 Modélisation de la conduite autonome

Comme avec la modélisation faite avec l'outil du Dr. Hugot, la spécification de la conduite autonome est rapide. Le proctype de la conduite autonome attend en permanence de recevoir un message pour changer l'état de la conduite autonome. L'opération de réception du message et le changement de la valeur sont effectuées de manière atomique pour garantir que la réception du message est immédiatement suivie de la mise à jour de la valeur.

```
proctype auto_drive()
{
    /* Initialisation des variables de la conduite autonome */
```

```
4     message received;
5     /* La conduite autonome reçoit indéfiniment les messages de la carte
6     mère et met à jour son état interne */
7     do
8     ::  atomic{mainboard_to_auto_drive?received -> auto_drive_status = received.cmd;}
9     od
10 }
```

Listing 4.18 – Code du proctype de la conduite autonome

Avec tous les éléments nécessaires au fonctionnement de l'ISW, nous pouvons effectuer une première vérification de la modélisation. Nous ne proposons pas de représentation graphique de l'automate du système global généré par SPIN étant donné que ce n'est pas une fonctionnalité offerte par SPIN.

4.6 Première vérification avec SPIN

Les vérifications avec SPIN ne sont pas aussi visuelles que celles faites avec l'outil du Dr. Hugot. Lors d'une vérification, en cas d'erreur, une trace de l'exécution menant à la violation d'une propriété est produite, permettant ainsi de dérouler facilement le scénario en question. En cas d'erreur pour les vérifications, nous donnons la trace produite par SPIN. Avant de pouvoir passer à la vérification, nous devons définir les propriétés à vérifier sur le système. En effet, là où l'outil du Dr. Hugot utilisait la logique CTL, SPIN utilise LTL. Nous devons donc redéfinir des propriétés, les plus approchantes possibles de celles définies en CTL.

4.6.1 Définition des propriétés à vérifier sur le système global

Comme avec l'outil du Dr. Hugot, les propriétés à vérifier couvrent différents aspects de l'ISW.

Propriété de safety

$$\phi' = \Box(light\_status \Rightarrow auto\_drive\_status)$$

Propriété de disponibilité

$$\psi' = \Box(\neg\Box\Diamond(mainboard\_L\_pressed \vee mainboard\_R\_pressed) \Rightarrow \Diamond light\_status)$$

Propriété fonctionnelle

$$\xi' = \Box((auto\_drive\_status \wedge \neg act\_state \wedge \neg\Box\Diamond(mainboard\_L\_pressed \vee mainboard\_R\_pressed)) \Rightarrow \Diamond light\_status)$$

Ainsi, nous définissons à nouveau une propriété de **safety** ϕ' qui se traduit par "Toujours, si la lumière est allumée, alors la conduite autonome est activée". La deuxième propriété,

ψ' est une propriété de disponibilité "Toujours, si les boutons gauche ou droit ne sont pas infiniment souvent pressés, alors fatalement, la lumière s'allumera". Nous aurions pu définir ψ' de manière plus directe avec la formule suivante $\Diamond(light\_status)$ ("Fatalement, la lumière s'allumera"). Or, cette propriété n'aurait jamais été vérifiée sur le système. En effet, à la différence de CTL dont les propriétés sont évaluées sur les chemins possibles à partir d'un état donné, les propriétés LTL sont évaluées sur une trace d'exécution du système. Ainsi, s'il existe une trace violant une propriété, le système ne vérifie pas la propriété. Or, notre système peut indéfiniment alterner entre des pressions et des relâchements des boutons. Ainsi, il existe une infinité de traces violant la propriété $\Diamond(light\_status)$ (*i.e.* à partir du moment où le système boucle indéfiniment sur des pressions/relâchements de boutons). Pour proposer une propriété intéressante, nous ajoutons donc un prérequis, indiquant que les boutons gauche ou droit ne doivent pas être infiniment souvent pressés pour qu'un jour la lumière s'allume. Enfin ξ' est une propriété d'intégrité fonctionnelle "Toujours, si la conduite autonome est activée, et que *act_state* est faux et que les boutons gauche ou droit ne sont pas infiniment souvent pressés, alors fatalement la lumière s'allumera".

Avec ces propriétés, nous pouvons procéder à une première phase de vérification. Dans un premier temps, nous allons vérifier le système par rapport à ces 3 propriétés, sans attaque ni contrôle d'accès. Puis, nous vérifierons le système attaqué, mais non protégé.

4.6.2 Résultats de la vérification

Pour vérifier notre système, SPIN propose différents modes de vérification. Le premier que nous allons utiliser, est le mode *safety* (au sens de SPIN). Dans ce mode, SPIN recherche des violations d'assertions (non utilisées dans notre cas) ainsi que des deadlocks. Un deadlock au sens de SPIN correspond à un état ne possédant pas de transitions sortantes et n'étant pas final. Le second mode que nous utilisons est le mode *acceptance* permettant de vérifier des formules LTL sur le système.

Nous commençons donc par une première vérification, en mode *safety*, sans vérifier de propriétés LTL, pour nous assurer que le modèle ne peut pas se trouver en deadlock. Notre système ne comporte aucune erreur, comme le montre le Listing 4.19. Pour les prochaines vérifications, si le système ne comporte pas d'erreurs, nous ne donnerons pas la sortie générée par SPIN.

```
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000  -n
Pid: 15287

(Spin Version 6.5.1 -- 3 June 2021)
  + Partial Order Reduction

Full statespace search for:
  never claim           - (not selected)
  assertion violations  +
```

```
  cycle checks        - (disabled by -DSAFETY)
  invalid end states  +

State-vector 144 byte, depth reached 2717, errors: 0
    30407 states, stored
    17317 states, matched
    47724 transitions (= stored+matched)
        4 atomic steps
hash conflicts:        21 (resolved)

Stats on memory usage (in Megabytes):
    4.988 equivalent memory usage for states (stored*(State-vector + overhead))
    2.978 actual memory usage for states (compression: 59.70%)
          state-vector as stored = 75 byte + 28 byte overhead
  128.000 memory used for hash table (-w24)
    0.534 memory used for DFS stack (-m10000)
  131.464 total actual memory usage

pan: elapsed time 0.02 seconds
No errors found -- did you verify all claims?
```

Listing 4.19 – Output de la vérification du système, sans attaque ou contrôle d'accès, en mode *safety*

Nous effectuons ensuite une vérification de la conjonction des propriétés ϕ', ψ' et ξ' en mode *acceptance*. Là encore, la vérification ne génère pas d'erreurs, ce qui est attendu, le système n'étant pas attaqué.

Nous effectuons maintenant les mêmes vérifications mais en ajoutant l'attaquant. La vérification en mode *safety* ne produit aucune erreur. En revanche, lorsque nous vérifions nos 3 propriétés, SPIN trouve un total de 19056 erreurs. Comme avec l'outil du Dr. Hugot, c'est la propriété de **safety** ϕ' qui est violée, ce qui est attendu étant donné la nature de l'attaque. SPIN génère donc une trace de la violation, retranscrite dans le Listing 4.20. On peut voir des lignes 1 à 14 le démarrage de chacun des proctypes depuis le bloc `init` du Listing 4.13 : "Prélude du code PROMELA". La partie tronquée du listing décrit des déclarations de variables pour les messages. Notre attaquant envoie un message d'allumage de la lumière à la ligne 16, reçu par la lumière à la ligne 17. Puis la lumière met à jour sa valeur interne ligne 18 et SPIN nous montre la violation lignes 19 à 21. On voit en effet que l'assertion violée est la suivante : $assert(!(!((!(light\_status)||auto\_drive\_status))))$ ce qui ne ressemble pas à ϕ'. SPIN réécrit ϕ' sous la forme $\phi' = \Box(!light\_status||auto\_drive\_status)$ et cherche une trace qui satisfait la négation de cette formule (*cf.* ligne 19 du Listing 4.20). Si la formule de la ligne 19 est satisfaite, alors une assertion est violée (*cf.* ligne 21).

```
    proc  - (phi:1) _spin_nvr.tmp:4 (state 4) [(1)]
Never claim moves to line 4 [(1)]
Starting button with pid 2
    proc  0 (:init::1) isw.pml:59 (state 1) [(run button(left_id))]
```

```
Starting button with pid 3
    proc  0 (:init::1) isw.pml:60 (state 2) [(run button(right_id))]
Starting light with pid 4
    proc  0 (:init::1) isw.pml:64 (state 3) [(run light())]
Starting mainboard_tmp with pid 5
    proc  0 (:init::1) isw.pml:65 (state 4) [(run mainboard_tmp())]
Starting auto_drive with pid 6
    proc  0 (:init::1) isw.pml:66 (state 5) [(run auto_drive())]
Starting attacker with pid 7
    proc  0 (:init::1) isw.pml:68 (state 6) [(run attacker())]
    [...] /* Sortie tronquée pour plus de lisibilité */
    proc  6 (attacker:1) isw.pml:349 (state 6)  [mainboard_to_light!sent.src,sent.dst,sent.cmd,
    sent.button_pressed]
    proc  3 (light:1) isw.pml:165 (state 7) [mainboard_to_light?received.src,received.dst,
    received.cmd,received.button_pressed]
    proc  3 (light:1) isw.pml:165 (state 8) [light_status = received.cmd]
    proc  - (phi:1) _spin_nvr.tmp:3 (state 1) [(!((!(light_status)||auto_drive_status)))]
spin: _spin_nvr.tmp:3, Error: assertion violated
spin: text of failed assertion: assert(!(!((!(light_status)||auto_drive_status))))
```

Listing 4.20 – Exemple de trace menant à une violation de ϕ lorsque le système est attaqué

En résumé, l'échange de messages menant à la violation de ϕ', est décrit par le diagramme de séquence de la Figure 4.25. Cet échange peut avoir lieu à n'importe quel moment de l'exécution du système, tant que la lumière peut recevoir le message, ce qui explique les 19056 traces menant à une violation de ϕ'.

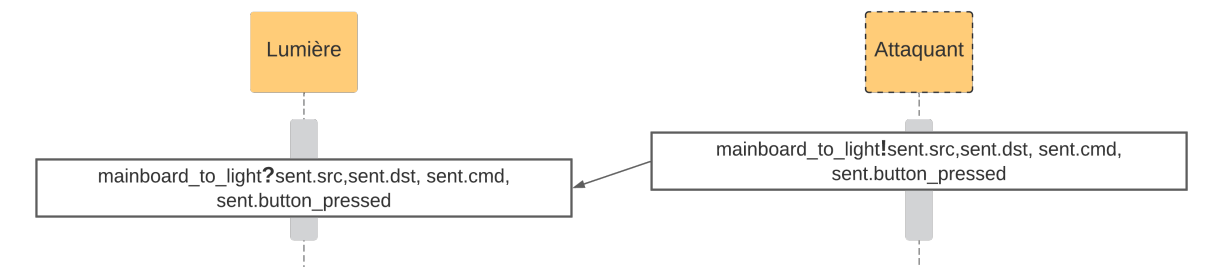


FIGURE 4.25 – Diagramme de séquence (inspiré de SPIN) menant à la violation de ϕ'

En revanche, l'attaque ne provoque pas de violation de ψ' et ξ'.

Notre attaquant perturbe le fonctionnement du système du point de vue de la **safety** ce qui n'est pas tolérable dans le secteur automobile. Pour y remédier, nous allons voir dans la section suivante comment mettre en place une politique de contrôle d'accès afin de contraindre l'exécution du système pour préserver sa **safety** même en cas d'attaque.

4.7 Enrichissement de la modélisation

Pour mettre en place notre politique de contrôle d'accès, nous devons – contrairement à la modélisation du Dr. Hugot – modifier la spécification de certains des composants. En effet, SPIN ne propose pas de manière standard un canal de communication de un vers plusieurs (*one-to-many*), nous devons donc mettre en place notre politique d'une autre manière. Ainsi, nous allons commencer par détailler le proctype de la politique, puis nous présentons les modifications apportées aux proctypes de la lumière et de la carte mère.

4.7.1 Modélisation de la politique de contrôle d'accès

Pour rappel, la politique de contrôle d'accès garantit la séquence suivante :

(1) activation de la conduite autonome ;

(2) allumage de la lumière ;

(3) extinction de la lumière ;

(4) désactivation de la conduite autonome.

Le proctype de la politique de contrôle d'accès est donné dans le Listing 4.21.

```
proctype light_AC()
{
    /* Déclaration des messages */
    message received;
    message sent;

    /* Le premier message autorisé est celui de l'activation de la
    conduite autonome, de fait, le contrôle d'accès attend ce message
    pour passer dans le bloc light_on.
    Si un message de la lumière est reçu (correspondant à une demande
    d'autorisation d'allumage), alors le contrôle d'accès envoie une
    interdiction, symbolisée par sent.cmd = 0 */
drive_on:
    sent.src = light_ac_id
    do
    ::  mainboard_to_light_AC?received ->
        if
        ::  received.cmd == 1 -> goto light_on;
        fi
    ::  light_to_light_AC?received -> sent.dst = light_id; sent.cmd = 0 -> light_to_light_AC!sent
    ;
    od

    /* Une fois que la conduite autonome est activée, on attend un
    message d'allumage de la lumière. Si ce message est reçu, on passe
    dans le bloc light_off.
    Si un message de la lumière est reçu (correspondant à une demande
    d'autorisation d'allumage), alors le contrôle d'accès envoie une
```

```
28      interdiction, symbolisée par sent.cmd = 0 */
29  light_on:
30      light_to_light_AC?received ->
31      /* Si on ne vérifie pas la source de la commande, alors peu importe d'où elle vient, elle
        sera exécutée */
32      /* Si SOURCE_CHECKER == 1 alors la commande est seulement exécutée lorsqu'elle ne provient
        pas de l'attaquant */
33      if
34      ::  (received.cmd == 1 && received.src != attacker_id) ->
35          sent.dst = light_id; sent.cmd = received.cmd; sent.button_pressed = 0 ->
36          light_to_light_AC!sent ->
37          if
38          :: received.button_pressed == attacker_id -> attack_succeed = 1 -> goto light_off;
39          :: else -> goto light_off;
40          fi
41      #if SOURCE_CHECKER == 0
42      ::  (received.cmd == 1 && (received.src == attacker_id || received.button_pressed ==
        attacker_id)) -> attack_succeed = 1 ->
43          sent.dst = light_id; sent.cmd = received.cmd ->
44          light_to_light_AC!sent -> goto light_off;
45      #endif
46      ::  else -> sent.dst = light_id; sent.cmd = 0 -> light_to_light_AC!sent -> goto light_on;
47      fi
48
49      /* Une fois que la lumière est allumée, on attend un
50      message d'extinction de la lumière. Si ce message est reçu, on passe
51      dans le bloc drive_off.
52      Si un message avec une autre commande est reçu, alors la lumière reste
53      allumée.*/
54  light_off:
55      light_to_light_AC?received ->
56      if
57      ::  received.cmd == 0 ->
58          sent.dst = light_id; sent.cmd = 0; sent.button_pressed = 0 ->
59          light_to_light_AC!sent ->
60          mainboard_to_light_AC!received; goto drive_off;
61      :: else -> sent.dst = light_id; sent.cmd = 1 -> light_to_light_AC!sent -> goto light_off;
62      fi
63
64      /* Une fois que la lumière est éteinte, on attend un
65      message de désactivation de la conduite autonome.
66      Si ce message est reçu, on revient dans le bloc drive_on pour pouvoir
67      procéder à une nouvelle activation.
68      Si on reçoit un message de la lumière entre temps, alors elle reste
69      éteinte.*/
70  drive_off:
71      do
72      ::  mainboard_to_light_AC?received ->
73          if
74          ::  received.cmd == 0 -> goto drive_on;
75          fi
```

```
76      ::  light_to_light_AC?received -> sent.dst = light_id; sent.cmd = 0 ->
77          light_to_light_AC!sent;
78      od
79  }
```

Listing 4.21 – Code du proctype de la politique de contrôle d'accès

Le proctype du contrôle d'accès utilise le canal de communication `light_to_light_AC` permettant l'échange de messages entre la lumière et le contrôle d'accès. En créant un canal dédié, cela nous permet de modéliser le fait que le contrôle d'accès est hébergé sur la lumière. Nous mettons aussi en place un canal de communication `mainboard_to_light_AC`. Cela permet à la carte mère de signaler l'état de la conduite autonome. En effet, nous rappelons que la conduite autonome communique uniquement avec la carte mère via un bus dédié. Or, SPIN ne permet pas une synchronisation sur ces messages (contrairement à l'outil du Dr. Hugot), nous obligeant à mettre en place un canal de communication supplémentaire. Une autre solution aurait été de modifier la carte mère pour qu'elle notifie la lumière de l'activation de la conduite autonome, afin de synchroniser la politique de contrôle d'accès. Nous trouvions cette approche plus complexe à mettre en place et avons donc choisi cette approche, plus directe.

Cette approche implique néanmoins de modifier la spécification de la lumière et de la carte mère afin de prendre en compte les échanges avec le contrôle d'accès.

4.7.2 Modification des proctypes de la lumière et de la carte mère

Pour la lumière, la modification consiste à ajouter dans chacune des situations (`start`, `forward_left`, `forward_right`, `forward_both`) une synchronisation avec le contrôle d'accès. Lorsque la lumière reçoit un message (d'allumage ou d'extinction) dans le canal `mainboard_to_light`, ce message est envoyé au contrôle d'accès, puis la lumière attend de recevoir une réponse du contrôle d'accès pour mettre à jour son état. Un exemple est donné dans le Listing 4.22, lignes 18-19.

```
1   proctype light()
2   {
3       /* Initialisation des variables de la lumière et des paramètres du message*/
4       bool light_L_pressed = 0, light_R_pressed = 0;
5       message received;
6       message sent;
7       message sent_2;
8
9       /* Etat start, la lumière attend une commande d'un des deux boutons. En fonction
10      de l'ID du bouton, la lumière passe dans le bloc forward_left ou forward_right
11      Si un message est reçu dans le canal de communication avec la carte mère est reçu,
12      l'état de la lumière est mis à jour */
13  start:
14      do
15      ::  buttons[(left_id % 2)]?received -> light_L_pressed = received.cmd -> goto forward_left;
16      ::  buttons[(right_id % 2)]?received -> light_R_pressed = received.cmd -> goto forward_right;
```

```
    #if AC == 1
    ::  mainboard_to_light?received -> light_to_light_AC!received ->
        atomic{light_to_light_AC?received -> light_status = received.cmd;}; /* Ligne en plus pour
     faire appel au contrôle d'accès */
    #else
    ::  atomic{mainboard_to_light?received; light_status = received.cmd;}
    #endif
    od
    [...] /* Même code que précédemment, avec la ligne supplémentaire ci-dessus dans chaque bloc
    (forward_left, forward_right, forward_both) */
}
```

Listing 4.22 – Code du proctype de la lumière pour la mise en place du contrôle d'accès

En ce qui concerne la carte mère, les modifications se trouvent au niveau de l'activation et la désactivation de la conduite autonome. Ces modifications sont données dans le Listing 4.23 : "Code du proctype de la carte mère pour la mise en place du contrôle d'accès".

```
proctype mainboard()
{
    /* Si le contrôle d'accès est activé, un tableau de deux canaux de communication
    est défini */
    #if AC == 1
    mcast_group_of_2 AC_light_group;
    AC_light_group.group[0] = mainboard_to_auto_drive;
    AC_light_group.group[1] = mainboard_to_light_AC;
    #endif

    /* Déclaration des variables utilisées par la carte mère */
    message received;
    message sent;
    bool light_off = 1;

    /* La carte mère effectue différentes actions dans une boucle infinie */
    do
    [...]

    /* La carte mère attend une confirmation du contrôle d'accès que la lumière est bien éteinte
    */
    ::  mainboard_to_light_AC?received -> light_off = 1;

    /* Si les deux boutons sont pressés, la conduite autonome désactivée,
    la lumière éteinte, et que l'on attend pas un relâchement des boutons,
    alors la carte mère envoie un message d'activation à la conduite autonome */
    ::  mainboard_L_pressed && mainboard_R_pressed && !mainboard_auto_drive_status &&
            !mainboard_light_status && !act_state->
        /* Si le contrôle d'accès est activé,  */
        #if AC == 1
        send_to(AC_light_group, 1, mainboard_id) -> mainboard_auto_drive_status = 1;
        /* Si le contrôle est désactivé, on garde le même fonctionnement que précédemment */
```

```
31          #else
32          sent.src = mainboard_id;
33          sent.dst = auto_drive_id;
34          sent.cmd = 1;
35          mainboard_to_auto_drive!sent -> mainboard_auto_drive_status = 1;
36          #endif
37
38      /* Si les deux boutons sont pressés, la conduite autonome activée,
39      la lumière éteinte, et que l'on attend pas un relâchement des boutons,
40      alors la carte mère envoie un message d'allumage à la lumière,
41      il n'y a pas d'interaction avec le contrôle d'accès car c'est la lumière
42      qui gère les appels avec le contrôle d'accès pour son état */
43      ::  mainboard_L_pressed && mainboard_R_pressed && mainboard_auto_drive_status &&
              !mainboard_light_status && !act_state->
44          sent.dst = light_id;
45          mainboard_to_light!sent -> mainboard_light_status = 1; light_off = 0;
46
47      /* Si les deux boutons sont pressés, la conduite autonome activée,
48      la lumière allumée, et que les boutons ont été relâchés,
49      alors la carte mère envoie un message d'extinction de la lumière,
50      il n'y a pas d'interaction avec le contrôle d'accès car c'est la lumière
51      qui gère les appels avec le contrôle d'accès pour son état */
52      ::  mainboard_L_pressed && mainboard_R_pressed &&  mainboard_auto_drive_status &&
              mainboard_light_status && act_state ->
53          sent.src = mainboard_id;
54          sent.dst = light_id;
55          sent.cmd = 0;
56          mainboard_to_light!sent -> mainboard_light_status = 0; /* on demande à la lumière de s'é
        teindre */
57
58      /* Si les deux boutons sont pressés, la conduite autonome activée,
59      la lumière éteinte, et que les boutons ont été relâchés,
60      alors la carte mère envoie un message de désactivation de la conduite autonome */
61      ::  mainboard_L_pressed && mainboard_R_pressed &&  mainboard_auto_drive_status &&
              !mainboard_light_status && act_state ->
62          /* Si le contrôle est activé, */
63          #if AC == 1
64          if
65          /* Si la lumière est bien éteinte */
66          ::  light_off == 1 ->
67              send_to(AC_light_group, 0, mainboard_id) -> mainboard_auto_drive_status = 0;
68          /* Sinon on passe à autre chose */
69          ::  else -> skip;
70          fi
71          /* Si le contrôle est désactivé, on garde le même fonctionnement que précédemment */
72          #else
73          sent.dst = auto_drive_id;
74          mainboard_to_auto_drive!sent -> mainboard_auto_drive_status = 0;
75          #endif
76
77      [...]
```

```
78      od
79  }
```

Listing 4.23 – Code du proctype de la carte mère pour la mise en place du contrôle d'accès

La politique de contrôle d'accès doit être au courant de l'état de la conduite autonome afin d'autoriser ou non l'allumage et l'extinction de la lumière. Les parties d'allumage et d'extinction de la lumière ne changent pas, la lumière gérant elle même les appels au contrôle d'accès pour autoriser ou non son changement d'état.

À titre d'exemple, nous allons détailler la séquence de désactivation de la conduite autonome. La désactivation de la conduite autonome commence à la ligne 61 et la condition suivante : `mainboard_L_pressed && mainboard_R_pressed && mainboard_auto_drive_status && !mainboard_light_status && act_state`. Ensuite, si le contrôle d'accès est activé, et si `light_off == 1` alors un message demandant la désactivation de la conduite autonome est envoyé à la conduite autonome, puis au contrôle d'accès. La présence de la variable `light_off` peut paraître surprenante étant donné que la carte mère a déjà une variable `mainboard_light_status`, mais est nécessaire pour des raisons de synchronisation. Lorsque la carte mère envoie le message d'extinction à la lumière, la carte mère et la lumière ne sont plus synchronisées : selon la carte mère la lumière est éteinte, mais la lumière doit vérifier que cette opération est autorisée par le contrôle d'accès, et reste allumée pendant ce temps.

Sans la confirmation d'extinction de la lumière envoyée par le contrôle d'accès, on pourrait se trouver dans le cas décrit par la Figure 4.26. La carte mère demande l'extinction, et demande la désactivation de la conduite autonome, alors que la lumière n'a pas encore reçu l'autorisation d'extinction du contrôle d'accès. Cette situation viole donc la safety, car la conduite autonome est désactivée alors que la lumière est allumée.

Ainsi, nous ajoutons une confirmation d'extinction, produisant le diagramme de séquence de la Figure 4.27. Ici, avant d'envoyer le message de désactivation de la conduite autonome, la carte mère attend une confirmation d'extinction, passant la variable `light_off` à 1, permettant de préserver la safety du système.

Avec la spécification de la politique de contrôle d'accès complète, et les modifications aux spécifications de la lumière et de la carte mère, nous pouvons maintenant passer à une première vérification du système avec la présence d'un attaquant.

4.7.3 Vérification avec la politique de contrôle d'accès, en cas d'attaque

La vérification avec le mode *safety* de SPIN indique que le système ne comporte aucune erreur, donc aucun deadlock. Si nous vérifions nos propriétés LTL – ϕ', ψ' et ξ' – on constate que SPIN trouve un cycle d'acceptation (*acceptance cycle*). En vérifiant les

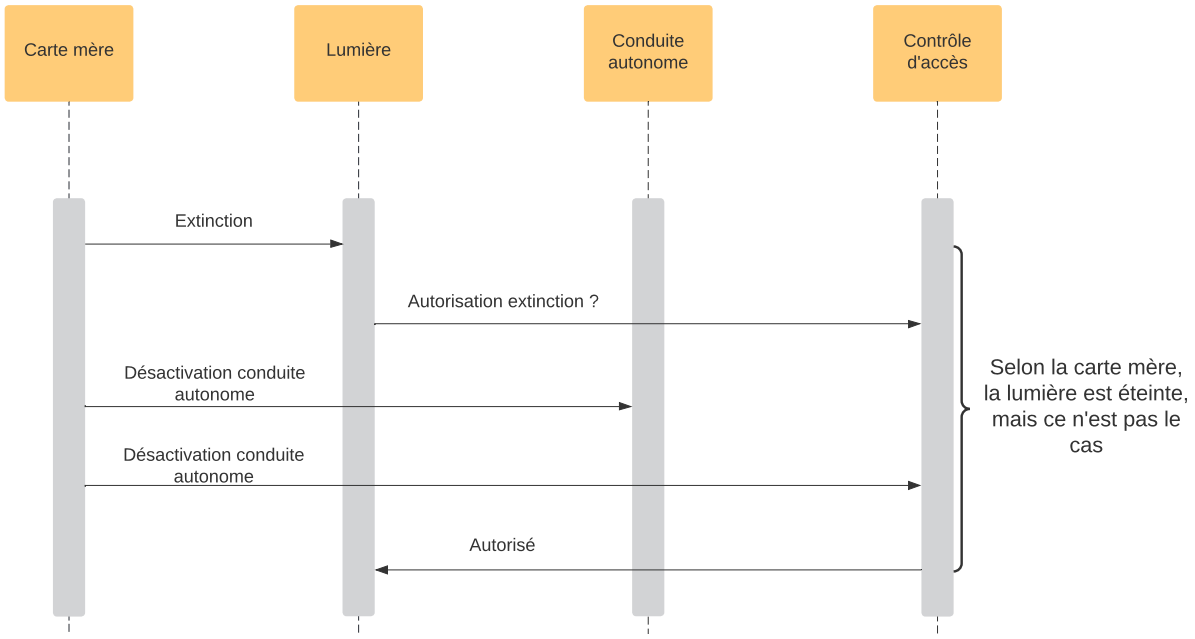


FIGURE 4.26 – Diagramme de séquence sans confirmation d'extinction de la lumière

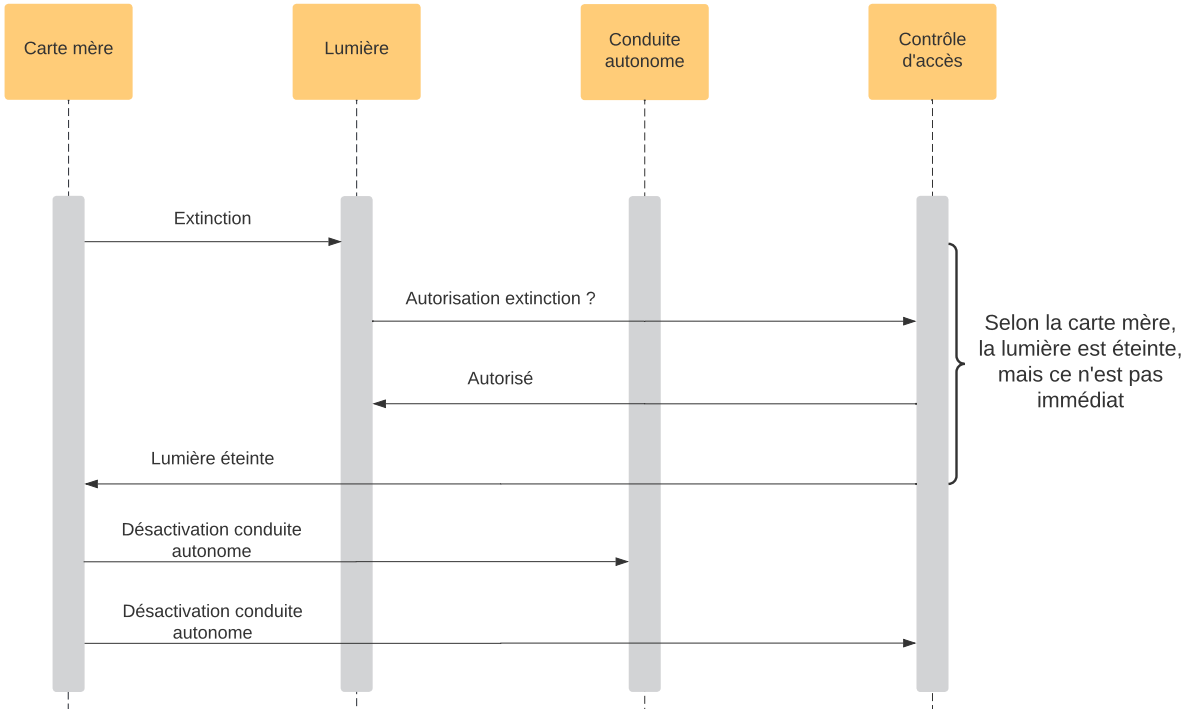


FIGURE 4.27 – Diagramme de séquence avec confirmation d'extinction de la lumière

formules une à une, un cycle est trouvé pour ψ' et ξ', la safety est donc préservée. La trace de la violation est donnée par le Listing 4.24.

```
1  [...]
2  <<<<<START OF CYCLE>>>>>
3  283:  proc  - (manuscrit:1) _spin_nvr.tmp:132 (state 13)  [(((!(((!(mainboard_L_pressed)||!(
       mainboard_R_pressed)))&&!(light_status)))]
4  284:  proc  3 (light_AC:1) isw.pml:105 (state 8)  [sent.dst = light_id]
5  285:  proc  - (manuscrit:1) _spin_nvr.tmp:132 (state 13)  [(((!(((!(mainboard_L_pressed)||!(
       mainboard_R_pressed)))&&!(light_status)))]
6  286:  proc  3 (light_AC:1) isw.pml:105 (state 9)  [sent.cmd = 0]
7  287:  proc  - (manuscrit:1) _spin_nvr.tmp:132 (state 13)  [(((!(((!(mainboard_L_pressed)||!(
       mainboard_R_pressed)))&&!(light_status)))]
8  288:  proc  3 (light_AC:1) isw.pml:105 (state 10) [light_to_light_AC!sent.src,sent.dst,sent.cmd,
       sent.button_pressed]
9  289:  proc  4 (light:1) utility.pml:23 (state 9)  [light_to_light_AC?received.src,received.dst,
       received.cmd,received.button_pressed]
10 289:  proc  4 (light:1) utility.pml:23 (state 10) [light_status = received.cmd]
11 290:  proc  - (manuscrit:1) _spin_nvr.tmp:132 (state 13)  [(((!(((!(mainboard_L_pressed)||!(
       mainboard_R_pressed)))&&!(light_status)))]
12 291:  proc  7 (attacker:1) isw.pml:348 (state 6)  [mainboard_to_light!sent.src,sent.dst,sent.cmd,
       sent.button_pressed]
13 292:  proc  4 (light:1) isw.pml:162 (state 7) [mainboard_to_light?received.src,received.dst,
       received.cmd,received.button_pressed]
14 293:  proc  - (manuscrit:1) _spin_nvr.tmp:132 (state 13)  [(((!(((!(mainboard_L_pressed)||!(
       mainboard_R_pressed)))&&!(light_status)))]
15 294:  proc  4 (light:1) utility.pml:22 (state 8)  [light_to_light_AC!received.src,received.dst,
       received.cmd,received.button_pressed]
16 295:  proc  3 (light_AC:1) isw.pml:105 (state 7)  [light_to_light_AC?received.src,received.dst,
       received.cmd,received.button_pressed]
17 spin: trail ends after 295 steps
```

Listing 4.24 – Exemple de trace menant à une violation lorsque le système est attaqué, avec le contrôle d'accès

On constate que SPIN trouve un cycle (ligne 2), causé par les envois infinis de messages de l'attaquant (ligne 12), causant un déni de service sur le système (qui passe son temps à recevoir ces messages et à les contrôler). De fait, la disponibilité de certaines fonctions n'est plus assurée – ψ' – et certains comportements fonctionnels ne sont pas remplis – ξ'. Ces violations auraient dû apparaître lors de la vérification précédente, mais étant donné que l'attaquant allumait la lumière, ces deux propriétés étaient vérifiées. Notre mécanisme de contrôle d'accès n'a pas vocation à nous protéger de ce type d'attaque. Nous n'empêchons pas l'émission des messages, mais leur action à leur réception (*e.g.* la lumière reçoit le message d'allumage de l'attaquant, mais ne s'allume pas).

En permettant à l'attaquant d'envoyer un seul message, SPIN trouve toujours un cycle d'acceptation, mais cette fois sans lien avec l'attaquant. La trace est donnée en Listing 4.25.

```
1  [...]
```

```
<<<<<START OF CYCLE>>>>>
478:  proc  - (manuscrit:1) _spin_nvr.tmp:132 (state 13)  [(((!((!(mainboard_L_pressed)||!(
    mainboard_R_pressed)))&&!(light_status)))]
Never claim moves to line 132 [(((!((!(mainboard_L_pressed)||!(mainboard_R_pressed)))&&!(
    light_status)))]
479:  proc  5 (mainboard:1) isw.pml:446 (state 48)  [((((((mainboard_L_pressed&&
    mainboard_R_pressed)&&mainboard_auto_drive_status)&&!(mainboard_light_status))&&act_state))]
480:  proc  - (manuscrit:1) _spin_nvr.tmp:132 (state 13)  [(((!((!(mainboard_L_pressed)||!(
    mainboard_R_pressed)))&&!(light_status)))]
481:  proc  5 (mainboard:1) isw.pml:451 (state 71)  [else]
482:  proc  - (manuscrit:1) _spin_nvr.tmp:132 (state 13)  [(((!((!(mainboard_L_pressed)||!(
    mainboard_R_pressed)))&&!(light_status)))]
483:  proc  5 (mainboard:1) isw.pml:451 (state 72)  [(1)]
spin: trail ends after 483 steps
```

Listing 4.25 – Exemple de trace menant à une violation lorsque le système est attaqué une fois, avec le contrôle d'accès

Ici, on constate que c'est la carte mère qui boucle indéfiniment sur la condition : `(((((mainboard_L_pressed && mainboard_R_pressed) && mainboard_auto_drive_status) && !(mainboard_light_status)) && act_state))` (ligne 61 du Listing 4.23 : "Code du proctype de la carte mère pour la mise en place du contrôle d'accès"). Ce comportement est dû à une spécificité de SPIN [(c)] : lorsque plusieurs options d'une boucle sont exécutables, SPIN sélectionne de manière non déterministe une de ces options. Ainsi, il existe des exécutions où la carte mère choisit indéfiniment l'option `(((((mainboard_L_pressed && mainboard_R_pressed) && mainboard_auto_drive_status) && !(mainboard_light_status)) && act_state))` ce qui cause la détection d'un cycle d'acceptation dans les formules LTL. Cette fois, seule la propriété ψ' est violée, ϕ' et ξ' sont vérifiées, la safety est donc toujours préservée.

Si nous modifions cette condition, pour qu'elle ne soit pas exécutable sans avoir reçu la confirmation d'allumage de la lumière (*cf.* Listing 4.26), forçant ainsi l'exécution du contrôle d'accès pour avoir la confirmation d'extinction de la lumière, le système est à nouveau correct par rapport à toutes nos propriétés.

```
do
[...]
::  mainboard_L_pressed && mainboard_R_pressed &&  mainboard_auto_drive_status && !
    mainboard_light_status && act_state && light_off == 1->
        #if AC == 1
        send_to(AC_light_group, 0, mainboard_id) -> mainboard_auto_drive_status = 0; /* on dé
    sactive la conduite autonome */
        #else
        sent.dst = auto_drive_id;
        mainboard_to_auto_drive!sent -> mainboard_auto_drive_status = 0; /* on désactive la
    conduite autonome */
        #endif
```

(c). https ://spinroot.com/spin/Man/do.html

```
10 [...]
```

Listing 4.26 – Modification du proctype de la carte mère pour empêcher le cycle d'acceptation

Ainsi, bien que SPIN détecte certains cycles d'acceptation (qui, contrairement à leur nom, ne sont pas une bonne chose), nous voyons qu'en les corrigeant, le système est totalement protégé. Dans toutes les situations, la **safety** du système est toujours préservée – ϕ' est toujours vérifiée. Les propriétés de disponibilité – ψ' – et d'intégrité fonctionnelle – ξ' – sont en revanche violées en cas de déni de service, causé par un attaquant envoyant un nombre illimité de messages. SPIN nous a aussi permis de détecter que la carte mère pouvait mener à un cycle violant une propriété de disponibilité. La **safety** du système n'a été violée à aucun moment. Ainsi, notre politique de contrôle d'accès a bien l'effet attendu sur le système et permet de nous protéger de notre attaquant, dont les capacités ont été formalisées en LTL. Nous allons maintenant voir quels sont les impacts de la politique sur le système, lorsqu'il n'est pas attaqué.

4.7.4 Vérification avec la politique de contrôle d'accès, sans attaque

En préservant les modifications faites à la carte mère dans la section précédente (modification de la condition ligne 3 du Listing 4.26), SPIN ne trouve aucune erreur en mode *safety*. Ainsi, la politique n'introduit pas de deadlock dans le système. La vérification de nos formules, ϕ', ψ' et ξ' ne produit là encore aucune erreur. La politique de contrôle d'accès ne perturbe pas le fonctionnement du système d'une manière qui est observable avec nos propriétés.

Ainsi, notre politique de contrôle d'accès ne perturbe pas le comportement du système. Nous préservons la **safety**, ainsi que des propriétés de disponibilité et d'intégrité fonctionnelle.

4.8 Conclusion sur la modélisation avec SPIN

Contrairement à l'outil du Dr. Hugot, SPIN est beaucoup plus difficile à prendre en main étant donné qu'il propose plus d'abstractions. Bien que fournissant un exemple de trace menant à une erreur, la correction de ces dernières reste complexe. En utilisant SPIN, nous avons à nouveau modélisé notre exemple de l'ISW, ainsi que des propriétés plus adaptées que celles que nous pouvions exprimer avec l'outil du Dr. Hugot.

En effectuant une première vérification des propriétés, nous avons constaté que notre modélisation vérifiait toutes les propriétés énoncées lorsqu'il n'y a pas d'attaque, ou de contrôle d'accès.

Puis nous avons effectué une vérification de ces mêmes propriétés, mais cette fois en ajoutant notre attaquant à la modélisation, engendrant une violation de la propriété ϕ' : le système perd sa propriété de **safety**.

Afin de préserver la safety du système, nous avons mis en place notre politique de contrôle d'accès après avoir modifié la spécification de la carte mère et de la lumière. Lors de la vérification en présence de l'attaquant, nous avons constaté une violation de ψ' et ξ'. La **safety** du système est à nouveau préservée, mais au détriment de propriétés de disponibilité et d'intégrité fonctionnelle. SPIN détecte des cycles, correspondant à des dénis de service causés par l'attaquant. En réalité, ces dénis de service étaient déjà existants lors de la vérification sans contrôle d'accès. Cependant, telles que nous les avons spécifiées, les formules ψ' et ξ' ne sont pas violées dès lors que la lumière est allumée, et n'étaient donc pas violées lors de la vérification sans contrôle d'accès.

Enfin, nous avons terminé sur la vérification du système uniquement avec la politique de contrôle d'accès. Toutes les propriétés du système sont préservées.

4.9 Conclusion de la modélisation

Notre approche visant à **préserver la safety** d'un système automobile en cas d'attaque a été validée par deux outils différents. Dans les deux cas, nous avons montré que notre système de base possédait un certain nombre de propriétés dans une situation nominale.

Puis, nous avons perturbé le fonctionnement en introduisant un attaquant, dont le but était de violer une propriété de safety, nous permettant d'observer si notre politique de contrôle d'accès avait l'effet escompté.

L'ajout de notre politique de contrôle d'accès a permis, dans les deux cas, de préserver la safety du système en cas d'attaque.

Enfin, nous avons montré que la politique de contrôle d'accès, ne perturbe pas le fonctionnement du système, qui préserve toutes ses propriétés lorsqu'il n'est pas attaqué.

La vérification formelle est un outil puissant afin de vérifier des propriétés sur des systèmes. Néanmoins, nous avons vu qu'il est facile de tomber dans certains pièges. Par exemple certaines propriétés peuvent être vérifiées, mais une partie du système peut être totalement bloquée, sans être détectée, si les propriétés à vérifier ne concerne pas la partie en question.

Ainsi, c'est une pratique qui nécessite un investissement en termes de vérification pour en tirer des avantages, tels que prouver la préservation de la **safety** d'un système automobile.

Chapitre 5

Expérimentations supplémentaires

Dans le chapitre précédent, nous avons décrit la modélisation et la vérification de notre système. Ces étapes sont nécessaires pour garantir que l'intégration du mécanisme de contrôle d'accès dans le système ne viole pas de propriétés safety, d'intégrité, ou de disponibilité. Pour cela, nous avons présenté notre modélisation avec deux outils. Le premier, développé par le Dr. Hugot, utilise la logique CTL et propose une représentation graphique du système généré. Le deuxième outil que nous avons utilisé est SPIN/PROMELA, un model checker utilisé dans l'industrie, offrant un autre formalisme et permettant la vérification de formules LTL plus appropriées à certains aspects de notre cas d'usage réel.

Notre approche permet de protéger avec succès le système présenté d'une attaque. Néanmoins, l'intégration du mécanisme de contrôle d'accès apporte aussi son lot de contraintes, que nous allons pour certaines essayer d'adresser dans ce chapitre.

Pour cela, nous commençons par rappeler notre cas d'usage réel de l'ISW ainsi que le profil de l'attaquant que nous souhaitons bloquer. Puis nous abordons les problèmes de synchronisation qui peuvent survenir avec l'intégration de notre mécanisme de contrôle d'accès, et comment les résoudre. Enfin, nous terminons en abordant la tolérance aux fautes de notre mécanisme.

5.1 Rappel du cas d'usage

5.1.1 Volant intelligent

Pour notre cas d'usage, nous reprenons l'ISW déjà présenté dans la Section 3.2 : "Présentation du cas d'usage réel". L'architecture physique de ce composant automobile est rappelée par la Figure 3.20 : "Architecture physique de l'ISW".

Pour rappel, le fonctionnement de l'ISW est comme suit : dans l'état initial, la conduite autonome est désactivée, la lumière d'indication est éteinte et les deux boutons sont

relâchés. Lorsque les deux boutons sont pressés en même temps, la carte mère envoie un message pour activer la conduite autonome. Lorsque la conduite autonome est activée, la carte mère envoie un message pour allumer la lumière. Lorsque les deux boutons sont à nouveau pressés, la carte mère éteint d'abord la lumière, puis désactive la conduite autonome. L'ECU en charge de la conduite autonome est en dehors de notre champ d'action.

Pour tester notre approche, nous modélisons aussi un attaquant, dont nous rappelons le profil ci après.

5.1.2 Profil de l'attaquant

L'objectif de notre attaquant est le suivant : allumer la lumière d'indication de la conduite autonome à tout moment. Les succès de l'attaquant peuvent mener à deux situations :

(1) la lumière s'allume alors que la conduite autonome est désactivée (*i.e.* menant à une situation anormale, en dehors du comportement nominal du système)

(2) la lumière s'allume alors que la conduite autonome est activée (*i.e.* menant à une situation normale, dans le comportement nominal du système, mais pouvant éviter certaines transitions)

Dans le premier cas, le conducteur pense qu'il peut relâcher son attention car la conduite autonome semble activée alors que non, créant une situation où la safety n'est pas préservée. Dans le second cas, la lumière est allumée et la conduite autonome est activée. Néanmoins, cette situation peut créer des désynchronisations et de potentiels dénis de service comme nous allons le voir.

Maintenant que nous avons rappelé les spécificités de notre cas d'usage réel et de notre attaquant, nous allons voir dans la section suivante comment se comporte notre système avec notre mécanisme de contrôle d'accès obligatoire dynamique.

5.2 Synchronisation

Reprenons la vérification faite avec l'outil du Dr. Hugot avec un attaquant et notre politique de contrôle d'accès.

La vérification des propriétés ϕ, ψ et ξ a produit la Figure 4.21 : "Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ en cas d'attaque, avec une politique de contrôle d'accès". Sur cette figure, tous les états sont bleus, indiquant que tous les états vérifient toutes les propriétés données.

Or, si notre attaquant envoie son message d'allumage au bon moment, cet envoi est synchronisé avec la transition de l'automate de contrôle d'accès, et l'automate de la lumière. Ainsi, la lumière s'allume. Or, la carte mère va aussi devoir envoyer ce même message pour poursuivre son exécution, ce qui ne va pas être possible car la lumière et

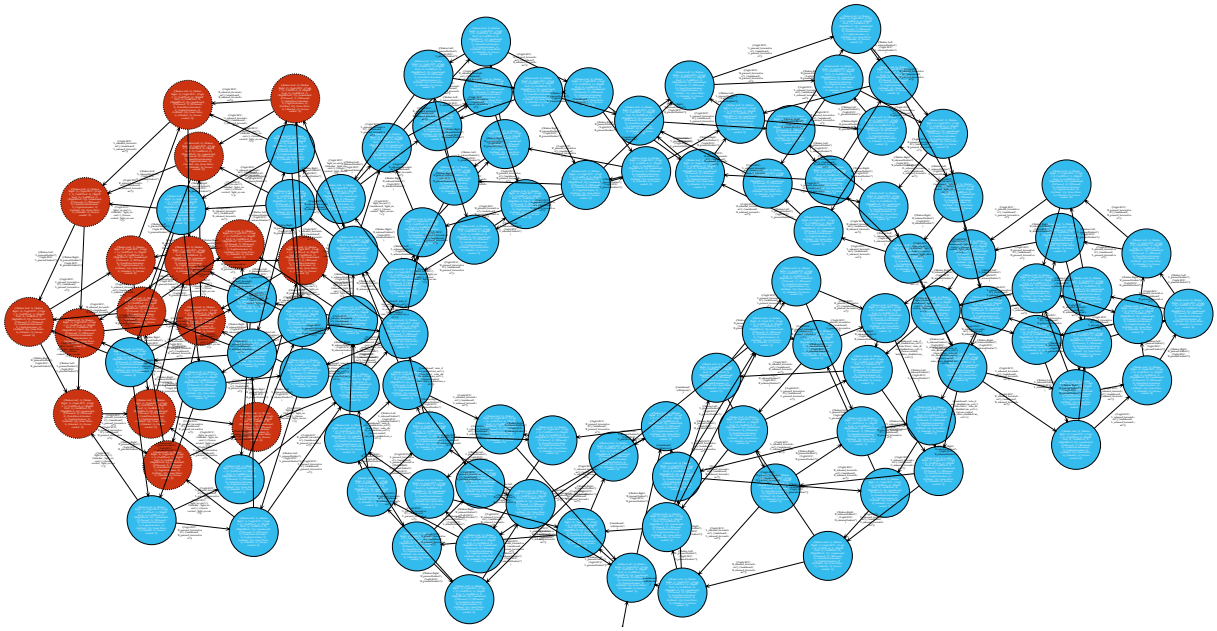


FIGURE 5.1 – Automate de vérification de l'ISW pour la propriété 5.1

le moniteur qui ont déjà effectué cette transition. On se trouve dans un cas où il y a une désynchronisation entre la carte mère, la lumière et le moniteur.

Ainsi, si nous effectuons une vérification de cette modélisation pour la propriété suivante (identique à la conjonction des propriétés 4.3 et 4.4) :

Propriété fonctionnelle

$$(AutoDriveActivated \ \wedge \ \neg ActStateON) \Rightarrow \exists\Diamond(LightActivated)) \wedge$$
$$(\neg LightActivated \ \wedge \ ActStateON) \Rightarrow \exists\Diamond(\neg AutoDriveActivated)) \tag{5.1}$$

nous obtenons la Figure 5.1.

La propriété 5.1 s'intéresse à l'état de la conduite autonome (`AutoDriveActivated`) et de la lumière (`LightActivated`) connu par la carte mère. Ainsi, on constate que le système contient des erreurs pour cette propriété, alors que toutes les propriétés que nous avions définies précédemment étaient vérifiées. Il s'agit en réalité d'un problème de synchronisation, qui peut s'expliquer avec le diagramme de séquence de la Figure 5.2 : "Diagramme de séquence en cas de désynchronisation". Sur ce diagramme, on constate que le message d'activation de la conduite autonome (`auto_drive_enabled.can_ext!+`) est envoyé par la carte mère et reçu par la conduite autonome et le contrôle d'accès. Ces trois composants se synchronisent sur ce message. D'après l'automate de notre moniteur de contrôle d'accès (*cf.* Figure 4.18), la prochaine synchronisation se fera sur le message `light_on.can`. Dans notre exemple, ce message est envoyé par l'attaquant et se synchronise avec le contrôle d'accès et la lumière. Or, lorsque la carte mère va envoyer ce même message, elle ne va pas être synchronisée avec le contrôle d'accès ou la lumière et va donc être bloquée. Elle ne pourra donc jamais envoyer le message d'allumage, puis celui d'extinction et enfin celui de désactivation de la conduite autonome.

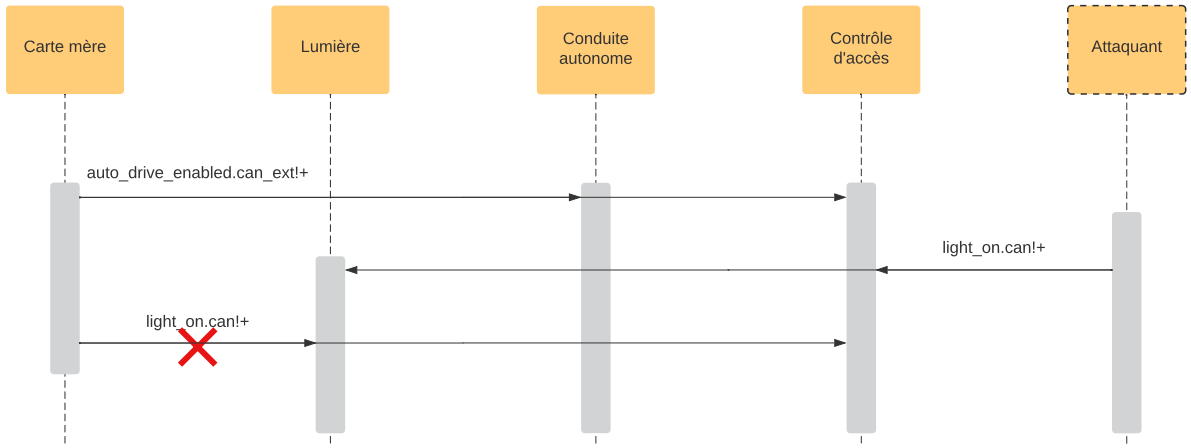


FIGURE 5.2 – Diagramme de séquence en cas de désynchronisation [(a)]

Ainsi, bien que le système préserve sa safety d'après notre propriété, la lumière ne pourra jamais être éteinte, et donc la conduite autonome ne pourra jamais être désactivée.

Notons que cette désynchronisation n'est pas exclusive à la modélisation intégrant notre mécanisme de contrôle d'accès. En effet, si nous vérifions la même propriété sur un système attaqué et non protégé par notre mécanisme de contrôle d'accès, nous obtenons la Figure 5.3 : "Automate de vérification du système en cas de désynchronisation, sans contrôle d'accès, pour la propriété 5.1", sur laquelle se trouvent des états rouges. La désynchronisation a toujours lieu car la lumière n'accepte qu'un seul message d'allumage.

Pour éviter cette désynchronisation, deux solutions sont possibles :

(1) authentifier l'émetteur du message ;

(2) autoriser la réception multiple de certains messages.

Avec la solution 1, les automates emprunteraient la transition que si l'authentification est réussie. Or cela implique d'avoir une connaissance encore plus fine du système (en plus de savoir quels messages doivent être envoyés/reçus, il faudrait savoir quelle application doit envoyer chaque message), et introduit des contraintes supplémentaires (latences). Ainsi, nous choisissons de ne pas aborder les mécanismes d'authentification.

De fait, pour resynchroniser la carte mère avec la lumière et le moniteur de contrôle d'accès, nous choisissons la solution 2. Cette approche nécessite de modifier le moniteur de contrôle d'accès, ainsi qu'une partie de la spécification de la lumière. Pour les deux spécifications, il faut autoriser la réception de plusieurs messages d'allumage de la lumière, produisant ainsi l'automate de contrôle d'accès de la Figure 5.4. Cette modification permet au moniteur de recevoir un ou plusieurs messages d'allumage de la lumière.

(a). Deux flèches sur la même ligne indiquent un envoie synchronisé d'un émetteur vers plusieurs destinataires

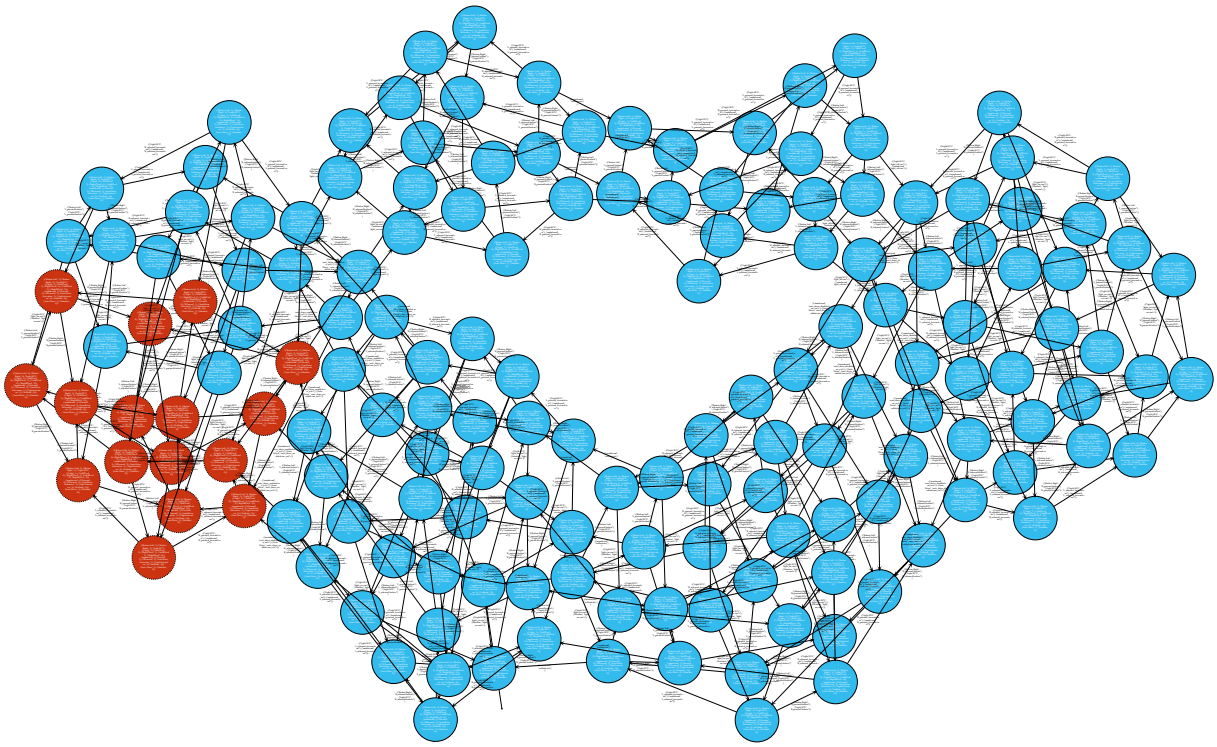


FIGURE 5.3 – Automate de vérification du système en cas de désynchronisation, sans contrôle d'accès, pour la propriété 5.1

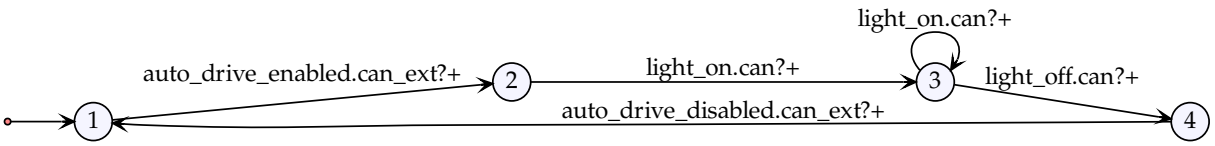


FIGURE 5.4 – Automate de spécification de la politique de contrôle d'accès acceptant plusieurs fois le message d'allumage de la lumière

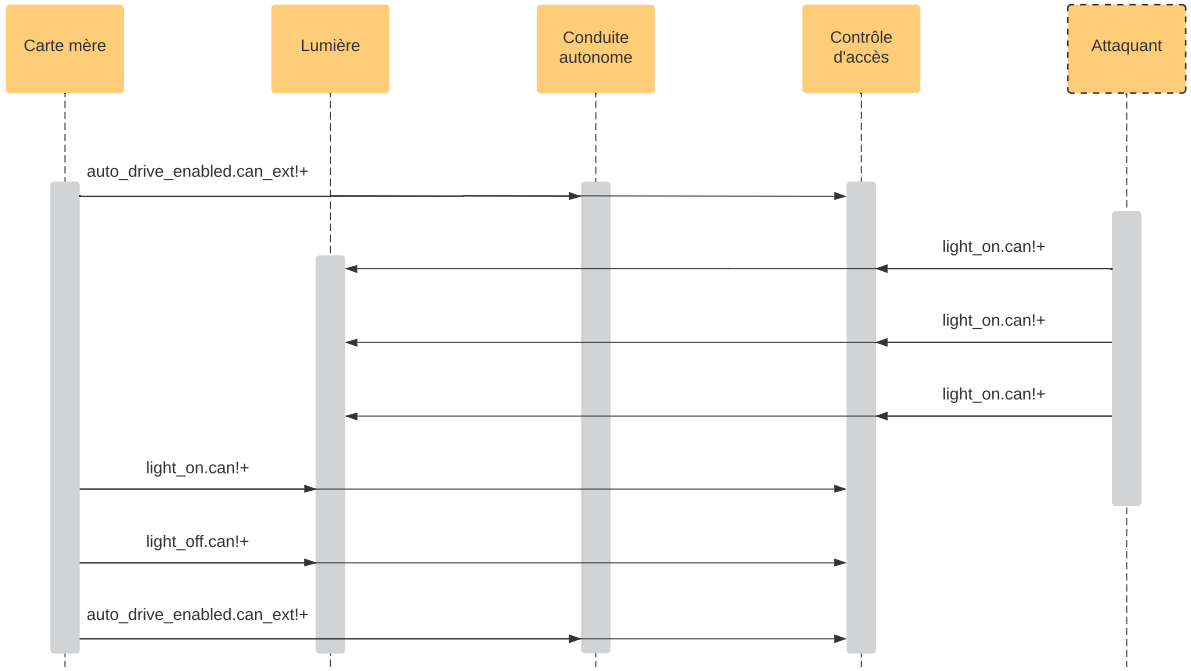


FIGURE 5.5 – Diagramme de séquence en cas de resynchronisation [(b)]

On constate que l'automate de spécification de la Figure 5.4 est similaire à celui de la Figure 4.18 : "Automate de spécification de la politique de contrôle d'accès", à la différence que celui de la Figure 5.4 possède une transition supplémentaire sur le 3^{e} état. Cette transition autorise la réception de plusieurs messages d'allumage après en avoir reçu un premier pour changer d'état. Ainsi, si l'attaquant envoie le message d'allumage avant la carte mère, la carte mère pourra toujours se resynchroniser grâce à cette transition, comme le montre le diagramme de séquence de la Figure 5.5 : "Diagramme de séquence en cas de resynchronisation".

La spécification de la lumière est modifiée d'une manière similaire (un cycle est ajouté sur les états où la lumière est déjà allumée, permettant ainsi de recevoir à nouveau le même message). Ces modifications permettent à la carte mère de se resynchroniser avec le reste du système, même en présence de l'attaquant, sans avoir besoin d'identifier quel composant a envoyé le message d'allumage de la lumière. Ainsi, si nous vérifions à nouveau le système avec ces modifications, nous obtenons l'automate de la Figure 5.6, tous les états sont bleus, indiquant que le système est cette fois-ci correct.

La vérification de ce système par rapport à ϕ, ψ et ξ produit aussi un automate où tous les états sont bleus. Notre approche permet donc de préserver la safety du système, en appliquant des restrictions sur l'envoi et la réception de certains messages, en fonction des précédents messages observés. Ainsi, un attaquant est contraint par cette séquence de message et notre mécanisme peut être facilement adapté pour gérer les

(b). Deux flèches sur la même ligne indiquent un envoie synchronisé d'un émetteur vers plusieurs destinataires

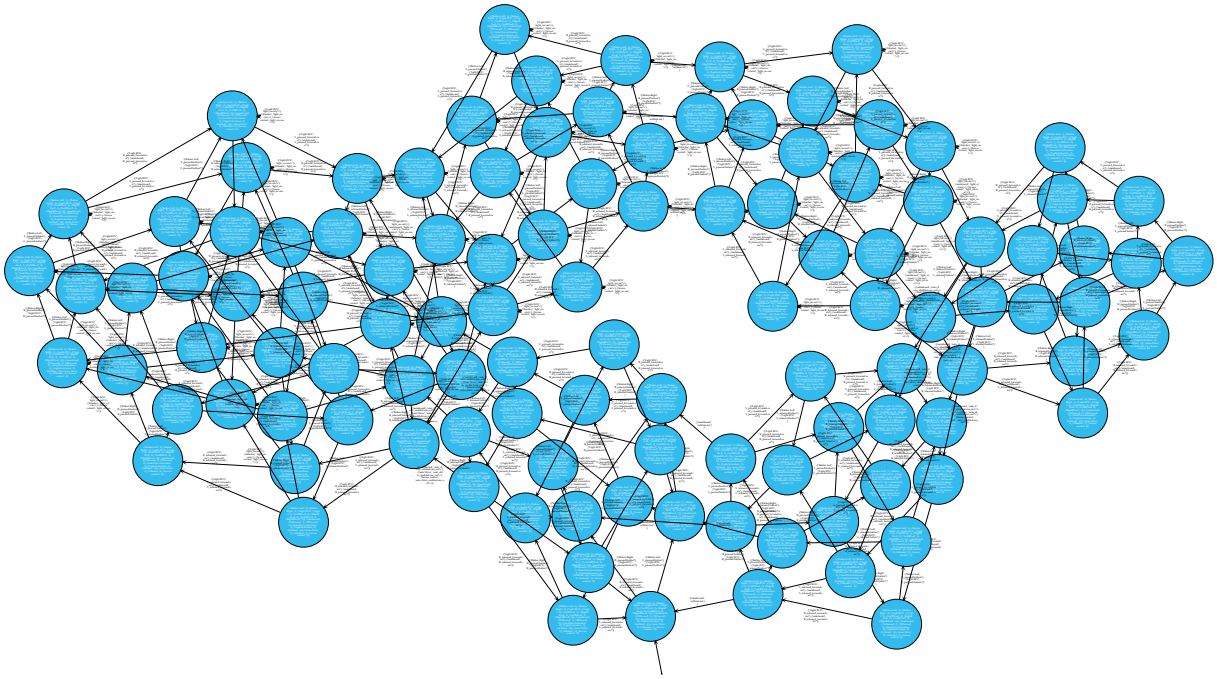


FIGURE 5.6 – Automate de vérification de l'ISW pour la propriété 5.1 avec resynchronisation

désynchronisations qui peuvent survenir si l'attaquant respecte cette séquence. Sans authentifier l'émetteur d'un message, nous sommes capables de protéger le système. Cependant, le système que nous avons utilisé jusqu'ici représente un scénario idyllique, où les échanges sont instantanés, sans pertes et donc non représentatifs d'un système réel. C'est pourquoi la section suivante s'intéresse à la tolérance aux pertes de messages.

5.3 Pertes de messages

Les vérifications effectuées dans le Chapitre 4 : "Modélisation formelle de notre approche" se sont déroulées dans des scénarios idéaux : sans pertes. Or, dans un système réel les pertes existent. De fait, notre mécanisme de contrôle d'accès doit les tolérer.

Pour illustrer les points de cette section, nous utilisons toujours l'exemple de l'ISW, mais en considérant que le message d'extinction de la lumière peut être perdu. Nous utilisons l'outil du Dr. Hugot avec les opérateurs asynchrones non bloquants présentés en Section 3.1.3 : "Ajouts des nouveaux opérateurs".

Comme le message d'allumage, le message d'extinction de la lumière d'indication de la conduite autonome présente un intérêt safety. En effet, s'il n'est pas reçu par la lumière, elle restera allumée alors que la carte mère désactivera la conduite autonome.

Pour illustrer ce point, nous reprenons la modélisation de l'ISW mais en autorisant les pertes lors de l'envoi du message d'extinction (*i.e.* nous remplaçons les transitions

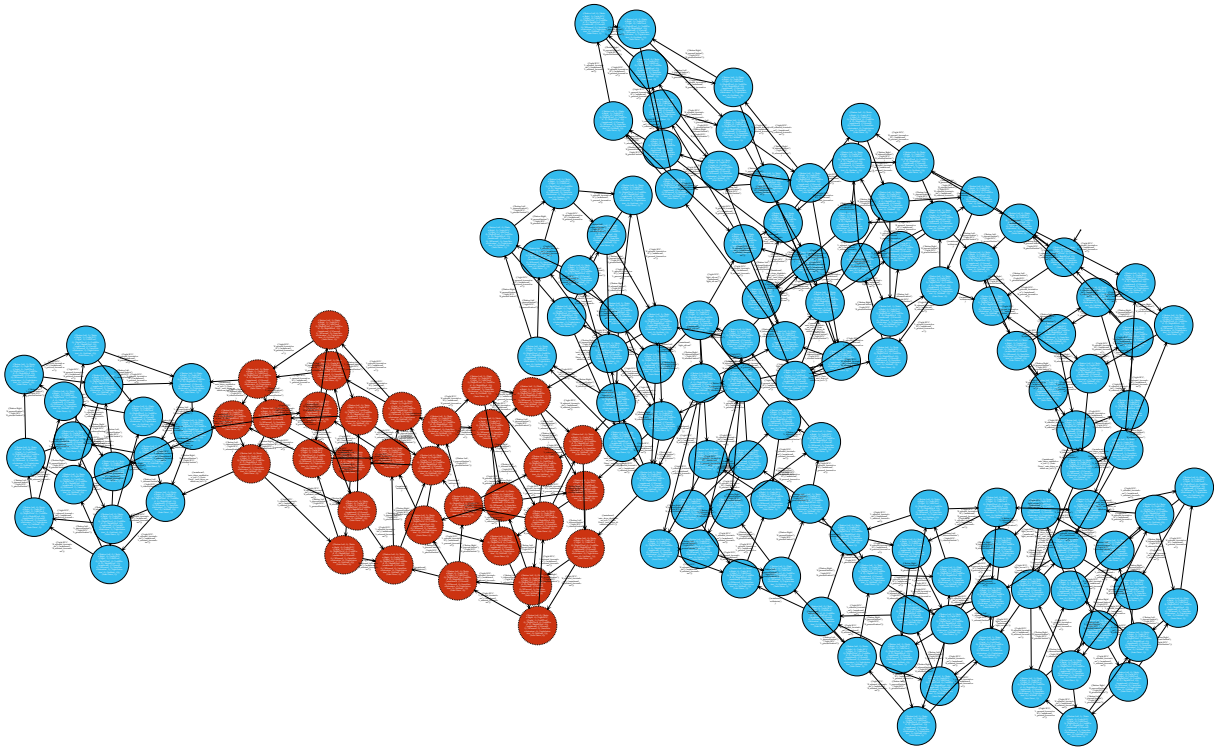


FIGURE 5.7 – Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ en cas de perte du message d'extinction de la lumière

`light_off.can!+` et `light_off.can?+` par `light_off.can!*` et `light_off.can?*` respectivement).

Ainsi, sans attaquant ou contrôle d'accès, nous obtenons l'automate de vérification de la Figure 5.7 : "Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ en cas de perte du message d'extinction de la lumière".

On constate que toutes les propriétés ne sont pas vérifiées. Les états rouges sont produits par la violation de ϕ, la safety n'est donc pas préservée. La situation est identique avec notre politique de contrôle d'accès, qui produit en plus une violation de ψ, comme illustré par la Figure 5.8.

Pour préserver la safety même si un message est perdu, nous devons mettre en place un mécanisme de rejeu, permettant de resynchroniser les différents automates, qu'ils aient reçu ou non les précédents messages d'extinction. Pour cela, la carte mère doit envoyer 2 messages d'extinction :

— le premier avec les opérateurs asynchrones non bloquants pour simuler des pertes ;

— le second avec les opérateurs synchrones bloquants pour simuler un retour à la normale du système (sans pertes).

La spécification de la carte mère avec ce second envoi est donnée dans le Listing 5.1. Nous ajoutons un tuple supplémentaire `("Replay", 0)` permettant d'indiquer si le message

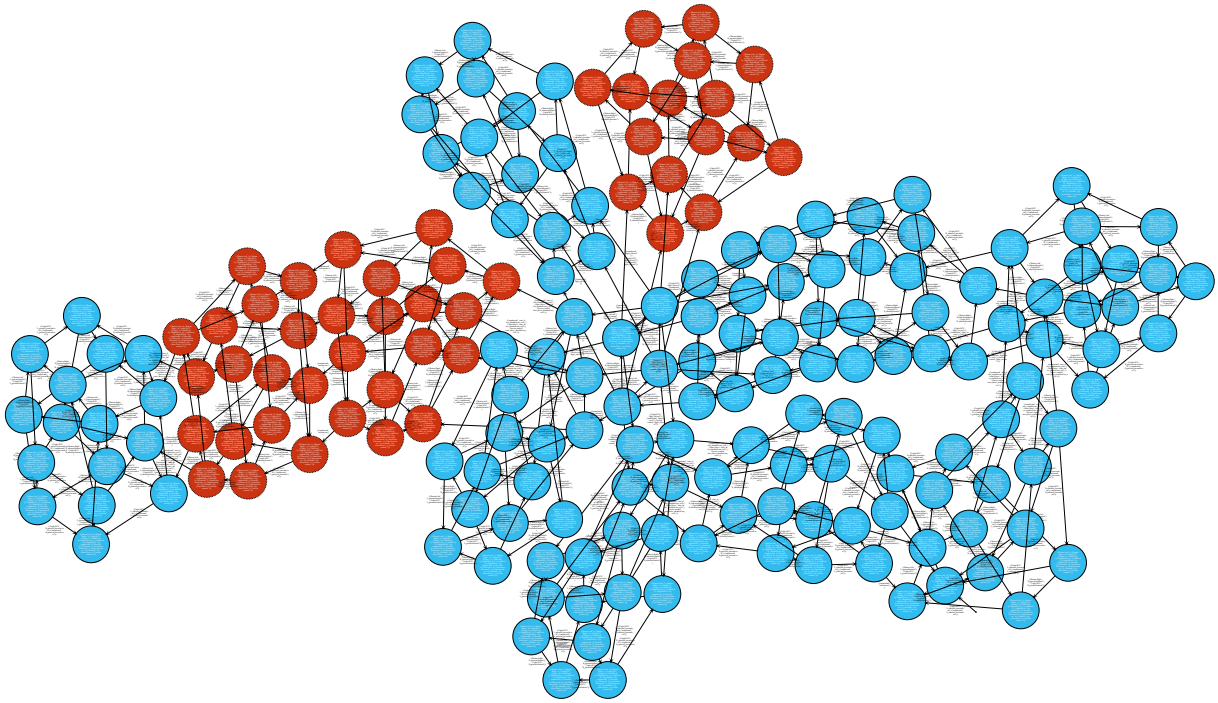


FIGURE 5.8 – Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ en cas de perte du message d'extinction de la lumière, avec la politique

exploitant les opérateurs synchrones a été envoyé. Une condition supplémentaire est aussi ajoutée (lignes 45 à 50) pour l'envoi de ce message.

```
1  mainboard = NFA(
2      {(("LPressed", 0), ("RPressed", 0), ("AutoDriveActivation", 0), ("LightActivation", 0), ("
       ActState", 0), ("Replay", 0))},
3      set(),
4      set(),
5      name="mainboard",
6      worder=tuple
7  )
8
9  def grow(A):
10     has = False
11
12     def extend(q):
13         nonlocal has
14         qdict = dict(q)
15         if qdict["LPressed"] == 0:
16             newd = qdict.copy()
17             newd["LPressed"] = 1
18             has = A.try_rule(q, "L_pressed_forward.can?", tuple(newd.items())) or has
19         if qdict["LPressed"] == 1:
20             newd = qdict.copy()
21             newd["LPressed"] = 0
22             has = A.try_rule(q, "L_released_forward.can?", tuple(newd.items())) or has
23         if qdict["RPressed"] == 0:
```

```
            newd = qdict.copy()
            newd["RPressed"] = 1
            has = A.try_rule(q, "R_pressed_forward.can?", tuple(newd.items())) or has
        if qdict["RPressed"] == 1:
            newd = qdict.copy()
            newd["RPressed"] = 0
            has = A.try_rule(q, "R_released_forward.can?", tuple(newd.items())) or has

        if qdict["LightActivation"] == 0:
            newd = qdict.copy()
            newd["LightActivation"] = 1
            if qdict["LPressed"] == 1 and qdict["RPressed"] == 1 and qdict["AutoDriveActivation"]
     == 1 and qdict[
                "ActState"] == 0:
                has = A.try_rule(q, "light_on.can!+", tuple(newd.items())) or has
        if qdict["LightActivation"] == 1:
            newd = qdict.copy()
            newd["LightActivation"] = 0
            if qdict["LPressed"] == 1 and qdict["RPressed"] == 1 and qdict["AutoDriveActivation"]
     == 1 and qdict[
                "ActState"] == 1:
                has = A.try_rule(q, "light_off.can!*", tuple(newd.items())) or has

        if qdict["Replay"] == 0:
            if qdict["LPressed"] == 1 and qdict["RPressed"] == 1 and qdict["LightActivation"] ==
    0 and qdict[
                "ActState"] == 1:
                newd = qdict.copy()
                newd["Replay"] = 1
                has = A.try_rule(q, "light_off.can!+", tuple(newd.items())) or has

        if qdict["AutoDriveActivation"] == 0:
            newd = qdict.copy()
            newd["AutoDriveActivation"] = 1
            if qdict["LPressed"] == 1 and qdict["RPressed"] == 1 and qdict["LightActivation"] ==
    0 and qdict[
                "ActState"] == 0:
                has = A.try_rule(q, "auto_drive_enabled.can_ext!+", tuple(newd.items())) or has
        if qdict["AutoDriveActivation"] == 1:
            newd = qdict.copy()
            newd["AutoDriveActivation"] = 0
            if qdict["LPressed"] == 1 and qdict["RPressed"] == 1 and qdict["LightActivation"] ==
    0 and qdict[
                "ActState"] == 1 and qdict["Replay"] == 1:
                has = A.try_rule(q, "auto_drive_disabled.can_ext!+", tuple(newd.items())) or has

        if qdict["ActState"] == 0:
            newd = qdict.copy()
            newd["ActState"] = 1
            if qdict["LPressed"] == 0 and qdict["RPressed"] == 0 and qdict["LightActivation"] ==
    1 and qdict[
```

```
                "AutoDriveActivation"] == 1:
                has = A.try_rule(q, "actState.int;", tuple(newd.items())) or has
        if qdict["ActState"] == 1:
            newd = qdict.copy()
            newd["ActState"] = 0
            newd["Replay"] = 0
            if qdict["LPressed"] == 0 and qdict["RPressed"] == 0 and qdict["LightActivation"] ==
    0 and qdict[
                "AutoDriveActivation"] == 0:
                has = A.try_rule(q, "actState.int;", tuple(newd.items())) or has

    for q in A.Q.copy():
        extend(q)
    return has

mainboard.growtofixpoint(grow, record_steps=True).visu()
```

Listing 5.1 – Code de génération de l'automate de la carte mère avec rejeu du message d'extinction

De fait, chaque composant qui réceptionne le message d'extinction doit être en mesure de se synchroniser sur le message envoyé :

(1) avec les opérateurs asynchrones ;

(2) avec les opérateurs synchrones ;

(3) avec les deux (on ne sait pas quel composant s'est synchronisé avec les opérateurs asynchrones).

La spécification de la politique de contrôle d'accès permettant les rejeux est présentée par le Listing 5.2 produisant la Figure 5.9. On constate que l'automate de la politique de contrôle d'accès peut aller de l'état 3 à l'état 4 en recevant le message `light_off.can?*` ou le message `light_off.can?+` permettant de prendre en compte les situations 1 et 2. De plus, une boucle sur l'état 4 permet de prendre en compte la situation 3 si l'automate s'est déjà synchronisé avec les opérateurs asynchrones et doit maintenant se synchroniser avec les opérateurs synchrones.

```
access_control = NFA.spec("""
    1
    __
    1 auto_drive_enabled.can_ext?+ 2
    2 light_on.can?+ 3
    3 light_off.can?* 4
    3 light_off.can?+ 4
    4 light_off.can?+ 4
    4 auto_drive_disabled.can_ext?+ 1""").named("Access control").visu()
```

Listing 5.2 – Code de génération de l'automate du contrôle d'accès avec gestion du rejeu

L'automate de spécification de la lumière d'indication de la conduite autonome évolue d'une manière similaire à celui de la politique (2 transitions en parallèle pour le message

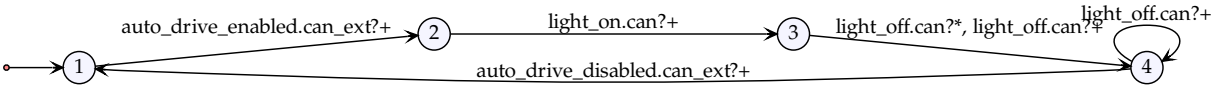


FIGURE 5.9 – Automate de spécification de la politique de contrôle d'accès avec gestion du rejeu

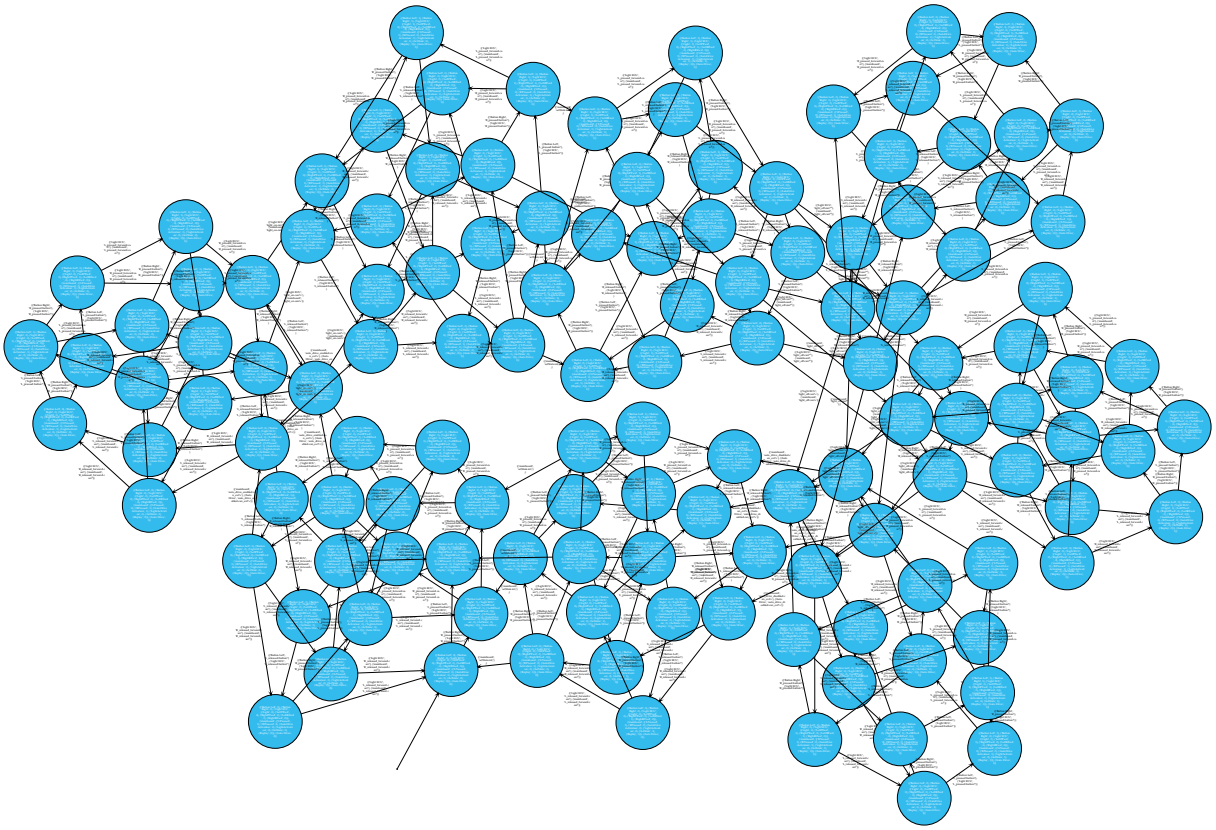


FIGURE 5.10 – Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ en cas de perte du message d'extinction de la lumière avec gestion du rejeu

d'extinction et une boucle sur l'état d'arrivée de ces transitions).

Avec ces modifications, nous pouvons à nouveau effectuer une vérification de l'ISW, qui produit l'automate de vérification de la Figure 5.10 : "Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ en cas de perte du message d'extinction de la lumière avec gestion du rejeu" pour la vérification sans la politique, et la Figure 5.11 : "Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ en cas de perte du message d'extinction de la lumière, avec la politique et gestion du rejeu" lorsque la politique est inclue dans la vérification.

Sur ces deux figures, nous constatons que tous les états sont bleus, indiquant que tous les états vérifient les propriétés ϕ, ψ et ξ, même en cas de perte du message d'extinction. Cependant, ces modélisations décrivent une situation où il n'y a qu'une seule perte (la carte mère n'envoie qu'un seul message avec les opérateurs asynchrones, *cf.* ligne 43 du Listing 5.1 : "Code de génération de l'automate de la carte mère avec rejeu du message

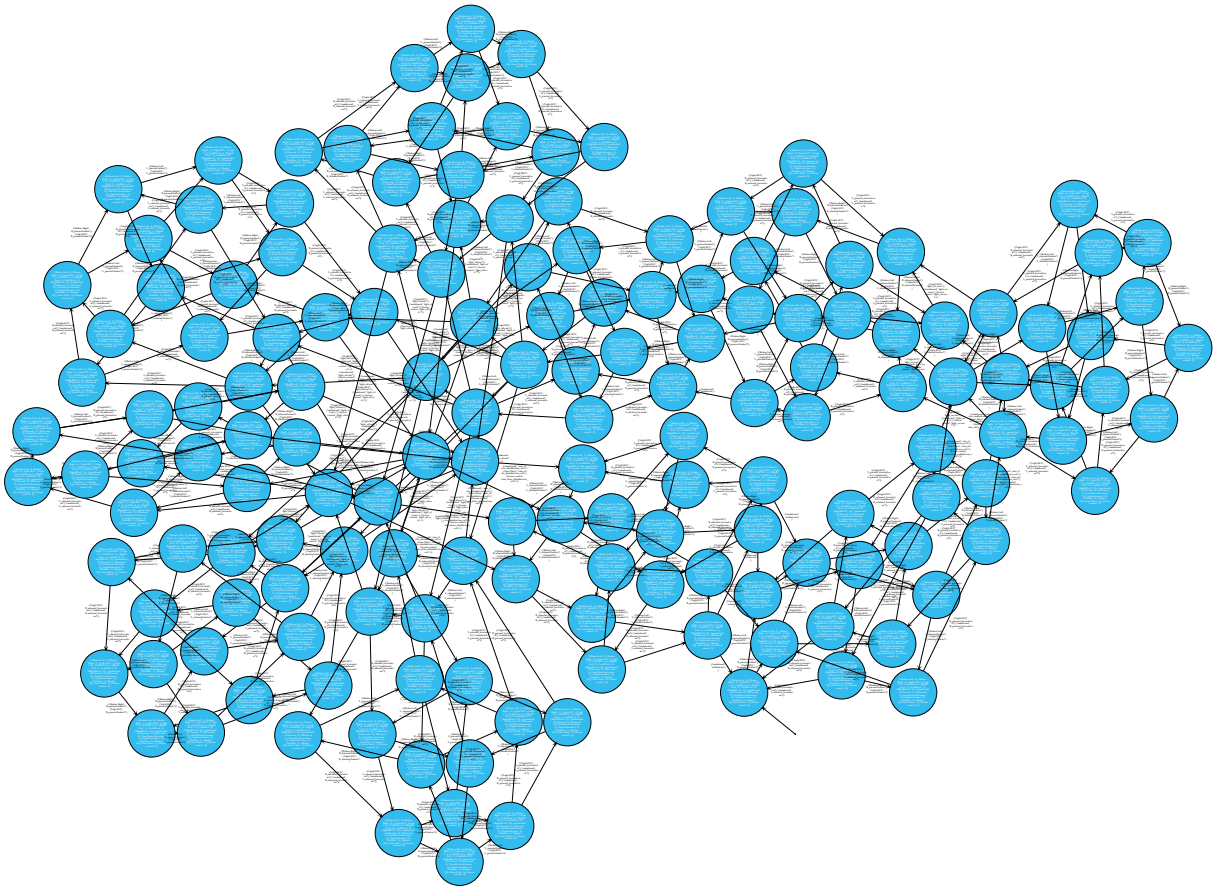


FIGURE 5.11 – Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ en cas de perte du message d'extinction de la lumière, avec la politique et gestion du rejeu

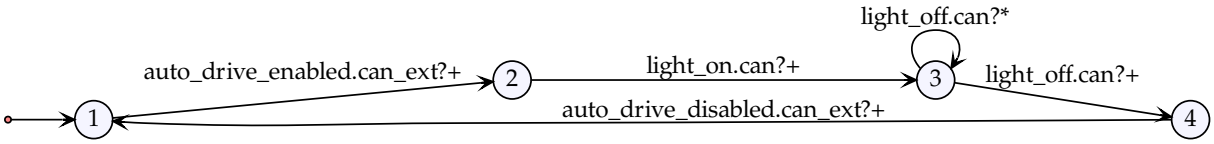


FIGURE 5.12 – Automate de spécification de la politique de contrôle d'accès avec gestion de plusieurs pertes et du rejeu

d'extinction"). Or, il est possible que des pertes se produisent lors de plusieurs envois.

Pour illustrer ce point, nous modifions nos spécifications pour permettre à la carte mère d'envoyer plusieurs messages avec les opérateurs asynchrones comme illustré par le Listing 5.3.

```
[...]
if qdict["LightActivation"] == 1:
    newd = qdict.copy()
    newd["LightActivation"] = 0
    if qdict["LPressed"] == 1 and qdict["RPressed"] == 1 and qdict["AutoDriveActivation"] == 1
    and qdict[
        "ActState"] == 1:
        # Premier envoi avec pertes
        has = A.try_rule(q, "light_off.can!*", tuple(newd.items())) or has
        # Multiples envois avec pertes
        has = A.try_rule(tuple(newd.items()), "light_off.can!*", tuple(newd.items())) or has
[...]
```

Listing 5.3 – Code de génération de l'automate de la carte mère avec plusieurs pertes et rejeu du message d'extinction

Nous modifions aussi les spécifications de la lumière et du contrôle d'accès pour leur permettre de recevoir plusieurs messages avec les opérateurs asynchrones. Nous obtenons ainsi l'automate de spécification de la Figure 5.12 pour le contrôle d'accès. La spécification de la lumière est donnée par le Listing 5.4 sur lequel nous ajoutons les réceptions des messages envoyés avec les opérateurs asynchrones lignes 23 et 26.

```
isw_light = NFA(
        {(("Light", 0), ("LeftPFwd", 0), ("RightPFwd", 0), ("LeftRFwd", 0), ("RightRFwd", 0))},
        set(),
        set(),
        name="Light ECU",
        worder=tuple
    )

def grow(A):
    has = False

    def extend(q):
        nonlocal has
        qdict = dict(q)
        if qdict["Light"] == 0:
```

```
            newd = qdict.copy()
            newd["Light"] = 1
            has = A.try_rule(q, "light_on.can?+", tuple(newd.items())) or has
        if qdict["Light"] == 1:
            newd = qdict.copy()
            newd["Light"] = 0
            # Lorsque l'on éteint la lumière, on peut recevoir un message envoyé avec les opé
    rateurs asynchrones ou bien synchrones
            has = A.try_rule(q, "light_off.can?*", tuple(newd.items())) or has
            has = A.try_rule(q, "light_off.can?+", tuple(newd.items())) or has
            # Lorsque la lumière est éteinte, on peut recevoir une infinité de  messages envoyés
    avec les opérateurs asynchrones ou bien synchrones
            has = A.try_rule(tuple(newd.items()), "light_off.can?*", tuple(newd.items())) or has
            has = A.try_rule(tuple(newd.items()), "light_off.can?+", tuple(newd.items())) or has

        if qdict["LeftPFwd"] == 0 and qdict["LeftRFwd"] == 0:
            newd = qdict.copy()
            newd["LeftPFwd"] = 1
            has = A.try_rule(q, "L_pressed.button?", tuple(newd.items())) or has
        if qdict["LeftPFwd"] == 1:
            newd = qdict.copy()
            newd["LeftPFwd"] = 0
            has = A.try_rule(q, "L_pressed_forward.can!", tuple(newd.items())) or has

        if qdict["RightPFwd"] == 0 and qdict["RightRFwd"] == 0:
            newd = qdict.copy()
            newd["RightPFwd"] = 1
            has = A.try_rule(q, "R_pressed.button?", tuple(newd.items())) or has
        if qdict["RightPFwd"] == 1:
            newd = qdict.copy()
            newd["RightPFwd"] = 0
            has = A.try_rule(q, "R_pressed_forward.can!", tuple(newd.items())) or has

        if qdict["LeftRFwd"] == 0 and qdict["LeftPFwd"] == 0:
            newd = qdict.copy()
            newd["LeftRFwd"] = 1
            has = A.try_rule(q, "L_released.button?", tuple(newd.items())) or has
        if qdict["LeftRFwd"] == 1:
            newd = qdict.copy()
            newd["LeftRFwd"] = 0
            has = A.try_rule(q, "L_released_forward.can!", tuple(newd.items())) or has

        if qdict["RightRFwd"] == 0 and qdict["RightPFwd"] == 0:
            newd = qdict.copy()
            newd["RightRFwd"] = 1
            has = A.try_rule(q, "R_released.button?", tuple(newd.items())) or has
        if qdict["RightRFwd"] == 1:
            newd = qdict.copy()
            newd["RightRFwd"] = 0
            has = A.try_rule(q, "R_released_forward.can!", tuple(newd.items())) or has

```

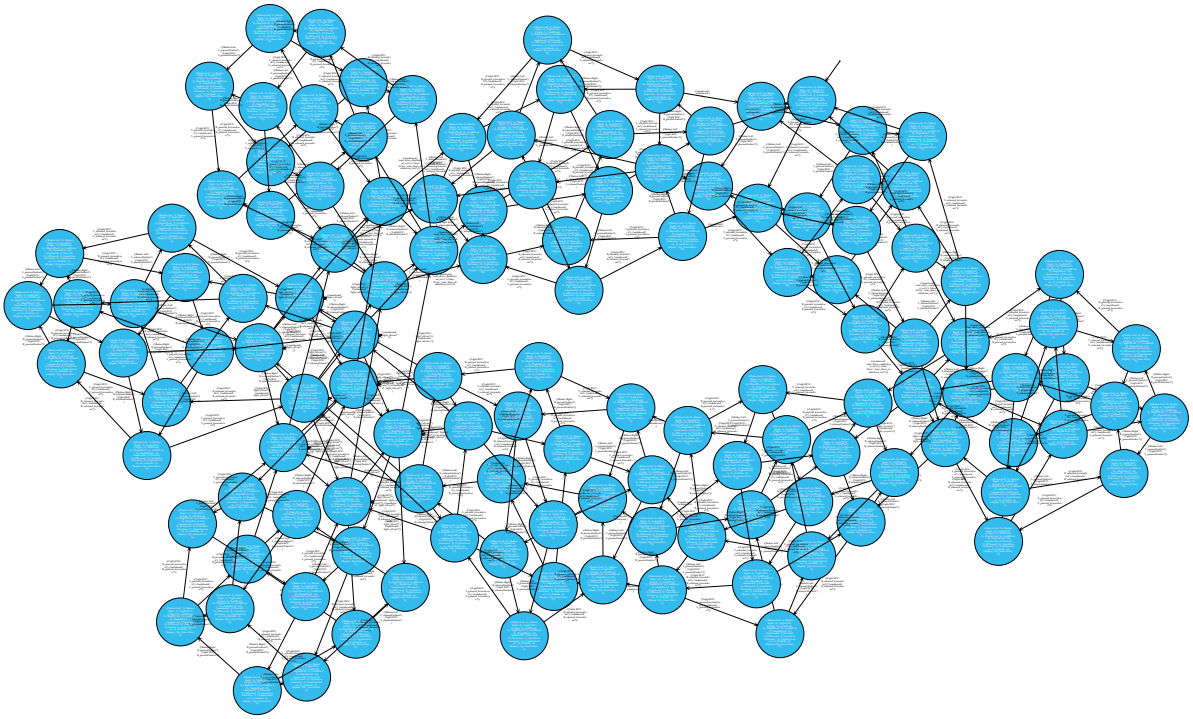


FIGURE 5.13 – Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ en cas de multiples pertes du message d'extinction de la lumière avec gestion du rejeu

```
65      for q in A.Q.copy():
66          extend(q)
67      return has
68
69  isw_light.growtofixpoint(grow, record_steps=True).visu()
```

Listing 5.4 – Code de génération de l'automate de la lumière avec plusieurs pertes et rejeu du message d'extinction

Nous préservons l'envoi et la réception d'un message avec les opérateurs synchrones pour permettre la resynchronisation de tous les automates.

Ces spécifications produisent les automates de vérification de la Figure 5.13 sans la politique et de la Figure 5.14 avec la politique. Nous constatons que les deux automates comportent uniquement des états bleus, indiquant que les propriétés ϕ, ψ et ξ sont vérifiées dans tous les états.

En adaptant notre mécanisme de contrôle d'accès et certaines spécifications du système, nous pouvons rejouer des messages de manière à rendre le système tolérant à une ou plusieurs pertes, et ce même avec la politique. Ainsi, nous sommes en mesure de préserver la safety, la disponibilité et l'intégrité fonctionnelle du système.

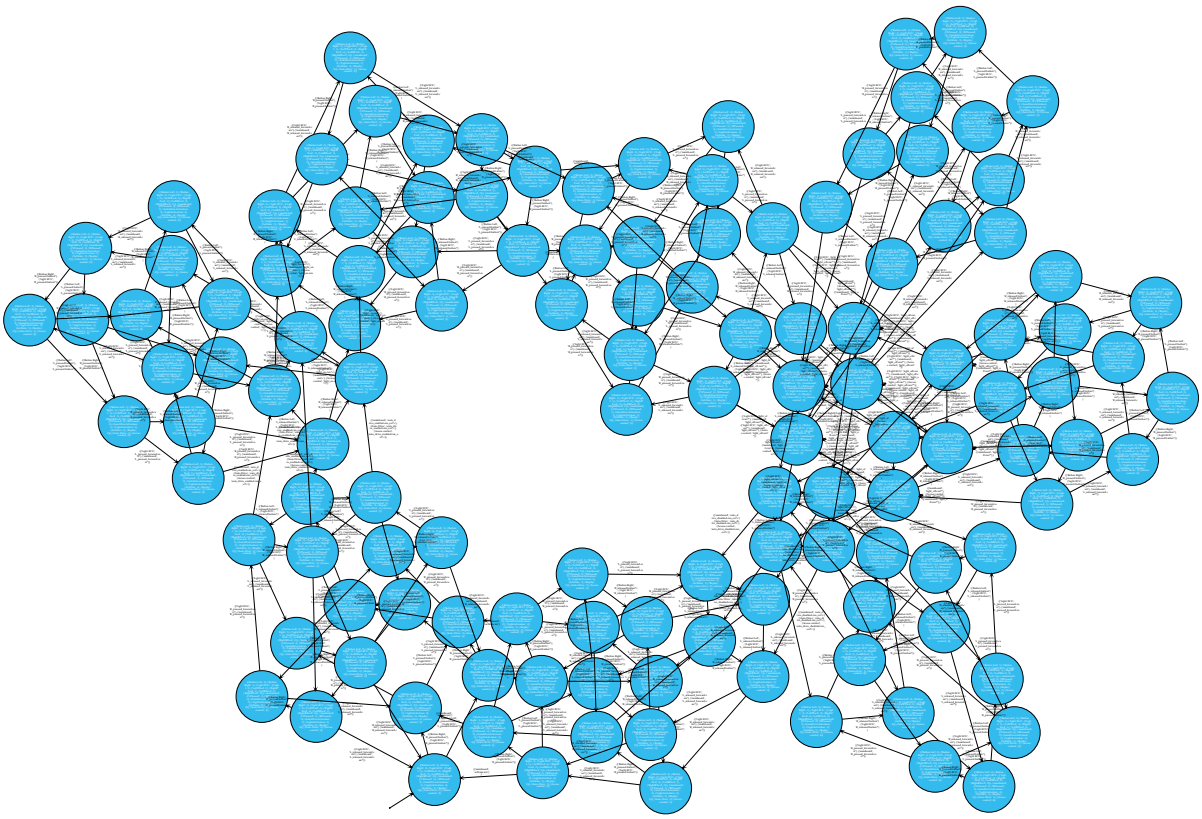


FIGURE 5.14 – Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ en cas de multiples pertes du message d'extinction de la lumière, avec la politique et gestion du rejeu

5.4 Conclusion

Après avoir rappelé le comportement de l'ISW ainsi que le profil de notre attaquant, nous avons détaillé un problème de synchronisation.

Bien que la vérification de l'ISW présentée par la Figure 4.21 : "Automate de vérification de l'ISW pour les propriétés ϕ, ψ et ξ en cas d'attaque, avec une politique de contrôle d'accès" semble vérifier toutes les propriétés, certains composants sont désynchronisés. Dans le cas présenté, cette désynchronisation est attendue et n'a pas d'impact safety sur le système.

En revanche, cette désynchronisation peut bloquer l'envoi de certains messages du système. La politique doit donc être modifié afin que tous les composants puissent se resynchroniser.

Nous avons ensuite abordé la question des pertes de messages. Dans ce cas, la politique doit recevoir certains messages plusieurs fois, afin de pouvoir gérer de multiples rejeux (si un message est perdu plusieurs fois).

En adaptant la spécification de la politique, ainsi que nos composants (afin qu'ils puissent aussi envoyer ou recevoir plusieurs fois certains messages), nous avons montré que nous sommes capables de garantir toutes les propriétés du système.

Ainsi, notre politique de contrôle d'accès peut facilement être modifiée pour autoriser des resynchronisations ou des mécanismes de rejeu. Nous sommes capables de gérer des désynchronisations entre composants, ainsi que de multiples pertes de messages.

Chapitre 6

Conclusion et perspectives

Ces dernières années, la surface d'attaque des systèmes automobiles s'est étendue, permettant de nombreuses attaques. Dans certains cas, la safety des passagers n'est plus garantie, ce qui n'est pas tolérable dans un système automobile.

Afin de protéger ces systèmes, nous nous sommes concentrés sur la mise en place d'un mécanisme de contrôle d'accès obligatoire dans un système automobile. L'état de l'art montre que les mécanismes de contrôle d'accès disponibles ne permettent pas d'identifier finement les applications du système. De plus, la mise en place d'une politique de contrôle d'accès statique n'est pas adaptée à un système automobile, car elle ne permet pas de prendre en compte l'évolution du contexte de fonctionnement du système, et les besoins safety.

De fait, la politique doit suivre l'évolution du système pour s'adapter à ce dernier : elle doit être dynamique. Bien que plus appropriée qu'une approche statique, l'approche dynamique vient aussi avec ses contraintes, notamment l'impact de la dynamicité de la politique sur le système, et donc la prise en compte d'une propriété essentielle : la safety.

La prise en compte du contexte de fonctionnement du système dépend de l'architecture (*i.e.* il faut pouvoir collecter les informations pour déterminer le contexte). Historiquement, les systèmes automobiles sont des systèmes à diffusion, permettant d'échanger facilement des informations entre les différents composants du système. Étant donné leur complexité, les systèmes automobiles sont séparés en différents domaines de diffusion. Chaque domaine donne accès à un ensemble d'informations qui sont diffusées sur le domaine. Ce type d'architecture se limite à une sécurité périmétrique. Dès lors que l'on est connecté à un domaine, toutes les informations qui sont échangées et tous les composants connectés à ce domaine sont accessibles. Ce type de sécurité n'est pas adéquat lorsqu'un système propose un nombre croissant d'interfaces vers l'extérieur.

Afin d'identifier les points les plus sensibles d'un système automobile, des analyses de risques sont effectuées. Elles permettent de faire des recommandations sur les contre-mesures à déployer et surtout où les déployer.

Le contrôle d'accès dynamique est une méthode de vérification en ligne. Une vérification hors ligne permet de garantir que le contrôle d'accès ne viole pas la safety. L'état de l'art a montré que ces deux approches sont complémentaires et que leur usage dépend des contraintes sur le système à vérifier.

Dans un premier temps, nous avons détaillé l'approche que nous avons employée. L'ajout d'un mécanisme de contrôle d'accès peut se faire grâce à un composant matériel ou logiciel. Du fait des contraintes de poids, de coûts, de latence et du besoin de pouvoir bloquer une attaque, l'approche matérielle ne nous semble pas appropriée à un système automobile. Un composant dédié devrait être ajouté à chaque domaine de diffusion et introduirait une latence importante. À l'opposé, l'approche logicielle peut être déployée sur plusieurs composants, sans ajouter de poids supplémentaire, tout en permettant de détecter une attaque et même de la bloquer. Déployer un composant logiciel est la meilleure option pour limiter les contraintes sur le véhicule et contrôler le plus grand nombre de composants.

La modélisation d'un système automobile et de notre mécanisme de contrôle d'accès a requis l'ajout de nouveaux opérateurs à l'outil du Dr. Hugot pour modéliser la diffusion. Les deux premiers sont des opérateurs synchrones bloquants, permettant d'abstraire simplement un mécanisme de contrôle d'accès en coupure. Les deux autres sont des opérateurs asynchrones non bloquants qui nous sont utiles pour modéliser des fautes.

Avec tous les éléments nécessaires à la modélisation, nous avons décrit notre cas d'usage : un volant intelligent, issu de l'expérience de Valeo en tant qu'équipementier. Nous avons aussi défini un profil d'attaquant, cherchant uniquement à violer une propriété de safety.

Pour bloquer l'attaque définie, nous avons construit itérativement une politique de contrôle d'accès dynamique évoluant en fonction des messages observés dans le système. Cette politique cherche à préserver une propriété de safety de l'ISW : la lumière d'indication de la conduite autonome n'est jamais allumée si la conduite autonome est désactivée. Cette construction itérative montre que le système à protéger doit être connu en profondeur pour définir la séquence de messages à observer.

Enfin, nous avons terminé la description de notre approche en présentant les types de propriétés qui ont été utilisées pour vérifier le système modélisé. Quatre types de propriétés ont été identifiées : safety, disponibilité, intégrité fonctionnelle et cybersécurité. Ces types de propriétés permettent de vérifier que le système préserve sa safety, tout en restant opérationnel.

Le système a ensuite été modélisé avec deux outils différents, l'outil du Dr. Hugot et SPIN. L'utilisation de ces deux outils a été motivée par la facilité de spécification et d'analyse des résultats pour le premier et des fonctionnalités plus avancées pour l'autre.

Étant donné que les deux outils n'utilisent pas le même formalisme, notre cas d'usage réel de l'ISW a été modélisé pour chaque outil, avec la volonté de garder les deux

modélisations aussi similaires que possible. De mêmes, les propriétés de safety, de disponibilité et d'intégrité fonctionnelle définies pour vérifier l'ISW ont aussi été adaptées pour chaque outil. Nous avons ensuite effectué plusieurs vérifications dans différentes situations avec chacun des outils.

La vérification du système avec les deux outils dans une situation nominale a montrée que le système a bien le fonctionnement attendu : toutes les propriétés ont été vérifiées. De plus, l'outil du Dr. Hugot nous a permis de vérifier individuellement les composants, un point intéressant pour détecter au plus tôt une erreur de spécification.

Nous sommes ensuite passé à la vérification du système en cas d'attaque. Cette étape a produit des erreurs, ce qui était attendu étant donné que le profil d'attaquant spécifié doit violer la propriété safety du système.

Pour préserver la safety du système en cas d'attaque, nous avons ajouté notre politique de contrôle d'accès dynamique à la modélisation du système. Avec l'outil du Dr. Hugot, nous n'avons trouvé aucune erreur, la politique protège le système de l'attaque et préserve toutes les propriétés. En revanche, SPIN a détecté des erreurs liées à l'envoi infini de messages de l'attaquant. Ces erreurs sont liées à la modélisation même du profil de notre attaquant et non à l'ajout de notre politique de contrôle d'accès. Ainsi, nous sommes capables de modéliser un attaquant perturbant le système et une politique pour préserver la **safety** de ce système en cas d'attaque.

Cependant, cela ne garantit pas que la politique n'a pas d'effets négatifs sur le système lorsqu'il n'est pas attaqué. Nous avons donc utilisé les deux outils pour vérifier le système et la politique sans attaque. La vérification n'a trouvé aucune erreur dans les deux cas. De fait, notre politique n'a pas d'effets négatifs sur le système.

Toutes ces vérifications ont été effectuées dans une situation idéale, où les messages ne sont jamais perdus. Or, les pertes sont fréquentes sur un réseau. En cas de pertes, notre politique ne doit pas amplifier les effets de ces dernières. De fait, nous nous sommes intéressés aux problématiques de désynchronisation entre les différents automates. Nous avons montré que notre politique est facilement adaptable pour permettre une resynchronisation des différents automates tout en préservant les propriétés du système. Nous avons aussi étudié le comportement de notre mécanisme de contrôle d'accès en cas de pertes de messages. Nous avons montré qu'il est possible de traiter plusieurs pertes tout en préservant les propriétés du système.

À partir de ce travail de recherche, plusieurs pistes peuvent être explorées pour enrichir nos résultats.

Tout d'abord, la spécification des propriétés à vérifier sur le système. Ici, cette étape a été effectuée manuellement, dans le langage utilisé par l'outil de vérification (CTL pour l'outil du Dr. Hugot, LTL pour SPIN). Or, il n'est pas réaliste de demander à des employés du secteur automobile d'effectuer cette tâche sans un langage d'abstraction. L'utilisation de pattern [MdS15, DAC99] peut être une première piste pour développer

un langage plus complet et naturel permettant de spécifier les propriétés à vérifier.

Toujours du côté de la spécification des propriétés, les moniteurs de contrôle d'accès sont actuellement spécifiés manuellement. Comme nous l'avons vu, en cas d'erreur lors de cette étape, le système peut devenir totalement inopérant. Là encore, un langage plus haut niveau pour spécifier le moniteur de contrôle d'accès permettrait de limiter les occurrences de ce type d'erreur.

Enfin, nous avons utilisé un unique modèle d'attaque cherchant à violer une propriété de safety. Comme nous l'avons vu, un système automobile possède de multiples propriétés de différentes natures (safety, disponibilité, intégrité fonctionnelle). Dans certains cas, l'attaquant que nous avons spécifié ne perturbait pas la safety, mais pouvait indirectement perturber d'autres propriétés. Ainsi, spécifier différents profils d'attaquants permettrait de tester plus amplement l'intégration de la politique de contrôle d'accès dans le système.

Bibliographie

[Aut17] Autosar. *Specification of Secure Onboard Communication v4.3.1*, December 2017.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking : 1020 states and beyond. *Information and Computation*, 98(2) :142–170, 1992.

[BD22] D. Berard and V. Dehors. Tesla pwn2own 0-click. https ://www.linkedin.com/posts/synacktiv_compromising-a-tesla-model-3-with-a-0-click-activity-6933336100224565249-Y_HT/, May 2022.

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 403–418, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BK09] Dirk Beyer and M. Erkan Keremoglu. Cpachecker : A tool for configurable software verification. *CoRR*, abs/0902.0019, 2009.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4), sep 2011.

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[CGP01] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT press, 01 2001.

[Cla14] Pierre Clairet. *Approche algorithmique pour l'amélioration des performances du système de détection d'intrusions PIGA*. Theses, Université d'Orléans, June 2014.

[DAC99] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 411–420, 1999.

[dAH01a] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In A Min Tjoa and Volker Gruhn, editors, *ESEC / SIGSOFT FSE*, pages 109–120. ACM, 2001.

[dAH01b] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software*, pages 148–165, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[dAH01c] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2001.

[dAH05] Luca de Alfaro and Thomas A. Henzinger. Interface-based design. In Manfred Broy, Johannes Grünbauer, David Harel, and Tony Hoare, editors, *Engineering Theories of Software Intensive Systems*, pages 83–104, Dordrecht, 2005. Springer Netherlands.

[DLLF+16] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, October 2016.

[DNR20] Yuri Gil Dantas, Vivek Nigam, and Harald Ruess. Security engineering for ISO 21434. *CoRR*, abs/2012.15080, 2020.

[EF20] Amir Eaman and Amy P. Felty. Formal verification of a certified policy language. In *VECoS*, 2020.

[EH83] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited : On branching versus linear time (preliminary report). In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, page 127–140, New York, NY, USA, 1983. Association for Computing Machinery.

[Eme08] E. Allen Emerson. The beginning of model checking : A personal perspective. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 27–45. Springer, 2008.

[ESF17] Amir Eaman, Bahman Sistany, and Amy Felty. Review of existing analysis tools for selinux security policies : Challenges and a proposed solution. In *E-Technologies : Embracing the Internet of Things*, pages 116–135, Cham, 2017. Springer International Publishing.

[FMFR11] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime Enforcement Monitors : composition, synthesis, and

enforcement abilities. *Formal Methods in System Design*, 38(3) :223–262, June 2011.

[FS15] Adrian Francalanza and Aldrin Seychell. Synthesising correct concurrent runtime monitors. *Formal Methods in System Design*, 46 :226–261, 09 2015.

[Gmb04] Robert Bosch GmbH. *BOSCH Automotive Handbook*. Bosch Handbooks. Robert Bosch, 2004.

[GPVW96] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. *Simple On-the-fly Automatic Verification of Linear Temporal Logic*, pages 3–18. Springer US, Boston, MA, 1996.

[Gre15] A. Greenberg. Hackers remotely kill a jeep on the highway - with me in it. Online, July 2015.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.

[Hol91] Gerald J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, NJ, USA, 1991.

[Hol97] G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5) :279–295, 1997.

[Hol11] Gerard Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2011.

[Hol15] Reid Holmes. Non-functional properties. https ://www.cs.ubc.ca/ rtholmes/teaching/2015t1/cpsc410/slides/410_04_NFPs.pdf, 2015. p.4.

[Hoo94] Jozef Hooman. Extending hoare logic to real-time. *Formal Asp. Comput.*, 6(6A) :801–826, 1994.

[HP00] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *Int. J. Softw. Tools Technol. Transf.*, 2(4) :366–381, 2000.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science : Modelling and Reasoning about Systems*. Cambridge University Press, USA, 2004.

[HSZ14] Timothy L. Hinrichs, A. Prasad Sistla, and Lenore D. Zuck. Model check what you can, runtime verify the rest. In Andrei Voronkov and Margarita Korovina, editors, *HOWARD-60. A Festschrift on the Occasion of Howard Barringer's 60th Birthday*, volume 42 of *EPiC Series in Computing*, pages 234–244. EasyChair, 2014.

[Hug20] V. Hugot. Nfa framework for 4a class on verification / model-checking. Online, October 2020.

[IEE18] IEEE. IEEE 802.3 IEEE Standard for Ethernet, 2018.

[Ing19] Felix Ingrand. Recent trends in formal validation and verification of autonomous robots software. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 321–328, 2019.

[ISO94] ISO. ISO 7498 Information technology — Open Systems Interconnection, 1994.

[ISO13] ISO. ISO 17458 Road vehicles – FlexRay communications system, 2013.

[ISO15] ISO. ISO 11898 Road vehicles – Controller area network (CAN), 2015.

[ISO18] ISO. ISO 26262 Road vehicles – Functional safety, 2018.

[ISO20] ISO. ISO 21806 Road vehicles – Media Oriented Systems Transport (MOST), 2020.

[ISO21a] ISO. ISO 15765-4 Road vehicles — Diagnostic communication over Controller Area Network (DoCAN), 2021.

[ISO21b] ISO. ISO/SAE CD 21434 Road Vehicles – Cybersecurity engineering, 2021.

[Kar96] Pim Kars. The application of promela and spin in the bos project, 1996.

[Kri63] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16 :83–94, 1963.

[Lab18] Tencent Keen Security Lab. Experimental security assessment of bmw cars : A summary report. Technical report, Tencent Keen Security Lab, May 2018.

[Lam80] Leslie Lamport. The 'hoare logic' of concurrent programs. *Acta Inf.*, 14 :21–37, 1980.

[Mar22] Matthew Martin. Functional vs. non functional requirements : Differences. https ://www.guru99.com/functional-vs-non-functional-requirements.html, 2022.

[MdS15] Frédéric Mallet and Robert de Simone. Correctness Issues on MARTE/CCSL constraints. *Science of Computer Programming*, 106 :78–92, August 2015.

[Mou11] Sebti Mouelhi. *Contributions à la vérification de la sûreté de l'assemblage et à l'adaptation de composants réutilisables*. Theses, Université de Franche-Comté, August 2011.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In Cynthia Dwork, editor, *PODC*, pages 377–410. ACM, 1990.

[oST18] National Institute of Standards and Technology. Systems security engineering : Considerations for a multidisciplinary approach in the engineering of trustworthy secure systems (p.166). Technical report, U.S. Department of Commerce, Washington, D.C., 2018.

[Par19] European Parliament. General Safety Regulation 2019/2144, 2019.

[Pnu77] Amir Pnueli. The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977.

[Pol17] West Midlans Police. Relay attack solihull. Online, November 2017.

[PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 364–380, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[PRD21] Christophe Ponsard, Valery Ramon, and Jean-Christophe Deprez. Goal and threat modelling for driving automotive cybersecurity risk analysis conforming to iso/sae 21434. In *Proceedings of the 18th International Conference on Security and Cryptography - Volume 1 : SECRYPT,*, pages 833–838. INSTICC, SciTePress, 2021.

[R.12] Gerth R. Promela manual pages, 2012.

[RCB08] Grigore Rosu, Feng Chen, and Thomas Ball. Synthesizing monitors for safety properties : This time with calls and returns. In Martin Leucker, editor, *RV*, volume 5289 of *Lecture Notes in Computer Science*, pages 51–68. Springer, 2008.

[S+13] Y. Shoukry et al. Non-invasive spoofing attacks for anti-lock braking systems. In *Cryptographic Hardware and Embedded Systems - CHES 2013*, pages 55–72, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Sec17] Argus Cyber Security. A remote attack on the bosch drivelog connector dongle. Technical report, Argus Cyber Security, April5 2017.

[Syn] Synopsys. What is asil ? https ://www.synopsys.com/automotive/what-is-asil.html.

[TG+15] M. Turker Garip et al. Congestion attacks to autonomous cars using vehicular botnets. In *NDSS 2015*, January 2015.

[UAcLR01] Algirdas Avizienis Ucla, Algirdas Avizienis, Jean claude Laprie, and Brian Randell. Fundamental concepts of dependability, 2001.

[Ven15] Benjamin Venelle. *Contrôle d'accès obligatoire pour systèmes à objets : défense en profondeur des objets Java*. These, Université d'Orléans, July 2015.

Adrien JOUSSE

Protection obligatoire vérifiée au regard des objectifs safety du secteur automobile

Résumé : La safety d'un système automobile (sécurité des personnes et des biens) dépend du contexte de fonctionnement du système embarqué. Afin de préserver la safety, les mécanismes de sécurité doivent prendre en compte dynamiquement le contexte de fonctionnement du système. Or, comment garantir qu'un mécanisme de sécurité dynamique préserve la safety dans toutes les configurations possibles, sans introduire d'effets indésirables ? Afin de répondre à ces problématiques, nous proposons d'ajouter un mécanisme de contrôle d'accès obligatoire dynamique vérifié au regard des objectifs safety. Nous commençons par montrer que les mécanismes de contrôle d'accès existants ne sont pas suffisants. Nous détaillons ensuite certaines spécificités du secteur automobile et les approches de vérification formelle que nous utiliserons. Puis nous présentons la conception de notre mécanisme de contrôle d'accès, le cas d'usage réel sur lequel il sera testé, ainsi que les capacités de l'attaquant considéré. Nous détaillons aussi la politique de contrôle d'accès ainsi que les propriétés que nous vérifions sur notre cas d'usage. Nous passons ensuite à la modélisation de notre cas d'usage afin de le vérifier avec deux outils. Différentes vérifications sont effectuées afin de vérifier que le système possède les propriétés voulues, qu'elles ne sont pas garanties sans notre contrôle d'accès mais préservées par notre politique de contrôle d'accès. Enfin, nous montrons que notre mécanisme de contrôle d'accès est tolérant aux pertes lorsque des mécanismes de rejeu appropriés sont mis en place.
Mots clés : contrôle d'accès, automobile, vérification, safety, automates

Mandatory protection checked against the automotive sector's safety objectives

Abstract : The safety of an automotive system (security of persons and goods) depends on the system's operating context. In order to preserve safety, safety mechanisms must dynamically consider the context of the embedded system. However, how can we guarantee that a dynamic safety mechanism preserves the safety in all possible configurations, without introducing undesirable effects ? In order to address these issues, we propose to add a dynamic mandatory access control mechanism that is verified with respect to safety objectives. We start by showing that existing access control mechanisms are not sufficient. We then detail some specificities of the automotive sector and the formal verification approaches we will use. We then present the design of our access control mechanism, the real use case on and the capabilities of the attacker. We also detail the access control policy and the properties that have to be enforced. We then move on to model our use case in order to verify it with two tools. Different checks are performed to verify that the system has the desired properties, that they are not satisfied without our access control but are preserved by the enforcement of our access control policy. Finally, we show that our access control mechanism is loss tolerant when appropriate replay mechanisms are implemented.
Keywords : access control, automotive, verification, safety, automata

LIFO - Bâtiment IIIA
Rue Léonard de Vinci
B.P. 6759
F-45067 ORLEANS
Cedex 2