Conception de systèmes de reconnaissance d'activités humaines.

Ines Sarray

Université de Nice - Sophia Antipolis – UFR Sciences
École Doctorale STIC

THÈSE

Présentée pour obtenir le titre de :
Docteur en Sciences de l'Université de Nice - Sophia Antipolis
Spécialité : INFORMATIQUE
Soutenue le : 21 MARS 2019

par

Ines SARRAY

Conception de systèmes de reconnaissance d'activités humaines

Directeur de thèse : **Mme Sabine MOISAN**
Co-encadrant de thèse : **Mme Annie RESSOUCHE**

Jury

M. Jean-Paul BODEVEIX,	Professeur	Rapporteur
M. Frédéric BOULANGER,	Professeur	Rapporteur
M. Frédéric MALLET,	Professeur	Examinateur
Mme Térésa COLOMBI,	Directrice LUDOTIC	Invitée
Mme Sabine MOISAN,	Chargée de recherches	Directeur de thèse

Résumé

La reconnaissance d'activités est un domaine de recherche qui vise à décrire, analyser, reconnaître, comprendre et suivre les activités et les mouvements de personnes, d'animaux, ou d'objets animés.

De nombreux domaines d'applications, importants et critiques, tels que la surveillance, la sécurité ou la santé, nécessitent une certaine forme de reconnaissance d'activités (humaines). Dans ces domaines, la reconnaissance d'activités peut être utile pour détecter tôt les comportements anormaux de certaines personnes : actes de vandalisme ou difficultés dues à l'âge ou la maladie.

Les systèmes de reconnaissance doivent être temps réel, réactifs, corrects, complets et fiables. Ces exigences strictes nous amènent à l'utilisation de méthodes formelles pour décrire, analyser, vérifier et générer des systèmes de reconnaissance efficaces et corrects. L'objectif de cette thèse est de contribuer à la définition d'un tel système en se focalisant sur les aspects de description et de vérification.

Parmi les nombreuses approches envisageables, nous proposons d'étudier comment le paradigme synchrone peut s'appliquer aux besoins de la reconnaissance d'activités. En effet, cette approche possède des atouts qui semblent intéressants : une sémantique bien fondée, l'assurance du déterminisme, une composition parallèle sûre, et la possibilité de vérification grâce au model checking.

Les langages synchrones existants peuvent être utilisés pour décrire des modèles d'activités, mais ils sont difficiles à maîtriser par des non informaticiens (ex : médecins). Nous proposons donc un nouveau langage dédié à ce type d'utilisateurs pour décrire les activités qu'ils souhaitent reconnaitre. Ce langage nommé ADeL (Activity Description Language) propose deux formats équivalents, l'un textuel et l'autre graphique.

Afin de permettre à la fois les vérification et l'implémentation, nous munissons notre langage de deux sémantiques synchrones complémentaires. D'abord, une sémantique comportementale qui donne une définition référentielle du comportement d'un programme en utilisant des règles de réécriture. Deuxièmement, une sémantique opérationnelle qui décrit le comportement d'une manière constructive et peut être directement mise en œuvre.

Comme l'environnement des systèmes de reconnaissance n'est généralement pas conforme aux hypothèses du paradigme synchrone, notre système doit comporter un transformateur asynchrone/synchrone. Ce transformateur, que nous appelons "synchroniseur", reçoit les évènements asynchrones de l'environnement, les filtre, décide lesquels peuvent être considérés comme "simultanés", et les regroupe en instants logiques selon des politiques prédéfinies pour les envoyer au moteur de reconnaissance d'activités.

Abstract

The research area of activity recognition aims at describing, analyzing, recognizing, understanding and following the activities and movements of persons, animals, or animated objects.

Numerous important and critical application domains, such as surveillance or health-care, require a certain form of recognition of (human) activities. In these domains, activity recognition can be useful for the early detection of abnormal behavior of people, such as vandalism, troubles due to age, or illness.

Recognition systems must be real-time, reactive, correct, complete, and reliable. These stringent requirements led us to use formal methods to describe, analyze, verify, and generate effective and correct recognition systems. This thesis aims at contributing to define such a system while focusing on description and verification issues.

Among many possible approaches, we propose to study how the synchronous paradigm can cope with the requirements of activity recognition. Indeed, this approach has several major assets such as well founded semantics, assurance of determinism, safe parallel composition, and possibility of verification owing to model checking.

Existing synchronous languages can be used to describe models of activities, but they are difficult to master by non specialists (e.g., doctors). Therefore, we propose a new language to allow this kind of users to describe the activities that they wish to recognize. This language, nammed ADeL (Activity Description Language), proposes two input formats, the first textual, the other graphic.

In order to make both verification and implementation possible, we supply this language with two synchronous and complementary semantics. First, a behavioral semantics gives a reference definition of program behavior using rewriting rules. Second, an operational semantics describes the behavior in a constructive way and can be directly implemented.

The environment of recognition systems does not usually comply with the hypotheses of the synchronous paradigm. Hence, we propose an asynchronous/synchronous adapter. This adapter, that we call "synchronizer", receives the asynchronous events from the environment, filters them, decides on which ones can be considered as "simultaneous", groups them in logical instants according to predefined politics, and send them to the activity recognition engine.

Remerciement

J'aimerais remercier toutes les personnes qui ont contribué de près ou de loin à l'achèvement de cette thèse.

Je tiens à exprimer ma profonde reconnaissance envers Madame Annie RESSOUCHE, pour m'avoir encadrée durant mes trois années de thèse. Je la remercie particulièrement pour m'avoir dirigée, guidée et conseillée dans une ambiance toujours formidable. J'apprécie énormément ses hautes qualités scientifiques et ses valeurs humaines exemplaires et pour tout le temps qu'elle m'a consacré. Je la remercie parce qu'elle a été toujours à mes côté pour m'encourager et m'aider. J'ai pris beaucoup de plaisir à travailler avec elle. Les mots ne peuvent exprimer toute ma gratitude. J'ai de la chance de l'avoir comme encadrante.

J'ai aussi de la chance d'avoir Madame Sabine MOISAN comme directrice de thèse. Je suis profondément reconnaissante et je lui exprime toute ma sincère gratitude. Elle a su à chaque fois me donner les meilleurs conseils et les bonnes orientations. Elle a toujours cherché des solutions pour mon bien, pour m'aider et pour me pousser vers l'avant. Elle m'a toujours défendue, encouragée et m'a remontée souvent le moral quand je passais par des périodes difficiles. Je la remercie pour la magnifique ambiance, pour cette inoubliable expérience, pour son sourire, pour son aide et sa confiance, sans elle, je n'aboutirai pas à un tel résultat.

Un grand merci à Monsieur Jean-Paul RIGAULT, l'une des personnes qui m'ont beaucoup inspirée et qui m'inspireront toujours. Je le remercie pour sa gentillesse, pour ses orientations et ses judicieux conseils, associés à sa sagesse qui m'ont permis d'améliorer la qualité de mes travaux et du présent mémoire. Je tiens à lui exprimer mon respect le plus profond et mes remerciements les plus sincères.

Je tiens aussi à remercier Monsieur Daniel GAFFE pour son amabilité, son aide importante et ses orientations judicieuses, ses critiques pertinentes et ses conseils qui m'ont été bénéfiques tout au long de cette thèse. Je le remercie également pour sa confiance, son soutien et son encouragement.

J'adresse mes sincères remerciements à Madame Térésa COLOMBI et Madame Michèle DELACROIX, de LUDOTIC, pour leurs contributions à cette thèse, pour leurs remarques, conseils et aide, pour le temps qu'elles m'ont consacré malgré leurs responsabilités et pour leurs encouragements. J'ai pris beaucoup de plaisir à travailler avec elles.

Mes remerciements d'adressent également aux membres de jury : Monsieur Jean-Paul BODEVEIX, Professeur à l'Université Paul Sabatier de Toulouse et Monsieur Frédéric BOULANGER, Professeur à SUPELEC de Paris pour l'honneur qu'ils m'ont fait en acceptant d'être mes rapporteurs de thèse. Je tiens à remercier également Monsieur Frédéric MALLET, Professeur à l'Université Côte d'Azur, qui m'a honorée par sa présence dans le jury de ce travail.

Je tiens aussi à remercier Madame Susane GRAF pour ses remarques et conseils avisés qui m'ont beaucoup servi à améliorer ce mémoire.

Je remercie aussi tous les membres de l'équipe STARS d'INRIA, ma deuxième famille, avec lesquels j'ai passé mes trois ans de thèse. Merci pour tout ce temps passé ! Merci pour votre amitié et pour le climat convivial et chaleureux que vous m'avez apporté.

Je remercie également Madame Chantal GOLAZ pour son accueil chaleureux, sa gentillesse et ses encouragements.

Un énorme et un grand merci à mes parents : Houcine et Habiba. Aucun hommage ne pourrait être à la hauteur de l'amour dont ils ne cessent de me combler. Vous m'avez toujours soutenu tout au long de mon parcours et vous n'avez jamais cessé de m'encourager. Vos conseils ont toujours guidé mes pas vers la réussite. Vous avez su m'inculquer le sens des responsabilités, l'optimisme et la confiance en soi face aux difficultés de la vie. Tout ce que j'ai pu réaliser est grâce à vous. Merci pour tous vos sacrifices. Je vous aime beaucoup ! Que Dieu vous procure bonne santé et longue vie.

A ma chère sœur Oumaima et mon cher frère Cherif, merci pour vos encouragements, votre soutien moral et votre confiance, merci d'avoir été toujours à mes côtés pendant mes moments difficiles. Vous êtes toujours dans mes pensées et dans mon cœur. Que Dieu vous aide à concrétiser tous vos rêves.

A mon amie, ma sœur et ma source d'énergie positive : Olfa ! merci d'avoir été toujours là à m'écouter quand j'avais besoin de parler, merci pour ton soutien et pour l'énergie positive que tu ne cesse de m'offrir à chaque fois que j'atteins mes limites ou quand je ne me sens pas bien.

A ma chère amie et sœur Salwa, merci pour tous les moments inoubliables que nous avons passé ensemble durant ces années de thèse, merci pour tes encouragements, merci d'avoir été toujours là pour m'écouter et me soutenir.

A mes amis : Lotfi, Monia, Douja, Abdelrahman, Andrew, Rachid, merci pour votre soutien et encouragements, merci d'avoir été toujours là pour moi, pour m'aider, merci de m'avoir fait sentir au sein d'une famille, je vous aime !

A tous ceux qui m'aiment et pour qui ma réussite est importante, je vous remercie de tout mon cœur.

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1

Introduction générale

Sommaire

1.1 Contexte de la thèse

La reconnaissance d'activités est un domaine de recherche qui vise à analyser, reconnaître, comprendre et suivre les activités et les mouvements d'une personne, d'un animal, ou d'un objet animé, en employant différentes techniques et méthodes.

De nombreuses applications importantes et critiques telles que la surveillance ou le domaine de la santé nécessitent une certaine forme de reconnaissance d'activités (humaines).

Domaine de la santé
L'espérance de vie croît régulièrement. Au cours des 60 dernières années, la durée de vie moyenne dans le monde a augmenté de plus de 20 ans : elle n'atteignait que 46,6 ans dans les années 1950-1955 [1]. En 2017, en France, l'espérance de vie a atteint 82.5 ans [2]. Le 1er janvier 2017, les habitants âgés de 75 ans ou plus, représentaient près d'un français sur dix [3]. De plus, un grand nombre de personnes agées vivent seules aujourd'hui : selon une étude de l'INSEE en 2015, 21% des hommes et 53% des femmes de 75 ans ou plus vivent seuls à leurs domiciles [4]. Ceci peut créer un sentiment d'insécurité chez eux : si quelque chose nécessite une intervention urgente et qu'ils sont seuls, cela peut avoir de dangereuses conséquences. Ceci explique la nécessité de leur suivi.

De plus, les maladies neurobiologiques qui nécessitent un diagnostic rapide, une surveillance et un suivi continu commencent à être de plus en plus fréquentes ; on peut citer par exemple la maladie d'Alzheimer et l'autisme. En France, selon le ministère de la santé, 900 000 personnes étaient atteintes de la maladie d'Alzheimer en 2015 et ce nombre pourrait atteindre les 3 millions en 2020 [5]. Pour l'autisme, 700 000 personnes sont atteintes de cette maladie, selon l'INSERM, et ce nombre risque d'augmenter dans les années à venir [6].

Tout cela montre la nécessité et l'importance de la reconnaissance d'activités dans le domaine médical. En effet, la reconnaissance d'activités permet aux médecins de surveiller le comportement et les activités des patients (par exemple, personnes âgées et/ou atteintes d'Alzheimer) même à distance et d'agir rapidement en cas d'urgence ou de danger. Elle leur permet aussi de mieux connaître le patient, de diagnostiquer ses pathologies, et de développer de nouvelles stratégies d'intervention en matière de prévention et d'accompagnement.

Domaine de la sécurité
Le domaine de la sécurité et de la surveillance devient de plus en plus important surtout avec l'augmentation du nombre d'attaques terroristes. En effet, en France,

1. https://www.ined.fr/fr/tout-savoir-population/memos-demo/focus/la-duree-de-vie-dans-le-monde/
2. https://www.insee.fr/fr/statistiques/3303354?sommaire=3353488#consulter
3. https://www.insee.fr/fr/statistiques/2582785?sommaire=2587886
4. https://www.insee.fr/fr/statistiques/1285396
5. http://alzheimer-recherche.org/la-maladie-alzheimer/quest-maladie-dalzheimer/definition-et-chiffres/
6. https://www.inserm.fr/index.php/information-en-sante/dossiers-information/autisme

depuis 2013, il y a eu 11 attaques terroristes, faisant 245 morts et plus de 900 blessés [7]. La reconnaissance d'activités peut jouer un rôle important dans le domaine de la surveillance en aidant à détecter les mouvements et activités suspects de certaines personnes. Elle permet donc de réduire le risque d'attentats, d'intrusions, et aussi de vols et d'agressions. La reconnaissance d'activités peut détecter tôt les comportements douteux, ce qui augmente la sécurité des personnes dans les lieux publics ou privés et la protection des sites sensibles comme les banques, les départements militaires ou les frontières.

Dans cette thèse, nous collaborons avec les médecins de l'Institut Claude Pompidou [8] de Nice dans le cadre du laboratoire CoBTeK [9] de l'université de Nice Côte d'Azur dont notre équipe fait partie. Nous souhaitons principalement reconnaître les activités de personnes agées et/ou atteintes d'Alzheimer. Nos tests seront principalement dirigés vers le domaine du suivi médical. Cependant, notre système de reconnaissance est générique et il pourra être utilisé dans d'autres domaines, comme la sécurité.

1.2 Positionnement du sujet de recherche

Depuis longtemps, l'équipe STARS travaille sur la génération de systèmes de reconnaissance d'activités. Ces systèmes correspondent à une succession d'algorithmes de *clustering* et de *pattern matching*, combinés à des représentations de connaissances adéquates (par exemple, la topologie de la scène, les contraintes temporelles, etc.). En raison de la large variété de domaines d'application (surveillance, sécurité, soins de santé, etc.), nous proposons une approche générique pour concevoir des moteurs de reconnaissance d'activités. De plus, ces domaines nécessitent une haute fiabilité en raison de possibles problèmes de sécurité. Ainsi, notre approche doit également s'appuyer sur des méthodes formelles pour décrire, analyser, vérifier et générer des moteurs de reconnaissance efficaces et sûrs.

1.2.1 Les systèmes de reconnaissance d'activités

Aujourd'hui, dans le domaine de la vision par ordinateur en particulier, il est facile pour la plupart des systèmes existants de reconnaître des activités simples et courtes comme "marcher" "se lever" ou "attraper un objet". Mais dans des scénarios d'activités plus complexes, on doit aussi traiter des activités parallèles, imbriquées et/ou qui se déroulent sur une longue durée. On désigne par *activité complexe* une activité qui englobe plusieurs activités simples comme "marcher" ou "boire". Ces activités simples peuvent être séquentielles ou parallèles. Une activité complexe peut faire appel à une autre sous activité complexe. Elle peut introduire plusieurs personnes et décrire plusieurs intéractions avec des personnes et/ou des équipements. Une activité complexe peut aussi contenir des activités entrelacées, c'est à dire des activités mises en arrêt momentané en faveur d'une autre activité, et des activités ambigües, c'est à dire qui contiennent des activités simples qui ne peuvent pas être reconnues à coup sûr sans avoir d'informations supplémentaires.

7. https://www.lemonde.fr/societe/article/2018/03/30/de-2013-a-2018-la-france\-au-rythme-des-attentats_5278453_3224.html
8. http://www.institut-claude-pompidou.fr/
9. http ://unice.fr/en/research/the-laboratories-1/cobtek

La reconnaissance d'activités complexes reste un problème difficile. Les systèmes de reconnaissance doivent être temps réel, réactifs, corrects, complets et fiables. L'objectif de cette thèse est de créer un tel système de reconnaissance. En raison du grand nombre de domaines d'application possibles, nous voulons un système *générique*. Nous souhaitons aussi que notre système de reconnaissance d'activités soit facile à manipuler par les utilisateurs non informaticiens.

En fait, nous pouvons considérer les moteurs de reconnaissance d'activités comme des systèmes réactifs qui réagissent aux événements d'entrée de leur environnement et produisent des événements de sortie sous la forme d'alarmes ou de notifications (voir section 1.2.2). Ces moteurs sont intrinsèquement temps réel, réactifs et évoluent en temps discret. En conséquence, pour reconnaître des scénarios d'activités, nous avons choisi d'adapter les techniques de l'approche de modélisation *synchrone*. Cette approche facilite la validation des descriptions d'activités et nous permet de générer un "reconnaisseur" pour chaque d'activité.

1.2.2 Les systèmes réactifs

Les systèmes réactifs (figure 1.1)[10] sont des systèmes qui dépendent de leur environnement et sont en interaction permanente avec lui. Leur réveil dépend du changement des entrées provenant de l'environnement (E_i). Ils les écoutent, effectuent le traitement nécessaire (F_i) en fonction de leur état interne et envoient les sorties en réaction (S_i). Les systèmes réactifs doivent répondre aux stimuli de l'environnement en respectant certaines contraintes dont la plus importante est la contrainte temporelle. Nous considérons les systèmes de reconnaissance d'activités comme des systèmes réactifs temporels qui écoutent les capteurs de l'environnement et identifient les activités selon les informations reçues. Dans notre cas, les réactions sont des alarmes levées vers l'environnement. Par exemple, il peut s'agir d'un message envoyé à un soignant pour intervention, d'un message vocal rappelant une consigne à un patient, ou d'une alarme sonore pour réveiller quelqu'un qui s'endort. Ces alarmes peuvent déclencher des actions qui peuvent changer les futurs instants du système. Par exemple, lors d'un jeu sérieux, un patient qui reçoit un message vocal rappelant une consigne du jeu peut changer son attitude et cela a un impact sur la reconnaissance.

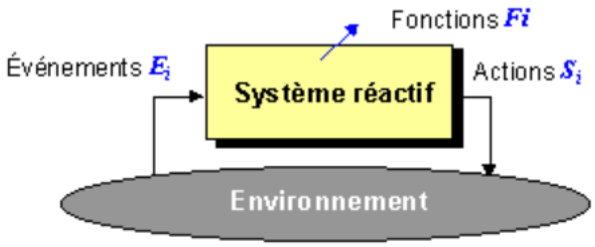


FIGURE 1.1 – Représentation usuelle de la communication d'un système réactif avec son environnement

10. http://www.rennes.supelec.fr/ren/perso/pchlique/tpsreel/sysreac.htm

1.2.3 Pourquoi s'appuyer sur des méthodes formelles pour construire un système de reconnaissance ?

Les simulations arrivent à leurs limites avec la complexité croissante des systèmes numériques, ce qui impose de chercher de nouvelles méthodes de validation. Les exigences strictes des systèmes de reconnaissance nous amènent à l'utilisation de méthodes formelles pour décrire, analyser, vérifier et générer une reconnaissance efficace. Nous avons décidé d'utiliser l'approche synchrone.

1.2.4 Pourquoi l'approche synchrone ?

Le modèle synchrone est un moyen de réduire la complexité de description du comportement d'un système en simplifiant la notion du temps. En effet, l'hypothèse synchrone modélise et simplifie le temps réel (temps de l'environnement ou d'un observateur externe, que nous appellerons par la suite "temps de la montre") et considère l'évolution des systèmes selon des instants discrets successifs qui constituent un *temps logique*. Cette hypothèse considère aussi que la réaction des systèmes aux évènements provenant de l'environnement est atomique, c'est-à-dire que le système ne prend plus en considération les variations de l'environnement pendant son traitement et ce jusqu'à la fin de sa réaction. Ainsi, nous pourrons considérer que le système réagit aux évènements d'entrée et génère les évènements de sortie dans le même instant logique.

La modélisation synchrone permet de définir des systèmes réactifs (dont les systèmes de reconnaissance) sûrs. Le paradigme synchrone a plusieurs atouts : d'abord, il assure le "parallèle déterministe", une propriété qui permet de traiter les activités parallèles tout en gardant le déterminisme du système sans avoir des problèmes de concurrence critique. Ensuite, c'est l'un des paradigmes qui aident à simplifier et à faciliter la description et la modélisation des activités complexes, en permettant de diviser un système en plusieurs sous-systèmes plus simples. Ceci facilite non seulement la représentation de l'activité, mais aussi la génération de formules de logique temporelle qui peuvent être introduites facilement dans les model-checkers. D'autre part, le paradigme synchrone est bien fondé, il s'appuie sur une sémantique bien définie.

Le paradigme synchrone n'a pas été utilisé dans le domaine de la reconnaissance d'activités auparavant, ce qui fait l'originalité de ce travail. Un des buts de cette thèse est donc d'étudier l'applicabilité de cette approche à la reconnaissance d'activités.

1.2.5 Comment rendre l'approche synchrone accessible à des utilisateurs non-informaticiens ?

Les langages synchrones permettent de décrire des modèles synchrones qui représentent des automates à états finis et qui peuvent être des modèles d'entrée pour les outils de model checking. Il existe plusieurs langages synchrones qui permettent de modéliser les activités simples et complexes, mais aucun d'entre eux n'est destiné aux non-informaticiens. Dans cette thèse, nous proposons un langage synchrone destiné à ce type d'utilisateurs qui est capable de modéliser et de décrire la plupart des activités. Ce langage s'appelle ADeL (Activity Description Language) et il a été défini en collaboration avec des ergonomes pour s'assurer de l'acceptabilité du langage par des non informaticiens. Ce langage de description d'activités propose les notions de rôles (typés), d'actions, de sous-activités et de flux de contrôle. Il repose sur des opérateurs de contrôle classiques et temporels formellement spécifiés, et supporte le parallélisme,

le choix, les répétitions, et la description de contraintes (voir chapitre 3).

1.2.6 Comment donner des bases formelles au langage ADeL ?

En se basant sur l'approche synchrone, nous avons associé à notre langage deux sémantiques complémentaires. Tout d'abord, une sémantique comportementale donne une définition de référence du comportement du programme en utilisant des règles de réécriture [Plo81]. Cette sémantique présente un moyen "naturel" de décrire le comportement de chaque opérateur et en donne ainsi une interprétation claire. Deuxièmement, une sémantique opérationnelle décrit le comportement d'une manière constructive et peut être directement mise en œuvre. Cette seconde sémantique transforme un programme en un système d'équations booléennes. En utilisant ces équations, nous pouvons générer l'automate de reconnaissance ainsi qu'un code efficace pour plusieurs outils cibles tels que des model-cherckers, des simulateurs et notre moteur de reconnaissance d'activités. Nous avons deux sémantiques différentes, il est donc obligatoire d'établir leur relation. En fait, nous prouvons dans cette thèse que l'exécution d'un programme basé sur la sémantique opérationnelle est conforme à la sémantique comportementale (voir chapitre 4).

1.2.7 Comment intégrer un système synchrone dans le monde asynchrone ?

Le problème qui se pose ici est que notre système synchrone n'accepte que des données synchrones en entrée et n'émet que des données synchrones en sortie. Or, l'environnement est généralement asynchrone, et si on souhaite traiter des informations qui proviennent des capteurs et des caméras, ces données seront asynchrones. En fait, un système synchrone pourrait traiter des données asynchrones : il suffirait d'accrocher chaque événement asynchrone à un instant logique différent et de garantir que le système réagit assez vite avant l'arrivée de l'évènement asynchrone suivant. Ceci a deux effets, d'abord le système serait sollicité beaucoup trop souvent ; ensuite, deux évènements asynchrones très proches temporellement peuvent signifier un phénomène particulier (par exemple un double clic souris) ; dans ce cas, il est intéressant de pouvoir les mettre dans le même instant logique et d'envoyer cet instant au système réactif synchrone. Nous proposons dans cette thèse une adaptation automatique et continue des échanges entre l'environnement et notre système de reconnaissance, en créant un *synchroniseur* (ou transformateur asynchrone/synchrone) qui va transformer des données asynchrones en données synchrones avant de les présenter au système de reconnaissance (voir chapitre 5).

1.3 Contributions : système de reconnaissance proposé

La figure 1.2 présente la structure globale de notre système générique de reconnaissance d'activités. La première étape consiste à décrire les activités à reconnaître à l'aide de notre langage de description. Ces activités sont indépendantes et chacune correspondra à un programme autonome. Ensuite, en se basant sur les règles de la sémantique opérationnelle, le compilateur génère une représentation des activités décrites, sous la forme d'une machine à états finis, qui est intégrée dans le moteur

de reconnaissance. Le compilateur peut aussi générer une représentation en d'autres formats pour différentes cibles, telles que les model-checkers et les simulateurs.

Le synchroniseur (voir chapitre 5) extrait de façon continue des évènements et des objets à partir des informations qui proviennent des capteurs (par exemple, vidéo, audio). Ces informations étant asynchrones, le synchroniseur filtre ensuite ces évènements et les regroupe en des instants logiques suivant des stratégies prédéfinies. Il les envoie au moteur de reconnaissance, dont le rôle est de reconnaître au moment de l'exécution toutes les instances d'activités répondant à au moins un modèle d'activité. A chaque instant, ce moteur fait progresser les automates des différentes instances d'activités selon les évènements d'entrées, et il collecte aussi les évènements de sortie (dans notre cas, des alarmes ou des terminaisons d'activité).

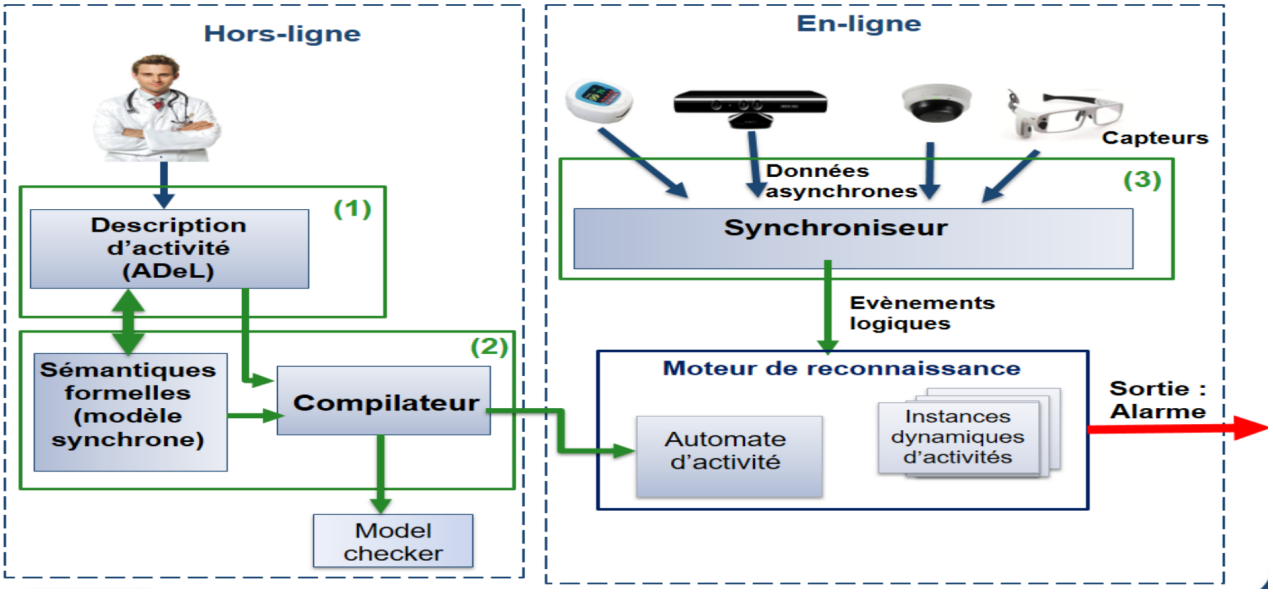


FIGURE 1.2 – Vue globale de notre système de reconnaissance d'activités : les contributions de la thèse sont mentionnés en vert

Les contributions dans cette thèse sont mentionnées en vert dans la figure 1.2. Une première contribution est la définition du langage de description d'activités. Notre objectif est de définir un langage générique qui pourra être utilisé pour décrire plusieurs types d'activités.

La seconde partie consiste en la définition des deux sémantiques formelles déjà évoquées. Une sémantique comportementale permet de décrire mathématiquement le comportement des programmes ADeL. La seconde sémantique est opérationnelle ; son implémentation permet la définition d'un compilateur qui génère l'automate de représentation de l'activité, à partir de l'activité décrite par l'utilisateur à l'aide du langage. Ce compilateur peut générer plusieurs formats du modèle de représentation d'activités, dont l'un est utilisé pour faire la vérification et la validation de l'activité en l'envoyant à un model-checker.

Enfin, puisque nous avons choisi de travailler avec le paradigme synchrone, notre troisième contribution porte sur le développement d'un "synchroniseur" dont le rôle est de regrouper les données asynchrones de l'environnement extérieur en des instants logiques pour les envoyer à notre système synchrone.

1.4 Plan du manuscrit

Cinq parties principales structurent ce mémoire. Dans le chapitre 2, nous étudions les définitions du terme “activité” selon les travaux existants, et nous spécifions celle qui nous intéresse dans notre travail. Nous étudions également les techniques existantes pour la reconnaissance d'activités. Nous concluons ce chapitre par la description de l'approche choisie.

Le chapitre 3 présente notre première contribution. Nous comparons l'approche synchrone et l'approche asynchrone pour mettre en avant les avantages attendus du synchrone pour modéliser les activités. Nous étudions ensuite différents langages synchrones et nous expliquons les raisons pour lesquelles nous avons défini un nouveau langage de description d'activités. Nous présentons aussi le langage et ses formats, nous expliquons son utilisation à l'aide d'un cas d'utilisation simple. Nous concluons par un sondage fait auprès de publics divers sur l'utilisation du langage.

Le chapitre 4 décrit notre deuxième contribution. Pour assurer des fondations formelles à ce langage, nous avons conçu deux sémantiques : comportementale et opérationnelle. Nous définissons les règles de ces deux sémantiques et leur contexte mathématique. Nous montrons ensuite la relation entre elles et nous expliquons enfin la compilation et la validation de notre langage.

Le chapitre suivant commence par décrire la vue globale de notre système de reconnaissance et par définir le rôle de chaque composant. Ensuite, nous présentons notre troisième contribution, un composant important nécessaire pour la communication entre un système synchrone et des données asynchrones : le synchroniseur. Nous présentons son architecture, ses stratégies ainsi que son fonctionnement. Nous concluons ce chapitre par un cas d'utilisation qui présente une étude comparative de différentes stratégies.

Le chapitre 6 illustre notre approche avec trois cas d'utilisation issus des problématiques de notre équipe. Ces cas d'utilisation commencent par la description d'une activité, puis la compilation de cette description en générant trois formats : un format pour sa simulation, un format pour sa validation et un autre format qui représente l'automate de reconnaissance à intégrer dans le moteur de reconnaissance.

Enfin, le dernier chapitre conclut cette thèse et identifie des perspectives de futurs travaux de recherche.

Chapitre 2

Reconnaissance d'activités

L'essentiel : Ce chapitre présente les différents types d'activités à reconnaître et définit ce que nous désignons comme "activité" tout au long de notre travail. Plus précisément, nous allons voir qu'une activité représente tout groupe d'actions et/ou d'intéractions et/ou des activités de groupe. Nous décrivons aussi différentes approches pour reconnaître une activité. Il existe deux types d'approches : celles qui traitent les données de bas niveau, et celles qui s'occupent du traitement haut niveau des données. Nous nous intéressons au deuxième type d'approche.

Sommaire

2.1 Introduction

Actuellement, les travaux en reconnaissance d'activités se situent à plusieurs niveaux : la reconnaissance des émotions [DBB+16], la reconnaissance du genre à partir du visage et du sourire [BDB16], la ré-identification des personnes [BBSB12], etc. On trouve surtout des recherches sur la reconnaissance d'actions et d'activités humaines. Les termes "action" et "activité" sont fréquemment utilisés de façon interchangeable. Dans cette thèse nous travaillons spécifiquement sur la reconnaissance d'activités. Il est donc nécessaire de donner une définition exacte à ce que nous appelons activité. Il existe aussi plusieurs techniques pour reconnaître des activités. On trouve des systèmes qui se basent sur le traitement direct (bas niveau) des informations qui proviennent des capteurs de l'environnement. D'autres utilisent des informations de haut niveau pour faire la reconnaissance. Plusieurs systèmes se basent aussi sur des modèles d'activités. Ces modèles présentent la description de l'enchainement souhaité de l'activité. Dans ce chapitre nous précisons la définition du terme activité, nous donnons plusieurs exemples d'approches utilisables pour décrire et reconnaître une activité et nous décrivons notre approche en fin de chapitre.

2.2 Qu'est ce qu'une activité ?

Aggarwal et Ryoo [AR11] présentent 4 catégories d'activités humaines classées selon leur complexité : les gestes, les actions, les intéractions et les activités de groupe.

- **Les gestes**
  Ce sont les "mouvements élémentaires" du corps humain. Ils présentent les composants atomiques qui donnent une description significative aux mouvements de la personne ; "lever la main" ou "lever un pied" sont des exemples de gestes.
- **Les actions**
  Les actions présentent une activité simple d'une personne. Les actions sont composées d'un ensemble de gestes organisés temporellement. On peut citer quelques exemples d'actions comme "marcher", "parler", "boire", "s'asseoir", etc.
- **Les interactions**
  Les interactions sont les activités humaines qui impliquent plusieurs personnes et/ou plusieurs objets. Par exemple : deux personnes qui se serrent la main est une interaction qui implique deux personnes, et un voleur qui vole le sac d'une femme est une interaction qui implique deux personnes et un objet.
- **Les activités de groupe**
  Une activité de groupe est l'ensemble des activités réalisées par des groupes de plusieurs personnes et/ou objets. " un groupe de personnes qui se battent entre elles" ou "un groupe de personnes qui font une réunion" sont des exemples de telles activités.

Dans notre cas, on désigne par *activité* un ensemble d'évènements élémentaires qui sont partiellement ordonnées dans le temps et qui respectent certaines contraintes. Ces évènements élémentaires décrivent les gestes ou les actions d'une ou plusieurs personnes et nous sont fournis par les capteurs. En se basant sur la classification mentionnée ci-dessous, une activité peut comporter :

- Un ensemble d'actions d'une seule personne par exemple : "marcher et compter en même temps" est une activité composée de deux actions qui se déroulent en parallèle.

- Une interaction ou un ensemble d'interactions d'une personne avec un ou plusieurs objets, par exemple : "parler au téléphone" ou "prendre un médicament".
- Une interaction ou un ensemble d'interactions entre plusieurs personnes et/ou des objets, par exemple : " un voleur vole le sac d'une femme" .
- Une activité de groupe, par exemple : "une réunion" peut se décomposer en : une personne entre, une autre personne entre, elles se serrent la main.

2.3 Reconnaissance d'activités

Le but de la reconnaissance d'activités est d'analyser, de comprendre et d'interpréter les mouvements et les actions d'objets animés, en particulier de personnes, pour donner une idée sur leurs comportements, voire leurs intentions.

Les recherches existantes ont réussi à reconnaître les activités simples des personnes. Mais reconnaître une activité complexe reste toujours un challenge. Les types de challenges en reconnaissance d'activités complexes selon [KHC10] sont :

- **la reconnaissance d'activités concurrentes**
  On peut dans certains cas avoir des activités qui se déroulent simultanément. Une personne peut exécuter plusieurs activités en même temps, elle peut cuisiner pendant qu'elle parle à quelqu'un. Ce genre d'activité doit être reconnu avec une approche spécifique, autre que celle qui traite les activités séquentielles par exemple. Le traitement de ces activités parallèles peut engendrer des courses critiques donc du non déterminisme et des comportements de reconnaissance non souhaités.
- **la reconnaissance d'activités entrelacées**
  Certaines activités peuvent être suspendues ou entrelacées pour en faire une autre. Par exemple, si une maman qui cuisine entend son enfant pleurer, elle va mettre en pause la tâche de préparation du plat et va voir son enfant, après elle retourne cuisiner.
- **l'ambiguïté d'interprétation de certaines activités**
  Des interprétations d'activités similaires peuvent être différentes d'une situation à une autre. Par exemple : "prendre une cuillère" peut faire partie de plusieurs activités différentes comme "cuisiner", "préparer un café" ou "prendre un médicament".
- **la reconnaissance d'activités impliquant plusieurs personnes (activités de groupe)**
  On peut avoir plusieurs personnes présentes dans une activité et chacune de ces personnes peut faire une activité en parallèle (par exemple, dans une réunion une personne parle et une autre écrit) ou plusieurs personnes qui font la même activité (par exemple, "une chorégraphie").

Toujours selon Kim [KHC10], comprendre une activité englobe la reconnaissance d'activités et la découverte de modèles d'activités ("activity pattern discovery"). La première approche consiste à détecter des activités en se basant sur des modèles prédéfinis. Par conséquent, dans cette approche on construit d'abord un modèle conceptuel d'activité de haut niveau, puis on met en œuvre le modèle en l'intégrant dans un système de reconnaissance.

Une deuxième approche consiste à trouver des modèles inconnus d'activités directement à partir des données de capteurs de bas niveau, sans aucun modèle et/ou hypothèse prédéfini.

Même si les deux approches sont différentes, elles peuvent être complémentaires : un modèle découvert peut être utilisé ensuite pour reconnaître les activités.

D'autres chercheurs comme Aggarwal *et al.* [AR11] par exemple, englobent les deux approches dans le terme reconnaissance d'activités. De plus, en utilisant une taxonomie des approches, ils présentent plusieurs méthodologies de reconnaissance d'activités et les classifient en deux grandes catégories : les approches à une seule couche et les approches hiérarchiques. Dans leur cas, les approches à une seule couche représentent et reconnaissent les activités humaines directement à partir de séquences d'images. En prenant en considération leur nature, ces approches mono-couches conviennent à la reconnaissance d'actions gestuelles et séquentielles. D'un autre côté, les approches hiérarchiques représentent les activités humaines de haut niveau en les décrivant en termes d'autres activités plus simples, que Aggarwal *et al.* appellent des sous-événements. Les systèmes de reconnaissance à plusieurs couches sont construits pour faciliter l'analyse d'activités complexes.

Chacune de ces deux catégories est classée en plusieurs types. Cette classification est présentée dans la figure 2.1 [AR11] :

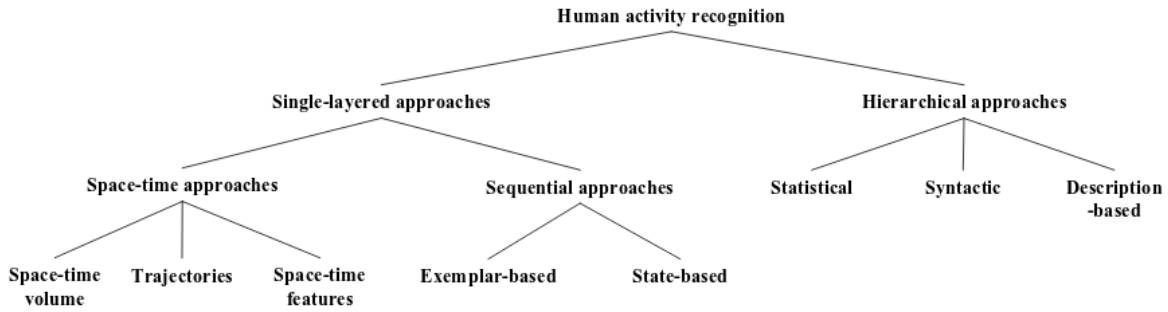


FIGURE 2.1 – Classification basée sur la taxonomie des approches de reconnaissance d'activité

Dans notre cas, comme Kim [KHC10], nous considérons que le traitement de données pour la reconnaissance d'activités se fait en deux niveaux :

— D'abord, le traitement des données provenant des images, des séquences vidéos ou de différents capteurs (audio, binaires, analogiques...). Ce traitement permet le filtrage et l'extraction de caractéristiques nécessaires à la reconnaissance (appelées aussi évènements de bas niveau), il englobe le calibrage de la scène, la détection automatique d'objets et leur suivi, leur reconnaissance et leur classification.

— Ensuite, ces informations de bas niveau sont recueillies et transformées en entrées d'un moteur de reconnaissance d'activités. Nous considérons ces entrées comme des "événements primitifs" ou des situations (exemple : une personne dort, mange, regarde la télévision ...). Le traitement haut niveau analyse et interprète ces évènements, à partir desquels on arrive à reconnaître une activité complexe.

En respectant la classification de Kim, nous avons fait une étude non exhaustive de différentes techniques utilisées dans chaque approche.

2.4 Techniques de reconnaissance

Il existe plusieurs approches dans la littérature qui peuvent être utilisables pour reconnaître une activité. Depuis les années 60, des modèles de systèmes évoluant à temps discret ou continu ont été introduits afin de les analyser. Ensuite, des techniques de description de scénarios sont apparues pour décrire le cycle de vie d'un système et se sont révélées intéressantes pour faire de la reconnaissance en se basant sur des modèles. Plus récemment, des techniques d'apprentissage automatique se sont largement imposées.

2.4.1 Modèles formels applicables dans les systèmes de reconnaissance

Les modèles formels, en particulier mathématiques, permettent non seulement de décrire une activité mais aussi de la reconnaître et de la valider.

Les réseaux de Petri

Les réseaux de Petri [Mur89] ont été définis par Carl Petri comme un outil graphique et mathématique pour décrire des relations entre des conditions et des événements, représentées par des diagrammes de transition d'états. Ils présentent des graphes bipartites [OSPI16] composés de deux types de nœuds : les places et les transitions. Les places (représentées par des cercles) se réfèrent à l'état local d'une entité et les transitions (événements - représentées par des barres) indiquent les changements d'état de l'entité. On trouve aussi des liens PT (pré-conditions, représentées par des flèches provenant des places et se terminant aux transitions), des liens TP (post-conditions, représentées par des flèches provenant de transitions et se terminant aux places), et des jetons (caractérisant l'état local actif et représentés par des points)(voir figure 2.2) [1].

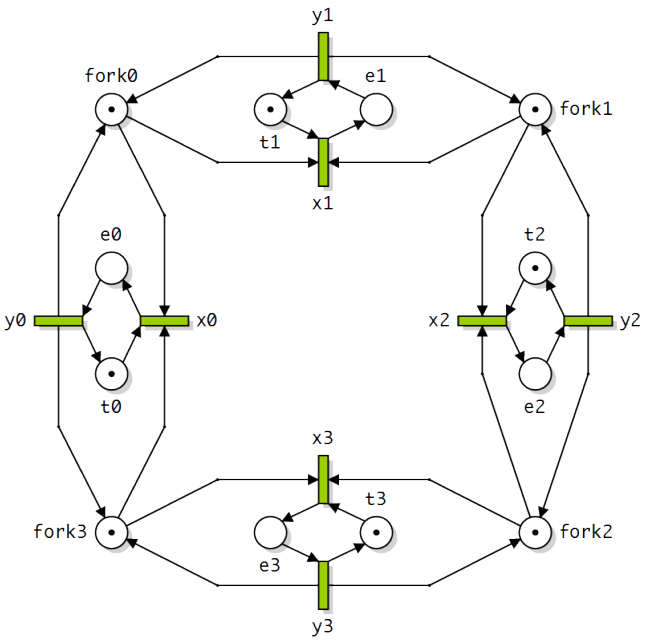


FIGURE 2.2 – Problème du dîner des philosophes modélisé à l'aide d'un réseau de Pétri

1. ParRp22\T1\textemdashTravailpersonnel,CCBY-SA3.0,https://commons.wikimedia.org/w/index.php?curid=4372927

Les réseaux de Petri sont utiles pour représenter différents systèmes, modéliser et visualiser des comportements dynamiques tels que la concurrence, la synchronisation et le partage de ressources et plus généralement tous les problèmes de vivacité ("liveness"). Ils sont utilisés pour la modélisation des jeux [FO14] et aussi pour la reconnaissance d'activités [ACM+08, LRR13]. Le problème avec les réseaux de Petri est que leurs graphes de marquages [MG17] peuvent devenir trop grands pour analyser tous les états du système, ce qui rend le modèle complexe et difficile à analyser [GBA16]. L'algorithme de dépliage ("unfoldings") de MacMillan [MP95] vise à construire un préfixe complet minimal représentant tous les marquages atteignables d'un réseau de Petri. Ces préfixes ont été introduits principalement pour résoudre le problème d'explosion d'états rencontré lors de la vérification de certaines propriétés de réseaux de Petri comme l'absence d'interblocage ("deadlock"). Toutefois, l'algorithme proposé par MacMillan peut générer des préfixes d'une taille considérablement supérieure à celle qui serait souhaitable pour le traitement.

Les réseaux de Petri temporels

Cette catégorie de réseaux de Petri a été introduite la première fois en 1974 par P.M. Merlin dans sa thèse [Mer74] pour modéliser des systèmes asynchrones dépendant de contraintes de temps. Dans ces réseaux, on introduit des intervalles de temps. On distingue essentiellement 2 types de réseaux de Petri temporels (RdPT) :

1. Les réseaux de Petri P-temporels : les intervalles de temps sont associés aux places, un intervalle de temps [a,b] associé à une place représente le temps de séjour minimum et maximum qu'un jeton doit passer dans la place. C'est un formalisme puissant pour spécifier et analyser des systèmes à contraintes de temps.
2. Les réseaux de Petri T-temporels : les intervalles de temps sont sur les transitions. Ils ne modélisent que les contraintes de temps minimum ou borné. Ils sont dédiés à la spécification des protocoles et des processus temporels. Ils sont traduisibles structurellement en produits synchronisés d'automates temporisés.

Il existe aussi des réseaux de Petri A-temporels où les intervalles de temps sont sur les transitions. L'intervalle de temps donne alors la date au plus tôt du tir et la date au plus tard.

Pour vérifier les réseaux de Petri temporels, une boite à outils, Tina (Time Petri Nets Analyser) [BRV04], a été développée au LAAS à Toulouse [2]. Les techniques utilisées dans Tina sont des techniques issues du model-checking. Pour construire les modèles sur lesquels les propriétés seront vérifiées, Tina s'appuie sur différentes méthodes d'abstraction qui garantissent la préservation de certaines classes de propriétés. Ces abstractions sont indispensables, car en général, les espaces d'états des RdPT sont infinis. Tina offre des méthodes de vérification pour des propriétés de logique temporelle LTL et Mu-Calcul. De plus, Tina propose une exportation vers CADP [GLMS11] et MEC [GV04], deux boites à outils proposant différents model checkers pour diverses logiques temporelles et le Mu-Calcul [HP72] ainsi que des algorithmes efficaces de vérification.

2. http://www.laas.fr/tina

Les automates temporisés

Les automates temporisés sont un modèle des systèmes réactifs à temps continu, proposé par Alur et Dill en 1991 [AD94]. Ce sont des machines d'états finis étendues avec un ensemble fini d'horloges, utilisées pour exprimer des contraintes de temps sur les transitions. Les horloges peuvent être remises à zéro et leurs valeurs augmentent uniformément avec le temps. A chaque instant, la valeur d'une horloge correspond au temps passé depuis sa dernière remise à zéro. Une transition est déclenchée uniquement si sa contrainte de temps est satisfaite par les valeurs de ses horloges. Les modèles des automates temporisés sont des systèmes de transitions étiquetées avec un ensemble infini d'états, ce qui rend *a priori* leur analyse directe par des techniques de model-checking impossible. Toutefois, Alur [AD94] a montré que le model-checking est décidable pour les automates temporisés en utilisant une relation d'équivalence sur les valuations des horloges et en transformant les contraintes sur les horloges en contraintes entières. Chaque classe d'équivalence constitue une région et le graphe d'états du système devient un graphe de régions. Toutefois, ce graphe des régions peut rester difficile à analyser car on a souvent un nombre exponentiel de régions. Une solution est de manipuler des unions convexes de régions : les zones. Ces dernières permettent de représenter efficacement le comportement temporel du système et le graphe d'états pourra être remplacé par le graphe des zones. Le processus d'analyse d'un outil comme Uppall [LPY97] permet la vérification "à la volée" de propriétés de la logique temporelle CTL en utilisant des graphes de zones. Un autre model-checker, Kronos [BDM+98], permet de vérifier des propriétés d'une logique temporelle spécifique (TCTL) de systèmes temps réél représentés par des automates temporisés. TCTL est une extension de la logique temporelle CTL qui permet un raisonnement temporel quantitatif sur le temps. Toutefois, même si l'utilisation du graphe de zones permet une représentation de l'espace d'état plus compacte, la représentation des zones en mémoire reste dépendante du nombre de contraintes sur les horloges et les processus d'analyse peuvent rester complexes.

Le modèle de Markov caché (Hidden Markov Model (HMM))

Les chaines de Markov [Nor97] se présentent sous la forme d'automates probabilistes et elles permettent d'identifier certains comportements et propriétés des systèmes que l'on veut modéliser. Dans les chaines de Markov, les transitions des automates sont cachées et leurs états sont connus de l'utilisateur.

Les modèles de Markov cachés sont une dérivée des chaines de Markov. Dans un modèle de Markov caché, les états des automates sont caché *et* inconnus de l'utilisateur. Chaque état génère des symboles ("observations") en s'appuyant sur une certaine loi de probabilité, et ce sont ces symboles qui sont observables. Donc, avec les modèles cachés de Markov, on ne traite pas les séquences d'états du modèle mais la séquence de leurs observations générées. Plus précisément, un objectif principal de ce modèle est de déterminer la séquence d'états cachés (y1,y2 ... yt) à partir de la séquence de sortie observée (x1, x2,..., xt) que ces états ont généré. Cette observation rend les HMM capables de construire progressivement le modèle d'un système, qui peut être ajusté, étendu et réutilisé.

Par exemple, dans la figure 2.3 [KHC10], les auteurs utilisent le modèle de Markov caché pour reconnaitre l'activité "manger". Dans cet exemple (partie (a)), les états en bleu sont des états cachés qui représentent des activités simples ("Have Soup", "Cut

Steak", "Drink", "Pick Food"). Pour permettre de les reconnaitre, ces états génèrent une séquence d'observations comme dans l'exemple (partie (b)) : l'état caché de l'activité "Have soup" génère l'observation "Spoon", l'état caché de l'activité "Cut Steak" génère l'observation "Knife", etc.

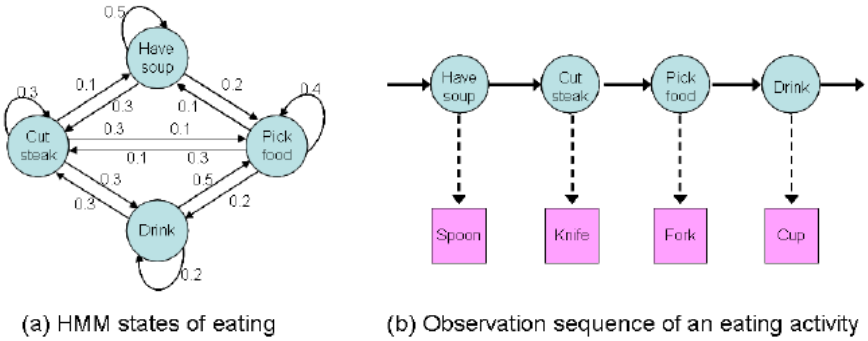


FIGURE 2.3 – Un exemple HMM de l'activité "manger"

Les modèles de Markov cachés sont utilisés dans différents domaines comme la reconnaissance d'activités [KD16, SM15], la reconnaissance de la parole [RBD03], la reconnaissance d'écriture manuscrite [MMZ18], ou même la médecine [NKCN15].

L'avantage principal de ces modèles est d'utiliser les probabilités pour traiter les observations. Cette possibilité serait utile dans notre approche et nous envisageons de l'introduire dans le futur.

Les ontologies

Partant d'un point de vue différent, les ontologies et les théories logiques associées peuvent aussi être un moyen formel pour décrire une activité. Les ontologies ont été développées depuis 1973 grâce à F. Wolff [LLD07, Cor18]. Dans [Cor18], une des définitions d'ontologie est la suivante : *"an ontology is a theory concerning the kinds of entities and specifically the kinds of abstract entities that are to be admitted to a language system".*

L'utilisation des ontologies dans la modélisation et la reconnaissance d'activités a suscité un intérêt, mais les travaux étaient focalisés principalement sur les activités simples et principalement les relations statiques à l'intérieur des activités [CNW12, CN09, LLD07]. De nouveaux travaux comme [OCW14] ont introduit des relations temporelles et ainsi réussi à utiliser les ontologies pour modéliser et reconnaître des activités composées, en combinant les connaissances ontologiques et temporelles pour créer des modèles d'activité représentant les dépendances inter-activités (en utilisant des relations temporelles). Cette approche améliore les modèles d'activités ontologiques en ajoutant des connaissances fondées sur les relations logiques temporelles d'Allen [All83]. L'utilisation de cette approche a abouti à un taux encourageant de précision (88.26% pour les activités composites).

Conclusion

Les modèles présentés dans cette section sont des modèles généraux que l'on pourrait utiliser pour représenter des activités et en déduire leur reconnaissance tout en assurant la validité de certaines propriétés. Toutefois, leur manipulation est peu pratique pour décrire et modifier des activités complexes, surtout pour des non-experts de ces formalismes.

Les réseaux de Petri sont un modèle de haut niveau destiné principalement à la simulation et à l'analyse de propriétés de vivacité et de sureté (par exemple non blocage) de systèmes à événements discrets. Nous pourrions faire des preuves grâce à ce modèle mais il nous semble difficile d'en déduire un "reconnaisseur" de l'activité ce qui rend la tache de reconnaissance compliquée. Les versions temporelles de ces modèles (RdPT et automates temporisés) incluent explicitement le temps, ce qui permet de faire des preuves impliquant le temps. Des outils existent (Tina, Kronos, Uppal) pour faire ces preuves mais comme nous l'avons déjà évoqué leur mise en œuvre peut être difficile. Les problèmes d'accessibilité d'un nœud et l'existence d'une borne temporelle d'exécution d'un réseau de Petri sont indécidables. Une construction analogue au graphe des régions pour les automates temporisés est envisageable mais ce graphe peut être infini si le réseau n'est pas borné. Certes, certains RdPT peuvent être traduits en un ensemble d'automates temporisés synchronisés, mais la vérification sur ce modèle est elle aussi complexe. Par exemple, dans [GDAB05], les auteurs font face au problème de vérification de propriétés temporelles pour des systèmes distribués temps réel dans le domaine de l'automobile. Afin d'utiliser Uppall, ils doivent définir tout un ensemble d'abstractions leur permettant d'arriver à leurs fins.

Intégrer un certain degré d'incertitude lié aux capteurs et ayant une influence sur la présence des évènements écoutés est un point important dans notre approche. Dans cette thèse, nous n'avons pas envisagé d'en tenir compte mais c'est un passage obligatoire pour la suite de ces travaux et des modèles comme les HMM seront très utiles.

Concernant les ontologies, nous pouvons envisager d'en utiliser selon le domaine d'application pour décrire la partie "données" de notre langage, c'est à dire les rôles, plus précisément les relations entre les objets participant à la description d'activité. Cependant, les ontologies ne couvrent pas tous nos besoins pour les aspects dynamiques et en terme de généricité de description d'activités pour différents domaines.

2.4.2 Outils graphiques utilisables dans la reconnaissance d'activités

Des outils, dont certains dérivent des approches précédentes, ont été développés et largement utilisés dans le monde industriel.

Les grafcets

Le GRAFCET (Graphe Fonctionnel de Commande des Étapes et Transitions) [Gen00, IEC13] est un mode graphique de représentation et d'analyse du comportement d'un système, généralement automatisé, qui fonctionne en logique séquentielle. Il est dérivé du modèle mathématique des réseaux de Petri [DA92, Gen00]. Un grafcet est composé des éléments suivants (voir figure 2.4) [3].

Des étapes avec des actions associées : une étape est indiquée par un carré identifié par un numéro. Une étape active peut être marquée par un point au-dessous du numéro. Une étape initiale est mentionnée par un double carré. Les actions associées sont représentées d'une façon symbolique ou littérale, dans un rectangle relié à la partie droite. Une action s'exécute quand l'étape correspondante est active.

Des liaisons orientées entre les étapes, appelées aussi arcs, relient les étapes et les transitions. Elles sont implicitement orientées du haut vers le bas. Dans le cas contraire,

3. http://philippe.berger2.free.fr/automatique/cours/G7/le_grafcet.htm

elle doivent être indiquées par une flèche.

Des transitions présentent les conditions de changement d'états d'un système. Elles sont représentées par un trait horizontal placé entre deux étapes. Le passage d'un état du système à l'état suivant, respectivement d'une situation (ensemble des étapes actives) à la situation suivante, correspond au franchissement simultané d'un ensemble de transitions.

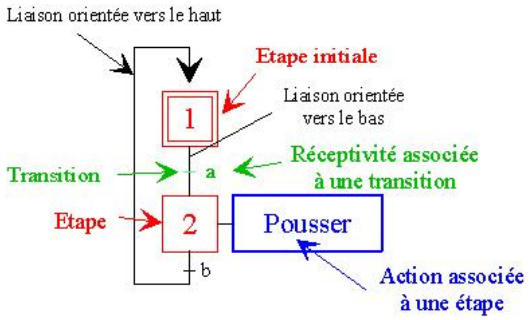


FIGURE 2.4 – Composition de grafcet : un exemple simple de grafcet unidirectionnel

Les grafcets constituent à la fois un langage de programmation et un modèle de représentation des comportements des systèmes et peuvent aussi exprimer le parallélisme, la concurrence et d'autres mécanismes utilisés dans les systèmes de contrôle. Lors de leur définition en 1977, aucune sémantique formelle n'a été imposée. En effet, la définition des grafcets est issue d'un consensus entre des industriels ayant préalablement défini leur propre langage métier, des chercheurs voulant harmoniser les représentations des machines d'états et des enseignants dont l'attente était clairement pédagogique. Cependant, des postulats doivent être respectés et des règles d'évolution également pour interpréter un grafcet. Il existe deux algorithmes d'interprétation des grafcets : avec recherche de stabilité (ARS) et sans recherche de stabilité (SRS). L'algorithme SRS n'assure pas le déterminisme du modèle. L'algorithme ARS assure le déterminisme mais peut entrainer des situations de blocage (si la stabilité n'est pas atteinte) qui sont difficiles et couteuses à vérifier. Ces inconvénients empêchent d'utiliser des méthodes d'analyse formelle, comme le model-checking, pour vérifier si un grafcet se comporte comme prévu une fois qu'il est construit.

Des chercheurs comme P. Leparc [Lep94] et D. Gaffé [Gaf96], ont réussi à résoudre ce problème de sémantique en donnant une sémantique formelle synchrone à des catégories de grafcets. Dans [PRF11], ce problème a été aussi résolu en fournissant une sémantique formelle à une catégorie de grafcets. Ceci est réalisé en représentant le comportement d'un modèle de grafcets sous la forme d'une machine à états finis, nommée Stable Location Automaton (SLA).

Statecharts

L'objectif principal des Statecharts [Har87a, Pro08] est de modéliser le cycle de vie d'un objet, de sa création à sa fin, sous forme de diagrammes d'états hiérarchiques. Ils servent à décrire le comportement des systèmes réactifs en utilisant des notations graphiques. Ils sont aussi une extension du formalisme classique des machines à états finis (FSM) et des diagrammes de transition d'états en incorporant les notions de hiérarchie, de parallélisme, d'événements composés et un mécanisme de diffusion pour la communication entre composants concurrents. Les Statecharts fournissent une notation

graphique efficace, non seulement pour la spécification et la conception de systèmes réactifs, mais également pour la simulation du comportement du système modélisé.

Depuis l'introduction des Statecharts il y a une trentaine d'années, des progrès importants ont été accomplis en ce qui concerne leur sémantique, leur analyse formelle et leur mise en œuvre.

Des variantes de Statecharts ont été développées et ont également été intégrées au langage de modélisation UML. Dans ce cas, ils décrivent différents états d'un composant dans un système.

Aujourd'hui, les Statecharts sont aussi supportés par plusieurs outils commerciaux, comme Matlab Simulink / Stateflow, Statemate ou Rational Rose.

Message Sequence Chart (MSC)

Messages Sequence Charts (MSCs) [GGR93, GMP04] est un langage de spécification qui permet de décrire les activités et les comportements des systèmes via des représentations graphiques simples, basées sur des lignes de vie et des messages échangés entre entités communicantes. Il a été standardisé dans les années 90 par ITU (International Telecommunication Union) dans sa recommandation Z.120 [ITU] La figure 2.5 montre une description d'une opération de retrait d'argent [4] auprès d'un Distributeur Automatique de Billets (DAB) avec les Message Sequence Charts. Ce diagramme décrit une séquence des messages entre l'utilisateur représentée par la composante "Utilisateur", le DAB représenté par la composante "ATM" et le système de base de données de la banque représenté par la composante "BANK". L'utilisateur insère sa carte bancaire, entre son code confidentiel, le DAB vérifie le code PIN avec la base de données de la banque. Le message "processing" signifie que le guichet automatique affiche un message indiquant qu'il traite le code PIN. Une fois que la base de donnée de la banque a indiqué que le code PIN est valide, le DAB présente les choix de transaction ("option") et l'utilisateur sélectionne le retrait. Le guichet automatique demande ensuite le montant à retirer.

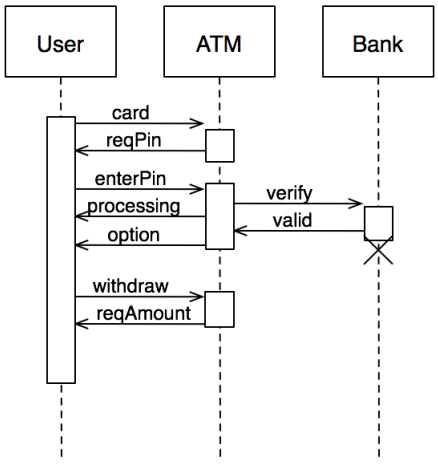


FIGURE 2.5 – Description d'une opération de retrait d'argent avec MSC

Il est également possible de coordonner les activités en utilisant des automates d'états finis MSC. Cette représentation est appelée High-level Message Sequence Charts (HMSC) et prend en charge la composition parallèle des activités ou des systèmes [GGH+07, AY99] (voir figure 2.5).

4. https://www.ics.uci.edu/~alspaugh/cls/shr/msc.html

Les travaux de recherche de Dan [DHC12], identifient des incohérences (appelées "pathologies") dans les MSCs : il peut y avoir des comportements incorrects à cause de certains défauts de configuration d'interactions et de spécifications de MSC. Par exemple, les problèmes de "course" peuvent provenir de la divergence entre l'ordre visuel défini par la sémantique et l'ordre imposé qui peut changer selon les implémentations, puisqu'il dépend des causalités décrites dans la spécification MSC ou (dans le cas) de la communication sous-jacente du système. Toutefois, ces problèmes sont facilement détectés dans les MSCs. Une autre pathologie provient de possibles choix entre les événements provenant d'autres processus. Une solution pour éviter cette pathologie est la vérification du modèle. Dans [AY99] les chercheurs illustrent les problèmes de vérification de modèles MSCs pour des interprétations synchrones et asynchrones et suggèrent différentes techniques pour résoudre ces problèmes de contrôle de modèle telles que l'utilisation de la théorie des automates ou la définition d'algorithmes pour éviter une dérive en temps et en mémoire de la vérification/du model-checking de certaines sémantiques de MSC.

Live Sequence Chart (LSC)

Un autre langage de modélisation et de spécification est le Live Sequence Charts (LSC) [DH99, BDK+04]. Il représente une extension des MSC et des diagrammes de séquence d'UML2 (voir figure 2.6), mais il est plus expressif et sémantiquement plus riche, ce qui le rend utile dans différentes étapes de développement de logiciels et de vérification de processus, et pour toutes sortes d'applications telles que les applications Web [LGS16].

Dans la figure 2.6 [BHS05], nous montrons un exemple de LSC universel (uLSC) décrivant le fonctionnement d'une machine à café. Comme montré dans la figure, ce diagramme a deux composants de base appelés *charts*. Le composant supérieur est entouré d'un hexagone en ligne pointillée et est nommé *prechart*, le composant inférieur est appelé *main chart* et est entouré par une ligne continue rectangulaire. Les événements du *prechart* sont des évènements d'activation et les réponses sont dans le *main chart*. On peut avoir des événements supplémentaires de restriction indiqués par la clause *restricts*. L'interprétation des LSC spécifie que l'exécution du *prechart* implique les réponses du *main chart*.

LSC a une sémantique formelle permettant l'analyse et la vérification. Le model-checking est possible mais difficile, même pour les diagrammes simples. Cependant, [KMB09] propose une solution plus efficace pour le model-cheching, mais seulement pour une classe particulière de LSC. Les modèles LSC sont utilisés pour spécifier le comportement des systèmes séquentiels ou parallèles. Ils peuvent également être transformés en automates [HK01, LGS16], ce qui aide à vérifier et tester des scénarios en utilisant une méthode de recherche en profondeur d'abord.

UML : Unified Modeling Language

Le langage UML (Unified Modeling Language) [RJB99, SBC+15] est un language de modélisation graphique très utilisé proposant un standard de modélisation et de représentation d'architecture logicielle des systèmes. UML tire ses racines des méthodes de programmation orientées objets. C'est un standard adopté par l'Object Management Group (OMG). Il est basé à l'origine sur les notations de la méthode Booch, la technique de modélisation d'objet (OMT) et l'ingénierie logicielle orientée objet (OOSE), qu'il

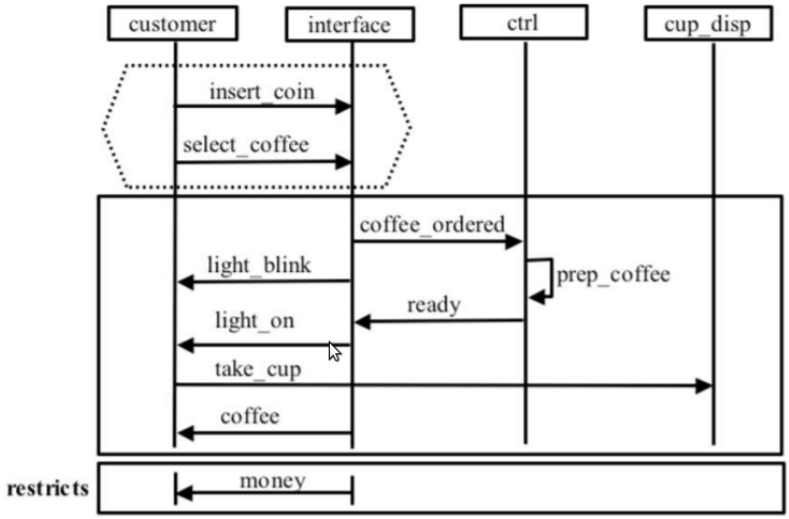


FIGURE 2.6 – Un exemple LSC universel décrivant le fonctionnement d'une machine à café [BHS05]

a fusionné dans un seul langage en 1995. Aujourd'hui, plusieurs autres modèles ont été intégrés dans UML comme les Statecharts, les MSC et les LSC. UML possède un ensemble intégré de différents types de diagrammes permettant de considérer un système sous différents points de vue dans plusieurs phases de son cycle de vie. On trouve plusieurs catégories de diagrammes :

— les diagrammes de structure ou diagrammes statiques
Ces diagrammes montrent la structure statique du système et de ses composants à différents niveaux d'abstraction et de mise en œuvre, et comment ils sont liés les uns aux autres. Les éléments d'un diagramme de structure représentent les concepts significatifs d'un système et peuvent inclure des concepts abstraits, concrets et d'implémentation. Il existe sept types de diagramme statiques.

— les diagrammes d'interaction ou diagrammes de comportement dynamiques
Ces diagrammes montrent le comportement des objets dans un système, qui peut être décrit comme une série de modifications apportées au système au fil du temps, il existe sept types de diagrammes de comportement dont certains sont basés sur les Statecharts, les LSC et les MSC présentés dans cette section.

UML est extensible et adaptable grâce à la notion de "profiles". Il permet ainsi de définir des langages-métiers. Par exemple, UML a été étendu pour modéliser les systèmes temps-réel : UML-RT est un "profile" dérivé de ROOM (Modélisation orientée objet en temps réel) [SGW94] et de l'UML.

Conclusion

Ces méthodologie graphiques permettent de décrire des activités et sont intéressantes dans notre approche. Toutefois, comme la plupart des formalismes graphiques, elles ont certaines limites d'utilisabilité.

Les grafcets en général n'ont pas de sémantique formelle normalisée. Les deux algorithmes d'interprétation existants montrent tous les deux des inconvénients (non déterminisme ou possibilité de blocage). Le grafcet est plutôt destiné à la conception des commandes des systèmes automatisés. En conséquence, l'aspect validation important

pour nous l'est aussi dans le grafcet. Pour cela, certains auteurs ont traduit le grafcet en langage synchrone Signal [VA98] ou bien utilisé des automates temporisés [L'H97]. D'une part, quitte à utiliser des modèles intermédiaires pour transcrire des systèmes de reconnaissance décrits en grafcet, nous pourrions utiliser directement ces modèles. D'autre part, nous expliquons dans ce manuscrit pourquoi Signal et les automates temporisés ne répondent pas à nos besoins (voir sections 2.4.1 et 3.3.1).

UML présente divers avantages pour la modélisation de systèmes. C'est un langage de grand pouvoir expressif. La plupart de ses diagrammes sont faciles à comprendre pour des utilisateurs non informaticiens, mais certains peuvent nécessiter une bonne maîtrise des relations entre leurs composants. On peut citer par exemple le débat sur la compréhension des relations <<include>> et <<extend>> du diagramme de cas d'utilisation. Nous nous intéressons dans notre cas aux diagrammes d'activités. Il permettent la description d'une activité à l'aide de ses composants qui est facile à comprendre par un utilisateur non-informaticien. Cependant, un diagramme d'activité peut devenir excessivement complexe. Ce type de diagramme nous a beaucoup inspiré dans la création de la partie graphique de notre langage. D'un autre côté, UML est un langage de description, à partir duquel on génère un code en général partiel alors que nous cherchons une génération entièrement automatique.

Les Statecharts peuvent être aussi un bon moyen pour décrire les activités. Ils rendent les systèmes compréhensibles et fournissent une bonne vue d'ensemble d'un système. Un avantage des Statecharts est leur intuitivité, mais la modélisation graphique d'applications réalistes donne souvent des Statecharts complexes ingérables. Il en résulte des graphiques très volumineux, compromettant leur lisibilité, leur utilisation pratique et leur simulation [Pro08]. Par conséquent, les erreurs peuvent être difficiles à localiser, même lors d'une simulation. Cela compromet l'utilisation pratique du formalisme Statecharts. Il faut aussi noter que les Statecharts ont plusieurs sémantiques (Statemate, Rhapsody, etc.) ce qui a pour conséquence une différence d'interprétation d'un modèle suivant les différentes sémantiques.

Toutefois, de manière générale, les diagrammes de séquence, les MSC et les LSC servent principalement à décrire des scénarios individuels (un flot de contrôle et ses variantes simple mais pas l'ensemble d'une activité complexe), dans notre cas nous voulons décrire une activité, c'est à dire tous les scénarios possibles de réalisation de cette activité. Le formalisme des diagrammes de séquences imposerait une description exhaustive de chaque scénario, notre approche se veut plus compacte.

2.4.3 Techniques de reconnaissance basées sur l'apprentissage

Il existe plusieurs techniques pour décrire une activité selon la deuxième approche de reconnaissance d'activités proposée par Kim. Ce sont principalement des méthodes probabilistes qui traitent les données de bas niveau. Certaines de ces techniques sont discutées ici.

L'apprentissage automatique

"*Machine Learning : Field of study that gives computers the ability to learn without being explicitly programmed.*" (Arthur Samuel, 1959) [AK15]

L'apprentissage automatique ou *machine learning* est un domaine de l'intelligence artificielle qui permet aux systèmes d'apprendre et d'améliorer automatiquement l'expérience sans être explicitement programmés. L'apprentissage automatique est

l'un des domaines de l'informatique dont la croissance d'utilisation et l'évolution sont rapides [SSBD14]. Le terme "apprentissage automatique" se réfère à la détection automatisée de motifs significatifs dans les données. Il se concentre sur le développement d'algorithmes utilisant ces motifs pour la reconnaissance. Il existe plusieurs types d'algorithmes en apprentissage automatique qui s'appuient sur des modèles de calcul existants depuis longtemps, nous en citons trois principaux.

• **Les arbres de décision (Decision trees)**

Un arbre de décision [KJS18] est un arbre graphique représentant les résultats possibles selon un ensemble de choix (interconnectés) (voir figure 2.7) [5]. Il permet l'évaluation de différentes actions/choix possibles en fonction de plusieurs paramètres comme la probabilité, le coût, etc, et détermine ensuite le meilleur choix de façon mathématique.

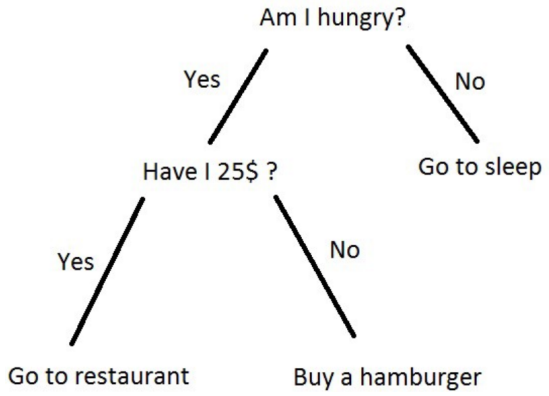


FIGURE 2.7 – Un exemple simple d'arbre de décision, les questions dans cet arbre représentent les nœuds internes, les "yes/no" sont des variables d'entrées et les actions de fin sont les feuilles de l'arbre

L'apprentissage par arbre de décision est une méthode qui se base sur l'utilisation d'un arbre de décision comme modèle prédictif automatisé. C'est une méthode classique en apprentissage automatique [NAPS11].

Les arbres de décisions sont faciles à comprendre, extensibles, et peuvent aider à prendre la meilleure décision, cependant, l'utilisation d'un grand nombre de choix et de variables peut conduire à un arbre très complexe. De plus, cette représentation n'est pas très appropriée pour décrire des activités dynamiques.

• **Les forêts aléatoires (Random forests)**

"To say it in simple words : Random forest builds multiple decision trees and merges them together to get a more accurate and stable prediction" [6].

Les forêts aléatoires ont été introduites en 2001 par L.Breiman [NEM09]. Elles comportent un certain nombre d'arbres décisionnels sur des sous-ensembles de données légèrement différents (appelés échantillons bootstrap), sur lesquelles est appliqué un apprentissage basé sur des méthodes d'inférences statistiques (appelées "bagging").

Les forêts aléatoires ont été abondamment utilisées dans le domaine de la reconnaissance d'activités [HCHP18, NFP17, FML15]. En effet, elles assurent une précision relativement élevée par rapport aux autres algorithmes de classification, elles peuvent gérer de grands ensembles de données, elles peuvent gérer les données

5. https://becominghuman.ai/understanding-decision-trees-43032111380f
6. https://towardsdatascience.com/the-random-forest-algorithm-d457d499ffcd

manquantes et elles sont rapides par rapport aux autres algorithmes. Cependant, elles présentent des limites dans le cas de très grands ensembles de données. Les expériences dans [Lou14] montrent que le sous-échantillonnage simultané des échantillons et caractéristiques augmente la performance, mais consomme plus de mémoire, ce qui aboutit à l'obligation d'utiliser un nombre limité d'arbres sinon on aura comme effet un calcul de données lent et une mémoire épuisée, ce qui ne convient pas à des systèmes temps réel traitant d'énormes données.

- **Les machine à vecteur de support (Support Vector Machine)**

Les machines à vecteurs de support (SVM) [HB06], appelées aussi séparateurs à vaste marge, ont été proposé par V. Vapnik et C. Cortes en 1995 [CV95]. C'est une classe d'apprentissage qui propose des modèles d'apprentissage supervisés associés à des algorithmes d'apprentissage automatique qui analysent les données utilisées pour résoudre des problèmes comme la classification, c'est-à-dire prédire l'appartenance d'une entrée à une classe/catégorie définie. Ceci se fait en définissant des frontières entre les différentes classes ; en effet, si la frontière est connue, on arrive facilement à définir à quel côté appartient la nouvelle entrée. Les SVM sont des classificateurs linéaires qui ont leur propre façon de définir ces frontières en utilisant la séparation linéaire des données. Pour définir ces frontières, les machines à vecteurs de support reposent sur le principe de maximisation de la marge, ce qui lui permet une grande capacité de généralisation. Un autre problème que les SVM sont capables de résoudre est le problème de la régression statistique.
Cependant, si la performance des SVM est élevée pour un ensemble de données limité, cette performance baisse quand l'ensemble de données grandit.

- **Les réseaux de neurones artificiels**

Les réseaux de neurones [Has95] représentent un système dont la conception est inspirée du fonctionnement des neurones biologiques. Ils sont basés sur un ensemble d'unités ou de nœuds connectés, appelés neurones artificiels. Chaque nœud peut transmettre un signal d'un neurone artificiel à un autre. Un neurone qui reçoit un signal peut le traiter puis le transmettre à des neurones qui lui sont connectés. Dans la plupart des implémentations de réseaux de neurones artificiel, l'information transmise par une connexion entre les neurones artificiels est représentée par un nombre réel, et la sortie de chaque neurone artificiel est calculée par une fonction souvent non linéaire à partir de la somme de ses entrées. Les réseaux de neurones sont généralement optimisés en utilisant des méthodes d'apprentissage probabilistes (comme par exemple les méthodes bayésiennes). Ils permettent une classification rapide de données et fournissent des informations d'entrée au raisonnement logique formel. A partir des réseaux de neurones, un concept particulier de l'apprentissage automatique a été crée : l'apprentissage profond (Deep Learning) [Gra16].

L'apprentissage profond

Le *deep learning* (apprentissage profond) [GBC16] est un concept dérivé de l'apprentissage automatique dans les années 2010. L'apprentissage profond est basé sur la combinaison de couches successives de réseaux de neurones simples. Les techniques de l'apprentissage profond ont permis des évolutions significatives dans plusieurs domaines comme le traitement du son et le traitement d'images, y compris la reconnaissance faciale, la reconnaissance vocale, le traitement automatisé des langages, la classification

des textes (par exemple reconnaissance de spams). Il existe plusieurs architectures dans le *deep learning*. Les plus connues sont les réseaux de neurones convolutifs (CNN) [LB98] et les réseaux de neurones récurrents (RNN) [JM99, SP97] .

Conclusion

Tous les outils d'apprentissage automatique et d'apprentissage profond ont des exigences comme une très grande capacité de calcul et de mémoire pour pouvoir traiter les données en temps-réel. Actuellement, malgré une évolution prodigieuse de ces capacités, l'apprentissage automatique reste limité par la taille des données, d'où une grande consommation de mémoire, ce qui peut ralentir le calcul, empêcher la reconnaissance en temps réel et, dans certains cas critiques, aboutir à des résultats dangereux. De plus, ces outils ne peuvent reconnaître que les données qui ont déjà été apprises, ils ne peuvent pas faire la reconnaissance à partir de nouvelles entrées qui n'existent pas dans leur ensemble de données, ce qui peut donner de mauvais résultats dans certains cas. Enfin, le contexte de notre travail nous permet de disposer de modèles *a-priori* décrits par nos utilisateurs. En effet, nous travaillons avec des médecins qui ont des protocoles bien précis et qui savent décrire les activités qu'ils souhaitent reconnaître, donc nous n'avons pas besoin de découvrir les modèles d'activités. L'apprentissage a donc peu d'intérêt dans notre cas.

2.5 Conclusion

Le premier objectif de ce chapitre était de donner une définition précise du terme "activité". Plusieurs travaux existants ont défini l'activité de différentes façons mais proches l'une de l'autre. En se basant sur ces travaux, nous avons donné notre propre définition de ce terme. Dans notre cas, on désigne par *activité* un ensemble d'évènements élémentaires qui sont partiellement ordonnées dans le temps et qui respectent certaines contraintes. Ces évènements élémentaires décrivent les gestes ou les actions d'une ou plusieurs personnes et nous sont fournis par les capteurs. Plusieurs technologies ont été utilisées pour reconnaître les activités, certaines d'entre elles s'appuient sur le traitement direct de données bas niveau, d'autres utilisent des modèles prédéfinis. Nous avons adopté cette seconde approche dans ce travail. Ces technologies peuvent être efficaces mais elles sont souvent difficiles à manipuler par un utilisateur non-informaticien. De plus, certaines technologies nécessitent de grandes capacités de calcul, ce qui pourrait être un problème dans un système de reconnaissance temps-réel, comme c'est notre cas. Dans cette thèse, nous avons adopté l'approche de reconnaissance d'activités à partir de modèles décrits *a priori*, en écoutant les événements de bas-niveau provenant de l'environnement.

Notre objectif principal est de créer des systèmes de reconnaissance temps-réel et faciles à utiliser par des non informaticiens. Par ailleurs, l'approche synchrone n'ayant pas été utilisée dans ce domaine auparavant, l'originalité de ce travail est aussi d'étendre cette approche à la reconnaissance d'activités. L'utilisation du modèle synchrone dans cette thèse, comme nous allons le détailler dans le chapitre suivant, permet de décrire, d'analyser, et de vérifier ces systèmes, en assurant le déterminisme et en résolvant le problème de la concurrence par la composition parallèle déterministe. Les problèmes particuliers de concurrence critique sont détectés par analyse statique et par des techniques de vérification sur le modèle (model-checking). De plus, les modèles

synchrones constituent une bonne solution pour réduire la complexité de tels systèmes et caractériser leurs évolutions.

Dans le chapitre suivant, nous introduisons explicitement cette approche synchrone et nous montrons comment nous l'avons utilisée pour définir un langage description d'activités et sa sémantique.

Chapitre 3

ADeL : Activity Description Language

L'essentiel : Ce chapitre décrit la première contribution de cette thèse. Nous justifions notre choix de créer un nouveau langage synchrone. Nous présentons notre langage "ADeL" (Activity Description Language) en définissant ses concepts, ses opérateurs et ses deux formats, graphique et textuel. Nous présentons aussi un cas d'utilisation pour illustrer ces formats. Finalement, nous présentons un sondage sur l'utilisabilité et l'acceptabilité de ce langage.

Sommaire

3.1 Introduction

Nous considérons les systèmes de reconnaissance d'activités comme des systèmes temps-réel réactifs. Pour les implémenter, il existe deux approches : synchrone ou asynchrone. Chaque approche a des avantages et des inconvénients. Nous aurions pu utiliser l'approche asynchrone car elle permet d'écouter directement les données provenant de l'environnement extérieur. Toutefois, nous avons une bonne expérience avec l'approche synchrone et nous voulions l'appliquer et étudier son utilité pour construire des systèmes de reconnaissance d'activités. Il reste à décider comment il est possible de modéliser des activités : soit en utilisant un des langages synchrones existants, soit en créant un nouveau langage. Il existe plusieurs langages synchrones, comme Esterel, Lustre, Scade ou Signal, mais ces langages sont plutôt destinés à des informaticiens. Dans notre cas, nous souhaitons travailler avec des utilisateurs non informaticiens. C'est ce qui nous a motivé à créer un nouveau langage. Pour ce faire, nous avons choisi de collaborer avec des ergonomes pour en définir les concepts de base et bénéficier de leur expérience pour définir une interface utilisateur adaptée.

3.2 Systèmes temps-réel

"*En informatique temps réel, le comportement correct d'un système dépend, non seulement des résultats logiques des traitements, mais aussi du temps auquel les résultats sont produits*" [Stankovic [Sta88]].

Les systèmes temps réel [Liu00] sont des systèmes souvent réactifs c'est-à-dire qui dépendent de leurs environnements et sont en interaction permanente avec eux. Ils doivent répondre aux stimuli de ces derniers en respectant certaines contraintes dont la plus importante est la contrainte temporelle. En effet, les systèmes temps réel sont des systèmes dont le temps de réaction doit être évaluable ou borné. De plus, leurs réactions aux stimuli peuvent avoir un effet sur l'environnement.

On distingue généralement trois catégories de systèmes temps réel :

— Les systèmes temps réel durs ou critiques (Hard Real Time) dont la réponse est indispensable au respect de la fonctionnalité ; elle doit être exacte et arriver dans un laps de temps donné.

— Les systèmes temps réel mous ou souples (Soft Real Time) dont la réponse tardive n'a pas d'effets dangereux, mais perd son intérêt au fur et à mesure qu'elle dépasse un temps limite.

— Les systèmes temps réel fermes (Firm Real Time) qui doivent répondre aux stimuli de l'environnement, mais leur réponse sera inutile si on dépasse un temps limite.

Dans notre thèse, nous ne prenons pas vraiment en compte ces différents types car les activités que nous souhaitons reconnaître peuvent être critiques, souples ou fermes. Par exemple "une personne âgée tombe" est une activité critique, "une personne âgée doit boire de l'eau un certain temps après avoir pris son médicament" est une activité ferme.

3.2.1 L'approche synchrone

L'approche synchrone vient du monde des circuits. Le calcul synchrone a été introduit en informatique par Robin Milner, lorsqu'il a établi ses théories sur le calcul des systèmes communicants : SCCS (Synchronous Calculus of Communicating Systems) en 1980 [Mil80, Mil83] [1]. Puis, le modèle et les langages synchrones sont aussi apparus, indépendamment et simultanément, au début des années 80, dans différents endroits. Esterel [BMR83] a été défini à Sophia-Antipolis en 1983 dans une équipe commune Ecole des Mines et INRIA. Lustre [BCP+85, NHR92] a été défini à Grenoble. Signal [GBBG86, BGJ91] a été développé à INRIA Rennes.

Définition de l'approche synchrone

L'hypothèse synchrone simplifie le temps réel (temps de la montre) et suppose l'existence d'un temps logique discret. Cette approche suppose également que les réactions des systèmes temps-réel (calcul et communication) sont atomiques (le système n'évolue pas pendant une réaction). Donc, ces réactions sont considérées comme ne prenant pas de temps (logique), ce qui fait que chaque réaction d'un système réactif synchrone peut être considérée comme instantanée. Ces réactions atomiques définissent les instants qui composent ce temps logique discret. Par conséquent, deux actions exécutées à un même instant t seront considérées comme simultanées (voir figure 3.1). En conséquence, le système produit ses sorties dans le même instant logique que ses entrées.

L'hypothèse synchrone ne s'applique pas dans le cas où le temps de réaction d'un système est significativement long par rapport à la période moyenne d'occurrence des évènements de son environnement. Dans ce cas, l'atomicité n'est plus assurée.

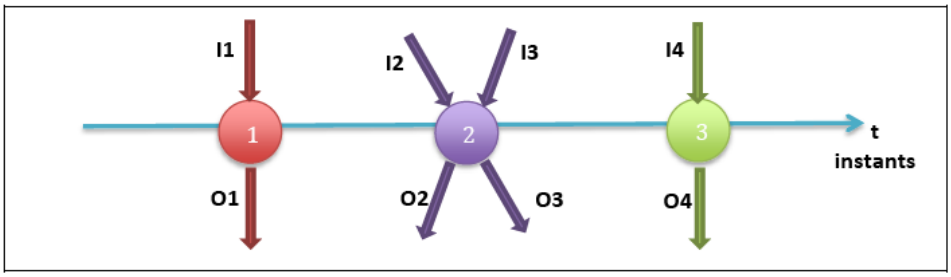


FIGURE 3.1 – Principe de l'approche synchrone

Grâce à l'atomicité, le système ne traite pas de nouvelles entrées tant qu'il n'a pas fini de réagir aux entrées précédentes. La simplification et l'abstraction de la notion du temps aide à réduire la complexité de programmation, en supposant que tous les évènements reçus dans un intervalle de temps $[t_1, t_2]$ ne seront pris en considération et traités qu'à la date t_2. Pour suivre cette hypothèse, les différentes parties du système suspendent leur exécution pour recevoir les nouveaux évènements et démarrer une nouvelle exécution. Ainsi, la gestion des délais et des exceptions est plus facile, ils sont gérés dans l'instant où ils arrivent. Avec cette abstraction du temps, toutes les réactions démarrent à la même date et leurs temps d'exécution sont identiques et égaux à zéro. Ceci génère un seul résultat si le système a une solution unique et garantit le déterminisme. Le déterminisme signifie qu'un système génère toujours la

1. http://www.esterel-technologies.com/about-us/scientific-historic-background/

même séquence de sortie à partir de la même séquence d'entrée. Ce déterminisme est assuré même dans le cas d'un système complexe qui contient plusieurs blocs en parallèle : l'approche synchrone coordonne ces blocs et gère les interactions entre eux. Le déterminisme est assuré car tous les flots de contrôle (ou blocs de systèmes) qui s'exécutent pendant une réaction voient le même statut pour un signal dans cette réaction (la communication est une diffusion instantanée).
Le déterminisme assure aussi la prédictibilité : il est plus facile de prédire le comportement (correct) du système, ce qui facilite sa modélisation, sa simulation et sa validation.

En se basant sur l'approche synchrone, on peut définir une sémantique formelle qui permet d'une part, l'analyse statique et la vérification formelle des propriétés et du comportement des systèmes et, d'autre part, une simplification considérable de l'implémentation et de la compilation de ces systèmes pour générer un code efficace.

Inconvénients de l'approche synchrone

Bien que l'approche synchrone assure des système sûrs et efficaces, elle a aussi des inconvénients : le problème principal est la possibilité d'avoir des "cycles de causalité". Ce problème s'avère parfois inévitable surtout dans le cas d'un système complexe, qui nécessite des intéractions elles aussi complexes.
La causalité apparait lorsqu'il y a un cycle de dépendance dans le calcul des statuts des signaux. Considérons l'exemple suivant, décrit dans [Zaf05] pour illustrer ce problème. Cet exemple est un programme en langage synchrone Esterel (présenté dans la section 3.3).

```
module Cyclique:
output ping, pong;
    present ping then emit pong end present
||
    present pong emit ping end present
end module
```

Ce code parait correct et pourtant il sera rejeté par le compilateur Esterel car il contient un cycle de causalité. La présence du signal "ping" va engendrer l'émission du signal "pong" et vice versa. Cependant, les deux branches de l'opérateur parallèle peuvent réagir instantanément à la présence ou à l'absence d'un signal, dans cet exemple, le statut de "pong" dépend de celui de "ping" et réciproquement. Il y a donc un cycle de causalité et le programme ne sera pas considéré comme correct.

Un autre problème est que les signaux produits par l'environnement ne sont pas synchrones. Il faut donc faire face au problème de communication entre le monde asynchrone et les systèmes synchrones. Ces systèmes doivent interagir avec leurs environnements asynchrones et traiter les événements asynchrones. Il faut donc chercher une solution pour plonger les systèmes synchrones dans le monde asynchrone.

3.2.2 L'approche asynchrone

Le monde réel est généralement asynchrone. Les systèmes asynchrones sont des systèmes faiblement couplés qui sont dirigés par des évènements pour gérer leurs communications. Celles-ci sont traitées de façon locale, à l'aide de protocoles de synchronisation entre blocs fonctionnels. Ils mémorisent les données et peuvent

Approche synchrone	**Approche asynchrone**
Disponibilité simultanée de données regroupées dans un instant logique	Disponibilité des données au fur et à mesure
Implémentation facile	Implémentation plus compliquée
Regroupement des communications au sein d'instants logiques	Communication possiblement différée mais en général bornée
Temps logique discret	Temps généralement physique avec possibles retards.
Déterminisme et prédictibilité	Indéterminisme possible
Problèmes de concurrence statiquement détectables	Problèmes de concurrence pas toujours détectables statiquement
Vérification formelle et model-checking faciles	Vérification formelle et model-checking plus difficiles
Présence de cycles de causalité détectable statiquement	Possibilité de problèmes de causalité à l'exécution
Communication compliquée avec l'environnement extérieur	Communication facile avec l'environnement extérieur

TABLEAU 3.1 – Comparaison entre l'approche synchrone et l'approche asynchrone

fonctionner en mode déconnecté. Dans l'environnement extérieur on peut avoir des retards, des pertes ou des distorsions d'information. Ceci peut engendrer des problèmes de courses critiques, des inter-blocages, des ordonnancements ambigus ou incorrects. Tous ces problèmes rendent les systèmes asynchrones souvent non déterministes, donc sans garantie d'avoir le même résultat pour la même séquence d'entrées. Ces problèmes, dont l'indéterminisme, font de l'analyse et de la vérification du comportement de ces systèmes une tâche difficile.

Pour remédier aux problèmes des systèmes asynchrones, il est indispensable d'utiliser des mécanismes appropriés comme l'ordonnancement des exécutions des signaux, la création d'intervalles de temps pour l'attente de signaux en retard ou l'horodatage, etc. Toutefois, on peut attribuer une sémantique formelle à ces systèmes pour faciliter la vérification de leur comportement, mais ceci ajoute une grande complexité à la tâche de modélisation et de réalisation d'un système réactif asynchrone.

3.2.3 Comparaison entre les deux approches

Le tableau 3.1 présente une comparaison globale des deux approches. L'approche synchrone présente de sérieux avantages pour nos objectifs. En effet, dans notre travail, nous visons à créer un système sûr, efficace et bien fondé, l'approche synchrone nous permet de créer un tel système et de faciliter cette création, grâce à la simplification du temps de la montre, le parallèle déterministe et la sémantique formelle. De plus, dans l'approche asynchrone, un postulat de départ est que deux évènements ne peuvent pas arriver simultanément. L'approche synchrone permet de se libérer de ce postulat : deux évènements peuvent arriver en même temps et on doit pouvoir en tenir compte autrement qu'en les "ordonnant".

3.2.4 Modélisation synchrone des activités

Comme mentionné dans la section précédente, l'approche synchrone nous permet d'assurer des propriétés de sûreté de fonctionnement aux systèmes construits. Un langage synchrone bénéficie du parallèle synchrone, ce qui permet d'écrire des programmes parallèles déterministes et d'éviter les problèmes causés par les courses critiques, l'indéterminisme, etc, qui peuvent ajouter des difficultés pour vérifier et valider un système.

Les langages synchrones sont utilisés pour décrire des systèmes sous forme de produits d'automates déterministes qui peuvent être composés sans perdre le déterminisme. Ces automates sont généralement représentés par les *machines de Mealy*. L'automate produit est séquentiel car le parallèle synchrone se traduit par un entrelacement du comportement de ses sous-automates (une sorte d'entrelacement compilé).

Les machines de Mealy que nous considérons dans notre cas sont des 5-uples de la forme : $< Q, q_{init}, I, O, T >$, où Q est un ensemble fini d'états, q_{init} dans Q est l'état initial, I (respectivement O) est un ensemble fini d'événements d'entrée (respectivement de sortie). $T \subseteq (Q \times I) \times (Q \times O)$ est la relation de transition.

Ceci est une représentation explicite des machines de Mealy en tant qu'automates. Mealy lui-même a introduit une autre représentation sous forme d'un système d'équations booléennes qui calcule à la fois les valeurs des événements de sortie et l'état suivant à partir des valeurs des événements d'entrée et de l'état actuel [Mea55]. Cette représentation est appelée *machine implicite* de Mealy. Plusieurs langages synchrones ont été définis pour décrire ces machines de Mealy [Hal93]. Un problème avec celles-ci est qu'elles sont difficiles à manipuler surtout par les utilisateurs qui ne sont pas familiers avec les automates en premier lieu et avec l'approche synchrone en deuxième lieu. Pour les automates, un utilisateur doit apprendre ce qu'est un état, une transition, etc. D'autre part, la conception d'un automate n'est pas une tâche simple : ajouter un état à un automate composé revient à reconstruire tout l'ensemble. Cette tâche est plus compliquée si l'automate est complexe ou parallèle. Par exemple, si on a un automate parallèle composé des deux automates A et B de 3 états chacun, le nombre de combinaisons d'états est $3 \times 3(9)$. Si on souhaite lui ajouter un $3^{ème}$ automate en parallèle, de 3 états par exemple, le nombre de combinaison d'états devient $3 \times 3 \times 3(27)$. D'un autre côté, l'utilisateur doit respecter les concepts de base du synchrone, comme le déterminisme par exemple, l'utilisateur doit veiller à ce qu'à un même ensemble d'entrées corresponde un même ensemble de sorties. Notre défi est de résoudre cette complexité et de rendre l'approche synchrone utilisable par un utilisateur non-expert.

3.3 Langages synchrones existants

Les langages synchrones sont destinés à décrire des systèmes réactifs en général et peuvent donc être utilisés pour décrire des activités humaines. Ils utilisent un temps logique, ce qui signifie que les systèmes réagissent seulement quand quelque chose de significatif se produit. Les langages synchrones sont une alternative pour décrire des machines de Mealy. En effet, tout programme synchrone se compile en machine de Mealy. Décrire une telle machine avec un programme est plus facile que de décrire un automate.

Il existe plusieurs langages synchrones ayant des caractéristiques variées. Certains sont impératifs, d'autres déclaratifs ou fonctionnels, certains proposent une forme

graphique d'édition des programmes, ou un environnement de développement intégrée comportant des vérifieurs, des simulateurs, des outils de modélisation, etc. Tous reposent de manière plus ou moins implicite sur la théorie des automates. Nous allons présenter ceux qui nous paraissent les plus significatifs.

3.3.1 Les langages synchrones historiques

Les langages synchrones existent depuis les années 80. Le premier langage, appelé Esterel, a été crée en 1983 [BMR83]. Il a été ensuite suivi de plusieurs autres, comme Lustre et Signal.

Esterel

Esterel est un langage synchrone impératif (événementiel) et modulaire [Ber96]. Bien que sa syntaxe soit plutôt simple pour les programmeurs Esterel, les comportements décrits par les programmes sont parfois difficiles à appréhender par certains utilisateurs finaux. Comme beaucoup de langages synchrones, Esterel manipule des *signaux* diffusés à toutes les entités de manière instantanée. Les signaux traités par Esterel peuvent contenir des informations sur leurs statuts de présence/absence et des valeurs typées. L'unité de base d'Esterel est le "module" qui possède des signaux d'entrée et de sortie. Esterel possède les opérateurs usuels de programmation et des opérateurs spécifiques : *present* qui teste la présence d'un signal dans l'environnement, un opérateur || qui implémente le parallèle synchrone ainsi que des opérateurs spécifiques au temps logique comme *await* qui attend la présence d'un signal dans l'environnement au bout d'un instant. De plus, un opérateur de préemption *abort* permet d'arrêter l'exécution d'une instruction quand un signal d'arrêt arrive. Un autre opérateur spécifique est l'opérateur *emit* qui émet des sorties dans l'environnement. Voici un exemple très simple de ce langage que nous exprimerons aussi avec Lustre et Signal :

```
module minute:
input SEC;
output MIN;
 every 60 SEC do
  emit MIN
 end every
end module
```

Ce programme émet un signal MIN toutes les 60 occurrences du signal SEC.

Comme la plupart des langages synchrones, Esterel se compile en un automate fini qui peut être représenté sous forme d'un système d'équations booléennes.

Esterel est utilisé en particulier pour le hardware. Il peut être compilé en des langages de description de matériel comme SystemC, VHDL ou Verilog [DPBB07]. Par exemple dans [HNT03], Esterel a été utilisé dans le processus de développement pour la conception, la certification et la génération d'un code VHDL pour assurer la fiabilité d'un dispositif de surveillance d'un système hydraulique aérospatial utilisant des composants FPGA. Adapter le langage Esterel à nos besoins aurait été difficile car nous ne maitrisons pas sa chaîne de compilation et son format de sortie n'est pas adapté à notre moteur de reconnaissance.

Lustre

Lustre est un langage fonctionnel de programmation synchrone déclaratif [BCP+85]. Il manipule des flots de données. Un flot est une suite possiblement infinie de valeurs de même type, couplé avec une horloge. Un programme Lustre est basé sur un réseau d'opérateurs qui consomment et produisent des flots de données à chaque cycle d'activation. Lustre est également un langage modulaire et son entité est le nœud. Un nœud est un ensemble d'équations qui définissent les variables de sortie. Lustre possède un opérateur de test (*if ..then..else*), un opérateur "*pre*" qui donne la valeur d'un flot à l'instant précédent et son complément "->" qui permet d'initialiser la première valeur d'un flot. Lustre possède une horloge de base et tout flot booléen peut être utilisé pour définir une sous-horloge avec l'opérateur *when*. En revanche, il n'y a pas d'opérateur explicite de parallèlisme, par définition, tous les flots sont calculés simultanément et c'est juste leur interdépendance qui définit un ordre d'évaluation. Nous reprenons en Lustre le même exemple déjà décrit en Esterel :

```
node minute (SEC:bool) returns (MIN:bool)
var S,ZS:int;
 let
  S = if SEC then ZS+1 else ZS;
  ZS = 0 -> pre(S);
  MIN = false -> (pre(S) mod 60 = 59) and (S mod 60 = 0);
 tel
```

Dans cet exemple, la variable locale S compte le nombre de fois où le flot SEC est vrai. Le flot MIN est vrai au passage de la $59^{ème}$ à la $60^{ème}$ fois où le flot SEC a été vrai. Comme tous les langages synchrones, le modèle des nœuds Lustre est un automate fini. Toutefois, la manipulation de réseaux d'opérateurs communicants par flots de données demande un certain niveau d'expertise. Lustre est utilisé pour la conception de logiciels critiques dans l'environnement de développement Scade (Safety Critical Application Development) [HCRP91] [CCM+03]. Il est aussi utilisé pour l'analyse et génération de code des architectures "globalement asynchrones et localement synchrones" (GALS) [GWVW08].

Signal

Signal [BGJ91] est un langage de programmation synchrone déclaratif développé principalement pour des applications en traitement du signal et qui utilise les flots de données synchronisés, comme Lustre. Ces flots sont appelés "signaux". Signal est modulaire et son unité de programme est le "process". Contrairement à Lustre, Signal est un langage relationnel et tout process définit une relation entre ses flots d'entrée et de sortie. Le style de programmation de Signal est proche de la programmation par contraintes. Un signal est un flot, c'est-à-dire une suite possiblement infinie de valeurs associée à une horloge, mais contrairement à Lustre, il n'y a pas d'horloge globale mais des "arbres d'horloges" calculés par le compilateur du langage en s'appuyant sur une algèbre tri-valuée. C'est au compilateur de vérifier le déterminisme du programme.

Signal possède 5 opérateurs de base (en plus des opérateurs usuels arithmétiques et booléens) pour exprimer les relations définissant les flots de sortie : *delay* : $Y := X_k$ met la $k^{ième}$ valeur de X dans la première valeur de Y ; *when* est voisin de l'opérateur Lustre, pour redéfinir l'horloge d'un signal avec un flot booléen ; *default* est un opérateur de fusion ; | est la composition parallèle synchrone ; \ est un opérateur de restriction

pour limiter la portée de certains signaux. A titre d'exemple, nous décrivons en Signal le process équivalent au module *minute* défini en Esterel et au node *minute* défini en Lustre.

```
process minute = (? event SEC; ! event MIN)
(|
S := (0 when MIN) default (ZS +1)
| ZS  := S $ 1
| MIN := SEC when (ZS = 59)
| synchro{S,SEC}
|)
where integer S, ZS init 0;
end
```

Dans ce processus, S compte le nombre d'occurrences de SEC par rapport à MIN (MIN correspond à 60 SEC). Il est remis à 0 à chaque occurrence de MIN, sinon il vaut ZS incrémenté de 1. ZS prend la valeur précédente de S (équivalent au *pre* de Lustre). Enfin *synchro* synchronise les horloges de S et SEC. Signal n'ayant pas d'horloge globale, ni de sous-horloges de celle-ci, son calcul d'horloge déduit des relations définies par l'utilisateur un ensemble d'arbres d'horloges pour les signaux.

Toutefois, l'assurance de la cohérence des horloges et l'approche relationnelle pour définir des flots rend Signal difficile à utiliser pour un non expert.

Conclusion

Ces langages historiques ont introduit les bases de l'approche synchrone. Ils constituent des solutions efficaces pour concevoir des systèmes réactifs sûrs et de grande taille. En début de thèse, nous avons essayé d'utiliser Lustre, mais comme pour Esterel et Signal nous ne maitrisons pas leur chaine de compilation ce qui nous empêchait d'étendre et de modifier aussi bien le langage que la sémantique pour les adapter à nos besoins.

Toutefois, nous nous sommes fortement inspirés de ces langages historiques en donnant la priorité à des constructions de langage plus adéquates à notre problématique. L'aspect impératif du langage Esterel nous semble plus naturel pour des utilisateurs non informaticiens. Nous avons défini un opérateur qui manipule le temps similaire à celui d'Esterel (*wait*). Nous avons aussi gardé l'idée d'une algèbre muti-valuée pour représenter le statut des évènements, notre approche définit les statuts des évènements dans une algèbre quadri-valuée (voir section 4.3.1) qui va nous permettre de traiter plus finement les problèmes de causalité inhérents aux langages synchrones.

3.3.2 Les langages synchrones graphiques

Argos

Argos [Mar92, MR01] est dans la lignée des langages synchrones comme Esterel et Lustre ; c'est un langage graphique conçu pour la programmation de systèmes réactifs, basé sur les Statecharts. Ses opérateurs permettent de composer des machines de Mealy. Argos s'inspire d'un sous-ensemble des Statecharts qui correspond aux diagrammes n'ayant pas de flèches traversant plusieurs niveaux. La sémantique d'Argos est une sémantique synchrone "à la Esterel". Argos a donné naissance à d'autres formalismes graphiques comme les "mode automata" [MR03] dans lesquels Argos a été combiné avec

Lustre afin de construire un langage synchrone puissant dans lequel des applications de nature complémentaire (automate + flots de données) peuvent être spécifiées. Ces deux langages graphiques ne sont plus vraiment utilisés maintenant mais le modèle des "mode automata" a été mis en œuvre dans la dernière version de Scade.

SyncCharts

Les SyncCharts [ABD98, And04] permettent la modélisation des systèmes réactifs. Ils ont été introduits en tant que format graphique pour le langage Esterel. Leur forme a été inspirée des Statecharts d'Harel [Har87b]. Comme les Statecharts, les SyncCharts modélisent les systèmes réactifs en utilisant des états, des états initiaux et finaux, des transitions. Ils communiquent avec des signaux et des évènements et assurent la hiérarchie, la modularité et le parallélisme. En plus, ils s'appuient sur l'approche synchrone en introduisant des opérateurs synchrones et assurent aussi le déterminisme. Les SyncCharts traitent aussi la préemption en adoptant la notion d'avortement et de suspension. Ils utilisent un ensemble restreint de primitives graphiques assez puissantes. Les SyncCharts ont été utilisés dans la modélisation de systèmes réactifs critiques comme les systèmes de contrôle des automobiles [BLPA98]. Ils ont été introduits dans Scade. Les SyncCharts sont basés sur Esterel, ce qui fait que on peut avoir les mêmes problèmes qu'avec ce dernier. De plus, pour utiliser les SyncCharts, il faut que l'utilisateur ait des bases en synchrone pour comprendre la notion d'instant et de transition.

Conclusion

S'inspirant des Statecharts, ces deux langages graphiques n'existent plus vraiment en tant que tels mais des outils commerciaux comme Scade ont intégré leurs philosophies et leurs techniques. Concernant notre approche, ces deux langages ont montré l'importance du graphique dans la description synchrone de systèmes réactifs. Nous pensons que l'aspect graphique peut être une façon parlante de permettre à un utilisateur non informaticien de décrire ses activités plus facilement. C'est pourquoi nous avons aussi opté pour une telle représentation graphique des activités.

3.3.3 Les environnements de modélisation synchrone

Scade (Safety-Critical Application Development Environment)

Scade [Tec] est un environnement de modélisation et de développement de systèmes critiques, en particulier les systèmes embarqués en avionique ou pour les automobiles. Il a été développé dans les années 90 par une collaboration entre Airbus, Merlin Gerin, Verilog et le laboratoire Verimag, en se basant sur les travaux de recherches de Lustre. A partir des années 2000, Esterel-Technologies a repris Scade et lui a rajouté des structures de contrôle basées sur Esterel et les SyncCharts [CPP17, ABD98] [2]. Scade peut modéliser des systèmes critiques à l'aide de machines à états finis ou en utilisant des diagrammes de flots de données. Scade est basé sur des opérateurs modulaires qui peuvent être représentés de façon graphique ou textuelle, et qui permettent à l'utilisateur de décrire des systèmes complexes (séquentiels, parallèles, modulaires, etc). Scade permet de vérifier la conformité d'un modèle par rapport aux

2. https://www.college-de-france.fr/site/gerard-berry/seminar-2013-04-23-11h00.htm

spécifications définies par l'utilisateur. Il permet non seulement la vérification formelle du comportement des systèmes, mais aussi leur simulation et la prédiction de leur performance. Il génère du code C certifié. Scade est un outil complet que nous pourrions utiliser pour la reconnaissance d'activités ; en revanche, la description des modèles d'activités en utilisant Scade sera une tâche difficile pour un non-informaticien. En effet, les modèles d'automates et de flots de données ne sont pas simples à appréhender pour des non spécialistes. De plus, c'est un outil payant et cela nécessiterait que tous nos utilisateurs l'achètent.

Polychrony

Polychrony [LGTLL03] [3] est un environnement de développement intégré et un démonstrateur technologique basé sur le langage Signal et conçu à IRISA. Il fournit un environnement unifié basé sur un modèle pour valider la conception des systèmes embarqués, temps réel et critiques à différents niveaux, pour affiner les descriptions dans une approche descendante, pour abstraire les propriétés nécessaires à la composition de boîtes noires et pour assembler des composants prédéfinis. Polychrony est composé d'un compilateur Signal fournissant un ensemble de fonctionnalités, comme les transformations de programmes, les optimisations, la vérification formelle, la compilation séparée, la génération de code, la simulation, le profilage temporel, etc. Il est aussi composé d'une interface utilisateur graphique (éditeur + accès interactif aux fonctionnalités de compilation) et de l'outil Sigali qui représente un système formel associé pour la vérification formelle et la synthèse du contrôleur. Polychrony est utilisé dans différents domaines comme l'avionique, le contrôle automobile ou les systèmes de contrôle de l'énergie nucléaire.

Conclusion

Scade est un outil maintenant populaire mais c'est un outil commercial que nous ne pouvons pas envisager d'utiliser. Polychrony est basé sur le langage Signal et le model-checker Sigali, qu'il est difficile d'appréhender pour un non spécialiste du synchrone déclaratif. Toutefois, l'aspect boîte à outils avec diverses fonctionnalités comme la simulation, la validation et la génération de code sont des aspects que nous avons adopté dans notre travail.

3.3.4 Autres langages synchrones

D'autres langages synchrones se basant sur les langages historiques ont vu le jour dans les années 90 et 2000.

Quartz

Quartz est un langage de programmation synchrone impératif, basé sur Esterel et développé à l'Université de Kaiserslautern (Allemagne) [Sch09]. Il permet la modélisation, la vérification et la mise en œuvre de systèmes réactifs. Quartz ressemble beaucoup à Esterel. Cependant, il utilise des instructions qui permettent l'exécution parallèle asynchrone des tâches, ce qui permet d'implémenter explicitement des systèmes indéterministes. Quartz a été aussi étendu avec des instructions dont l'effet est retardé au prochain instant. Ces instructions permettent ainsi de décrire des systèmes

3. http://www.irisa.fr/espresso/Polychrony/

logiciels (des algorithmes séquentiels) et matériels (circuits) [SB16]. Quartz propose aussi la notion de concurrence et de préemption. De plus, Quartz utilise une algèbre quadri-valuée pour vérifier la causalité des programmes.

Light Esterel (LE)

Light Esterel est un langage synchrone dédié à la spécification, la mise en œuvre et la vérification des systèmes de contrôle réactifs [GR08]. Trois syntaxes sont supportées : une représentation sous forme d'automates hiérarchiques, une syntaxe déclarative proche de Lustre, une syntaxe impérative inspirée du langage Esterel V5 (voir figure 3.2). Les automates sont des constructions natives du langage LE, ce dernier est capable de les compiler et d'intégrer leurs instances dans d'autres modules déjà compilés (il est incrémental).

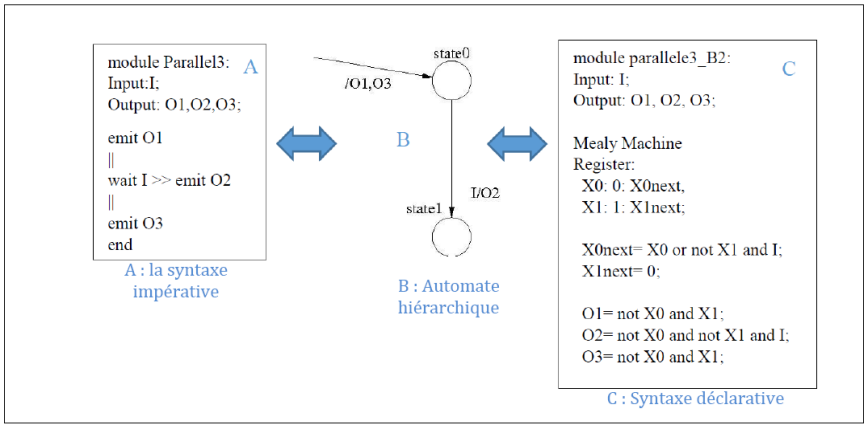


FIGURE 3.2 – Les syntaxes supportées par Light Esterel

Conclusion

Nous nous sommes fortement inspiré du langage Light Esterel. Toutefois, nous souhaitons avoir un langage ayant une syntaxe plus accessible. Certains opérateurs naturels pour des utilisateurs en dehors du monde synchrone n'existent pas. Comme il a été créé par la collaboration entre l'équipe MCSoc (LEAT) et notre équipe STARS (INRIA), nous avons décidé d'utiliser l'expérience acquise avec LE pour créer notre langage de description d'activités. Comme Quartz, Light Esterel s'appuie sur une algèbre quadri-valuée pour représenter les statuts des signaux manipulés par leurs programmes. Nous avons adopté cette idée dans notre travail. De plus, notre méthode de compilation est très similaire à celle de Light-Esterel.

3.4 Pourquoi un nouveau langage ?

Un point commun à tous les langages cités ci-dessus est la difficulté d'utilisation pour un non expert en synchrone. Actuellement, l'équipe STARS se sert d'un langage utilisateur non synchrone de description d'activités développé par V. Vu [VBT03, ZEV10]. Cependant, ce langage n'a pas de bases formelles et est très lié au domaine de la vision. Il ne fait pas de séparation entre la description d'une activité et les appels nécessaires aux algorithmes de vision. Ce que nous voulons faire dans cette thèse c'est fournir un langage ayant des bases formelles (s'appuyant sur le paradigme

synchrone), dédié à la description d'activités génériques (pas seulement en vision), mais en cachant les complexités de l'approche synchrone à nos utilisateurs non informaticiens (personnels médical, agents de sécurité, gérants de sociétés, etc.). Nous avons décidé de créer un nouveau langage plus adapté à leurs besoins et leurs utilisations. Ce langage étant destiné à décrire des modèles d'activités, nous l'avons appelé ADeL : Activity Description Language [SRM+17a]. Nous nous sommes inspirés des langages synchrones existants et de certains de leurs opérateurs. Nous avons principalement réutilisés certain aspects (style impératif, opérateurs temporels, algèbre quadri-valuée) de Light Esterel.

Toutefois, nous aurions pu définir un langage métier qui se traduit vers un langage synchrone existant, mais nous n'avons pas retenu ce choix essentiellement parce que nous désirons maîtriser la chaine de compilation du langage afin de générer automatiquement des automates de reconnaissance. Nous avons fait une première tentative en traduisant la description des activités en Lustre car il était relativement facile de programmer des automates d'activités dans ce langage. Cependant, comme pour Esterel, nous ne maitrisons ni la chaine de compilation ni le format de sortie. En effet, un des formats générés par Esterel et Lustre est le codage de l'automate de la description sous un format difficilement extensible. Or, notre système de reconnaissance d'activités général a besoin d'informations supplémentaires que l'on ne peut calculer que si nous avons la maîtrise de la chaine de compilation. Par exemple, nous désirons particulariser certains évènements ("timeout" par exemple) et permettre des optimisations du fonctionnement du synchroniseur (voir section 5.4.1) et du moteur de reconnaissance (voir figure 1.2).

3.4.1 Langage ADeL : mise en oeuvre

Nous nous focalisons surtout sur le travail avec les médecins, qui sera la catégorie d'utilisateurs avec laquelle nous allons principalement collaborer.
Pour aboutir à notre objectif, ADeL a été créé en collaboration avec les ergonomes de LudoTIC. LudoTIC est un cabinet français de conseil et d'expertise en ergonomie des IHM et UX (User eXperience), qui mène des audits, des tests utilisateurs, du maquettage, etc. Depuis 2004, LudoTIC investit dans la R&D concernant les IHM ludiques. Cette société fait aussi de la recherche scientifique de haut niveau dans le domaine de la psychologie cognitive [CB04].
Notre objectif est de permettre à des utilisateurs non experts en programmation de décrire des activités et des comportements. Pour ce faire, nous avons commencé avec LudoTIC par définir les concepts de base de la reconnaissance et les exigences principales des utilisateurs. Ces concepts sont présentés dans l'arbre de concepts de la figure 3.3 et définis comme suit.

3.4.2 Définition des concepts

Nous avons choisi de nous appuyer sur une approche ergonomique pour créer et définir un modèle conceptuel de notre langage. Définir les concepts de base de notre système est la première étape. Ci-dessous nous donnons la définition des principaux concepts utilisés dans la thèse.

Activité

Une activité correspond à un ensemble d'actions qui s'enchainent suivant différentes temporalités. Ces ensembles correspondent non seulement au déroulement typique

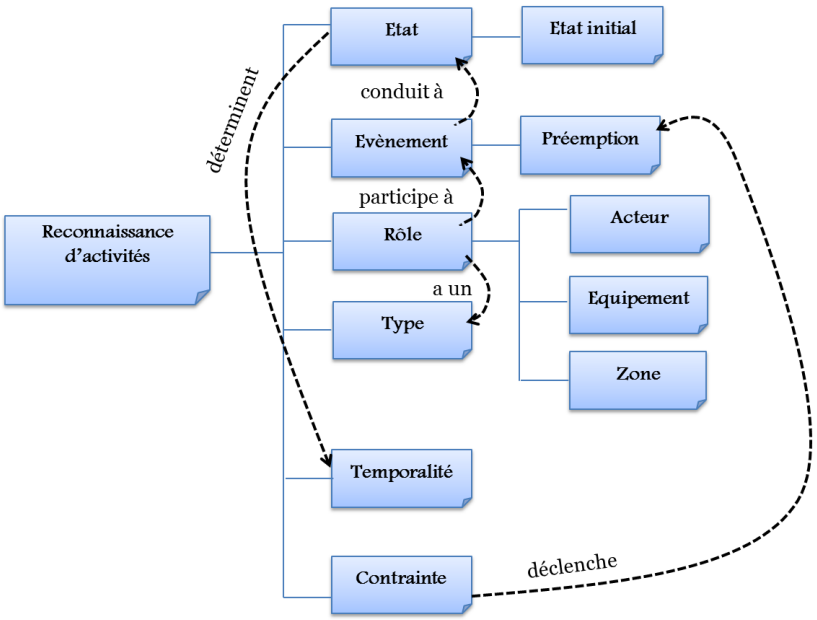


FIGURE 3.3 – Arbre de concepts du langage ADeL ("ontologie" minimale)

de l'activité mais aussi à ses variantes. Une activité est décrite à l'aide de rôles, d'évènements, d'états et de lieux.

Reconnaissance d'activité
La reconnaissance d'une activité est le fait de reconnaitre l'enchainement des événements et d'attribuer les rôles à des acteurs (objets reconnus par les capteurs en temps réel).

Temporalité
La temporalité correspond à une notion de temps (qui peut être le temps logique ou le temps de la montre, le temps relatif, etc). Exemple : la police arrive *après* le braquage.

Zone
C'est une partie du lieu où l'activité sera exécutée, par exemple, une chambre est une zone.

Acteur
Un acteur est un représentant particulier d'un type : par exemple, Lassie est un chien. Lassie est de type Chien.

Rôle
Un rôle présente un ensemble de caractéristiques et de comportements observables qui sera attribué à un acteur et qui est nécessaire à la reconnaissance de l'activité. Un acteur peut jouer plusieurs rôles et un même acteur peut changer de rôle. Exemples de rôles : médecin, infirmière, malade.

Etat
Un état représente une situation résultante des évènements passés. Dans chaque état on sait ce que l'on attend. Un état a une durée, un événement de début et un

événement de fin. Un état peut exister sans événement précédent, c'est l'état initial.

Etat initial
Un état initial est une situation de départ avec l'ensemble de ses rôles, ses zones... C'est un état où l'on attend les événements pour commencer l'activité, exemple : le médecin et le patient sont dans la chambre avec la fenêtre ouverte.

Evènement
Un évènement est quelque chose qui arrive et que l'on peut attendre. Un évènement est instantané (on dit aussi "impulsionnel"), c'est-à-dire que sa durée est négligeable dans le temps. Il peut déclencher la transition d'un état à un autre, ce qui fait avancer le déroulement de l'activité. Exemple d'évènement : le voleur entre dans la banque.

Contrainte
C'est une obligation logique créée par les règles applicatives, par les lois propres à un domaine, par une nécessité, etc. Une contrainte doit être observable et doit pouvoir être vérifiée. Une activité ne peut être reconnue que si ses contraintes sont vérifiées. On peut donner comme exemple les contraintes temporelles telles que *avant, pendant, après...*

Préemption
La préemption est un évènement qui, s'il arrive, arrête la reconnaissance de l'activité courante. Exemple : si un policier arrive dans la banque, on arrête la reconnaissance de l'activité "attaque de banque".

Ces concepts sont abstraits, nous souhaitons les mettre en pratique à l'aide des opérateurs qui les manipulent dans le langage.

3.4.3 Opérateurs du langage

Le choix des opérateurs d'ADeL repose sur plusieurs considérations : (1) ADeL est un langage synchrone, qui doit avoir des opérateurs qui manipulent le temps logique ; (2) ces opérateurs doivent constituer le noyau minimal à partir duquel toute activité peut se décrire ; (3) de plus, ce noyau doit être complet pour permettre aux utilisateurs d'exprimer les activités en mettant en œuvre tous les concepts définis dans la section 3.4.2. En respectant ces trois principes, nous avons choisi le noyau d'opérateurs décrit dans le tableau 3.2.
Les opérateurs **seq**, **while**, **stop when**, **if then else** et **parallel** sont fréquemment définis dans les langages synchrones et correspondent à nos besoins pour exprimer les différents enchainements d'événements dans une activité. L'opérateur **emit** et les alertes (**alert**) liées aux opérateurs **timeout** et **stop..when** changent le statut d'un évènement dans l'environnement de sortie. Ce sont les seuls opérateurs affectant le statut des évènements. L'opérateur **local** se justifie d'un point de vue synchrone, il introduit les évènements locaux permettant à deux branches d'un **parallel** de communiquer. Enfin, notre noyau d'opérateurs doit aussi comporter des opérateurs qui manipulent le temps logique. Si **wait** est classique dans les langages synchrones, nous introduisons des opérateurs spécifiques pour mettre en œuvre les concepts ci-dessus : **timeout** pour traiter le temps de l'horloge (qui sera traité par notre système comme n'importe quel autre évènement). Nous pouvons noter que **timeout** est bien

un opérateur noyau qui ne peut pas s'écrire simplement à l'aide de **stop** et **seq** parce qu'il faudrait modifier **stop** pour qu'il puisse traiter la durée, ce qui compliquerait cet opérateur. Il nous est apparu plus simple de séparer ces deux notions.
Les sémantiques d'ADeL définies au chapitre suivant considèrent les opérateurs du langage pour donner un sens aux programmes. Il est donc important que le noyau d'opérateurs choisi soit le plus petit possible. Il est aussi important que ce noyau soit complet car il va servir de base pour définir la syntaxe utilisateur. Un utilisateur doit pouvoir décrire toutes sortes d'activités grâce à cette syntaxe. Il faut noter que la syntaxe fournie aux utilisateurs peut comporter d'autres opérateurs pour se rapprocher du langage naturel dans certains cas grâce à du sucre syntaxique qui sera traduit à l'aide du noyau d'opérateurs. Par exemple, on pourrait introduire un opérateur de durée traduit par : **wait** *évènement temporel*, qui nécessite la génération de l'évènement temporel correspondant à la durée.

Notons que nous avons un opérateur pour appeler une sous-activité (**call**). Cet opérateur n'est pas un opérateur de base du langage, il correspond à une inclusion du code de la sous-activité dans l'activité principale appelante lors de la compilation, ce qui nous permettra d'avoir une compilation incrémentale (voir section 4.6.1).

3.4.4 Cas d'utilisation

Pour mieux décrire la démarche de notre modélisation d'activités, nous l'illustrons avec un cas d'utilisation simple de modélisation d'un "jeu sérieux" (serious game). Les jeux sérieux ne sont pas seulement destinés au divertissement mais aussi à l'enseignement, l'apprentissage, la formation, la communication et l'information. De nos jours, ils représentent une source d'intérêt importante dans de nombreux domaines, tels que la santé. Les médecins commencent à s'appuyer sur ce genre de jeux pour diagnostiquer les patients ayant des problèmes cognitifs, comme la maladie d'Alzheimer ou l'autisme.

Dans ce cas d'utilisation [PT17], nous décrivons l'activité d'un jeu sérieux pour évaluer le comportement et l'interaction des personnes Alzheimer (voir figure 3.4). Ce jeu consiste à afficher sur une tablette tactile une liste de différentes images, à présenter une image aléatoire au centre, et à demander au patient de choisir l'image correspondante dans la liste. Le rôle de la reconnaissance ici est de stocker dans un fichier le journal des informations sur les performances du patient (retards, erreurs, ...) afin que les médecins puissent avoir un enregistrement de l'évolution des patients.

Si le patient choisit la bonne image, un smiley "heureux" s'affiche. Une alerte disant que le patient a réussi est envoyée au fichier journal et le jeu affiche une autre image. Sinon, un smiley "triste" est affiché, et une alerte indiquant que le patient a choisi une mauvaise image est envoyée au fichier journal et le jeu demande au patient de réessayer. Si le patient n'interagit pas assez rapidement avec le jeu, le jeu demande à nouveau de choisir la bonne image dans la liste. Si le patient ne répond pas dans les 5 minutes, une alerte "game_cancelled" est envoyée et le jeu est annulé. Si le patient quitte la zone de jeu avant la fin, une alerte "test_failed" est envoyée et le jeu est abandonné.

3.4.5 Mode graphique

Nous proposons un outil graphique pour décrire les activités simples. Cet outil offre une description graphique de tous les opérateurs du tableau 3.2 et permet de représenter tous les concepts de la figure 3.3. Il existe donc une traduction structurelle du format

nothing	ne fait rien et se termine instantanément. **nothing** est un opérateur nécessaire à la définition de la sémantique. Cet opérateur est instantané.
[**wait**] S	attend l'événement S et suspend l'exécution de l'activité jusqu'à ce que S soit présent. L'opérateur **wait** peut être implicite ou explicite. L'exécution de cet opérateur dure au moins un instant.
p_1 **seq** p_2	la séquence globale commence quand p_1 commence ; p_2 commence quand p_1 se termine ; la séquence se termine quand p_2 est terminé. Cet opérateur est instantané si ses deux arguments le sont également.
p_1 **parallel** p_2	commence quand p_1 ou p_2 commence ; se termine lorsque les deux sont terminés. Cet opérateur est instantané si ses deux arguments le sont également.
while *condition* $\{p\}$	p est exécuté uniquement si la *condition* est vérifiée. Lorsque p se termine, la boucle redémarre jusqu'à ce que *condition* soit non valide. La condition est évaluée instantanément.
emit S	met l'évènement S présent dans l'environnement. Cet opérateur est instantané.
[**P**]**stop** $\{p\}$ **when** S [**alert** S_1]	exécute p lorsque S est absent, sinon lorsque S est présent, attend la terminaison de l'instant, puis envoie une alerte S_1 et se termine. L'option P signifie que cet opérateur est prioritaire. L'opérateur **Pstop** a le même comportement que **stop**, il génère un évènement *failure* en plus lorsque S est présent. L'exécution de cet opérateur est instantanée si p l'est également, dans ce cas S n'est pas testé.
if *condition* **then** p_1[**else** p_2]	exécute p_1 si *condition* est valide, sinon exécute p_2. Notons que *condition* est évaluée instantanément.
p [**P**]**timeout** S $\{p_1\}$ [**alert** S_1]	exécute p ; arrête si S survient avant que p se termine et envoie éventuellement une alerte S_1. L'option P signifie que cet opérateur est prioritaire. L'opérateur **Ptimout** génère un évènement *failure* en plus de l'alerte. Sinon les deux opérateurs exécutent p_1 lorsque p est terminé. L'exécution de cet opérateur est instantanée si p l'est également, dans ce cas S n'est pas testé.
local *event_list* $\{p\}$	déclare des événements internes pour communiquer entre des sous-parties de p. C'est un opérateur d'encapsulation.

TABLEAU 3.2 – Sémantique "intuitive" des opérateurs d'ADeL. S, S_1 sont des événements (reçus ou émis) ; p, p_1 et p_2 sont des instructions ; *condition* est soit un événement soit une combinaison booléenne de présence/absence d'événements [SRM+17b]. Les {} font partie de la syntaxe du langage.

FIGURE 3.4 – Interface du jeux sérieux de notre cas d'utilisation [PT17]

graphique vers le format textuel ce qui fournit au format graphique la même sémantique que le format textuel.

Nous avons deux types d'utilisateurs de cet outil : tout d'abord, l'administrateur responsable qui a une interface spéciale pour la conception des bibliothèques (de rôles, d'équipements, d'événements). L'administrateur définit la liste des évènements, des rôles et des équipements qu'un utilisateur final peut choisir. Pour cela, il se base sur les évènements que les capteurs sont capables de fournir. D'autre part, il définit aussi des catégories d'activités, c'est-à-dire l'ensemble des rôles, des lieux et des équipements spécifiques à un domaine particulier (exemple : médecine, sport, bureau, etc).

Nos principaux utilisateurs sont les utilisateurs finaux non informaticiens et notre effort porte surtout sur la définition d'une interface homme-machine adaptée à ce type d'utilisateur. Ce travail a été réalisé en collaboration avec LudoTIC. Ce deuxième type d'utilisateur peut uniquement décrire des activités génériques à l'aide des bibliothèques conçues par l'administrateur. Pour ces utilisateurs, notre outil propose plusieurs fenêtres. Les plus importantes sont celle pour la définition d'une activité, celle pour la déclaration des zones, des rôles et des équipements, celle pour la définition des évènements, et celle pour la description de l'enchainement des évènements dans l'activité.

Interface de définition des activités

Dans cette interface, les utilisateurs définissent le nom de l'activité et sa catégorie. Ils peuvent aussi modifier des activités existantes (voir figure 3.5 pour notre cas d'utilisation).

Interface de description de la scène et de définition de la situation initiale

Cette interface correspond aux concepts de type, de rôle et d'état initial de la figure 3.3. Elle permet de déclarer les acteurs et les équipements qui sont les seuls à pouvoir être impliqués dans les évènements constituant l'activité. L'ensemble des rôles possibles et leurs types associés sont proposés par l'administrateur. Dans cette

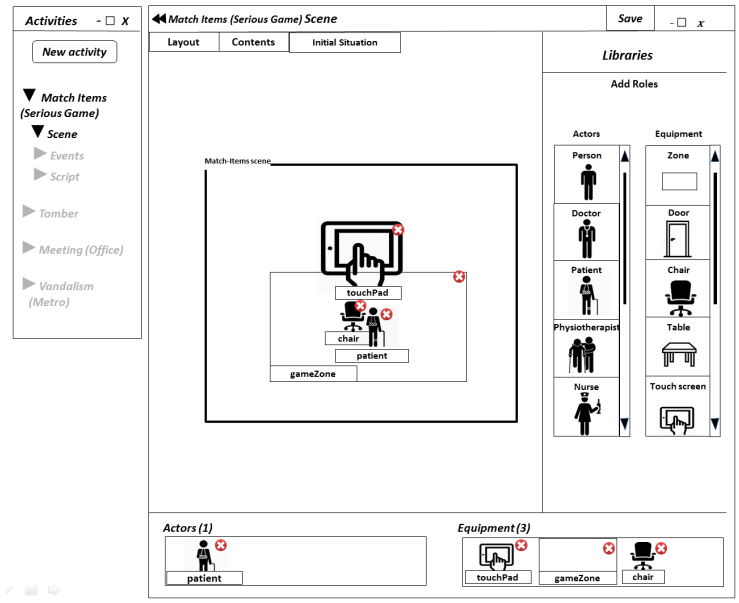


FIGURE 3.5 – Format graphique de la description du jeu sérieux de notre cas d'utilisation : description de la situation initiale de l'activité, à gauche on trouve la fenêtre qui liste les activités déjà définies et permet d'en rajouter

interface, les utilisateurs définissent d'abord la zone où se déroulera l'activité (la scène), ils sélectionnent les rôles de leurs acteurs et des équipements nécessaires dans l'activité à partir des bibliothèques définies par l'administrateur. Puis, pour définir la situation initiale, les utilisateurs positionnent dans la scène les acteur et les équipements nécessaires pour décrire cette situation, à partir de la liste des rôles et équipements déjà choisis. L'ajout se fait à l'aide d'un simple "glisser-déposer" (voir figure 3.5 pour notre cas d'utilisation).

Interface de définition des évènements :
Cette interface correspond au concept d'évènement de la figure 3.3. Dans la fenêtre de définition des évènements, les utilisateurs spécifient les événements prévus qui vont participer à la description de l'activité (à partir de la bibliothèque d'évènements définie par l'administrateur et en fonction de la catégorie, des rôles et des équipements choisis), et les associent aux rôles et équipements sélectionnés (les évènements du cas d'utilisation sont décrits dans la figure 3.6).

Interface de description de l'enchainement de l'activité
Après avoir défini la liste des évènements, les utilisateurs les organisent enfin dans la fenêtre de description de l'activité, dans un "story-board", en utilisant un panneau d'outils qui affiche les opérateurs ADeL du tableau 3.2, les sous-activités et les événements, pour en définir l'enchainement. Cette interface correspond aux concepts de temporalité et de contrainte de la figure 3.3. Le formalisme graphique reprend les notations des diagrammes d'états à la UML (Statecharts) mais la sémantique en est synchrone. De plus, les utilisateurs peuvent décrire leurs activités de manière hiérarchique et modulaire, ce qui signifie qu'ils peuvent créer une activité qui comprend plusieurs sous-activités. Cela rend la description plus facile à lire et à comprendre (voir figure 3.7).

L'outil graphique est en cours d'implémentation, l'interface de déclaration des activités, de description de la scène et l'interface des évènements sont implémentées,

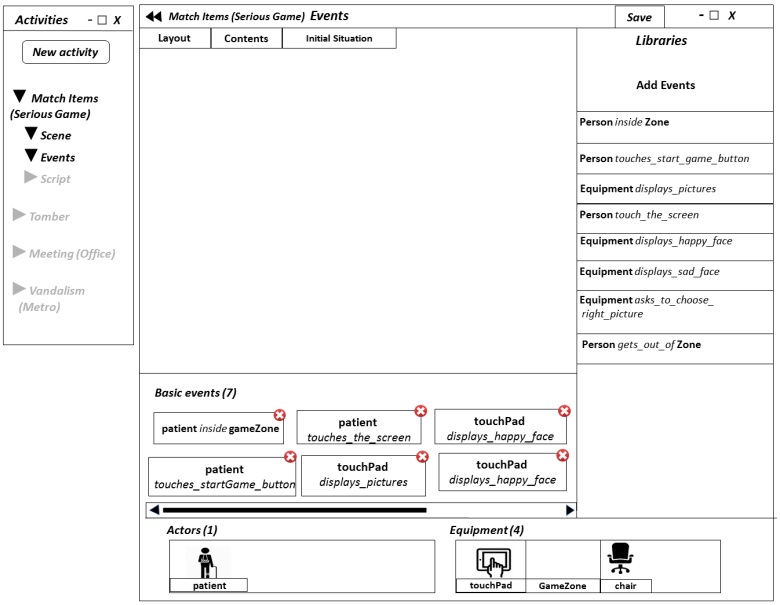


FIGURE 3.6 – Format graphique de la description de notre jeu sérieux : sélection des évènements à partir de la bibliothèque et association avec les acteurs et les équipements

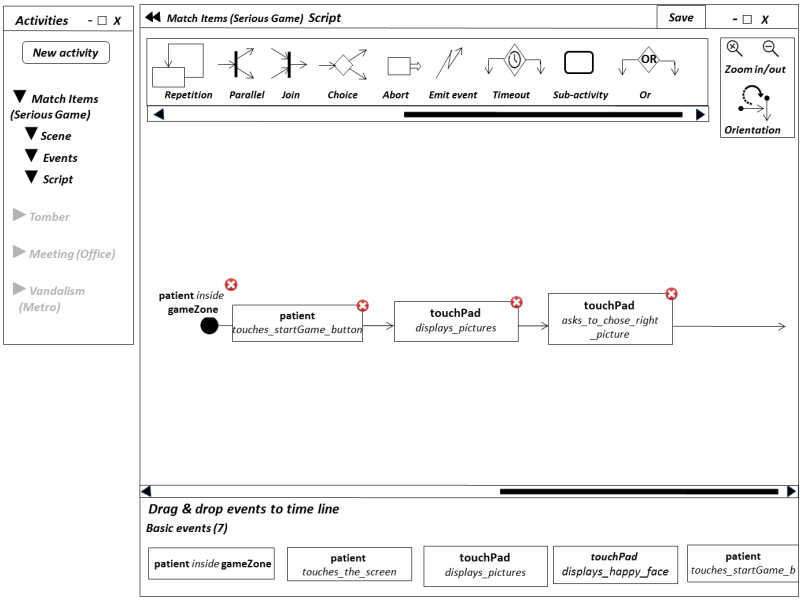


FIGURE 3.7 – Format graphique de la description de notre jeu sérieux : description de l'enchainement des évènements en les plaçant sur une ligne de temps. Si on change l'orientation de ce schéma, il sera au final comme montré figure 3.8

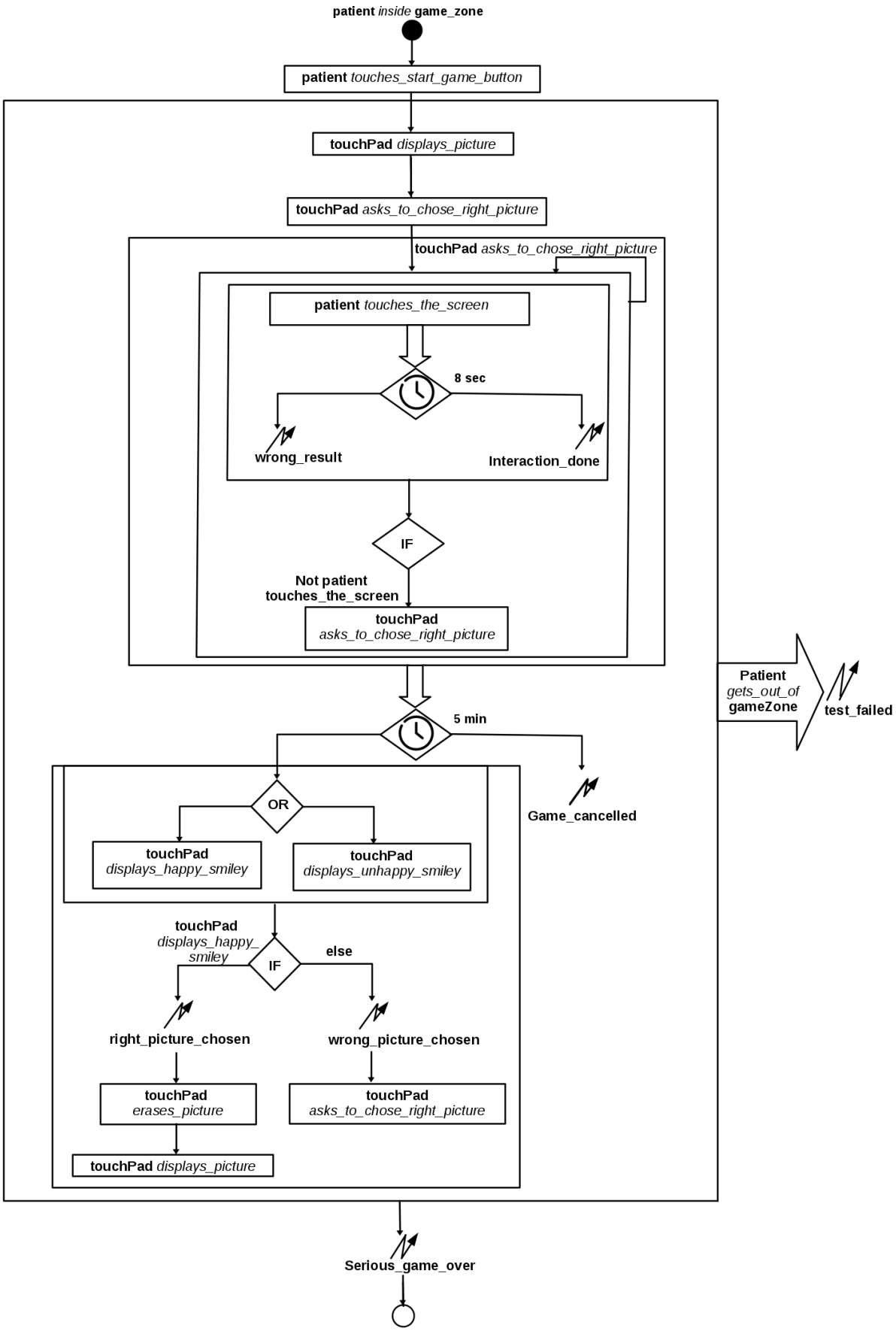


FIGURE 3.8 – Format de la description de l'enchainement des sous-activités du jeu sérieux sous forme graphique, notons qu'on peut avoir un opérateur **if** avec une seule branche (*then*).

et l'interface de description de l'enchainement de l'activité est en cours. Cependant, il peut être difficile d'exprimer des activités complexes en utilisant un outil purement graphique. En effet la manipulation de nombreux objets graphiques inter-connectés peut devenir difficile à maîtriser pour un utilisateur, c'est pourquoi nous proposons également un format textuel équivalent pour notre langage qui peut être plus manipulable. Notons qu'à partir de l'outil graphique, nous générerons du code textuel.

3.4.6 Mode textuel

Pour créer ce langage textuel, nous avons commencé par définir sa grammaire (voir en annexe A la forme BNF de la grammaire). La description des activités dans le format textuel se compose de plusieurs parties : définition des types et les rôles ; définition de l'état initial et des événements qui participent à la progression de l'activité. Enfin, dans le corps, l'utilisateur décrit le comportement attendu de l'activité et peut construire des instructions complexes par composition et combinaison d'opérateurs et d'événements.

Dans le format textuel d'ADeL, la première étape est de définir le nom de l'activité, de décrire les rôles et les types des objets animés ou non qui interviennent dans l'activité. Cette définition se fait dans la signature de l'activité, à la déclaration. Cette déclaration est de la forme : Nom_Activité(r1 : type, r2 : type2...). Pour le cas d'utilisation, on a 3 types : *Zone* où l'activité a lieu, *Person* et *Equipment*. Les rôles qui interviennent dans l'activité dans notre cas sont un patient de type Person, une tablette de type Equipment et une zone de jeu de type Zone.

Il s'agit ici de simples déclarations qui vont servir à vérifier que le code de l'activité fait référence à ces rôles et uniquement à eux. La déclaration est la suivante :

```
Activity seriousGame (patient:Person, touchPad:Equipment,gameZone:Zone)
```

Ensuite, les événements attendus et les sous-activités sont définis. Les sous-activités / évènements ont des paramètres qui déclarent les rôles (acteurs ou équipements) qui vont intervenir dans cette sous-activité/évènement. Par exemple : *gets_out_of_zone* est un évènement qui doit impliquer le *patient* et la *gameZone*. Ces informations seront utilisées plus tard dans le processus de reconnaissance.

Dans ce cas d'utilisation, on ne fait pas d'appel à des sous-activités extérieures, nous n'avons que des évènements. La déclaration de ces évènements est :

```
Events
 inside_zone(Person, Equipment, Zone);
 touches_start_game_button(Person);
 displays_picture(Equipment);
 asks_to_chose_right_picture(Equipment);
 touches_the_screen(Person);
 displays_happy_smiley(Equipment);
 displays_unhappy_smiley(Equipment);
 displays_sad_smiley(Equipment);
 erases_picture(Equipment);
 gets_out_of_zone(Person, Zone);
```

Enfin, l'activité du jeu sérieux est décrite en définissant l'état initial (cet état initial correspond à un évènement (ou plusieurs) attendu implicitement) et en faisant une combinaison des évènements et des opérateurs [4].

4. On remarque dans cet exemple un certain nombre de duplications de code préjudiciables à la lisibilité et l'élégance. Nous envisageons une amélioration de la syntaxe afin de les éviter

```
1  InitialState : inside_zone(patient, touchPad, gameZone)
2  Start
3      wait touches_start_game_button(patient)
4   seq
5   stop
6     {
7      wait displays_picture(touchPad)
8      seq
9      wait asks_to_chose_right_picture(touchPad)
10     seq
11     while asks_to_chose_right_picture(touchPad)
12     {
13      wait touches_the_screen(patient) timeout 8s
14      {
15       emit interaction_done
16      }
17      alert wrong_result
18      seq
19      If not touches_the_screen(patient)
20      then
21      wait asks_to_chose_right_picture(touchPad)
22     }
23     Ptimeout 5.0 min
24     {
25       wait (displays_happy_smiley(touchPad)
26              or
27             displays_unhappy_smiley(touchPad))
28
29       if displays_happy_smiley(touchPad)
30       then
31       {
32        emit right_picture_chosen
33        seq
34        wait erases_picture(touchPad)
35        seq
36        wait displays_picture(touchPad)
37       }
38       else
39       {
40        emit wrong_picture_chosen
41        seq
42        wait asks_to_chose_right_picture(touchPad)
43       }
44     }
45     alert game_cancelled
46    }
47    when gets_out_of_zone(patient, gameZone) alert test_failed
48   seq
49   emit serious_game_over
50  End
```

3.4.7 Comparaison entre ADeL et d'autres langages synchrones

ADeL manipule des évènements qui correspondent au "temps de la montre", par rapport à d'autres langages synchrones. Nous traitons ce temps de la montre comme

les autres évènements, mais nous offrons la possibilité d'en parler et de le décrire pour les utilisateurs non-experts grâce aux deux opérateurs **timeout** et **while**. Ensuite, nous faisons en interne ce que l'utilisateur d'un langage synchrone usuel est forcé de faire explicitement (voir chapitre 5). Par exemple, pour traiter les délais avec l'opérateur **timeout**, nous l'exprimons comme suit : p **timeout** S p_1 **alert** $alarm$ (S est un évènement temporisé, exemple $S = 2.0min$). En Esterel, cela correspondrait au code suivant :

```
weak abort {p} when S;
present S then emit alarm else p1;
```

Cette partie du code semble facile pour un programmeur mais ce n'est pas le cas pour les non informaticiens. Il est encore plus difficile de représenter cet opérateur dans un langage synchrone déclaratif comme Lustre (ce qui correspondrait au code ci-dessous).

```
node sequence(S, finp,finp1: bool)
  returns(debp,debp1,alert: bool)
  var Pfini,Sdetecte: bool;
let
  debp= true -> false;
  Sdetecte= S -> pre(Sdetecte) or S;
  Pfini= false -> pre(Pfini) or finp and not Sdetecte;
  debp1= false -> Pfini and not(pre(Pfini));
  alarm= S and not Pfini;
tel
```

3.4.8 Acceptabilité de l'approche choisie par les non-informaticiens

L'acceptabilité d'un langage comme ADeL par ses utilisateurs finaux est cruciale. C'est pourquoi, avec la collaboration de LudoTIC, nous avons conduit un sondage sur le Web, principalement auprès de personnels médicaux. Les premières questions déterminent le niveau informatique des participants et les questions suivantes portent sur des choix entre différents formalismes textuels et graphiques, dont ADeL.

Ce sondage est une première tentative pour avoir une idée de la recevabilité du langage pour différentes catégories d'utilisateurs. Pour avoir un grand nombre de retours exploitables nous avons volontairement proposé un exemple simple et traitable en 10 minutes. Pour ces raisons, cet exemple fait appel à une compréhension "intuitive" des langages. Pour le format textuel on propose un même exemple écrit dans trois langages synchrones : Esterel, Lustre et ADeL (les noms des langages n'étaient pas fournis aux participants).

Pour le format graphique on propose ce même exemple sous forme d'automate fini et sous la forme graphique d'ADeL. Dans cette première tentative, nous avons utilisé les outils auxquels nous avions facilement accès (ce qui exclut les outils commerciaux et/ou ceux dont les interfaces complexes nécessitent un apprentissage de la part des utilisateurs). L'outil à notre disposition (Galaxy) a été développé dans notre équipe. C'est une composante du langage synchrone Light Esterel (LE) qui offre la possibilité de représenter des automates hiérarchiques et parallèles avec des opérateurs graphiques dédiés ainsi que de décrire des SyncCharts [ABD98] qui sont dérivés des Statecharts.

Les objets représentés dans Galaxy ont une modélisation synchrone comparable aux activités décrites par ADeL graphique.

Nous avons eu 103 participants à ce questionnaire. Ces participants proviennent de différents pays : 36 participants de France, 48 de Tunisie, 15 du Canada, 1 de Roumanie, 1 d'Egypte, 1 du Royaume-Uni et 1 des Émirats Arabes Unis. Nous avons organisé les participants en trois catégories (voir figure 3.9).

1. **Informaticiens**
   Nous avons voulu avoir l'avis de professionnels de l'informatique sur les langages exemples, en particulier ADeL. 27 informaticiens ont répondu à ce sondage.
2. **Personnel médical**
   C'est le secteur qui nous intéresse le plus dans ce sondage, car nous nous adressons aux médecins en premier lieu dans ce travail. Les participants du secteur médical sont des médecins, des kinésithérapeutes, des étudiants en médecine, des psychologues, des médecins de l'équipe CoBTeK [5], etc. 34 participants du secteur médical ont répondu à ce sondage.
3. **Grand public (Autre)**
   Comme nous souhaitons que le langage ADeL soit destiné à tous les secteurs à terme, nous avons voulu avoir les avis des personnes d'autres secteurs aussi. Le groupe grand public contient des étudiants, des gérants de magasins, des entrepreneurs, des employés de banque, etc. 42 participants de ce groupe ont participé à ce sondage.

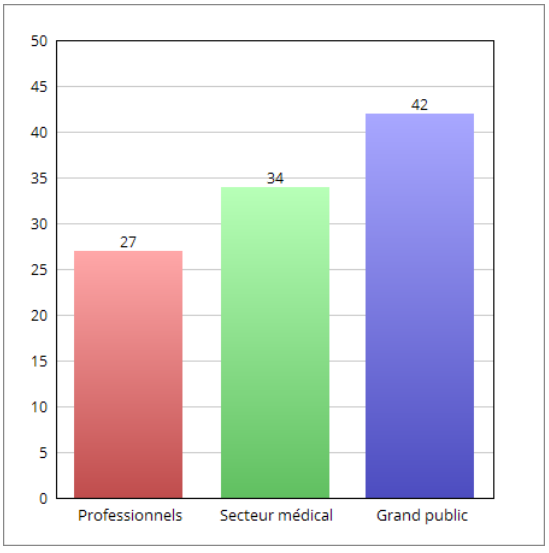


FIGURE 3.9 – Catégories des participants

Nous présentons en intégralité ici les trois questions qui nous permettent de comparer le langage ADeL et ses deux formats avec d'autres langages. Le texte du sondage complet est en annexe D.

Question sur le format textuel

Supposons qu'on souhaite décrire le scénario d'activité suivant à l'aide d'un langage textuel : une personne âgée tombe dans son salon, si elle ne se relève pas dans les 3

5. http ://www.innovation-alzheimer.fr/cobtek/

minutes, on lance une alerte de danger. Ci-jointes plusieurs descriptions de ce scénario en plusieurs langages, veuillez choisir qui vous parait le plus facile à comprendre (voir figure 3.10).
Les 3 langages proposés étaient :

1. Langage1 : ADeL.
2. Langage2 : Esterel
3. Langage3 : Lustre

1/ Langage1

```
Activity exemple (patient : Person, salon : Zone)
Events
tombe(Person, Zone)
se_releve(Person)
 bouge(Person)
InitialState : dans_zone(patient, salon);
start
        tombe(patient, salon)
        seq
        se_releve(patient) timeout 3.0 min
        {
          bouge(patient)
        } alert danger
end
```

2/ Langage2

```
module exemple:
  input: tombe_patient_zone, se_leve_patient, bouge_patient, timeout_3_minutes;
  output: danger;
      await tombe_patient_zone;
          abort
                await se_leve_patient
          when timeout_3_minutes ;
          present timeout_3_minutes then emit danger else await bouge_patient
       end
```

3/ Langage3

```
node exemple (tombe_patient_zone, se_leve_patient,  bouge_patient, timeout_3_minutes: bool)
returns (danger, ok :bool);
  var est_tombe : bool;
      let
         est_tombe = false -> if (tombe_patient_zone) then true else pre(est_tombe);
         danger = est_tombe and not se_leve_patient and timeout_3_minutes;
         ok = est_tombe and se_leve_patient and not timeout_3_minutes and bouge_patient;
      tel
```

FIGURE 3.10 – Propositions des formats textuels de trois langages

Parmi les 103 participants, 96 personnes ont répondu à cette question, les résultats sont montrés dans la figure 3.11 et détaillés par catégories ci-dessous.

— **Professionnels :**
26 professionnels ont répondu à cette question :
  — 14 personnes ont choisi le Langage 1
  — 6 personnes ont choisi le Langage 2
  — 6 personnes ont choisi le Langage 3
— **Secteur médical :**
32 personnes du secteur médical ont répondu à cette question :
  — 25 personnes ont choisi le Langage 1
  — 6 personnes ont choisi le Langage 2

— 1 personne a choisi le Langage 3

— **Grand public :**
38 personnes du grand public ont répondu à cette question :
— 26 personnes ont choisi le Langage 1
— 10 personnes ont choisi le Langage 2
— 2 personne ont choisi le Langage 3

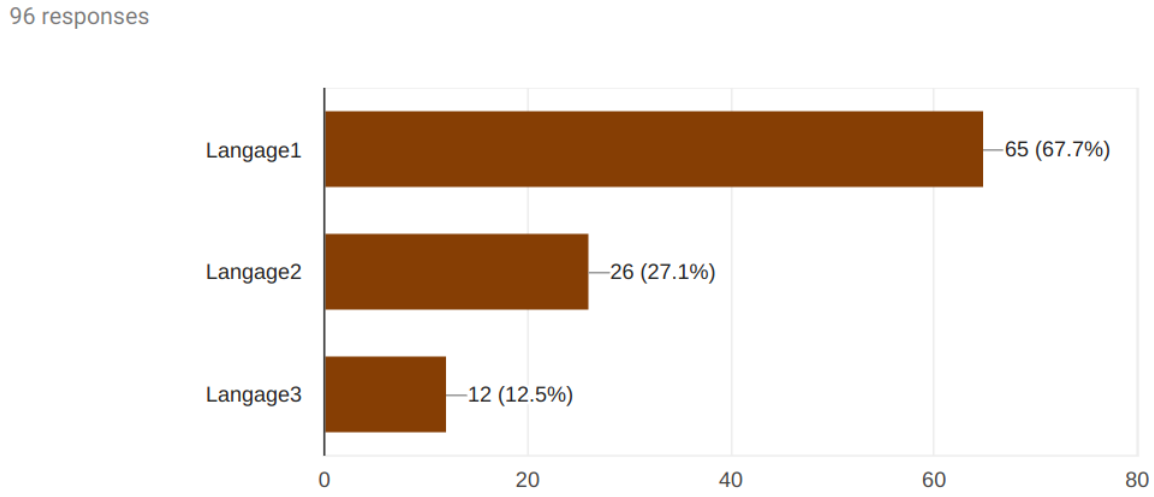


FIGURE 3.11 – Choix des formats textuels

Question sur le format graphique

On peut décrire graphiquement la même activité sous deux formes différentes, choisissez celle qui vous parait plus facile à comprendre et utiliser (voir figure 3.12 et figure 3.13).

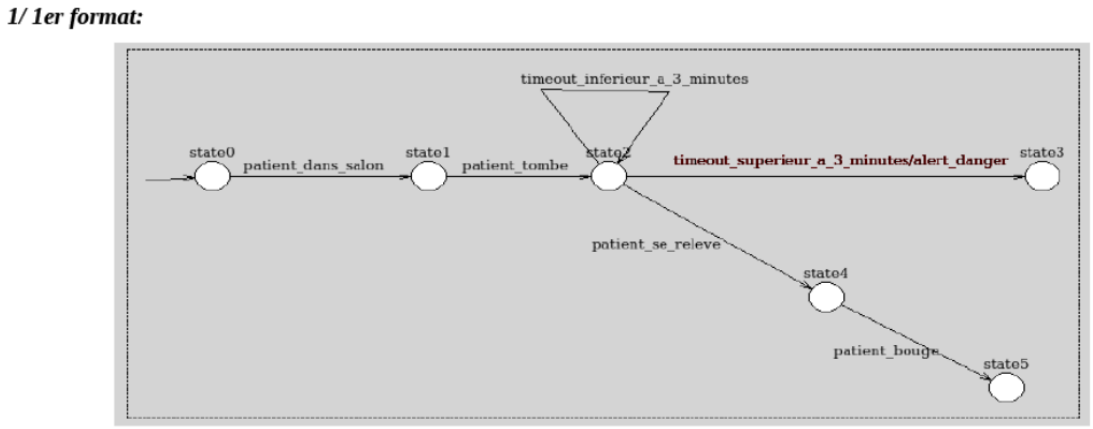


FIGURE 3.12 – Premier format graphique de l'exemple (automate fini)

Nous avons eu 94 réponses à cette question, les résultats sont montrés dans la figure 3.14) et détaillés par catégories ci-dessous.

— **Professionnels :**
26 professionnels ont répondu à cette question :
— 5 personnes ont choisi le premier format
— 21 personnes ont choisi le deuxième format

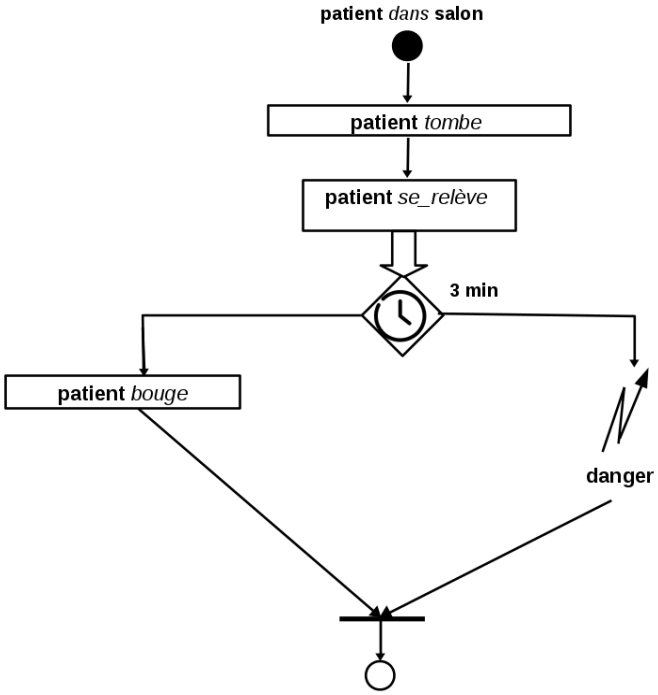


FIGURE 3.13 – Deuxième format graphique de l'exemple (ADeL)

— **Secteur médical :**
31 personnes du secteur médical ont répondu à cette question :
  — 7 personnes ont choisi le premier format
  — 24 personnes ont choisi le deuxième format
— **Grand public :**
37 personnes du grand public ont répondu à cette question :
  — 11 personnes ont choisi le premier format
  — 26 personnes ont choisi le deuxième format

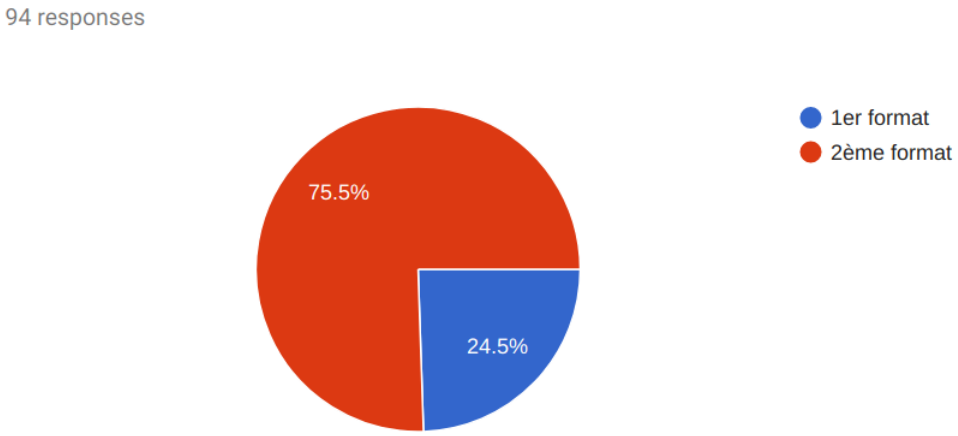


FIGURE 3.14 – Résultat des réponses à la question sur le format graphique

Question sur l'interface graphique

Dans un scénario, pour exprimer le fait qu'un patient est dans le salon par exemple,

nous pouvons vous proposer cette interface. Dans cette interface, on peut définir les rôles qui vont participer au scénario de l'activité, les équipements nécessaires et les zones, l'ajout se fait à l'aide d'un "drag and drop".Cette interface vous parait elle facile à utiliser ? Veuillez donner une note entre 0 et 5 pour représenter la facilité de cette interface en justifiant votre note : (0 : pas facile / 5 : très facile) (voir figure 3.15).

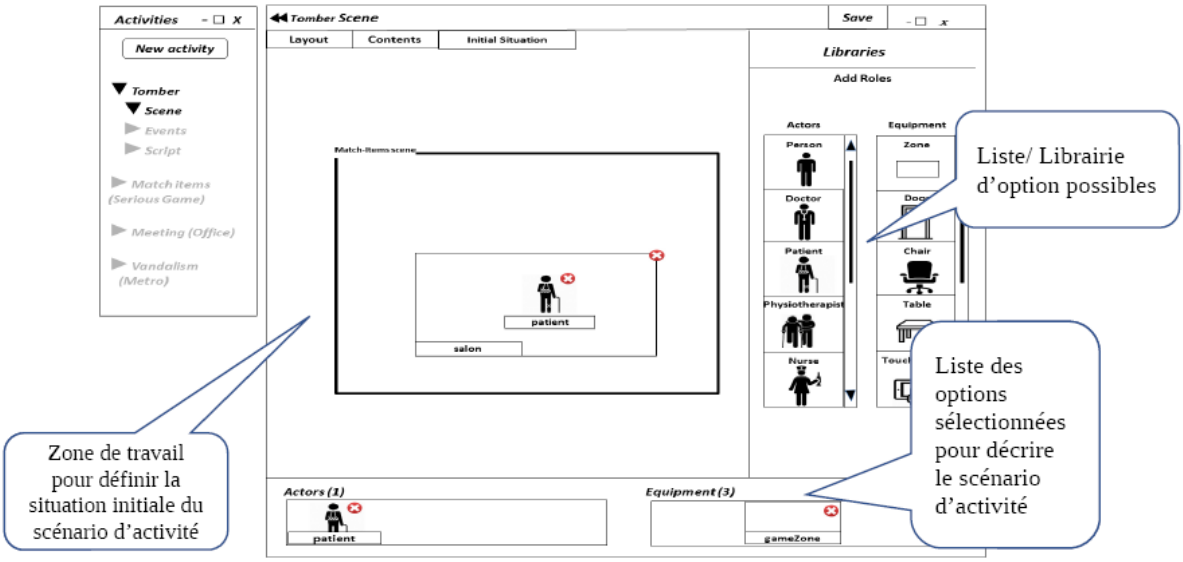


FIGURE 3.15 – Interface proposée pour la définition d'une scène

83 personnes parmi 103 personnes ont répondu à cette question, les résultats des réponses son comme suit (voir figure 3.16) :

1. 22.89% (19 personnes) ont donné la note 5
2. 30.12% (25 personnes) ont donné la note 4
3. 32.53% (27 personnes) ont donné la note 3
4. 8.43% (7 personnes) ont donné la note 2
5. 3.61% (3 personnes) ont donné la note 1
6. 2.40% (2 personnes) ont donné la note 0

Nous détaillons ces réponses par catégories d'utilisateurs comme suit :

— **Professionnels :**
24 professionnels ont répondu à cette question :
  — 7 personnes ont donné la note 5
  — 8 personnes ont donné la note 4
  — 8 personnes ont donné la note 3
  — 1 personnes a donné la note 2
  — 0 personnes ont donné la note 1
  — 0 personnes ont donné la note 0

— **Secteur médical :**
24 personnes du secteur médical ont répondu à cette question :
  — 3 personnes ont donné la note 5

- 7 personnes ont donné la note 4
- 11 personnes ont donné la note 3
- 1 personne a donné la note 2
- 2 personnes ont donné la note 1
- 0 personnes ont donné la note 0

— **Grand public :**
33 personnes du grand public ont répondu à cette question :

- 7 personnes ont donné la note 5
- 10 personnes ont donné la note 4
- 7 personnes ont donné la note 3
- 6 personnes ont donné la note 2
- 1 personne a donné la note 1
- 2 personnes ont donné la note 0

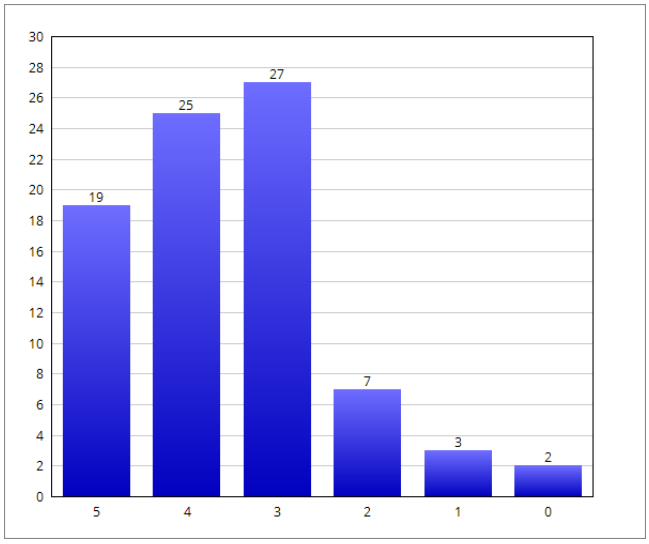


FIGURE 3.16 – Résultats de la réponse à la question 3

Les réponses à ce sondage ont montré que le langage ADeL sous ses deux formats est plus adéquat pour les utilisateurs non informaticiens. Nous avons même eu des commentaires comme :

1. ”Ce langage peut faciliter l’auto-formation”
2. ”Cela pourrait améliorer le temps de réaction et donc améliorer le résultat du traitement” (un kinésithérapeute)

Ce sondage n’est évidemment pas suffisant pour juger la recevabilité d’ADeL, c’est pourquoi nous envisageons des journées de sensibilisation avec un petit groupe de médecins pendant lesquelles nous allons expliquer plus en détails l’utilisation précise du langage et où nous demanderons aux participants de décrire et de tester eux-mêmes des activités plus complexes.

3.5 Conclusion

Dans ce chapitre, nous avons expliqué pourquoi nous avons choisi de créer un nouveau langage synchrone en étudiant des langages synchrones existants. Nous avons présenté ses concepts de base et ses opérateurs en insistant sur la possibilité du traitement du temps de la montre comme les autres évènements et la nécessité d'un noyau minimal et complet à partir duquel un utilisateur non-informaticien peut décrire une activité. ADeL a deux formats : textuel et graphique. Le langage textuel et l'interface graphique ont été définis avec l'aide des ergonomes de LudoTIC et validés par un nombre significatif d'utilisateurs. La syntaxe de ce langage concerne actuellement uniquement le noyau temporel minimal c'est-à-dire l'enchainement des actions. Il est clair qu'il faudra compléter la syntaxe concrète pour faciliter sa prise en main par des non-informaticiens. Nous envisageons d'ajouter du sucre syntaxique et des "patterns" d'opérations récurrentes ainsi que des compteurs pour faciliter l'utilisation intuitive du langage : le langage doit pouvoir être utilisé sans avoir une connaissance détaillée de la sémantique synchrone !

A partir d'une représentation d'activité avec ADeL, la compilation génère plusieurs formats pour plusieurs cibles : un model-checker pour la vérification ou notre moteur de reconnaissance d'activités en temps-réel. Pour une compilation facile et efficace, nous nous sommes appuyés sur une sémantique formelle qui sera décrite dans le chapitre suivant.

Chapitre 4

Sémantique et compilation du langage ADeL

L'essentiel : Ce chapitre décrit l'approche formelle sur laquelle se base notre langage. Nous présentons les deux sémantiques (comportementale et opérationnelle) que nous avons définies pour décrire le comportement des opérateurs et faciliter la compilation des programmes ADeL. Nous présentons le contexte mathématique sur lequel elles sont basées, nous décrivons leurs règles et nous démontrons ensuite la relation entre elles. Enfin, nous décrivons la compilation et la validation d'ADeL.

Sommaire

4.1 Introduction

Pour fournir des bases solides à notre langage, nous nous sommes tournés vers l'approche de la sémantique formelle. Nous avons défini deux sémantiques pour ADeL [SRMG18, SRM+18]. Premièrement, nous avons une *sémantique comportementale* à l'aide de règles de réécriture conditionnelles, qui sont une manière classique et plutôt naturelle pour exprimer formellement une sémantique intuitive. Cette forme de sémantique comportementale donne une description abstraite du comportement d'un programme et facilite son analyse. Cependant, elle est difficile à mettre en œuvre et n'est ni efficace pour la compilation ni adaptée aux preuves (par exemple, pour utiliser la technique du "model checking"). Par conséquent, nous avons également défini une *sémantique opérationnelle* qui traduit un programme ADeL dans un système d'équations booléennes représentant sa machine d'états. Le compilateur ADeL peut facilement traduire ce système d'équations en un code efficace. L'utilisation d'une telle sémantique est traditionnelle dans les langages synchrones [Ber00].

Comme nous avons deux sémantiques différentes, il est nécessaire d'étudier leur relation. Nous avons prouvé que ce qui est exécuté sur la base de la sémantique opérationnelle, se conforme également à la sémantique comportementale qui est la sémantique de référence. Nous commençons par décrire cette dernière.

4.2 Sémantique comportementale

Nous rappelons qu'un concept fondamental de l'approche synchrone est la notion d'instant logique (voir tableau 3.1). Un instant logique correspond à une réaction (voir section 3.2.1) et le but de la sémantique comportementale est d'exprimer ce qui se passe au cours d'une réaction, et de calculer formellement un ensemble d'événements "de sortie" à partir d'un ensemble d'événements "d'entrée" et de l'état.

Cette notion d'ensemble d'événements dans lequel évolue un programme est fondamentale dans l'expression des sémantiques. Nous l'appellerons *environnement*. Les définitions formelles d'environnement diffèrent dans la sémantique comportementale et dans la sémantique opérationnelle, toutefois ils représentent toujours un ensemble d'évènements manipulés au cours de l'exécution d'un programme et sont basés sur la *sorte* de celui-ci.

La sorte d'un programme ADeL P est constituée des instances des événements "génériques" déclarés dans le programme avec le mot clé **Events** et utilisées dans les instructions de P [1] ; elles constitueront les événements d'entrée de P. Les paramètres des instructions **emit** et **alert** font partie de la sorte de P et sont ses événements de sortie. Finalement, les événements déclarés dans l'instruction **local** constituent les événements locaux de la sorte de P [2].

Si l'on appelle $\mathcal{S}(\mathcal{P})$ la sorte d'un programme P, un environnement $E \subseteq \mathcal{S}(\mathcal{P})$ est l'ensemble des événements de la sorte de P présents.

1. Par exemple, lorsqu'un programme P possède une déclaration : **Events** $entre(Personne)$ et deux instructions $entre(p_1)$ et $entre(p_2)$, la sorte de P contiendra ces deux instances.

2. La syntaxe donnée dans la grammaire du langage exprime des événements de la forme $ident(ident_1, .., ident_n)$ avec $(ident_1, .., ident_n)$ optionnel. Du point de vue de l'expression de la sémantique, cette forme syntaxique n'a pas d'importance. Nous dénoterons les événements de la sorte d'un programme par une simple lettre (majuscule ou minuscule), sauf si nous avons besoin de nous référer à la syntaxe exacte.

La sémantique comportementale constitue la définition de référence d'ADeL. Pour la définir, nous avons adopté la même approche que [Ber93] pour le langage Esterel.

La sémantique comportementale définit un ensemble de règles de réécriture sous la forme $P \xrightarrow[I]{O} P'$ dont chacune décrit l'exécution du programme, où P est un programme ADeL, I est un environnement d'entrée, O est un environnement de sortie comportant les événements émis pendant la réaction de P sur I, et P' est la dérivée de P, qui représente le nouveau programme qui réagira au prochain environnement d'entrée dans l'instant suivant. Ainsi une réaction O_1, O_2, ..., O_n, ... à une suite d'environnements d'entrée I_1, I_2, ..., I_n, ... qui représentent les instants, est constituée d'une chaine de réactions élémentaires :

$$P \xrightarrow[I_1]{O_1} P_1 \xrightarrow[I_2]{O_2} .....P_n \xrightarrow[I_{n+1}]{O_{n+1}} P_{n+1}.....$$

La sémantique comportementale s'exécute structurellement sur un programme en appliquant des règles de réécriture définies pour chaque opérateur. Ainsi, les règles de réécriture de la sémantique s'appliquent à partir de l'instruction racine, en suivant structurellement l'arbre syntaxique du programme.

Les seuls opérateurs d'ADeL qui ajoutent un événement en tant que sortie sont l'opérateur **emit** et les opérateurs **stop when** et **timeout** lorsqu'ils lancent une alerte. Une *loi de cohérence logique*, introduite par G. Berry pour Esterel [Ber96] dit : "un signal S est présent dans une réaction si et seulement si une instruction **emit** S est exécutée". Dans notre cas, les alertes des opérateurs **stop when** et **timeout** sont similaires à **emit** (nous avons choisi le mot clé équivalent **alert** pour ces deux opérateurs dans un souci de précision). Les règles de réécriture sont une formalisation de cette loi. Une manière élégante pour exprimer ces règles de réécriture est d'utiliser le style "S.O.S" introduit par G. Plotkin [Plo81]. Ce style introduit des règles de déduction de la forme :

$$\frac{Prémisse(1)\ Prémisse(2)\ ...\ Prémisse(n)}{Conclusion}$$

signifiant : $\bigwedge_{i=1}^{i=n} Prémisse(i) \Rightarrow Conclusion.$

Une règle a la forme :

$$p \xrightarrow[E]{E',term} p'$$

où p et p' sont deux instructions d'ADeL, E est l'environnement courant dans lequel p s'exécute et qui précise l'état de présence des événements dans la sorte de p ; E' est l'environnement de sortie constitué des événements émis lors de la réaction, *term* est une expression booléenne[3] qui indique si p se termine dans la réaction courante et, dans ce cas, que les autres instructions en séquence peuvent commencer à se réécrire ; au contraire si p ne se termine pas dans la réaction courante, la réécriture est terminée pour cet instant.

E est composé de tous les événements présents dans l'instant considéré. Il doit donc inclure les événements de E', à cause la loi de cohérence logique évoquée ci-dessus. Mais E' dépend lui aussi de E. E et E' sont donc des points fixes.

Si p est l'instruction racine d'un programme P, la relation entre la sémantique comportementale de P et le système de réécriture de p est la suivante :

$$P \xrightarrow[I]{O} P' \text{ ssi } p \xrightarrow[I \cup O]{O, term_p} p'$$

3. Dans la suite, nous appellerons cette expression le "booléen de terminaison de p".

pour un booléen de terminaison $term_p$.

Nous définissons maintenant les règles de réécriture pour chaque opérateur d'ADeL.

Opérateur nothing

L'opérateur **nothing** n'a pas d'influence sur l'environnement courant, il commence et se termine au même instant et le booléen de terminaison *term* est toujours vrai.

$$\textbf{nothing} \xrightarrow[E]{\emptyset,\ tt} \textbf{nothing}$$

Opérateur wait

La sémantique de l'opérateur **wait** indique que ce dernier ne regarde pas si l'évènement est présent dans l'environnement au premier instant. Nous introduisons donc un opérateur intermédiaire que nous appelons **iwait** pour exprimer le comportement de wait. Ainsi, tout d'abord, **wait** S ne termine pas et se réécrit en **iwait** S.

$$\textbf{wait S} \xrightarrow[E]{\emptyset,ff} \textbf{iwait S} \qquad (wait)$$

iwait S réagit instantanément à la présence de l'événement S. Si S est présent, la valeur de *term* devient vraie et **iwait** se termine. Il se réécrit en **nothing** et il n'a aucune influence sur l'environnement courant.

$$\frac{S \in E}{\textbf{iwait S} \xrightarrow[E]{\emptyset,tt} \textbf{nothing}} \qquad (iwait1)$$

Si S n'est pas présent, la valeur de *term* est fausse et le **iwait** S n'évolue pas à cet instant. Il continue d'attendre S dans les instants suivants sans avoir aucune influence sur l'environnement.

$$\frac{S \notin E}{\textbf{iwait S} \xrightarrow[E]{\emptyset,ff} \textbf{iwait S}} \qquad (iwait2)$$

iwait n'est pas primitif, il peut se réécrire en **if** S **nothing else wait S**.

Il est à noter que si S est présent au premier instant et plus jamais par la suite, l'opérateur ne se terminera jamais et restera bloqué sur l'attente de S. Ce qui est le comportement souhaité.

Opérateur emit

L'opérateur **emit** met un événement présent dans l'environnement de sortie, il commence et se termine au même instant et change le booléen de terminaison en vrai :

$$\textbf{emit S} \xrightarrow[E\cup\{S\}]{\{S\},\ tt} \textbf{nothing}$$

Opérateur seq

L'opérateur **seq** a deux arguments. Il se comporte comme un opérateur de séquençage, le calcul du deuxième argument ne peut pas commencer tant que le premier argument n'est pas terminé. Si le premier argument ne se termine pas dans l'instant

(cas d'un **wait** par exemple), la séquence ne termine pas et se comporte comme son premier argument (règle $seq1$).

$$\frac{p_1 \xrightarrow[E]{E_1,\ ff} p_1'}{p_1 \textbf{ seq } p_2 \xrightarrow[E]{E_1,\ ff} p_1' \textbf{ seq } p_2} \qquad (seq1)$$

Lorsque le premier argument se termine, le second argument démarre dans la même réaction et la séquence se comporte comme ce dernier. De plus, les environnements de sortie respectifs sont fusionnés (règle $seq2$).

$$\frac{p_1 \xrightarrow[E]{E_1,\ tt} \textbf{nothing} \quad , \quad p_2 \xrightarrow[E]{E_2, term_{p_2}} p_2'}{p_1 \textbf{ seq } p_2 \xrightarrow[E]{E_1 \cup E_2, term_{p_2}} p_2'} \qquad (seq2)$$

Opérateur parallel

L'opérateur **parallel** (||) a deux arguments, il les calcule simultanément, éventuellement en écoutant et affectant les statuts des événements locaux. L'évolution des deux instructions peut avoir un impact à la fois sur les deux environnements. L'opérateur se termine lorsque les deux instructions se terminent, c'est-à-dire quand $term_{p_1}$ et $term_{p_2}$ deviennent vrais [4] et l'environnement de sortie résultant est la fusion des environnements résultants respectifs calculés pour p_1 et p_2.

$$\frac{p_1 \xrightarrow[E]{E_1, term_{p_1}} p_1' \quad , \quad p_2 \xrightarrow[E]{E_2, term_{p_2}} p_2'}{p_1 \| p_2 \xrightarrow[E]{E_1 \cup E_2,\ term_{p_1} \cdot\ term_{p_2}} p_1' \| p_2'} \qquad (parallel)$$

Opérateur while

L'opérateur **while** se comporte comme une boucle qui se termine uniquement quand sa condition n'est plus vraie.

Le corps de la boucle ne peut pas être instantané pour éviter en particulier des problèmes de causalité. En pratique, nous testons à l'analyse syntaxique que les boucles ne sont pas instantannées.

La condition d'un **while** est une combinaison booléenne de présence d'événement (S_1 and not S_2, par exemple).

Si la condition est évaluée à tt (noté $(cond, E) \rightarrowtail tt$ dans les règles), l'opérateur **while**($cond$, p) ne se termine pas dans l'instant et se comporte comme p' **seq while** ($cond$, p) (règle $while1$).

$$\frac{p \xrightarrow[E]{E_1, ff} p' \ ,\ (cond, E) \rightarrowtail tt}{\textbf{while}(cond, p) \xrightarrow[E]{E_1,\ ff} p' \textbf{ seq while}(cond, p)} \qquad (while1)$$

Si la condition n'est plus vraie (évaluée à ff), l'opérateur **while** se termine et se réécrit en **nothing**, son booléen de terminaison devient vrai. L'environnement final est

4. Dans la règle de l'opérateur ($parallel$), "$term_{p_1}.term_{p_2}$" représente la conjonction booléenne de $term_{p_1}$ et de $term_{p_2}$.

l'environnement de sortie E' résultant de l'exécution de p (règle $while2$).

$$\frac{p \xrightarrow[E]{E_1, ff} p' \ , \ (cond, E) \rightarrowtail \ ff}{\textbf{while}(cond, p) \xrightarrow[E]{E_1, \ tt} \textbf{nothing}} \quad (while2)$$

Opérateur if..then..else

Cet opérateur a le comportement habituel des opérateurs de test. Son comportement dépend de la condition à vérifier. Si la condition est évaluée vraie par rapport à l'environnement d'entrée, l'instruction **then** est exécutée :

$$\frac{p_1 \xrightarrow[E]{E_1, term_{p_1}} p_1', \ (cond, E) \rightarrowtail tt}{\textbf{if}(cond, p_1, p_2) \xrightarrow[E]{E_1, term_{p_1}} p_1'} \quad (if1)$$

Sinon, l'instruction **else** est exécutée :

$$\frac{p_2 \xrightarrow[E]{E_2, term_{p_2}} p_2', (cond, E) \rightarrowtail ff}{\textbf{if}(cond, p_1, p_2) \xrightarrow[E]{E_2, term_{p_2}} p_2'} \quad (if2)$$

Comme dans le cas de l'opérateur **while**, la condition peut être une expression booléenne de présence d'événements.

Opérateur stop..when..

Le comportement de cet opérateur dépend de la réécriture de son argument p et de la présence de l'événement d'abandon S. Cet opérateur ne regarde pas la valeur de S au premier instant. Ainsi, si p est instantané l'opérateur se termine.

$$\frac{p \xrightarrow[E]{E_1, tt} \textbf{nothing}}{\textbf{stop}(p, \ S, \ S_1) \xrightarrow[E]{E_1, \ tt} \textbf{nothing}} \quad (stop1)$$

En revanche, si p ne se termine pas dans l'instant, S sera testé dans l'instant suivant et le comportement de l'opérateur dépendra du résultat du test. La dérivée peut s'exprimer avec un **if..then..else** :

$$\frac{p \xrightarrow[E]{E_1, \ ff} p'}{\textbf{stop}(p, \ S, \ S_1) \xrightarrow[E]{E_1, \ ff} \textbf{if } S \textbf{ then emit } S_1 \textbf{ else stop}(p', \ S, \ S_1)} \quad (stop2)$$

Pstop a le même comportement que **stop**, sauf que dans le cas où S est présent, il génère un deuxième évènement $failure$ en plus. L'opérateur **Pstop** a deux règles, la première est la même que la règle $stop1$, et la seconde est la règle $Pstop2$:

$$\frac{p \xrightarrow[E]{E_1, \ ff} p'}{\textbf{Pstop}(p, \ S, \ S_1) \xrightarrow[E]{E_1, \ ff} \textbf{if } S \textbf{ then emit } S_1 \textbf{ seq emit } failure \textbf{ else Pstop}(p', \ S, \ S_1)} \quad (Pstop2)$$

Opérateur timeout

Le comportement de l'opérateur p **timeout** $S\{p_1\}$ **alert** S_1 dépend du calcul de son instruction p et du statut de S.

Cet opérateur permet d'arrêter son argument p si une condition de temps représentée par l'évènement S est vraie avant la terminaison de p. Sinon, p_1 s'exécute. Comme **stop**, cet opérateur n'écoute pas S au premier instant et nous le dérivons en **if..then..else** tant que l'évaluation de p n'est pas terminée (règle *timeout*2). En revanche, dès que p termine son évaluation ou bien, s'il est instantané, l'opérateur se comporte comme p_1 (règle *timeout*1). Même si S est simultané avec la terminaison normale de l'argument p, la préemption l'emporte, p_1 n'est pas exécuté et l'alerte est émise.

$$\frac{p \xrightarrow[E]{E_1, tt} \textbf{nothing}, p_1 \xrightarrow[E]{E_2, term_{p_1}} p_1'}{\textbf{timeout}(p,\ S,\ p_1, S_1) \xrightarrow[E]{E_1 \cup E_2, term_{p_1}} p_1'} \quad (timeout1)$$

$$\frac{p \xrightarrow[E]{E_1, ff} p'}{\textbf{timeout}(p,\ S,\ p_1, S_1) \xrightarrow[E]{E_1, ff} \textbf{if } S \textbf{ then emit } S_1 \textbf{ else timeout}(p',\ S,\ p_1, S_1)} \quad (timeout2)$$

Les règles de l'opérateur **Ptimeout** de timeout prioritaire sont calquées sur celle de **Pstop**. Cet opérateur émet juste un événement $failure$ supplémentaire en cas de préemption prioritaire. La règle *timeout*1 qui concerne la terminaison normale de p reste inchangée. En revanche,la règle *timeout*2 qui teste l'événement S devient :

$$\frac{p \xrightarrow[E]{E_1, ff} p'}{\textbf{Ptimeout}(p,\ S,\ p_1, S_1) \xrightarrow[E]{E_1, ff} \textbf{if } S \textbf{ then } Ptimeout\_fail \textbf{ else Ptimeout1}(p',\ S,\ p_1, S_1)} \quad (Ptimeou2)$$

et $Ptimeout\_fail \equiv$ **emit** S_1 **seq emit** $failure$

Opérateur local

L'opérateur **local** S $\{p\}$ se comporte comme une encapsulation. Il est utilisé pour introduire des événements permettant la communication entre instructions dans un programme. Un événement local peut être émis et écouté dans son argument. Les règles de réécriture distinguent le cas où S est émis dans p ou non. En effet, en accord avec la loi de cohérence logique, si S n'est pas émis dans p, il n'appartiendra pas à E et les instructions de p qui l'écoutent réagiront en conséquence. La première règle ($local1$) considère le cas où S est dans E', la seconde ($local2$) celui où il ne l'est pas.

$$\frac{p \xrightarrow[E \cup \{S\}]{E' \cup \{S\},\ term_p} p'}{\textbf{local}(S, p) \xrightarrow[E]{E',\ term_p} \textbf{local}(S, p')} \quad (local1)$$

$$\frac{p \xrightarrow[E \setminus \{S\}]{E',\ term_p} p', \quad S \notin E'}{\textbf{local}(S, p) \xrightarrow[E]{E',\ term_p} \textbf{local}(S, p')} \quad (local2)$$

Arbres de preuve

A titre d'illustration, nous détaillons la suite des réécritures (appelée arbre de preuve) qui caractérisent un instant pour l'instruction suivante :

```
local S { if S then emit T else nothing parallel emit S }
```

$$\dfrac{\dfrac{\dfrac{}{\textbf{emit } T \xrightarrow[\{S,T\}]{\{T\},tt} \textbf{nothing}}, \quad \textbf{emit } S \xrightarrow[\{S,T\}]{\{S\},tt} \textbf{nothing}}{\textbf{if } S \textbf{ then emit } T \textbf{ else nothing} \xrightarrow[\{S,T\}]{\{T\},tt} \textbf{nothing}}}{\textbf{if } S \textbf{ then emit } T \textbf{ else nothing parallel emit } S \xrightarrow[\{S,T\}]{\{S,T\},tt} \textbf{nothing}}}{\textbf{local } S \ \{\textbf{if } S \textbf{ then emit } T \textbf{ else nothing parallel emit } S\ \} \xrightarrow[\{T\}]{\{T\},tt} \textbf{nothing}}$$

Les points fixes sont calculés dans la sémantique comportementale, puisque E' dépend de E, car d'une part, on calcule les événements émis à partir de E et d'autre part, E dépend de E' puisque tout événement émis est dans E à cause de l'hypothèse synchrone. On a donc une suite de "micro steps" qui ajoutent des événements de E' dans E pour calculer le micro step suivant. Tant qu'une instruction a pour booléen de terminaison vrai, on continue à dérouler ces micro steps jusqu'à stabilisation de E et E'.

Un instant est ainsi défini : soit l'on a un micro step de la forme $p \xrightarrow[E]{E',ff} p'$ ou bien le terme se réécrit en **nothing**. Cette instruction n'affectant ni E ni E', on atteint la stabilisation.

La sémantique comportementale est structurelle. Un macro step (c'est à dire une stabilisation de micro steps) définit un instant. Toutefois elle n'exprime pas comment construire les points fixes. Elle accepte des programmes qui ne sont pas "logiquement correct" c'est à dire non réactifs ou non déterministes. Pour appliquer les règles de la sémantique comportementale, des suppositions sur l'état de présence ou d'absence de certains événements est nécessaire. Par exemple, il est bien connu [Ber93] que le programme :

```
local S {if S then emit S else nothing}      (P1)
```

n'est pas déterministe. En effet, si l'on considère l'instruction du **local**, on peut aussi bien supposer S présent ou absent. Si S est présent, la règle *local*1 s'applique ; si S est absent c'est la règle *local*2. Le programme a deux solutions et il n'est pas déterministe. Quand au programme

```
local S {if S then nothing else emit S}      (P2)
```

aucune supposition ne permet d'appliquer une règle. Le programme n'a pas de solution et il n'est pas réactif. En fait, ces deux programmes ont des cycles de dépendance entre S et lui même. Dans l'approche synchrone, il est facile d'écrire de tels cycles. Ce problème de cycles de dépendance est un problème de causalité et pour l'éviter notre compilateur vérifiera que la dépendance des événements est *acyclique* (voir section 4.6.1). Dans ce cas, un événement a un statut de présence unique et les programmes sont réactifs et déterministes.

La sémantique comportementale donne juste une définition logique du comportement des programmes ADeL. Elle est simple et intuitive mais ne peut servir de support à une implémentation car elle est inefficace et ne permet pas de caratériser les programmes qui ont une solution unique. C'est pourquoi nous allons considérer une

sémantique "opérationnelle" effective et qui permet de vérifier facilement la causalité des événements d'un programme.

4.3 Contexte algébrique de la sémantique opérationnelle

La sémantique opérationnelle d'ADeL repose sur la notion d'environnement qui est un ensemble fini d'événements qui connaissent des informations sur leur état de présence ou d'absence, appelé "statut". Les environnements mémorisent les statuts de leurs événements à chaque instant.

Afin de définir une sémantique constructive qui ne fait pas de suppositions sur les statuts des évènements comme la sémantique comportementale, mais qui propage l'information depuis les événements d'entrée vers les événements de sortie, nous introduisons un contexte algébrique quadri-valué [Bel77] pour représenter les statuts des événements. Dans un tel contexte, un statut pourra évoluer de $\perp$ vers 0 ou 1.

Notre but est de définir une sémantique constructive effective qui nous permet de compiler les programmes ADeL. Dans une telle sémantique, les cycles de causalité se détectent en étudiant les chaines de causalité qui relient les événements de sortie aux événements d'entrée. Mais l'ordre induit par la dépendance des évènements entre eux n'est pas toujours total, il y a des événements indépendants et en général on a un ensemble d'ordres partiels qui décrivent la causalité des événements d'un programme. Les approches constructives cherchent à construire un ordre de dépendance total à partir de ces ordres partiels, dans chaque instant. Mais fixer un ordre total à chaque instant peut introduire de faux cycles de causalité. Dans notre approche, nous voulons définir une sémantique opérationnelle nous permettant de construire facilement les ordres partiels de dépendance d'un programme. Une algèbre quadri-valuée ξ va nous permettre d'étudier finement comment l'information croit de $\perp$ à $\top$ (voir figure 4.1) et d'en déduire une étude de la causalité qui s'appuie sur la construction des ordres partiels de dépendance des événements. Nous détaillerons cette approche dans la section dédiée à la compilation, mais avant de continuer nous décrivons l'algèbre ξ.

4.3.1 L'algèbre ξ quadri-valuée

Dans notre équipe, plusieurs algèbres ont été testées pour exprimer les échanges d'information entre les systèmes synchrones. Notre recherche a convergé il y a quelques années vers une algèbre à 4 valeurs ($\xi = \{\perp, 0, 1, \top\}$) [Bel77] qui représente le statut des événements et qui permet d'exprimer simplement l'union des environnements de signaux calculés pour chaque système synchrone parallèle [GR13].

$\perp$ signifie que le statut de l'événement n'est pas encore déterminé, 0 que l'événement est absent, 1 que l'événement est présent, et $\top$ que l'événement a deux statuts incompatibles dans le même instant (par exemple, il reçoit les valeurs 0 et 1 par différentes parties du programme). Nous verrons que cette algèbre nous permet de calculer l'information relative aux statuts des évènements de façon plus fine qu'avec une algèbre booléenne.

Nous nous appuyons sur la sémantique opérationnelle pour faire la vérification et assurer une compilation correcte vers un système d'équations à valeur dans ξ à chaque programme. Dans chaque réaction, le système d'équations nous aide à calculer l'environnement de sortie à partir d'un environnement d'entrée. Pour cela, nous allons

munir l'algèbre ξ des opérateurs logiques usuels, notés ici $\neg$, $\boxplus$, $\boxdot$ qui doivent avoir une interprétation "booléenne" pour pouvoir exprimer le statut des sorties et des évènements locaux à partir des statuts des entrées et des évènements locaux introduits par l'opérateur **local**. De plus, pour tous les opérateurs prenant en entrée deux systèmes d'équation P_1 et P_2, le système d'équations global doit être déduit des systèmes d'équation de P_1 et P_2. En particulier, pour l'opérateur P_1 **parallel** P_2 ($P_1 \| P_2$), lorsque le même événement a des statuts différents dans chaque système d'équations, son statut dans le système d'équations résultant devrait être "l'unification" de l'information portée par ses statuts respectifs dans les systèmes d'équations de P_1 et de P_2.

Par exemple, supposons que l'événement S a le statut $\perp$ dans le système d'équations de P_1 et le statut 1 dans le système d'équations de P_2, alors il doit avoir le statut 1 dans le système d'équations de $P_1 \| P_2$. Ainsi, nous devrons aussi munir ξ d'une telle opération, appelée *unification* ($\sqcup$) qui effectue l'union de l'information concernant un événement dans des environnements différents.

D'autre part, nous avons besoin d'un contexte algébrique bien fondé dans lequel la notion d'accroissement de l'information a une définition mathématique. En effet, à chaque accroissement de l'information, on raffine le statut des signaux et on cherche un plus petit point fixe pour assurer une solution unique du calcul de l'environnement de sortie. Pour s'assurer que les plus petits points fixes existent et peuvent être calculés, nous avons besoin d'une algèbre quadri-valuée avec des opérateurs qui rendent l'évolution de l'information monotone [Tar55].

Pour remplir ces conditions, nous nous intéressons aux algèbres quadri-valuées munies d'une structure de bitreillis (ou bilattice) [Gin88]. Les bitreillis sont des structures mathématiques ayant deux ordres partiels distincts notés $\leq_B$ et $\leq_K$ et une opération de ξ dans ξ (une extension de la négation booléenne) $\neg$. $\leq_B$ représente une extension de l'ordre booléen habituel et $\leq_K$ exprime le niveau de connaissance sur la présence d'un événement.

Définition 1 *(Ginsberg [Gin88]) Un* ***bitreillis*** *est une structure* $(\mathcal{B}, \leq_B \leq_K, \neg)$ *qui possède un ensemble non vide* $\mathcal{B}$*, deux ordres partiels* $\leq_B$ *et* $\leq_K$ *et une fonction* $\neg : \mathcal{B} \mapsto \mathcal{B}$ *tel que :*

1. $(\mathcal{B}, \leq_B)$ *et* $(\mathcal{B}, \leq_K)$ *sont des treillis complets*
2. $x \leq_B y \Rightarrow \neg y \leq_B \neg x$, $\forall x, y \in \mathcal{B}$
3. $x \leq_K y \Rightarrow \neg x \leq_K \neg y$, $\forall x, y \in \mathcal{B}$
4. $\neg\neg x = x$, $\forall x \in \mathcal{B}$

Dans ξ, nous définissons les deux ordres partiels comme suit (ce qui correspond à la figure 4.1.

$\perp$	$\leq_K$	0	$\leq_K$	$\top$
$\perp$	$\leq_K$	1	$\leq_K$	$\top$

0	$\leq_B$	$\perp$	$\leq_B$	1
0	$\leq_B$	$\top$	$\leq_B$	1

Ainsi, dans $\leq_B$ $\perp$ et $\top$ sont incomparables car d'un point de vue booléen, le niveau de connaissance n'est pas significatif. En revanche, dans $\leq_K$, ce sont 0 et 1 qui ne sont pas comparables car on n'a pas plus de connaissance dans un cas comme dans l'autre.

Notre but est de munir $(\xi, \leq_B, \leq_K, \neg)$ d'une structure de bitreillis. Pour cela, nous devons doter $(\xi, \leq_K)$ et $(\xi, \leq_B)$ d'une structure de treillis.

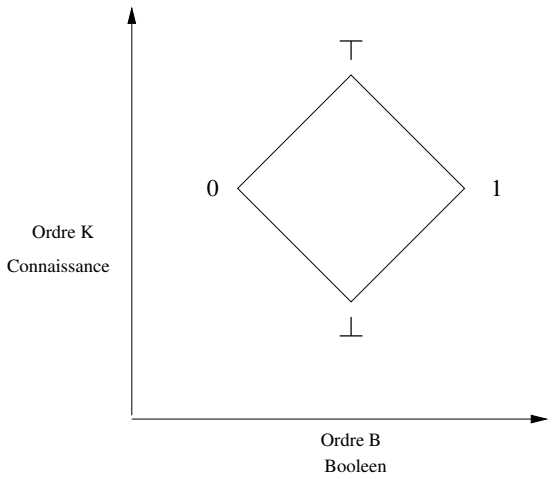


FIGURE 4.1 – Ordres partiels de ξ

Définition 2 *Un **treillis** est un ensemble muni d'un ordre partiel (L,≤) dans lequel chaque paire d'éléments (a, b) a une borne supérieure (a∨b) et borne inférieure (a∧b). L est complet si ∀X ⊆ L, X a une plus petite borne supérieure et une plus grande borne inférieure.*

Nous définissons aussi $\sqcup$ et $\sqcap$comme les opérations de borne supérieure et de borne inférieure pour l'ordre $\leq_K$; compte tenu de la définition de $\leq_K$, nous obtenons la table de vérité suivante :

$\sqcup$	1	0	$\top$	$\bot$
1	1	$\top$	$\top$	1
0	$\top$	0	$\top$	0
$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	1	0	$\top$	$\bot$

$\sqcap$	1	0	$\top$	$\bot$
1	1	$\bot$	1	$\bot$
0	$\bot$	0	0	$\bot$
$\top$	1	0	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

(ξ, $\leq_B$) doit également être un treillis. Ainsi, nous définissons de même $\boxplus$et $\boxdot$comme la borne supérieure et la borne inférieure pour $\leq_B$.

$\boxplus$	1	0	$\top$	$\bot$
1	1	1	1	1
0	1	0	$\top$	$\bot$
$\top$	1	$\top$	$\top$	1
$\bot$	1	$\bot$	1	$\bot$

$\boxdot$	1	0	$\top$	$\bot$
1	1	0	$\top$	$\bot$
0	0	0	0	0
$\top$	$\top$	0	$\top$	0
$\bot$	$\bot$	0	0	$\bot$

Finalement, nous définissons l'opération $\neg$. En effet, nous voulons que cette opération inverse la valeur booléenne, mais son rôle par rapport à $\leq_K$ doit rester transparent : nous n'avons pas plus de connaissance au sujet de x que de $\neg x$:

x	$\neg\ x$
1	0
0	1
$\top$	$\top$
$\bot$	$\bot$

$x \leq_K (x \sqcup y)$	$x \leq_B (x \boxplus y)$
$x \leq_B y$ et $z \leq_B t \Rightarrow x \sqcup z \leq_B y \sqcup t$	$x \leq_K y \Rightarrow (x \sqcup z) \leq_K (y \sqcup z)$
$(x \boxdot y) \leq_B x$	$x \leq_B y$ et $z \leq_B t \Rightarrow x \sqcap z \leq_B y \sqcap t$
$x \leq_K y \Rightarrow (x \sqcap z) \leq_K (y \sqcap z)$	$x \leq_B y \Rightarrow (x \boxdot z) \leq_B (y \boxdot z)$
$x \leq_K y$ et $z \leq_K t \Rightarrow x \boxplus z \leq_K y \boxplus t$	$\bot \leq_K (x \sqcup y) \leq_K \top$
$x \leq_B y \Rightarrow (x \boxdot z) \leq_B (y \boxdot z)$	$x \leq_K y$ et $z \leq_K t \Rightarrow x \boxdot z \leq_K y \boxdot t$
$\bot \leq_K (x \sqcap y) \leq_K \top$	$0 \leq_B (x \boxplus y) \leq_B 1$
$x \leq_K y \Rightarrow \neg x \leq_K \neg y$	$0 \leq_B (x \boxdot y) \leq_B 1$
$x \leq_B y \Rightarrow \neg y \leq_B \neg x$	

TABLEAU 4.1 – Les propriétés des ordres $\leq_K$ et $\leq_B$

Avec ces définitions, les ordres ont les propriétés décrites dans le tableau 4.1 et prouvées dans [GR12].

La structure $(\xi, \leq_B, \leq_K, \neg)$ est un bitreillis : (1) par construction, $(\xi, \leq_K)$ est un treillis avec $\bot$ et $\top$ comme extremum, ainsi que $(\xi, \leq_B)$ avec 0 et 1 comme extremum. Selon le tableau 4.1, nous pouvons assurer que l'opérateur $\neg$ inverse l'ordre booléen et préserve l'ordre des connaissance. Enfin, évidemment, $\neg\neg x = x$, pour chaque élément de ξ.

Bien que l'algèbre ξ puisse être considérée comme une extension de l'algèbre booléenne, elle n'est pas elle-même une algèbre booléenne. Par exemple, $x \boxplus \neg x$ n'est pas toujours égal à 1. Ainsi, nous ne pouvons pas appliquer les lois de l'algèbre booléenne. Par conséquent, nous étudions lesquelles de ces lois sont toujours vraies dans le bitreillis $(\xi, \leq_B, \leq_K, \neg)$. Les résultats sont détaillés dans le tableau 4.2 et les preuves sont dans [GR12].

Ces lois sont utiles pour calculer les solutions des systèmes d'équations à valeur dans ξ. De plus, la distributivité est importante pour appliquer les propriétés de bitreillis.

Nous allons montrer que l'avantage de considérer une algèbre quadri-valuée est que tout élément peut être encodé par une paire de booléens. Ceci permet de construire des systèmes d'équations booléennes à partir de systèmes d'équations à valeur dans ξ. Ainsi, nous aurons un moyen effectif de calculer les solutions de ces derniers en utilisant une approche booléenne classique.

4.3.2 L'encodage de l'algèbre ξ

Comme nous l'avons dit, l'objectif de la sémantique opérationnelle est d'associer à chaque programme un système d'équations de l'algèbre ξ qui calcule le statut de ses événements à chaque instant. Pour rendre ce calcul effectif, nous définissons un encodage des éléments de ξ en paires de booléens. D'un point de vue pratique, nous nous appuyons sur un tel encodage pour implémenter les automates implicites construits par la sémantique opérationnelle sous forme de système d'équations booléennes. Ces dernières seront représentées à l'aide d'un module de BDD (Binary Decision Diagrams), développé dans l'équipe, dédié à la représentation d'équations booléennes, sachant que toutes autres bibliothèques manipulant des expressions booléennes auraient répondu à notre attente. Il existe plusieurs fonctions d'encodage possibles et nous avons choisi celle qui nous semble la plus appropriée pour exprimer l'information croissante par rapport à l'ordre $\leq_K$.

Lois des éléments neutres et absorbants:

$\bot \sqcup x = x \quad 1 \boxplus x = 1 \quad 0 \boxdot x = 0 \quad \top \sqcup x = \top$

$\bot \sqcap x = \bot \quad 0 \boxplus x = x \quad 1 \boxdot x = x \quad \top \sqcap x = x$

Lois de distributivité:

$(x \boxplus y) \boxdot z = (x \boxdot z) \boxplus (y \boxdot z) \quad (x \sqcup y) \sqcap z = (x \sqcap z) \sqcup (y \sqcap z)$

$(x \sqcup y) \boxplus z = (x \boxplus z) \sqcup (y \boxplus z) \quad (x \boxdot y) \boxplus z = (x \boxplus z) \boxdot (y \boxplus z)$

$(x \sqcap y) \sqcup z = (x \sqcup z) \sqcap (y \sqcup z) \quad (x \sqcup y) \boxdot z = (x \boxdot z) \sqcup (y \boxdot z)$

$(x \sqcap y) \boxplus z = (x \boxplus z) \sqcap (y \boxplus z) \quad (x \boxplus y) \sqcup z = x \sqcup z \boxplus y \sqcup z$

$(x \boxdot y) \sqcup z = x \sqcup z \boxdot y \sqcup z \quad (x \sqcap y) \boxdot z = (x \boxdot z) \sqcap (y \boxdot z)$

$(x \boxplus y) \sqcap z = x \sqcap z \boxplus y \sqcap z \quad (x \boxdot y) \sqcap z = x \sqcap z \boxdot y \sqcap z$

Lois d'associativité:

$(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z) \quad (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$

$(x \boxplus y) \boxplus z = x \boxplus (y \boxplus z) \quad (x \boxdot y) \boxdot z = x \boxdot (y \boxdot z)$

Lois d'absorption:

$(x \sqcup y) \sqcap x = x \quad (x \sqcap y) \sqcup x = x$

$(x \boxplus y) \boxdot x = x \quad (x \boxdot y) \boxplus x = x$

Lois d'idempotence:

$x \sqcup x = x \quad x \sqcap x = x \quad x \boxplus x = x \quad x \boxdot x = x$

Lois De Morgan:

$\neg(x \boxdot y) = \neg x \boxplus \neg y \quad \neg(x \boxplus y) = \neg x \boxdot \neg y$

$\neg(x \sqcup y) = \neg(x) \sqcup \neg(y) \quad \neg(x \sqcap y) = \neg(x) \sqcap \neg(y)$

TABLEAU 4.2 – Les propriétés de l'algèbre ξ

Pour une comparaison des trois encodages acceptables voir [GR12].

$$e : \xi \mapsto \mathbb{B} \times \mathbb{B} : x \in \xi,\ e(x) = (x_h, x_l)$$

Ici $\mathbb{B}$ est l'ensemble booléen habituel avec deux éléments : tt, ff et les opérations $+$ et $.$ ainsi que la négation ($\overline{x}$). e est défini comme suit :

$$\begin{array}{lcl} \bot & \mapsto & (ff, ff) \\ 0 & \mapsto & (ff, tt) \\ 1 & \mapsto & (tt, ff) \\ \top & \mapsto & (tt, tt) \end{array}$$

La fonction de codage décrite précédemment s'étend aux opérateurs de ξ. La structure $(\mathbb{B}, \leq)$ est un treillis complet pour l'ordre $ff \leq tt$. De plus, $a + b$ et $a \cdot b$ sont respectivement la borne inférieure et la borne supérieure de a et b dans $(\mathbb{B}, \leq)$. La structure : $\mathbb{B} \bigodot \mathbb{B} = (\mathbb{B} \times \mathbb{B}, \leq_B, \leq_K, \neg)$ définie comme suit

$$\begin{array}{lll} (x_1, x_2) \leq_B (y_1, y_2) & \text{iff} & x_1 \leq y_1 \ and \ y_2 \leq x_2 \\ (x_1, x_2) \leq_K (y_1, y_2) & \text{iff} & x_1 \leq y_1 \ and \ x_2 \leq y_2 \\ \neg(x_1, x_2) = (x_2, x_1) & & \end{array}$$

est un bitreillis.

Dans la structure $\mathbb{B} \bigodot \mathbb{B}$, il est façile de vérifier [BMS97] que les opérations sont définies de la façon suivante [5].

5. Dans $\mathbb{B} \bigodot \mathbb{B}$, nous dénotons la borne inférieure et la borne supérieure de $\leq_B$ par $\boxdot$ et $\boxplus$; la borne inférieure et la borne supérieure de $\leq_K$ par $\sqcup$ et $\sqcap$

$$\begin{array}{rcl}(c_1,d_1)\sqcup(c_2,d_2) & = & (c_1+c_2,d_1+d_2)\\ (c_1,d_1)\sqcap(c_2,d_2) & = & (c_1.c_2,d_1.d_2)\\ (c_1,d_1)\boxplus(c_2,d_2) & = & (c_1+c_2,d_1.d_2)\\ (c_1,d_1)\boxdot(c_2,d_2) & = & (c_1.c_2,d_1+d_2)\end{array}$$

Le théorème suivant nous permet de plonger les équations quadri-valuées dans le monde booléen.

Théorème 1 *$(\xi, \leq_B, \leq_K, \neg)$ et $\mathbb{B}\bigodot\mathbb{B}$ sont isomorphes.*

Pour prouver ce théorème, nous montrons que l'encodage e précédemment défini est un isomorphisme entre $(\xi, \leq_B, \leq_K, \neg)$ et $\mathbb{B}\bigodot\mathbb{B}$. Pour cela nous montrons que les quatre opérations binaires et la négation du bitreillis $(\xi, \leq_B, \leq_K, \neg)$ sont conservées dans $\mathbb{B}\bigodot\mathbb{B}$. La preuve est détaillée en annexe B.

En conséquence du théorème, nous pouvons étendre l'encodage de e précédemment défini pour les éléments ξ aux opérateurs du bitreillis $(\xi, \leq_B, \leq_K, \neg)$:

$$\left[\begin{array}{rcl}e(x\sqcup y) & = & (x_h+y_h, x_l+y_l)\\ e(x\sqcap y) & = & (x_h.y_h, x_l.y_l)\\ e(x\boxplus y) & = & (x_h+y_h, x_l.y_l)\\ e(x\boxdot y) & = & (x_h.y_h, x_l+y_l)\end{array}\right]$$

Ainsi, nous pouvons convertir efficacement les systèmes d'équations de ξ dans l'univers booléen, ce qui nous permet de définir la sémantique opérationnelle.

4.4 Sémantique opérationnelle

La sémantique comportementale de la section 4.2 n'est pas efficace et trop laxiste car elle repose sur des suppositions de présence ou d'absence d'événements. Elle ne se soucie pas de la causalité des événements entre eux. Pour faire face à ce problème, la notion de sémantique constructive a été introduite pour Esterel [Ber96]. Dans l'approche constructive, l'idée de vérifier des suppositions sur les statuts de présence des événements est remplacée par la propagation d'information sur le flot de contrôle et les statuts des événements, en respectant la causalité. Par exemple, si l'on considère l'instruction ADeL : **if** I **then emit** O, la présence de I cause celle de O et son absence cause celle de O. Intuitivement, pour représenter la propagation des statuts des événements, il est pratique d'utiliser des équations. Ainsi la propagation de l'information sur les statuts des événements du programme :

```
local S1, S2
{
    if I then emit S1
  ||
    if S1 then nothing else emit S2
  ||
    if S2 then emit O
}
```

peut se représenter par le système d'équations [6] :

6. Dans ce système d'équations, les statuts sont booléens. On considère l'algèbre booléenne usuelle et nous dénoterons + l'opérateur "ou", . l'opérateur "et", $\overline{x}$ la négation de x.

$$\begin{array}{lcl} \text{S1} & = & \text{I} \\ \text{S2} & = & \overline{\text{S1}} \\ \text{O} & = & \text{S2} \end{array}$$

Donc, si le statut de I est *tt* (présent), celui de O sera ff c'est à dire absent et si I est absent le statut de O sera présent. D'une part ce système permet de calculer les statuts des événements de sortie et des locaux en propageant l'information. D'autre part, le test d'acyclicité est facile à faire. Par exemple, si l'on considère les programmes P1 et P2 introduits dans la section 4.2, leurs systèmes respectifs sont S = S pour P1 et S = $\overline{\text{S}}$ pour P2. Les cycles de causalité sont clairs. Pour des programmes plus compliqués, la détection reste simple. La sémantique opérationnelle met en œuvre cette démarche.

Nous rappelons que, comme tous les langages synchrones, ADeL utilise des évènements (appelés signaux ou flots dans les autres langages synchrones) à diffusion instantanée comme moyen de communication entre sous programmes et avec l'environnement. Un programme réagit à des événements d'entrée qui connaissent leur statut en produisant des évènements de sortie, avec eux aussi un statut. Dans la sémantique opérationnelle, la notion d'environnement diffère par rapport à la sémantique comportementale. Un environnement est toujours un ensemble fini d'événements avec un statut, mais ce dernier est dans l'algèbre ξ.

4.4.1 Environnements

Grâce à l'algèbre ξ, nous pouvons maintenant introduire formellement la notion d'*environnement*. Un environnement est un ensemble fini d'événements où chaque événement a un statut unique. Plus formellement, nous considérons un ensemble fini d'événements : $\mathcal{S} = \{S_0, S_1, ...S_n, ...\}$. Une évaluation $\mathcal{V} : \mathcal{S} \mapsto \xi$ est une fonction qui associe un événement $S \in \mathcal{S}$ à une valeur de ξ. Chaque évaluation $\mathcal{V}$ définit un environnement : $E = \{S^x \mid S \in \mathcal{S}, x \in \xi, \mathcal{V}(S) = x\}$. Le but de la sémantique est d'affiner le statut des événements d'un programme à chaque instant de $\bot$ à $\top$ selon l'ordre de connaissance ($\leq_K$).

Ainsi, pour chaque programme P construit avec des opérateurs d'ADeL, on note $\mathcal{S}(P)$ l'ensemble fini de ses événements et $\mathcal{E}(P)$ l'ensemble de tous les environnements possibles créés à partir de $\mathcal{S}(P)$. Les opérations dans (ξ, $\leq_B$, $\leq_K$, $\neg$) peuvent être étendues aux environnements. Nous présentons seulement les opérations nécessaires pour définir les deux sémantiques. Cependant, les cinq opérateurs de ξ peuvent être étendus de la même manière.

$$\begin{array}{lcl} \neg E & = & \{S^x | S^{\neg x} \in E\} \\ E \sqcup E' & = & \{S^z | \exists S^x \in E, \exists S^y \in E', z = x \sqcup y\} \\ & & \cup \{S^x | S^x \in E, \nexists y \in \xi, S^y \in E'\} \\ & & \cup \{S^y | S^y \in E', \nexists x \in \xi, S^x \in E\} \end{array}$$

Pour les environnements, l'opération $\sqcup$ est aussi appelée "unification". Nous introduisons une relation d'ordre ($\preceq$) pour les environnements comme suit :

$$E \preceq E' \text{ iff } \forall S^x \in E, \exists S^y \in E' | S^x \leq_K S^y$$

Ainsi $E \preceq E'$ signifie que chaque élément de E est inférieur à un élément de E' selon l'ordre de connaissance du treillis ξ. En conséquence, la relation $\preceq$ est un ordre partiel sur $\mathcal{E}(p)$ et les opérations $\sqcup$ et $\sqcap$ sont monotones selon $\preceq$. De plus, ($\mathcal{E}(P)$, $\preceq$) est un treillis complet, son plus grand élément est : $\{S^\top \mid S \in \mathcal{S}(P)\}$ et son plus petit élément est $\{S^\bot \mid S \in \mathcal{S}(P)\}$.

Pour exprimer la sémantique opérationnelle, nous avons besoin d'une opération nous permettant d'utiliser à l'instant futur une valeur mémorisée dans un environnement au cours d'un instant courant. Nous appelons cette opération Pre, et voici sa définition :

$$Pre(E) = \{S^{\perp} \mid S^x \in E\} \cup \{S^x_{pre} \mid S^x \in E\}$$

Cette opération crée une nouvelle occurrence d'un événement S dans l'environnement courant et mémorise la valeur de S de l'instant précédent. Ensuite, la valeur courante de S est mise à $\perp$, afin d'évoluer vers un statut plus grand selon l'ordre de connaissance, en fonction des règles de la sémantique qui seront appliquées, dans l'instant.

4.4.2 Description de la sémantique opérationnelle

La sémantique opérationnelle nous permet de faire une compilation incrémentale des programmes ADeL, en traduisant chaque programme en un système d'équations ξ. Un système d'équations est défini comme étant le quintuplet $< V, R^{init}, R, R^+, D >$ où V contient des variables représentant le statut des événements d'entrée, des événements de sortie et des événements locaux. R^{init}, R et R^+ sont les registres, c'est-à-dire les variables spécifiques agissant comme des mémoires pour enregistrer des valeurs utiles pour calculer l'instant suivant. R^{init}, R et R^+ ont la même cardinalité. Les variables de R sont les valeurs des registres, celles de R^+ portent les valeurs des registres dans l'instant suivant et celles de R^{init} représentent les valeurs initiales des registres. Ainsi, si nous considérons un registre $reg \in R$, il y aura un élément $reg^+ \in R^+$ pour stocker la valeur future de la mémoire reg et un élément $reg^{init} \in R^{init}$ donnant sa valeur initiale. Enfin, D est le système d'équations pour calculer le statut de chaque événement.

Comme pour la sémantique comportementale, nous calculons l'environnement de sortie d'un programme en appliquant les règles de sémantique à son instruction racine. Ainsi, nous définissons cette sémantique d'abord pour les opérateurs d'ADeL et ensuite nous étendons ces définitions aux programmes. La sémantique opérationnelle est une fonction $\mathcal{S}_o$ qui calcule un environnement de sortie à partir d'un environnement d'entrée. Un environnement d'entrée contient les événements de la sorte du programme où les événements présents ont 1 comme statut tandis que les événements de sortie ont $\perp$. De plus, il contient également les registres introduits pour certains opérateurs et nécessaires à l'exécution du programme ainsi que des événements locaux introduits pour chaque opérateur pour représenter la propagation de l'information. Ces derniers sont décrits ci-dessous. Soit p une instruction d'ADeL et E un environnement d'entrée. Nous notons $\mathcal{D}_p$, son système d'équations et $\langle p \rangle_E$ l'environnement de sortie résultant calculé par $\mathcal{S}_o$ Il est défini comme suit :

$$\mathcal{S}_o(p, E) = \langle p \rangle_E \quad ssi \quad (E \vdash \mathcal{D}_p) \hookrightarrow \langle p \rangle_E$$

$\hookrightarrow$ signifie qu'à partir des statuts des évènements de E, le système d'équations $\mathcal{D}_p$ calcule des statuts pour les évènements de sortie et les locaux de E, ainsi nous obtenons l'environnement résultant $\langle p \rangle_E$. Pour calculer les statuts des événements de sortie et les valeurs futures des registres, nous nous appuyons sur les lois de l'algèbre ξ détaillées dans le tableau 4.2.

Soit P un programme ADeL et E un environnement d'entrée (c'est-à-dire un environnement où les variables des sorties et des événements locaux ont $\perp$ comme statut et où tout registre reg a soit pour valeur reg^{init}, si l'on est dans la première réaction ou bien reg^+ calculé à l'instant précédent), la sémantique opérationnelle formalise une réaction de P par rapport à E et calcule un environnement E' si et

seulement si $\mathcal{S}_o(\beta(P), E) = E'$; $\beta(P)$ étant l'instruction représentant le corps de P, i.e, l'instruction racine de l'arbre syntaxique de P. On déduit donc le système d'équations d'une instruction à partir des règles sémantiques définies pour chaque opérateur du langage.

Pour exprimer ces règles, nous ajoutons à chaque opérateur trois événements locaux spécifiques pour propager l'information : un événement d'entrée START qui permet de propager l'information de démarrage de l'instruction aux arguments de l'opérateur, un événement d'entrée KILL pour tuer les arguments de l'instruction (en cas de préemption par exemple) et un événement de sortie FINISH qui propage l'information de terminaison à l'instruction englobante.

Pour toute instruction i, ses évènements START et KILL sont initialisés par l'instruction englobante, si i est la racine du programme, START $= 1$ et KILL $= 0$. Au contraire, comme FINISH envoie ses informations de terminaison à l'instruction englobante, si i est la racine du programme, son FINISH est la terminaison du programme.

Les systèmes d'équations des opérateurs sont définis à l'aide des règles sémantiques de ces derniers. Ils calculent le statut de FINISH, des événements de sortie et des événements locaux, en fonction du statut de START, KILL, ainsi que des événements d'entrées et des événements locaux. Dans cette sémantique, les statuts des événements dans l'environnement sont des ξ équations.

Nous décrivons la sémantique opérationnelle des opérateurs ADeL et nous présentons leurs règles ci-après.

Opérateur nothing

Cet opérateur termine instantanément en ne modifiant pas l'environnement courant. Sa terminaison est simultanée avec son démarrage :

$$\mathcal{D}_{nothing} = \left[\begin{array}{ccc} \text{FINISH} & = & \text{START} \end{array} \right]$$

Son environnement de sortie est calculé comme suit :

$$E \vdash \mathcal{D}_{nothing} \hookrightarrow \langle \textbf{nothing} \rangle_E$$

Opérateur wait

wait S est un opérateur temporel qui prend au moins un instant. C'est à dire que la présence de l'événement attendu n'est pas testée au premier instant. Dans son système d'équation, nous introduisons un registre (REG) pour mémoriser à l'instant suivant que l'opérateur a bien démarré. Ce registre est maintenu tant que l'on est au moins au deuxième instant et que l'événement attendu n'est pas là. La terminaison dépend de la présence de l'événement attendu, mais pas au premier instant.

$$\mathcal{D}_{wait} = \left[\begin{array}{ccl} \text{REG}^+ & = & \text{START} \boxdot \neg \text{KILL} \boxplus \text{REG} \boxdot \neg S \boxdot \neg \text{KILL} \\ \text{FINISH} & = & S \boxdot \text{REG} \end{array} \right]$$

Pour calculer son environnement de sortie $\langle wait\ S \rangle_E$, nous appliquons l'opération spécifique de "translation temporelle" *Pre* (défini section 4.4.1) à l'environnement de départ, afin de mémoriser les valeurs des statuts courants des événements locaux et de sortie.

L'environnement de sortie de l'opérateur **wait** est calculé comme suit :

$$Pre(E) \vdash \mathcal{D}_{wait\ S} \hookrightarrow \langle \textbf{wait}\ S \rangle_E$$

Opérateur seq

Dans le système d'équations de cet opérateur[7], le premier argument commence dès que l'opérateur commence tandis que le deuxième argument commence lorsque le premier argument se termine (équation 1). Par ailleurs dès que l'opérateur est tué, ses deux arguments le sont aussi. L'événement FINISH de l'opérateur est lié à la fin du second argument (équation 2).

$$\mathcal{D}_{seq} = \left[\begin{array}{lcll} \text{START}_{p_1} & = & \text{START} & \\ \text{KILL}_{p_1} & = & \text{KILL} & \\ \text{START}_{p_2} & = & \text{FINISH}_{p_1} & (1) \\ \text{KILL}_{p_2} & = & \text{KILL} & \\ \text{FINISH} & = & \text{FINISH}_{p_2} & (2) \end{array} \right]$$

L'environnement de sortie de cet opérateur est calculé comme suit :

$$\langle p_2 \rangle_{\langle p_1 \rangle_E} \vdash \mathcal{D}_{seq} \hookrightarrow \langle p_1 \textbf{seq}\ p_2 \rangle_E$$

Pour calculer cet environnement de sortie, on calcule l'environnement de sortie de p_1 à partir duquel on calcule l'environnement de sortie de p_2 et cet environnent résultant constitue l'environnement d'entrée pour évaluer les équations de l'opérateur. En effet, dans l'opérateur **seq**, on exécute p_1 et à partir des informations sur les statuts des événements de l'environnement résultant de cette exécution, on évalue p_2.

Opérateur parallel

Dans $\mathcal{D}_{p_1||p_2}$ deux registres REG_1 et REG_2 sont introduits pour enregistrer le statut des évènements FINISH des deux arguments du parallèle, puisque cet opérateur se termine lorsque ses deux opérandes sont terminés. Il faut tenir compte du fait que les opérandes peuvent terminer à des instants différents. REG_1 (resp. REG_2) mémorise la terminaison de premier (resp. second) argument. Par ailleurs, les deux arguments du parallèle commencent en même temps que l'opérateur et dès qu'il est tué ses arguments le sont également.

Ce système d'équations peut paraitre complexe, mais les différentes valeurs des deux registres permettent de représenter 3 états : un état initial qui est final si les deux arguments terminent instantanément, un état où le premier argument termine et où l'on attend la terminaison du second, et l'état dual où le second argument termine et où l'on attend la terminaison du premier pour transiter dans l'état initial. Il est clair que la valeur de FINISH est fonction de ces 3 états.

7. Dans la suite, pour exprimer les systèmes d'équations d'un opérateur, ses événements spécifiques seront notés START, KILL et FINISH alors que les événements spécifiques de ses arguments potentiels seront indexés avec les noms respectifs des arguments.

$$\mathcal{D}_{\|} = \left[\begin{array}{lcl}
\text{REG}_1^+ & = & \text{REG}_1 \boxdot \neg\text{FINISH}_{p_2} \boxdot \neg\text{KILL} \boxplus \neg\text{REG}_2 \boxdot \\
 & & \text{FINISH}_{p_1} \boxdot \neg\text{FINISH}_{p_2} \boxdot \neg\text{KILL} \\
\text{REG}_2^+ & = & \text{REG}_2 \boxdot \neg\text{FINISH}_{p_1} \boxdot \neg\text{KILL} \boxplus \neg\text{REG}_1 \boxdot \\
 & & \neg\text{FINISH}_{p_1} \boxdot \text{FINISH}_{p_2} \boxdot \neg\text{KILL} \\
\text{START}_{p_1} & = & \text{START} \boxdot \neg\text{KILL} \\
\text{START}_{p_2} & = & \text{START} \boxdot \neg\text{KILL} \\
\text{KILL}_{p_1} & = & \text{KILL} \\
\text{KILL}_{p_2} & = & \text{KILL} \\
\text{FINISH} & = & \text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \text{FINISH}_{p_2} \boxplus \text{REG}_2 \boxdot \neg\ \text{REG}_1 \boxdot \text{FINISH}_{p_1} \boxplus \\
 & & \neg\text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \text{FINISH}_{p_1} \boxdot \text{FINISH}_{p_2}
\end{array}\right]$$

On applique ce système d'équations sur l'unification (opération $\sqcup$) des environnements respectifs des deux opérandes de l'opérateur. En effet, l'opération $\sqcup$ permet de prendre la borne supérieure (selon l'ordre de connaissance) des statuts des événements dans les environnements de sortie respectifs des arguments. Ainsi, si un événement est émis dans un argument son statut sera 1 (ou $\top$ s'il est à 0 dans l'environnement de l'autre argument) et on tiendra compte de cette information pour calculer son nouveau statut en appliquant le système d'équations $\mathcal{D}_{\|}$. L'environnement de sortie est calculé à l'aide de la règle suivante.

$$\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_E \vdash \mathcal{D}_{\|} \hookrightarrow \langle p_1 \| p_2 \rangle_E$$

Opérateur while

Le système d'équations de cet opérateur est très simple. Son corps démarre quand l'opérateur démarre et que la condition est vraie et il redémarre quand le corps se termine et la condition est évaluée à vrai. Le système propage l'information KILL à son corps et met à vrai le KILL de ce dernier quand la condition n'est plus vraie. Le registre sert uniquement à ne pas tuer le corps si la condition n'est pas vraie dans le premier instant. En effet, dans ce cas on n'entre pas dans la boucle. L'instruction **while** se termine uniquement lorsque l'événement de condition n'est plus présent. Nous rappelons que le corps d'un **while** ne peut pas être instantané. Voici son système d'équations :

$$\mathcal{D}_{while} = \left[\begin{array}{lcl}
\text{REG}^+ & = & \text{REG} \boxplus \text{START} \\
\text{START}_p & = & \text{START} \boxplus \text{FINISH}_p \\
\text{KILL}_p & = & \text{KILL} \boxplus \neg cond \boxdot \text{REG} \\
\text{FINISH} & = & \neg cond
\end{array}\right]$$

L'environnement de sortie de cet opérateur est calculé comme suit :

$$\langle p \rangle_E \vdash \mathcal{D}_{while} \hookrightarrow \langle \textbf{while}(p, cond) \rangle_E.$$

Opérateur emit

L'opérateur **emit** est le seul opérateur qui peut mettre un événement à 1 dans l'environnement et faire ainsi grossir l'information de $\bot$ à 1. Son système d'équations est simple, il termine au même instant où il démarre et il met son événement à 1.

$$\mathcal{D}_{emit} = \left[\begin{array}{lcl}
\text{FINISH} & = & \text{START} \\
S & = & \text{FINISH}
\end{array}\right]$$

L'environnement de sortie est calculé à l'aide de la règle suivante :

$$E \vdash \mathcal{D}_{emit} \hookrightarrow \langle \textbf{alert S} \rangle_E$$

Opérateur **if** *cond* **then** p_1 **else** p_2

Dans le système d'équations de cet opérateur, l'événement START est lié à l'instruction **then** (instruction p_1) lorsque l'événement test est présent et à l'instruction **else** (instruction p_2) lorsqu'il ne l'est pas. L'événement FINISH de cet opérateur est présent lorsque l'événement FINISH de son argument sélectionné est présent.

$$\mathcal{D}_{if} = \left[\begin{array}{lcl} \text{START}_{p_1} & = & \text{START} \boxdot \mathit{cond} \\ \text{START}_{p_2} & = & \text{START} \boxdot \neg \mathit{cond} \\ \text{KILL}_{p_1} & = & \text{KILL} \\ \text{KILL}_{p_2} & = & \text{KILL} \\ \text{FINISH} & = & \text{FINISH}_{p_1} \boxplus \text{FINISH}_{p_2} \end{array} \right]$$

L'environnement de sortie de cet opérateur est calculé comme suit :

$$\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_E \vdash \mathcal{D}_{if} \hookrightarrow \langle \textbf{if}(\mathit{cond}, p_1, p_2) \rangle_E$$

L'environnement d'entrée pour appliquer les équations de l'opérateur est $\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_E$. En effet, dans les systèmes d' équations des opérateurs, nous pouvons constater que les événements de sortie ou locaux sont directement ou indirectement pilotés par le START de l'opérateur. Si nous considérons un événement S ayant $x_1 \sqcup x_2$ pour statut, x_1 est fonction directement ou indirectement de START_{p_1} et x_2 de START_{p_2}. Lors de l'application de $\mathcal{D}_{if}$, nous avons x_1 fonction de START$\boxdot cond$ et x_2 fonction de START$\boxdot \neg cond$. Dans le premier instant START vaut 1. Lorsque l'on applique $\mathcal{D}_{if}$, le statut de S sera fonction de *cond*, il sera de la forme $f(cond) \sqcup f(\neg cond)$, f étant une combinaison d'opérateurs de ξ. Ainsi, si *cond* vaut 0 ou 1, en vertu des définitions des opérateurs de ξ utilisés pour exprimer les équations, le statut de S sera fonction uniquement du calcul de son statut dans la branche qui est prise. Si $cond = \bot$, le statut de S sera 0 ou $\bot$ suivant les opérations faites dans f.

Opérateur **stop** p **when** S **alert** S_1

L'opérateur **stop** ne teste son événement d'abandon qu'à partir du second instant. Nous introduisons un registre pour caractériser ses deux états : un premier état où la transition est faite si l'opérateur démarre et si son argument ne termine pas au premier instant ; toutefois si l'argument est instantané on reste dans ce premier état et FINISH est vrai. Dans un second état on teste l'événement de préemption. Tant que ce dernier n'est pas vrai et que l'argument ne termine pas, on reste dans cet état, sinon on transite vers le premier état et FINISH est vrai.

$$\mathcal{D}_{stop} = \left[\begin{array}{lcl} \text{REG}^+ & = & \text{REG} \boxdot \neg \text{FINISH}_p \boxdot \neg S \boxdot \neg \text{KILL} \boxplus \\ & & \neg \text{REG} \boxdot \text{START} \boxdot \neg \text{FINISH}_p \boxdot \neg \text{KILL} \\ \text{START}_p & = & \text{START} \boxdot \neg \text{KILL} \\ \text{KILL}_p & = & = S \boxdot \neg \text{KILL} \boxdot \text{REG} \boxplus \text{KILL} \\ \text{FINISH} & = & \neg S \boxdot \text{FINISH}_p \ \boxplus \text{FINISH}_p \\ S_1 & = & \text{REG} \boxdot S \end{array} \right]$$

Comme pour l'opérateur précédent, l'environnement de sortie de l'opérateur **stop..when** résulte de l'application de $\mathcal{D}_{stop}$ aux statuts de l'environnement d'entrée :

$$\langle p \rangle_E \vdash \mathcal{D}_{stop} \hookrightarrow \langle \mathbf{stop}(p, S, S_1) \rangle_E.$$

Dans le cas de l'opérateur **Pstop**, un évènement de "*failure*" est généré en plus, son équation est comme suit :

$$\left[\; failure \;=\; \text{REG} \boxdot S \; \right]$$

Cet événement est certes redondant avec l'émission de l'événement d'alerte, mais il ne complexifie pas le système d'équations. Seul le nombre de registres est important pour la complexité. Il n'influe pas sur la terminaison de l'opérateur et a l'avantage d'être exploitable au runtime. De plus l'événement d'alerte S_1 est facultatif.

Opérateur p **timeout** S p_1 **alert** S_1

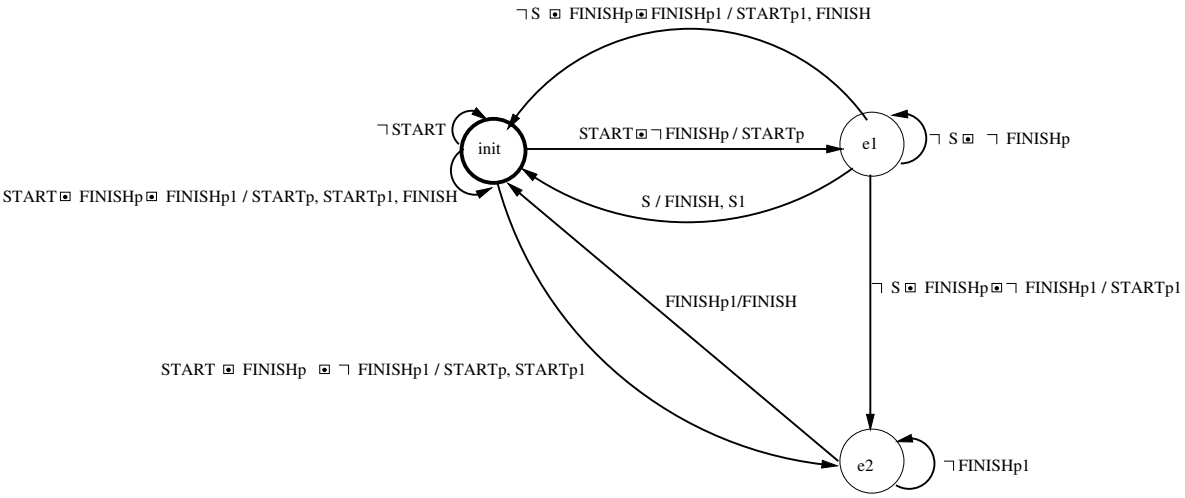


FIGURE 4.2 – L'automate explicite du comportement de l'opérateur **timeout** : les étiquettes sur les transitions sont de la forme a/b où a est le déclencheur de la transition et b l'ensemble des événements émis, c'est à dire avec un statut à 1. Pour alléger les expressions des déclencheurs, nous avons fait une simplification des notations, S signifie que le statut de S est à 1 et la transition est déclenchée si l'expression du déclencheur est 1.

Le comportement de l'opérateur **timeout** dépend de la présence de l'événement de (S) et des terminaisons respectives des deux arguments. Cet opérateur est à la fois une préemption si l'événement de timeout arrive avant la terminaison de p et une séquence car si cet événement de timeout n'arrive pas avant la terminaison de p, p_1 s'exécute en séquence. Par ailleurs, l'événement de timeout n'est pas testé au premier instant. Le système d'équations $\mathcal{D}_{timeout(p,S,p_1)}$ représente de façon implicite l'automate de comportement de l'opérateur. Comme ce dernier est complexe, nous avons détaillé dans la figure 4.2 une représentation explicite de cet automate. Ce dernier a trois états, ce qui justifie les deux registres du système d'équations dont les valeurs encodent ces trois états. Dans l'état initial (appelé init dans la figure) trois situations peuvent arriver :

1. on reste dans cet état init si START n'est pas à 1 ou bien si p et p_1 sont instantanés et terminent tous les deux dans le premier instant. Si p termine instantanément, cela signifie que dans une même transition START_p est émis et FINISH_p est reçu dans la partie déclenchement de la transition.

2. p n'est pas instantané, FINISH_p n'est pas à 1, on change d'état et on transite dans l'état e_1 en mettant START_p à 1.
3. p est instantané mais pas p_1, on va dans l'état où p est terminé et où l'on a plus besoin de tester S. On transite dans l'état e_2, en mettant START_p et START_{p_1} à 1.

Dans l'état e_1 on a démarré et ni p ni p_1 n'étaient instantanés, on teste alors le statut de S. Tant que S et FINISH_p ne sont pas à 1, on reste dans l'état ; p s'exécute et l'événement de timeout n'est pas présent. Si S arrive avant la fin de p, on retourne dans l'état init et on émet FINISH et S_1, car il y a préemption. Si S n'est pas présent et p termine, alors tout dépend de p_1. Si ce dernier termine instantanément (FINISH_{p_1} est présent, l'évaluation est terminée, on retourne dans l'état init en émettant START_{p_1} et FINISH. En revanche, si p_1 n'est pas instantané, on transite dans l'état e_2 en émettant START_{p_1}. Dans l'état e_2, on attend la terminaison de p_1 pour terminer en émettant FINISH. Pour plus de clarté, nous n'avons pas introduit les événements KILL dans la figure 4.2. Si cet événement est présent, on passe l'information à p et à p_1. De plus, lorsque l'on est dans l'état E_1, en présence de S, on tue les deux arguments.

Notons que dans cette première version du langage, timeout fait partie du noyau d'opérateurs bien qu'il puisse être considéré comme non primitif, nous envisageons de l'exprimer autrement dans le futur.

Voici le système d'équations qui correspond à ce comportement :

$$\mathcal{D}_{timeout} = \left[\begin{array}{lcl}
\text{REG}_1^+ & = & \text{REG}_1 \boxdot \neg S \boxdot \neg\text{FINISH}_p \boxdot \neg\text{KILL} \boxplus \\
 & & \neg\text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \text{START} \boxdot \neg\text{FINISH}_p \boxdot \neg\text{KILL} \\
\text{REG}_2^+ & = & \text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \neg S \boxdot \text{FINISH}_p \boxdot \neg\text{FINISH}_{p_1} \boxplus \\
 & & \neg\text{REG}_1 \boxdot \text{REG}_2 \boxdot \neg\text{FINISH}_{p_1} \boxplus \\
 & & \neg\text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \text{START} \boxdot \text{FINISH}_p \boxdot \neg\text{FINISH}_{p_1} \\
\text{START}_p & = & \neg\text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \text{START} \boxdot \text{FINISH}_p \boxdot \text{FINISH}_{p_1} \boxplus \\
 & & \neg\text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \text{START} \boxdot \neg\text{FINISH}_p \\
\text{START}_{p_1} & = & \text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \neg S \boxdot \text{FINISH}_p \boxdot \neg\text{FINISH}_{p_1} \boxplus \\
 & & \neg\text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \text{START} \boxdot \text{FINISH}_p \\
\text{KILL}_p & = & S \boxdot \neg\text{KILL} \boxdot \text{REG}_1 \boxdot \neg\text{REG}_2 \boxplus \text{KILL} \\
\text{KILL}_{p_1} & = & S \boxdot \neg\text{KILL} \boxdot \text{REG}_1 \boxdot \neg\text{REG}_2 \boxplus \text{KILL} \\
\text{FINISH} & = & \text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot S \boxplus \\
 & & \neg\text{REG}_1 \boxdot \text{REG}_2 \boxdot \text{FINISH}_{p_1} \boxplus \\
 & & \text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \text{FINISH}_p \boxdot \text{FINISH}_{p_1} \boxplus \\
 & & \neg\text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot \text{START} \boxdot \text{FINISH}_p \boxdot \text{FINISH}_{p_1} \\
S_1 & = & \text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot S
\end{array}\right]$$

L'environnement de sortie de l'instruction p **timeout** S $\{p_1\}$ **alert** S_1 est calculé comme suit :

$$\langle p_1 \rangle_{\langle p \rangle_E} \vdash \mathcal{D}_{timeout} \hookrightarrow \langle \textbf{timeout}(p, S, p_1, S_1) \rangle_E.$$

Comme l'opérateur **timeout** se comporte comme une séquence, l'environnement d'entrée à partir duquel on calcule celui de sortie est similaire à celui de l'opérateur **seq**. Ce sont les équations de l'opérateur qui propagent la préemption.

Dans le cas de l'opérateur **Ptimeout**, un évènement de "*failure*" est généré en plus, son équation est comme suit :

$$\left[\; \textit{failure} \;=\; \text{REG}_1 \boxdot \neg\text{REG}_2 \boxdot S \;\right]$$

Opérateur local

Les équations de cet opérateur sont simples. Le corps p démarre lorsque l'opérateur **local** commence son exécution, et la fin de p provoque la fin de l'opérateur **local**. Le système d'équations est le suivant :

$$\mathcal{D}_{local} = \left[\begin{array}{lcl} \text{START}_p & = & \text{START} \\ \text{KILL}_p & = & \text{KILL} \\ \text{FINISH} & = & \text{FINISH}_p \end{array} \right]$$

Nous considérons que les événements locaux de l'opérateur sont nouveaux et n'ont pas été utilisés dans une instruction englobante. C'est toujours possible à un renommage près. Dans l'environnement d'entrée $\langle p \rangle_E$, leur statut est $\bot$.

L'environnement de sortie de cet opérateur est calculé comme suit :

$$\langle p \rangle_E \vdash \mathcal{D}_{local} \hookrightarrow \langle \mathbf{local}(p, S) \rangle_E$$

4.5 Relation entre la sémantique comportementale et la sémantique opérationnelle

La sémantique comportementale est une “macro” étape de sémantique qui donne la signification d'une réaction pour chaque programme ADeL. Une réaction est un point fixe d'une suite de ”micro” étapes qui calcule une suite d'environnements de sortie à partir d'environnements initiaux, jusqu'à atteindre une stabilisation. En revanche, la sémantique opérationnelle nous permet de compiler le langage en associant un système d'équations à chaque opérateur. Pour vérifier qu'elle est correcte, nous allons prouver qu'elle est conforme à la sémantique comportementale. En effet, pour un programme P, nous allons montrer que les deux sémantiques s'accordent sur l'ensemble des événements de sortie ainsi que sur la valeur de la terminaison.

Mais la sémantique comportementale s'appuie sur une représentation booléenne des statuts des événements tandis que la sémantique opérationnelle considère les statuts dans l'algèbre ξ. Notre but est de montrer que la sémantique opérationnelle calcule des environnements de sortie identiques à la sémantique comportementale. En fait, dans la sémantique opérationnelle, nous avons une ”sur information” sur les statuts des événements et pour comparer les deux sémantiques, nous allons introduire une opération que nous avons appelée ”finalisation” qui permet de transformer un système d'équations quadri-valué en un système d'équations booléennes.

En se basant sur l'isomorphisme entre ξ et $\mathbb{B} \odot \mathbb{B}$, on peut faire correspondre un système d'équations à valeurs dans ξ à un système d'équations booléennes. Toutefois, chaque variable de ξ est associée à une paire de booléens et chaque équation se traduit par une paire d'équations booléennes. L'opération de finalisation est différente de l'encodage, car elle projette réellement tout ξ système dans le monde booléen en perdant de l'information. Nous allons nous appuyer sur cette opération pour montrer que les deux sémantiques concordent.

4.5.1 La finalisation

La finalisation consiste à remplacer $\bot$ par 0 et $\top$ par 1 dans un système d'équation. Il y a donc perte d'information, mais elle est nécessaire pour être dans le même contexte booléen que la sémantique comportementale. Pour faire effectivement ce remplacement

nous nous plaçons dans $\mathbb{B} \bigodot \mathbb{B}$. Tout élément x de ξ correspond de façon bijective à une paire de booléens, notée (x_h, x_l) [8](voir section 4.3.2) et nous avons 1 qui correspond à (tt, ff) et 0 à (ff, tt). Donc, si nous posons $x_l = \overline{x_h}$ [9], dans $\mathbb{B} \bigodot \mathbb{B}$ nous aurons remplacé l'encodage de $\top$ par 1 et celui de $\bot$ par 0 :

x	$\mathcal{F}(x)$
$\bot$ (ff, ff)	0 (ff, tt)
0 (ff, tt)	0 (ff, tt)
1 (tt, ff)	1 $(tt, ff$
$\top$ (tt, tt)	1 (tt, ff)

La finalisation s'étend aux systèmes d'équations de $\mathbb{B} \bigodot \mathbb{B}$, en remplaçant les composantes x_l par $\overline{x_h}$ pour toutes les variables d'un système. Nous montrons que la finalisation est compositionnelle pour les opérateurs $\boxplus$, $\boxdot$ et $\neg$. Une équation $x = f(z_1, .., z_n)$ dans ξ peut elle aussi être encodée dans $\mathbb{B} \bigodot \mathbb{B}$ par : $(x_h, x_l) = ((f(z_1, ..., z_n)_h, (f(z_1, ..., z_n)_l)$ et f est une combinaison d'opérateurs $\boxplus$, $\boxdot$ et $\neg$. Dans la section 4.3.2, nous avons montré que grâce au théorème 1 l'encodage s'exprime comme suit :

$$\begin{aligned} x \boxplus y &= (x_h + y_h, x_l.y_l) \\ x \boxdot y &= (x_h.y_h, x_l + y_l) \end{aligned}$$

De plus, par définition de $\neg$, nous avons $\neg x = (x_l, x_h)$. Un point important est la compositionnalité de la finalisation :

$\mathcal{F}(x \boxplus y) = (x_h + y_h, \overline{x_h + y_h}) = (x_h + y_h, \overline{x_h}.\overline{y_h}) = \mathcal{F}(x) \boxplus \mathcal{F}(y)$.
$\mathcal{F}(x \boxdot y) = (x_h.y.h, \overline{x_h}.\overline{y_h}) = (x_h.y_h, \overline{x_h} + \overline{y_h}) = \mathcal{F}(x) \boxdot \mathcal{F}(y)$.
$\mathcal{F}(\neg x) = (\overline{x_h}, x_h) = \neg \mathcal{F}(x)$.

Ainsi, l'équation $(x_h, x_l) = ((f(z_1, ..., z_n)_h, (f(z_1, ..., z_n)_l)$ s'écrit
$(x_h, x_l) = (f((z_{1_h}, z_{1_l}), ..., (z_{n_h}, z_{n_l}))$.

Pour calculer l'expression de la finalisation d'une équation de ξ, nous regardons l'application aux trois opérateurs utilisés dans les équations : $x = y \boxplus z$ qui s'écrit $(x_h, x_l) = (y_h, y_l) \boxplus (z_h, z_l)$ grâce à la remarque ci dessus. Si l'on pose $x_l = \overline{x_h}$ nous obtenons : $(x_h, \overline{x_h}) = (y_h, \overline{y_h}) \boxplus (z_h, \overline{z_h})$ et en appliquant l'expression de l'encodage de $\boxplus$ rappelée ci dessus, nous obtenons : $(x_h, \overline{x_h}) = (y_h + z_h, \overline{y_h}.\overline{z_h}) = (y_h + z_h, \overline{y_h + z_h})$. D'une façon duale, si nous considérons l'équation : $x = y \boxdot z$; nous obtenons : $(x_h, \overline{x_h}) = (y_h.z_h, \overline{y_h.z_h})$. Finalement, l'équation $x = \neg y)$ est finalisée comme suit : $(x_h, \overline{x_h} = (\overline{x_h}, x_h)$. Grâce aux lois de distributivité et de Morgan vérifiées dans ξ (voir table 4.2) et par isomorphisme dans $\mathbb{B} \bigodot \mathbb{B}$, nous avons : $(x_h, \overline{x_h}) = (f_{\mathbb{B}}(z_{1_h}, ..., z_{n_h}), \overline{f_{\mathbb{B}}(z_{1_h}, ..., z_{n_h})})$ avec $f_{\mathbb{B}}$ qui est la projection booléenne de f, c'est à dire la fonction obtenue quand on remplace $\boxplus$ par $+$, $\boxdot$ par . et $\neg$ par la négation booléenne. Il est clair que l'information est redondante et que l'on peut rester dans le monde booléen en ne considérant que la première projection des paires d'équations. Par abus, nous dénoterons $\mathcal{F}(S)$ le système d'équations où les variables sont la première composante de l'encodage des variables de S avec $\boxplus$ remplacé par $+$; $\boxdot$ par . et $\neg$ par la négation booléenne. Voici un exemple de finalisation pour le système de l'opérateur **if..then..else** :

8. Pour $x \in \xi$, son encodage $e(x)$ s'exprime dans $\mathbb{B} \bigodot \mathbb{B}$ comme une paire de booléen. Nous considérons que x s'exprime comme (x_h, x_l) et nous omettons la fonction d'encodage e quand il n'y a pas d'ambiguïté, afin de simplifier les notations

9. $\overline{x_h}$ est la négation de x_h dans $\mathbb{B}$

$$\mathcal{D}_{if} = \begin{bmatrix} \text{START}_{p_1} & = & \text{START} \boxdot \mathit{cond} \\ \text{START}_{p_2} & = & \text{START} \boxdot \neg \mathit{cond} \\ \text{KILL}_{p_1} & = & \text{KILL} \\ \text{KILL}_{p_2} & = & \text{KILL} \\ \text{FINISH} & = & \text{FINISH}_{p_1} \boxplus \text{FINISH}_{p_2} \end{bmatrix}$$

$$\mathcal{F}(\mathcal{D}_{if}) = \begin{bmatrix} \text{START}_{p_{1_h}} & = & \text{START}.\mathit{cond} \\ \text{START}_{p_{2_h}} & = & \text{START}.\overline{\mathit{cond}} \\ \text{KILL}_{p_{1_h}} & = & \text{KILL}_h \\ \text{KILL}_{p_{2_h}} & = & \text{KILL}_h \\ \text{FINISH}_h & = & \text{FINISH}_{p_{1_h}} + \text{FINISH}_{p_{2_h}} \end{bmatrix}$$

Cette opération de finalisation s'applique aussi aux environnements de la sémantique opérationnelle. Les statuts des environnements étant les solutions de systèmes d'équations, la finalisation d'un environnement E est définie par :

$$\mathcal{F}(E) = \{S^{x_h} \mid S^x \in E\}$$

La sémantique opérationnelle permet de calculer un environnement de sortie booléen ; ainsi pour une instruction p, les systèmes d'équations finalisés permettent de calculer des environnements finalisés $\mathcal{F}(E) \vdash \mathcal{F}(\mathcal{D}_p) \hookrightarrow \langle p \rangle_{\mathcal{F}(E)}$. De plus nous avons la propriété suivante :

Propriété 1 $\forall S^b \in \langle p \rangle_{\mathcal{F}(E)}$ *alors* $S^b \in \mathcal{F}(\langle p \rangle_E)$

Considérons $S^b \in \langle p \rangle_{\mathcal{F}(E)}$, alors

— soit $S^b \in \mathcal{F}(E)$ et aucune équation de $\mathcal{F}(\mathcal{D}_p)$ n'a modifié b. Par définition de la finalisation, il existe $x \in \xi$, et $S^{\mathcal{F}(x)} \in \mathcal{F}(E)$. L'encodage de x est (x_h, x_l) et $x_h = b$, car un événement a un statut unique dans un environnement. Si aucune équation de $\mathcal{F}(\mathcal{D}_p)$ ne change le statut de S dans $\mathcal{F}(E)$, alors aucune équation de $\mathcal{D}_p$ ne change le statut de S dans E. Donc $S^x \in \langle p \rangle_E$ et $S^{\mathcal{F}(x)} \in \mathcal{F}(\langle p \rangle_E)$ et $\mathcal{F}(x) = b$.

— soit il existe une équation dans $\mathcal{F}(\mathcal{D}_p)$ et b résulte de l'application de cette équation. Cette dernière est de la forme $b = f_{\mathbb{B}}(\overrightarrow{b})$; $f_{\mathbb{B}}$ est une combinaison d'opérateurs booléens "+","." et de la négation booléenne ; $\overrightarrow{b}$ est l'ensemble des variables de $\mathcal{F}(\mathcal{D}_p)$ à partir desquelles on calcule b. Comme précédemment, si une telle équation existe dans $\mathcal{F}(\mathcal{D}_p)$, alors dans $\mathcal{D}_p$, il y a une équation $x = f(\overrightarrow{y})$ et x est le statut de S dans $\langle p \rangle_E$; de plus f est la fonction quadri valuée correspondant à $f_{\mathbb{B}}$, c'est à dire que $\boxdot$ correspond à "+", $\boxdot$ à "." et $\neg$ à la négation booléenne . Nous allons étudier les différentes formes possibles pour f :

1. $x = y \boxplus z$: on a donc (grâce à l' encodage)une paire d'équations $(x_h, x_l) = (y_h, y_l) \boxplus (z_h, z_l)$; donc $(x_h, x_l) = (y_h + z_h, y_l.y_l)$. Ainsi dans $\langle p \rangle_E$, le statut de S est $(y_h + z_h, y_l.y_l)$ et dans $\mathcal{F}(\langle p \rangle_E)$, le statut de S est $y_h + z_h = x_h$. De plus nous savons aussi que par définition c'est l'équation $x_h = y_h + z_h$ (égale à $\mathcal{F}(x = y \boxplus z)$) qui permet de calculer le statut de S dans $\langle p \rangle_{\mathcal{F}(E)}$; ainsi nous avons $b = x_h$.
2. $x = y \boxdot z$: c'est le cas dual du premier item.

3. $x = \neg y$: on a donc $(x_h, x_l) = \neg(y_h, y_l) = (y_l, y_h)$ dans $\mathcal{D}_p$; ainsi dans $\langle p \rangle_E$ le statut de S est (y_l, y_h). Dans la finalisation, on pose $u_l = \overline{u_h}$, pour toute variable (u_h, u_l) de $\mathbb{B} \bigodot \mathbb{B}$ et on ne garde que la première composante de l'encodage. Ainsi dans $\mathcal{F}(\langle p \rangle_E)$, le statut de S sera $\overline{y_h} = x_h$. Dans $\mathcal{F}(\mathcal{D}_p)$ par construction nous aurons l'équation $x_h = \overline{y_h}$ pour calculer le statut de S ; ainsi le statut de S sera $x_h = \overline{y_h}$ dans $\langle p \rangle_{\mathcal{F}(E)}$.

Nous nous appuyons sur cette opération de finalisation pour montrer la concordance des deux sémantiques.

4.5.2 Concordance des deux sémantiques

La sémantique comportementale s'appuie sur une considération booléenne des statuts des événements. Afin d'être effective et de faire face au problème de causalité, la sémantique opérationnelle considère que leurs statuts sont dans l'algèbre quadri-valuée ξ. La finalisation va nous servir aussi à montrer que la sémantique opérationnelle, lorsqu'on la plonge dans le monde booléen, donne un résultat similaire à la sémantique comportementale, référence du langage.

Soit E un environnement d'entrée pour P et E' l'environnement de sortie calculé par la sémantique opérationnelle, nous voulons montrer qu'il existe une réécriture dans la sémantique comportementale $P \xrightarrow[I]{O} P'$ telle que I contient les événements d'entrée qui ont 1 pour statut dans E, et O les événements de sortie qui ont 1 comme statut dans E'.

Pour calculer l'environnement de sortie de la sémantique opérationnelle, on calcule $\mathcal{S}_o(\beta(P), E)$ [10] c'est à dire que l'on applique le système $\mathcal{D}_{\beta(P)}$ sur E et on obtient $\langle \beta(P) \rangle_E$. Pour projetter la sémantique opérationnelle dans le monde booléen, nous allons appliquer $\mathcal{F}(\mathcal{D}_{\beta(P)})$ sur $\mathcal{F}(E)$ et nous obtiendrons en environnement de sortie finalisé $\langle \beta(P) \rangle_{\mathcal{F}(E)}$. Dans la sémantique comportementale, on a une définition similaire : $P \xrightarrow[I]{O} P'$ ssi $\beta(P) \xrightarrow[I \cup O]{O, term_p} p'$, et p' est l'instruction dérivée de $\beta(P)$.

Nous allons montrer que les sémantiques des opérateurs d'ADeL coïncident :

Théorème 2 *Soit p une instruction ADeL et E un environnement d'entrée. Si $\langle p \rangle_E$ est l'environnement calculé par la sémantique opérationnelle tel que $\nexists S^\top \in \langle p \rangle_E$, alors la propriété suivante est vérifiée :*

$$\exists p' \text{ tel que } p \xrightarrow[E_\mathcal{C}]{E', \text{FINISH}_{p_h}} p' \text{ et } \forall o \in E',\ o^1 \in \langle p \rangle_E$$

On note $E_\mathcal{C}$ l'ensemble qui contient les événements d'entrée et de sortie de la sorte de p qui sont dans $\langle p \rangle_E$ avec un statut à 1. Les événements d'entrée de la sorte de p à 1 dans $\langle p \rangle_E$, sont ceux qui sont à 1 dans E car, par définition, les équations ne changent pas le statut des entrées. Le théorème signifie que lorsque la sémantique opérationnelle génère une solution, il existe également une solution comportementale avec les mêmes sorties. Ainsi, quand la sémantique opérationnelle calcule une valeur de terminaison (c'est-à-dire la valeur du statut de FINISH de l'instruction), la sémantique comportementale coïncide lorsque l'on projette les résultats de la sémantique opérationnelle dans le monde booléen. Ainsi nous ne considérerons plus que les premières composantes de l'encodage des éléments de ξ (la composante h).

10. nous rappelons que $\beta(P)$ est l'instruction racine de P.

La preuve de ce thèorème est une induction sur la taille des instructions, cette notion est définie en annexe dans la section B.2. La preuve est aussi détaillée dans cette section.

En corollaire, nous pouvons déduire que pour un programme ADeL P, les deux sémantiques coïncident quand au calcul des événements de sortie.

Corollaire 1 *Soit P un programme ADeL et E un environnement d'entrée pour la sémantique opérationnelle. Alors, dans la sémantique comportementale, $P \xrightarrow[I_\mathcal{C}]{O_\mathcal{C}} P'$ et $\forall o \in O, o^1 \in \langle P \rangle_E$. $I_\mathcal{C}$ contient les éléments d'entrée de E qui ont un statut à 1 et de même, $O_\mathcal{C}$ contient les éléments de sortie de E avec le même statut.*

Un environnement d'entrée E pour un programme P dans la sémantique opérationnelle est composé des événements de la sorte de P avec un statut déterminé dans ξ. Les événements d'entrée qui sont présents ont 1 pour statut et 0 sinon ; les autres événements de la sorte ont $\perp$ pour statut. De plus, E contient les événements locaux spécifiques (START, KILL, FINISH) avec pour statut $\perp$ sauf les événements $\text{START}_{\beta(P)}$ [11] qui est à 1 et $\text{KILL}_{\beta(P)}$ qui est à 0. E contient également les paires de variables qui représentent les registres (reg, $reg+$) introduits pour certains opérateurs, avec reg ayant pour statut soit le statut calculé pour reg^+ à l'instant précédent, soit le statut défini dans sa valeur initiale pour le premier instant.

Le théorème 2 nous permet d'affirmer qu'il existe une réécriture dans la sémantique comportementale telle que $\beta(P) \xrightarrow[E_\mathcal{C}]{E', term_{\beta(P)}} p'$ et $\forall o \in E', o^1 \in \langle \beta(P) \rangle_E$. Par définition, $\langle P \rangle_E = \langle \beta(P) \rangle_E$. E' est l'ensemble des sorties de P présentes dans la réaction de P à $E_\mathcal{C}$. Par ailleurs, nous savons que $E' \subseteq E_\mathcal{C}$. Posons $I_\mathcal{C} = \{i,$ les événements d'entrée $i \mid i^1 \in E\}$ et de façon duale $O_\mathcal{C} = \{o,$ les événements de sortie $o \mid o^1 \in \langle P \rangle_E\}$.

Par définition, $E' = O_\mathcal{C}$ et $E_\mathcal{C}$, $I_\mathcal{C} \cup O_\mathcal{C} = E_\mathcal{C}$. Comme, $P \xrightarrow[I_\mathcal{C}]{O_\mathcal{C}} P'$ ssi $\beta(P) \xrightarrow[I_\mathcal{C} \cup O_\mathcal{C}]{O_\mathcal{C}, term_{\beta(P)}} p'$ le corollaire est vérifié.

4.6 Compilation et validation

4.6.1 Compilation

Pour compiler un programme ADeL de façon efficace, nous le transformons d'abord en un système d'équations qui représente l'automate synchrone à l'aide de la sémantique opérationnelle, comme expliqué dans la section 4.4. L'isomorphisme et l'encodage défini dans la section 4.3.2 sont précieux d'un point de vue pratique car ils nous permettent de représenter un système d'équations quadri-valué et surtout de trier ce système afin de détecter les cycles de causalité. C'est la démarche que nous avons adoptée. Nous avons vu dans cette section que, dans la sémantique opérationnelle le test de la causalité des programmes se traduit par la recherche de cycles de dépendance dans les systèmes d'équations. Cela revient à trouver un ordre valide d'évaluation des équations qui respectent les dépendances des variables. Habituellement, dans l'approche synchrone, cet ordre est déterminé de manière statique. Mais choisir un ordre total à chaque instant oblige à ordonner des évènements qui a priori ne le sont pas. Considérons les trois scenarios suivants :

11. Nous rappelons que $\beta(P)$ est l'instruction racine de P.

```
scenario A:
while I
{
  if I1 then emit O1
   ||
  if I2 then emit O2
}
```

```
scenario B:
while I
{
  if I3 then emit O3
}
```

```
scenario FINAL:
local L1,L2
{
   call A [L2\I1, I\I2,
           O\O1, L1\O2]
  ||
   call B [L1\I3, L2\O3]
}
```

Dans cet exemple, dans les scenarios A et B O1, O2 et O3 sont indépendants et peuvent être évalués dans n'importe quel ordre. Le choix d'un ordre total pour les systèmes d'équations de A et de B, peut conduire à de faux cycles de causalité. Si l'on choisit l'ordre : {O2 = I2; O1 = I1; O3 = I3}, dans FINAL en tenant compte des renommages, nous obtenons : {L1 = I; O = L1; L2 = L1 } qui est correct. En revanche, si nous choisissons l'ordre : {O3 = I3; O2 = I2; O1 = I1 }, dans FINAL nous avons : {L2 = L1; O = L2; L1 = I } qui a un cycle. Ainsi, un faux cycle de causalité est apparu.

C'est pourquoi d'autres langages comme Light Esterel ont proposé une autre façon de trier les équations [RG11]. Les événements indépendants ne doivent pas être reliés par un ordre arbitraire. Seuls les ordres partiels déduit des dépendances des variables doivent être pris en compte quand on veut trier un système d'équations.Pour compiler un programme ADeL, nous construisons un ordre partiel incrémental pour son système d'équations. Pour cela, nous maintenons assez d'information sur la causalité des évènements grâce au statut de ξ attribué à chaque évènement. Nous allons introduire un algorithme de tri qui exploite le calcul des ordres partiels d'un système. Cet algorithme permet de trier séparément des sous systèmes d'équations et d'obtenir un système global trié sans avoir à recommencer le tri sur le système global. Ceci nous permet d'avoir une compilation incrémentale dans laquelle on peut calculer séparément le système d'équation d'un sous programme, le trier et le combiner au système trié du programme principal.

Pour construire ces ordres partiels, notre algorithme s'appuie sur des méthodes de tri bien connues dans la gestion de projets industriels pour affecter des dates à des activités. La technique la plus utilisée est celle du chemin critique (CPM : Critical Path Method) [12] [Fon61, AS80]. Cette méthode construit en premier un graphe acyclique dont les noeuds sont les activités. Les flèches entre les noeuds représentent la relation "doit être exécutée avant". De plus, des poids sont attachés aux nœuds pour tenir compte de la durée des activités. Les algorithmes de chemin critique calculent la date au plus tôt et la date au plus tard pour chaque activité.

Nous adoptons une méthode similaire pour calculer les dates au plus tôt et au plus tard auxquelles une variable peut et doit être évaluée. En pratique, à chaque variable du système d'équations nous associons deux variables entières (*CanDate*, *MustDate*) qui indiquent la date à laquelle la variable peut, respectivement doit, être évaluée. *MustDate* est la date finale à laquelle on doit prendre en compte une variable pour assurer que l'évaluation du système se déroule correctement. Cette notion de date est très proche de celle calculée par les algorithmes CPM. Toutefois, une date représente aussi un niveau d'évaluation. Les variables évaluées en premier ne dépendent d'aucune autre variable, elle sont caractérisées par la date 0. Les variables qui dépendent de variables de date n pour être évaluées sont représentée par la date $n+1$. Les variables de même date sont indépendantes et peuvent être évaluées dans n'importe quel ordre (entre elles).

12. http://pmbook.ce.cmu.edu/10_Fundamental_Scheduling_Procedures.html

Calcul des ordres partiels

E_1 :

$$\begin{aligned} a &= x \boxplus y \\ b &= x \boxplus \neg y \\ c &= a \boxplus t \\ d &= a \boxplus c \\ e &= a \boxdot t \end{aligned}$$

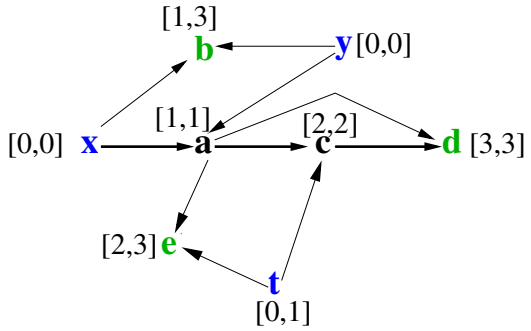


FIGURE 4.3 – Un système E_1 et $\Gamma(E_1)$, son graphe de dépendance. x, y et t sont les entrées de E_1 et b,d et e les sorties. Nous avons explicité sur le graphe les dates calculées pour les variables de E_1. t[0,1] signifie que la $CanDate$ de t est 0 et sa $MustDate$ est 1. Le chemin critique est en gras sur la figure.

Notre algorithme comprend deux phases. La première phase de l'algorithme va parcourir un système d'équations E_q et construire ensuite le graphe de dépendances $(\Gamma(E_q))$. $\Gamma(E_q) = (V_{E_q}, \rightarrow)$. V_{E_q} est l'ensemble des variables du système d'équations et représente les nœuds du graphe. $\rightarrow$ permet de construire les arcs du graphe : on aura $x \rightarrow x'$ s'il y a une équation $x' = f(x)$ dans E_q. La figure 4.3 montre un système d'équations dans ξ et son graphe de dépendance [13].

Nous ne donnons pas une définition de l'algorithme de tri en pseudo code, cette dernière est faite dans [RG11]. Nous décrivons juste son fonctionnement. Le but est d'associer à toutes les variables d'un système une $CanDate$ à partir de laquelle la variable peut être évaluée et une $MustDate$ à partir de laquelle la variable doit être évaluée pour ne pas bloquer le calcul des équations. Une date est un niveau de dépendance : les entrées du système ont pour $CanDate$ 0 et les dates des autres variables sont calculées en suivant les dépendances et en incrémentant de 1 à chaque dépendance ; les $MustDate$ se calculent à partir des sorties : on leur attribue N (la longueur du plus grand chemin dans le graphe) comme date et on décrémente de 1 pour obtenir la $MustDate$ des variables qui causent une sortie, toujours en respectant le graphe de dépendance. Pour cela, on se base sur le graphe de dépendance du système et on calcule les dates comme suit :

$$CanDate(x) = \begin{cases} 0 \text{ ssi x est une entrée} \\ max\{CanDate(y) + 1 \mid y \rightarrow x \in \Gamma(E_q)\} \text{ sinon} \end{cases}$$

$$MustDate(x) = \begin{cases} N \text{ ssi x est une sortie} \\ min\{MustDate(y) - 1 \mid x \rightarrow y \in \Gamma(E_q)\} \text{ sinon} \end{cases}$$

N est aussi la $CanDate$ la plus grande calculée. Cette remarque évite de calculer le plus long chemin du graphe.

Ainsi pour le tri de E_1 nous obtenons les dates décrites dans la figure 4.3. Pour calculer les $CanDate$, nous avons attribué 0 à x, y, t et ensuite nous avons suivi les dépendances de $\Gamma(E_1)$. Dans cet exemple, considérons la variable c, en partant de x sa $CanDate$ est 2 et en partant de t elle vaut 1, le résultat est 2. Comme la $CanDate$ maximum est 3, nous attribuons cette valeur à b, d, e pour calculer les $MustDate$ des

13. Nous n'avons pas considéré le système réel d'un programme ADeL, car son expression n'est pas simple et l'exemple E_1 suffit à illustrer le fonctionnement du tri dans la compilation d'ADeL.

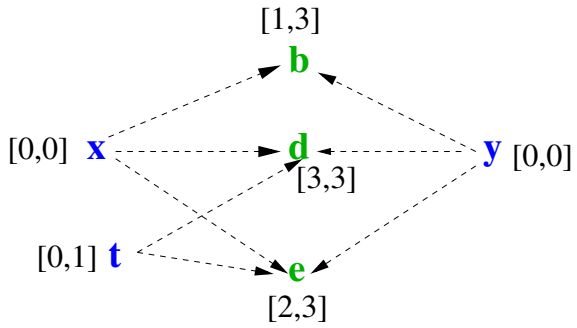


FIGURE 4.4 – Le graphe abstrait de E_1. Les chemins des entrées vers les sorties sont remplacés par de simples arcs.

variables, en suivant $\Gamma(E_1)$ en sens inverse des dépendances. Cet algorithme de tri nous permet de détecter les cycles, si une date est plus grande que le nombre de variables du système, il y a un cycle de causalité. Il nous permet aussi un tri "incrémental".

Dans un système d'équations E_q, nous pouvons isoler un sous système "clos" E_{q_1}, c'est à dire tel que toute variable de E_q qui n'appartient pas à E_{q_1} n'est reliée qu'à une entrée ou une sortie de E_{q_1}. Ceci est le cas dans ADeL, si l'on considère un sous programme d'un programme appelé avec une instruction **call**. Nous pouvons calculer les dates des variables de E_{q_1} et à partir de celles ci nous en déduisons les dates des variables de E_q. Pour cela nous calculons le graphe abstrait de E_{q_1}. Le graphe abstrait d'un système E_q est $\Gamma_a(E_q) = (V_{E_{q_a}}, \dashrightarrow, \rightarrow)$, $V_{E_{q_a}} \subseteq V_{E_q}$. $i \dashrightarrow o$ signifie qu'il existe un chemin $i \rightarrow x_1 \rightarrow x_2 .... \rightarrow o$ dans $\Gamma(E_q)$. Il existe un arc $x \rightarrow y$ si cet arc existe dans $\Gamma(E_q)$. On calcule, les dates des sommets de E en appliquant les règles suivantes :

$$CanDate(x) = \begin{cases} 0 \text{ ssi x est une entrée} \\ max\{CanDate(y) + [CanDate(y) - CanDate(x)] \mid y \dashrightarrow x \in \Gamma_a(E)\} \\ max\{CanDate(y) + 1 \mid y \rightarrow x \in \Gamma_a(E_q)\} \text{ sinon} \end{cases}$$

$$MustDate(x) = \begin{cases} N \text{ ssi x} \in \text{ est une sortie} \\ min\{MustDate(y) - [MustDate(y) - MustDate(x)] \mid x \rightarrow y \in \Gamma_a(\mathcal{E}_{\text{II}}) \\ min\{MustDate(y) - 1 \mid x \rightarrow y \in \Gamma_a(E_q)\} \text{ sinon} \end{cases}$$

N est toujours la $CanDate$ la plus grande calculée. En fait, on tient compte de la longueur du chemin calculée au préalable entre les entrées et les sorties du sous système.

Par exemple considérons le système E suivant :

$$E = \left\{ \begin{array}{l} x = i_1 \\ y = i_2 \\ o_1 = e \\ o_2 = b \\ o_3 = d \\ \left.\begin{array}{l} a = x + y \\ b = x + \neg y \\ c = a + t \\ d = a + c \\ e = a.t \end{array}\right\} E_1 \end{array} \right.$$

E contient le sous système E_1 que nous avons pris comme exemple et trié dans la figure 4.3. Ce sous système est clos. On peut donc dans un premier temps calculer son graphe abstrait $(\Gamma_a(E_1))$, visualisé dans la figure 4.4. Ensuite, on considère le graphe

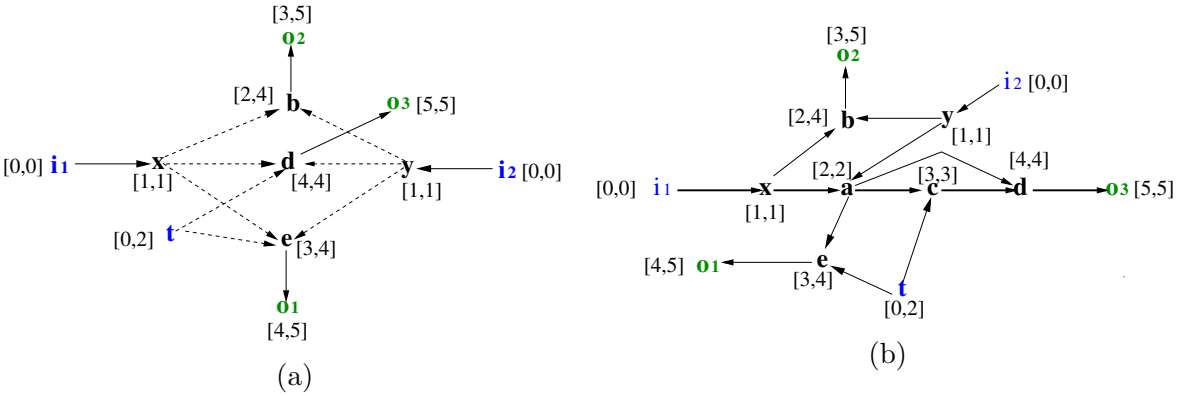


FIGURE 4.5 – (a) Le graphe abstrait $\Gamma_a(E)$ obtenu après abstraction de $\Gamma(E_1)$ et le calcul des dates à partir de celle de $\Gamma(E_1)$. (b) Le graphe de E ($\Gamma(E)$).

de E dans lequel le sous graphe correspondant à E_1 a été abstrait (voir figure 4.5(a)). Pour calculer les dates de E, nous partons des dates calculées pour E_1. Nous appliquons l'algorithme pour calculer les dates des nœuds de $\Gamma_a(E)$. Si l'on prend à titre d'exemple la variable d, ses dates après le calcul des dates de E_1 sont [3,3]. Nous mettons la $CanDate$ de i_2 à 0, la $CanDate$ de y devient 1 et la $CanDate$ de d devient 4 car $[CanDate(d) - CanDate(y) = 3]$ dans S_1. On itère le processus pour i_1, i_2, x, y, t, b, d, e, o_1, o_2 et o_3, en partant de $CanDate(i_1) = 0$ et $CanDate(i_2) = 0$. Pour les MustDate, on part des sorties o_1, o_2 et o_3 que l'on met à 5 (date maximale des $CanDate$ calculées et on décrémente en suivant les deux types d'arcs de $\Gamma(E)$. Ensuite, on doit à partir de ces calculs, faire une mise à jour des dates des variables internes de E_1. Si l'on calcule les dates de E, en considérant $\Gamma(E)$ (voir figure 4.5(b)), nous obtenons le même résultat.

Pratiquement, pour compiler un programme ADeL, nous construisons son système d'équations en appliquant les règles de la sémantique opérationnelle. De plus, l'encodage défini dans la section 4.3.2 nous permet de construire des systèmes quadri-valués encodés de façon isomorphe en paires d'équations booléennes (voir théorème 1). Ensuite nous trions ce système pour vérifier qu'il n'y a pas de cycle. L'avantage d'utiliser cet algorithme de tri est de pouvoir faire une compilation incrémentale. En effet, nous pouvons compiler les sous-programmes d'un programme et inclure leurs systèmes d'équations dans le système d'équation du programme principal sans recalculer l'ordre. Nous appliquons cette technique pour les sous-activités appelées à travers une instruction **call**. Nous compilons séparément un sous programme et nous incluons son système d'équations dans le système d'équations trié du programme appelant. Dans [RG11], un format a été défini afin de constituer des bibliothèques de programmes compilés.

Le but de la compilation d'ADeL est de permettre la génération du code C++ qui représente l'automate de reconnaissance et aussi celle des codes d'entrée pour la simulation et la validation formelle. Pour cela, nous devons projeter les équations quadri-valuées dans des systèmes booléens. Toutefois, avant de générer ce code, nous devons vérifier qu'aucun évènement n'a atteint le statut $\top$. Cette vérification est faite en cours d'exécution en ajoutant une équation par évènement qui teste leur projection en paire de booléens et s'assure que nous n'avons jamais le et logique de la paire égal à vrai. Nous finalisons (voir section 4.5.1) les systèmes avant de générer les différents codes de sortie. La finalisation reste cohérente vis à vis des statuts calculés des événements. Toutefois, une fois la finalisation faite, nous ne pouvons plus inclure les équations d'un sous programme dans un programme global. En effet, l'inclusion d'un système

d'équations trié dans un système global n'est possible que dans le monde quadri-valué. Une fois finalisé, un système n'est plus constructif, c'est pourquoi on n'applique cette opération sur le système d'un programme qu'à la fin de la compilation, au moment de générer du code par exemple.

4.6.2 Validation

Validation exhaustive

La représentation interne en tant que système d'équations booléennes rend également possible la validation formelle des programmes ADeL, en générant le format d'entrée du model-checker NuSMV [14][CCG+02].

NuSMV est l'un des premiers outils de vérification basé sur les diagrammes de décision binaire (ou BDD : Binary Decision Diagrams). Il définit des techniques de "model-checking borné" qui utilisent des SAT-solvers. NuSMV a été conçu en tant qu'architecture ouverte pour le model checking. Il offre une vérification fiable pour les modèles de taille industrielle. Il peut être utilisé comme arrière-plan d'autres outils de vérification et comme un outil de recherche pour les techniques de vérification formelle. NuSMV supporte l'analyse des spécifications exprimées en CTL et LTL [JGP00]. L'interaction avec l'utilisateur se fait à travers une interface textuelle.

Pour illustrer un exemple de preuves avec NuSMV à partir d'une description ADeL, nous allons utiliser le même exemple de cas d'utilisation que dans le chapitre 3. Dans cet exemple nous souhaitons montrer que recevoir l'évènement *displays_happy_smiley(touchPad)* et l'alerte *wrong_picture_chosen* en même temps génère une erreur (*Error*) parce que dans ce cas le comportement décrit est erroné. Pour faire cette preuve, nous souhaitons tout d'abord nous appuyer sur un observateur [HLR93, Rus12]. Un observateur représente un complément du modèle du système (programme ADeL) conçu pour vérifier la satisfaction de certaines propriétés. Il écoute les entrées et les sorties du programme à vérifier et lève un indicateur lorsqu'une condition est remplie. Un observateur permet de restreindre et de filtrer les comportements d'un programme. On peut aussi l'intégrer dans le model-checker si on souhaite vérifier des propriétés complexes mais d'une manière plus simple que par l'utilisation directe des logiques temporelles, puisqu'on peut le décrire à l'aide de notre langage.

Dans notre cas l'observateur est un programme ADeL qui sera exécuté en parallèle avec le programme initial (seriousGame) dans un programme ADeL global. Le programme de l'observateur est déclaré comme suit :

```
Activity observer (patient:Person,touchPad:Equipment,game_zone:Zone)

Events
inside_zone(Person, Equipment, Zone);
displays_happy_smiley(Equipment);
wrong_picture_chosen;
serious_game_over;

InitialState : inside_zone(patient, touchPad, game_zone);
Start
 stop when serious_game_over
 {
  wait displays_happy_smiley(touchPad) and wrong_picture_chosen
       seq
```

14. http://nusmv.fbk.eu/

```
  emit Error
 }
  alert test_ok
End
```

Dans ce code, nous remarquons que l'observateur a comme entrées un évènement d'entrée du programme initial *displays_happy_smiley(Equipment)* et aussi deux sorties du programme initial *wrong_picture_chosen* et *serious_game_over* qui sont générées sous la forme de deux alertes). Le programme de l'observateur indique qu'il émet *Error* si on reçoit les deux entrées *displays_happy_smiley(touchPad)* et *wrong_picture_chosen* avant la fin du programme initial (indiquée par la reception de l'évènement *serious_game_over*), sinon il envoie une alerte de succès *test_ok* qui indique que le programme a été vérifié et que cette propriété erronée n'existe pas. Les deux programmes ADeL (SeriousGame et l'observateur) sont appelés à l'aide de l'opérateur **call**. Le programme global est le suivant :

```
Activity seriousGameVerif(patient:Person,touchPad:Equipment,
                                    game_zone:Zone)
Events
inside_zone(Person, Equipment,Zone);
SubActivities
seriousGame(patient,touchPad,game_zone);
observer(patient,touchPad,game_zone);

InitialState : inside_zone(patient, touchPad, game_zone);
Start
 local wrong_picture_chosen, serious_game_over
 {
  call seriousGame parallel call observer
 }
End
```

A partir de ces programmes, nous avons généré un format compatible pour le model-checker NuSMV pour l'intégrer dans ce dernier. Dans notre cas, le modèle NuSMV est généré à partir du code global puisqu'il appelle l'observateur et le programme principal. On peut utiliser le model-checker NuSMV de deux façons : interactivement en introduisant des propriétés de logique temporelle dans un terminal, ou en *batch* en écrivant les propriétés de logique temporelle dans le module NuSMV généré à partir de notre programme et en le vérifiant automatiquement dès qu'on l'introduit dans le model-checker. Les propriétés que nous avons choisi d'introduire sont :

```
CTLSPEC AG ! mseriousGameVerif.Error;
CTLSPEC EF mseriousGameVerif.test_ok;
CTLSPEC AG(mseriousGameVerif.v_s_right_picture_chosen_seriousGame_i0
 ->mseriousGameVerif.test_ok);
```

mseriousGameVerif est le nom du module global (seriousGameVerif) généré dans le format NuSMV. La première propriété vérifie qu'il n'existe aucun chemin ou on génère une erreur *Error*, ce qui veut dire que notre programme est correct vis à vis de cette propriété et qu'on n'a jamais *displays_happy_smiley(Equipment)* et *wrong_picture_chosen* sur le même chemin.

La deuxième propriété vérifie qu'il doit exister au moins un chemin où on finit par générer l'alerte *test_ok*, ce qui veut dire que le programme a été exécuté avec succès et que les deux évènement *displays_happy_smiley(Equipment)* et *wrong_picture_chosen* n'arrivent pas ensemble.

La troisième propriété vérifie que pour tous les chemins, l'évènement d'alerte *test_ok* est présent si l'évènement *right_picture_chosen_seriousGame* l'est aussi. Ce dernier est présent lorsque *displays_happy_smiley(Equipment)* est présent.

Les résultats des preuves dans le model-checker NuSMV sont indiqués dans la figure 4.6.

```
[isarray@elacrab sam_09_2018]$ ./NuSMV seriousGameVerif.smv
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:36:56 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

-- specification AG !mseriousGameVerif.test_failure  is true
-- specification EF mseriousGameVerif.test_ok  is true
-- specification AG (mseriousGameVerif.v_s_right_picture_chosen_seriousGame_i0 -
> mseriousGameVerif.test_ok)  is true
[isarray@elacrab sam_09_2018]$
```

FIGURE 4.6 – Résultats de la vérification des propriétés dans le model-checker NuSMV

Simulation

Pour compléter la validation avec NuSMV qui se limite aux propriétés de la logique temporelle, notre système offre aussi la possibilité de simuler les programmes ADeL. Pour ce faire, à partir de notre description d'activité avec ADeL, nous générons un format spécifique *blif*[15] qui est interprété par l'outil *blif_Simul*[16]. Cet outil est un simulateur graphique d'automates implicites au format *blif*. C'est un logiciel qui aide à simuler et tester le fonctionnement des activités décrites avec ADeL, il permet d'afficher graphiquement les valeurs et le comportement des évènements des activités. Des exemples de simulations sont décrits dans le chapitre 6.

4.7 Conclusion

Nous avons doté le langage ADeL de deux sémantiques formelles, l'une pour décrire le comportement abstrait d'un programme, la deuxième pour compiler le programme en un automate décrit sous la forme d'un système d'équations. Cette deuxième sémantique nous permet une compilation modulaire d'un programme ADeL et assure une génération facile et efficace du code. Ces deux sémantiques s'appuient sur une algèbre quadri-valuée qui nous a permis d'exprimer le statut de chaque évènement de manière plus précise que l'algèbre booléenne. Décrire les sémantiques de certains opérateurs comme les opérateurs de gestion du temps de la montre (**timeout**) a été complexe.

15. bilf : (Berkeley Logic Interface Format) un standard crée par l'université de Berkeley
16. http://www.unice.fr/dgaffe/recherche/outils_blif.html

Dans ce chapitre, nous avons présenté ces deux sémantiques, leur contexte mathématique ainsi que leurs règles, montré leur relation et expliqué leur rôle important dans la compilation et la validation de notre langage. Cependant, décrire les activités et générer les automates correspondants ne constitue qu'une partie d'un système de reconnaissance. Dans le chapitre suivant, nous allons présenter notre système de reconnaissance global et expliquer le rôle de chaque composant de ce dernier et décrire plus précisément un composant important : le synchroniseur.

Chapitre 5

Transformation asynchrone/synchrone : le synchroniseur

L'essentiel : Dans ce chapitre, nous décrivons la structure et le fonctionnement du synchroniseur, un composant qui a un rôle important dans notre travail. Il s'agit de plonger un système synchrone dans le monde asynchrone, c'est-à-dire de réaliser une transformation asynchrone/synchrone. Nous présentons les hypothèses que nous avons choisies de suivre, ainsi que le paramétrage du synchroniseur. Nous décrivons son fonctionnement, et finalement, nous présentons une étude de cas comparant quelques stratégies.

Sommaire

5.1 Introduction

Dans cette thèse nous avons choisi de nous appuyer sur l'approche synchrone. Cette approche nous a facilité le travail en simplifiant la manipulation du temps, grâce au déterminisme et au parallèle synchrone, à la facilité de vérification formelle, etc. Donc, notre système de reconnaissance fonctionne de façon synchrone, c'est-à-dire qu'il n'accepte que des données synchrones en entrée et n'émet que des données synchrones en sortie. D'un autre côté, l'environnement des capteurs est un environnement asynchrone : chaque capteur est indépendant des autres et émet donc des données de façon asynchrone.

Pour valider notre approche, il était nécessaire de disposer d'un composant qui effectue cette transition de l'asynchrone vers le synchrone. C'est un problème difficile qui n'a pas de solution complète exacte. Dans ce chapitre nous proposons des pistes pour la structure d'un tel composant, que nous appelons *synchroniseur*, sous la forme d'une architecture paramétrable possible et de quelques idées sur les heuristiques.

Notre objectif pour cette première tentative est de créer un composant fonctionnel qui répond à notre besoin de validation.

5.2 Vue globale du système de reconnaissance

Construire un système de reconnaissance générique complet nécessite de nombreux composants. La figure 5.1 présente la structure globale de notre système de reconnaissance générique. Il y a deux parties principales : la *configuration*, qui se fait hors ligne et l'*exécution* en ligne.

Les contributions de la thèse interviennent dans la partie configuration (langage ADeL et son compilateur) mais aussi dans la partie en ligne (synchroniseur).

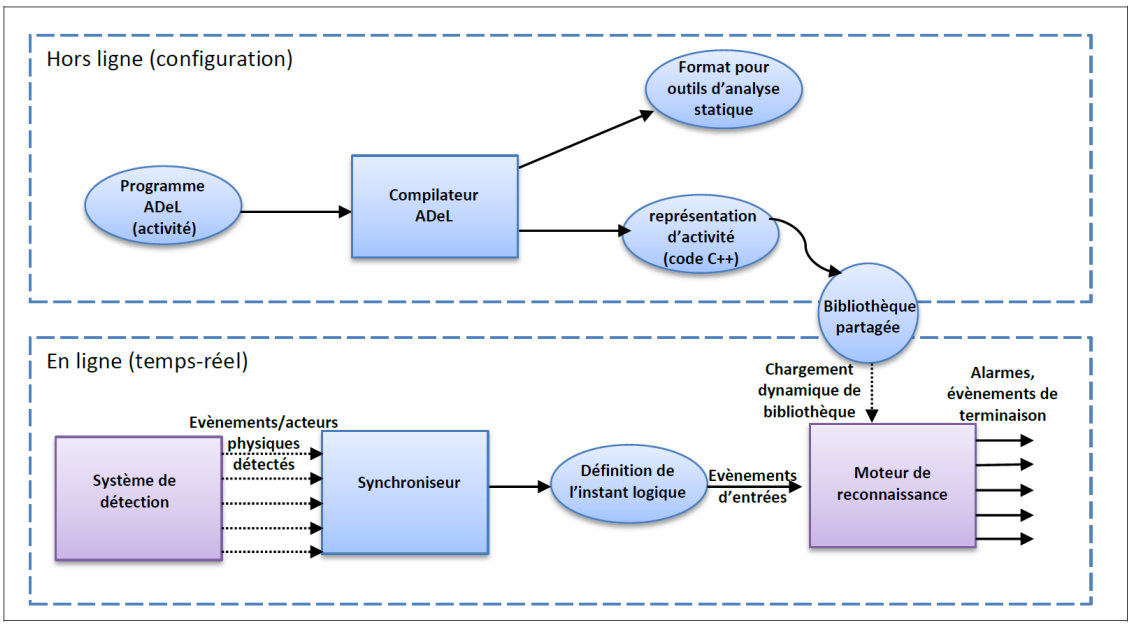


FIGURE 5.1 – Structure générale de notre système de reconnaissance d'activité (ADeL est utilisé lors de l'étape de configuration hors ligne) pour un automate (un programme ADeL).

5.2.1 Configuration du système

La première étape consiste à décrire les activités à reconnaître avec notre langage dédié ADeL. Chaque programme donne une description générique d'une activité (un *modèle* d'activité) en termes de *rôles* abstraits (au lieu des *acteurs* réels qui seront détectés au début de l'exécution).

Deuxièmement, chaque programme ADeL est compilé séparément, produisant finalement une bibliothèque (C++) partagée. Le système chargera ces bibliothèques au moment de l'exécution. Le moteur de reconnaissance est capable de reconnaître simultanément plusieurs activités correspondant à différents modèles prédéfinis. Les automates reconnaissant ces activités sont indépendants, c'est-à-dire qu'ils n'interagissent pas entre eux ; en revanche, ils peuvent partager des évènements d'entrée. Lorsqu'il reçoit les évènements d'entrée envoyés par le synchroniseur, le moteur de reconnaissance affecte les acteurs réels (objets détectés par les capteurs) contenus dans ces évènements d'entrée aux rôles correspondants dans le modèle d'activité.

Grâce à l'indépendance des automates mentionnée précédemment, il nous suffit de présenter le fonctionnement du synchroniseur pour une seule activité (et donc pour un seul programme ADeL) et c'est ce qui est fait dans la suite de ce chapitre.

En outre, le compilateur ADeL peut générer d'autres formats de sortie pour la simulation ou pour s'interfacer avec des outils d'analyse statique tels que les model-checkers.

5.2.2 Reconnaissance en temps réel

Le temps réel considéré dans cette thèse est celui lié à des activités "humaines" dont les constantes de temps ne sont pas forcément très rapides. Toutefois, ce temps réel peut être "dur", en particulier dans le cas de reconnaissance d'activités de personnes dans des conditions qui peuvent être dangereuses : une personne allongée et qui ne bouge pas pendant une durée supérieure à la normale, une personne qui tombe et qui ne se lève pas pendant un certain temps, ou encore une personne qui montre les symptômes d'une crise cardiaque. Dans ces exemples, il est obligatoire non seulement de ne jamais manquer un évènement mais aussi de le détecter dans un temps donné sous peine d'un fonctionnement invalide du système de reconnaissance et de conséquences graves.

Le système de détection extrait en permanence des événements et des objets (acteurs) à partir des informations issues de capteurs physiques (par exemple, vidéo, audio). Ces événements correspondent aux actions de base utilisables dans la description d'une activité. Le moteur de reconnaissance détecte à l'exécution toutes les activités correspondant à au moins un modèle à l'aide des informations envoyées par le système de détection. A chaque instant, il envoie les événements d'entrée vers les activités concernées, attribue des rôles aux acteurs et crée éventuellement de nouvelles instances d'activités. Il déclenche les transitions de toutes les activités en cours et collecte les événements de sortie (dans notre cas, les alarmes ou les terminaisons d'activité).

Comme notre moteur de reconnaissance est synchrone et que le système de détection est asynchrone, il faut traiter la communication entre ces deux composants. Pour ce faire, nous avons créé un composant intermédiaire, le *synchroniseur*, qui va filtrer ces événements asynchrones physiques et les regrouper en instants logiques, pour les envoyer au moteur de reconnaissance. Ce composant est décrit en détail dans les sections suivantes.

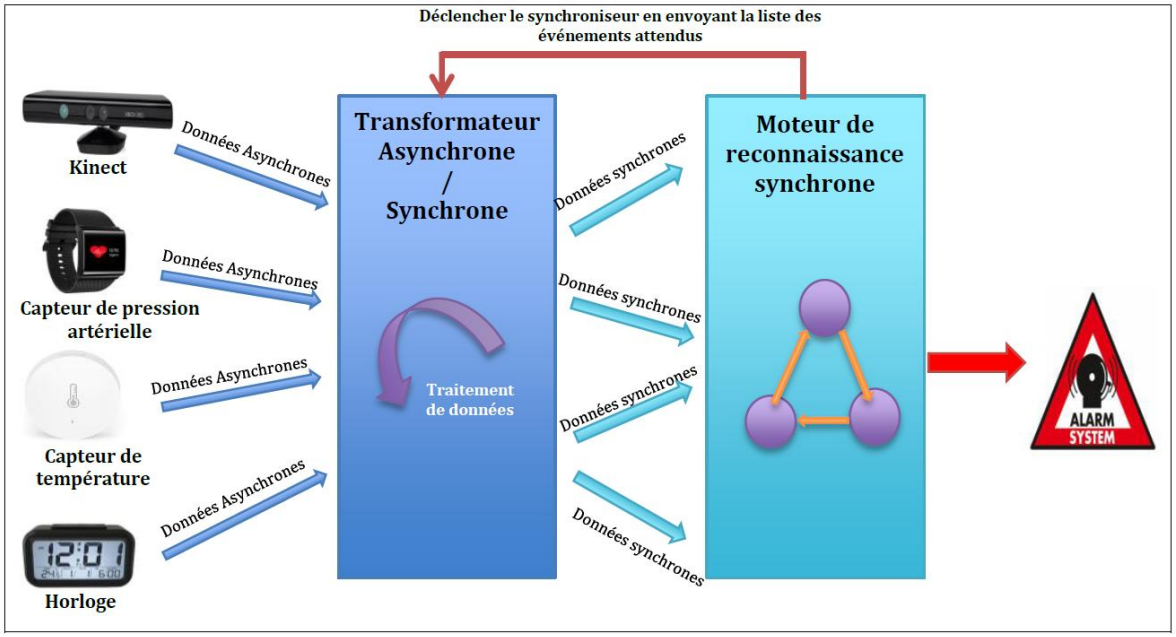


FIGURE 5.2 – Principe de fonctionnement du synchroniseur. Des exemples de capteurs sont mentionnés à gauche, les horloges sont des capteurs comme les autres, elles permettent une expression plus naturelle des ”timeout” pour l'utilisateur.

5.3 Quel est le rôle d'un synchroniseur ?

Un des buts de cette thèse est de pouvoir plonger notre système de reconnaissance synchrone dans un environnement de nature asynchrone. Pour transformer des évènements asynchrones en instants synchrones, il faut construire les instants logiques synchrones en fonction des évènements asynchrones donnés en entrée, pour ensuite les transmettre au système de reconnaissance synchrone. Pour cela nous introduisons le composant ”synchroniseur” entre les capteurs de l'environnement qui produisent des données bas niveau et notre moteur de reconnaissance qui traite des évènements de haut niveau. Le synchroniseur est chargé de transformer et de regrouper les données des capteurs pour les envoyer au moteur de reconnaissance afin de faire avancer l'automate de l'activité traitée par le moteur (voir figure 5.2).

La figure 5.3 montre la différence entre le temps physique et le temps logique créé à l'aide du synchroniseur. Elle illustre aussi la différence entre le comportement des systèmes asynchrones (en temps physique) et celui des systèmes synchrones (en temps logique). On remarque dans la figure que nous avons deux types de temps logique : un temps logique idéal et un temps logique opérationnel. Le temps logique idéal est celui créé en théorie et qui impose l'atomicité de la réaction du système (en temps nul). Dans ce temps logique idéal, la création de l'instant logique a lieu juste à la fin de la récupération de la dernière entrée. Le temps logique opérationnel représente ce qui se passe en réalité (lors du traitement de données). En effet, un instant logique peut être créé pendant la récupération des entrées de l'instant suivant. De plus, le temps d'exécution d'une transition n'est pas nul.

Il n'existe pas d'algorithme exact qui permette de faire la synchronisation de tout système asynchrone vers un système synchrone. Par exemple, quand décider d'arrêter l'attente d'un évènement absent ou que faire des évènements redondants ? Les algorithmes existants sont souvent destinés à un environnement ou à un fonctionnement précis. Ceci justifie l'introduction d'heuristiques pour concevoir un algorithme de

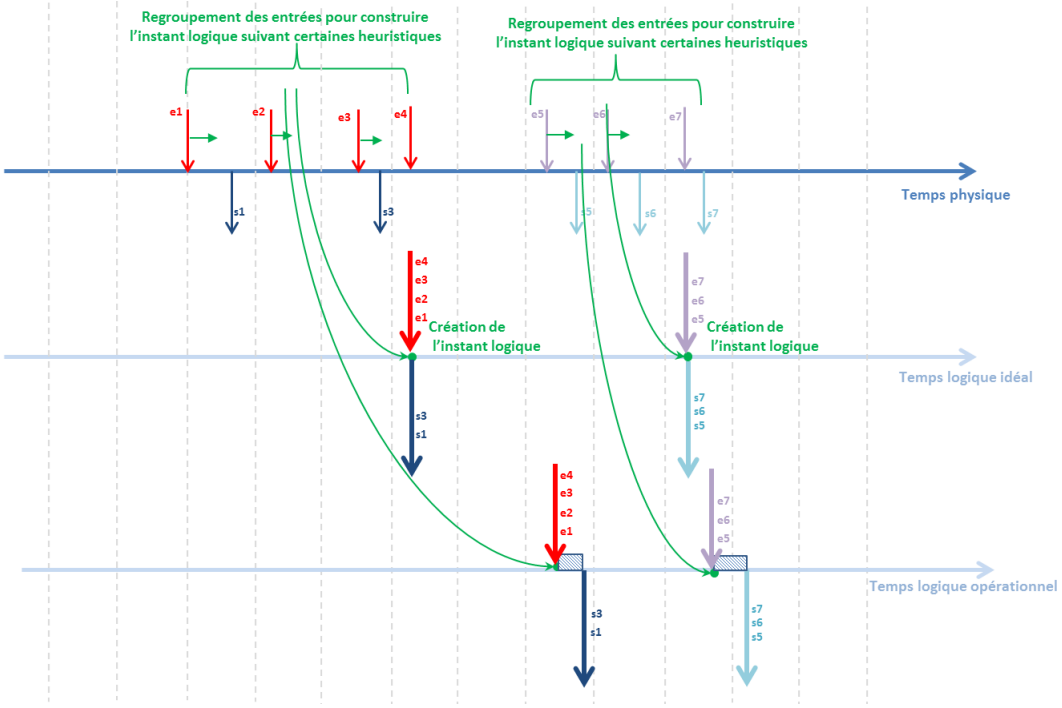


FIGURE 5.3 – Temps physique et temps logique (les flèches horizontales indiquent les regroupent d'évènements asynchrones pour constituer les instants logiques)

synchronisation générique qui peut fonctionner de plusieurs façons et dans différentes situations. Il y a deux points de variation principaux dans le synchroniseur où les heuristiques peuvent s'appliquer : au niveau du traitement des données issues des capteurs et au niveau de la constitution de l'instant logique. Nous appelons **stratégies** celles responsables du traitement de l'instant logique et nous appelons **tactiques** celles qui traitent les données provenant des capteurs. Par exemple, une stratégie peut décider de démarrer la construction de l'instant logique dès la réception du premier évènement attendu ou encore sous l'effet d'une horloge externe. Symétriquement, une stratégie peut décider de terminer un instant logique en satisfaisant un maximum d'évènements attendus dans un intervalle de temps donné. D'autre part, une tactique peut servir à résoudre le problème des données très nombreuses comme les informations sur les images qui arrivent au rythme de 25 fois par secondes en les fusionnant pour créer une donnée résultante.

Les données sont généralement issues des capteurs de l'environnement (caméra, capteur audio, capteurs sensoriels, chaine de traitement vidéo...). Ces données vont être traitées par les tactiques et elles seront envoyées au synchroniseur qui va créer des évènements à partir de ces données et consulter les stratégies pour regrouper ces derniers en un instant logique. Notre capteur privilégié est la chaine de traitement vidéo SUP [1] développée dans notre équipe STARS. SUP permet de percevoir, d'analyser, d'interpréter et de comprendre une scène dynamique 3D observée au moyen d'un réseau de capteurs.

1. https ://raweb.inria.fr/rapportsactivite/RA2010/pulsar/uid121.html

5.4 Synchroniseurs existants

On trouve peu de publications sur le problème de la transformation synchrone/asynchrone. Historiquement, les premiers travaux datent de 1991 avec Charles André, Jean-Paul Marmorat et Jean-Pierre Paris [AMP91], qui ont proposé une abstraction du processeur sur lequel Esterel peut être exécuté à l'aide d'une "machine d'exécution". Leur machine d'exécution abstraite était composée d'une unité de traitement d'entrée, d'une unité de traitement de sortie, d'une machine réactive, et d'une machine asynchrone. Les deux premières unités fournissent des fonctions d'interfaçage. L'unité de traitement d'entrée extrait les informations de l'environnement, les collecte et les rend disponibles pour le traitement ultérieur. La mise à jour de ces données est faite de façon asynchrone. L'unité de sortie permet les réactions du contrôleur vers son environnement.

La machine réactive est le cœur de la machine d'exécution, son rôle est de regrouper en instants les informations d'entrée de l'unité de stockage et d'exécuter la réaction selon un programme Esterel. Cette machine a trois composants. Tout d'abord, un générateur d'évènements a en charge la préparation des évènements d'entrée pour l'automate à partir des informations stockées par l'unité de traitement des données. Lorsque les évènements d'entrée sont définis, l'activation de l'automate peut être effectuée. Aucun changement dans les évènements d'entrée n'est autorisé pendant l'instant courant. Le deuxième composant est un automate qui calcule les états et qui gère la réaction du système à partir des évènements d'entrée. Enfin, une unité de traitement des actions synchrones qui exécute les actions associées à la transition d'un état à un autre, sous le contrôle de l'automate. Les actions sont dites synchrones parce qu'elles sont entièrement exécutées pendant un instant logique. La machine asynchrone est facultative, elle est utilisée pour l'exécution de quelques instructions. Elle est destinée à exécuter les actions qui durent sur plusieurs instants logiques.

La même année (1991), Daniel Gaffé [Gaf91] a aussi créé une machine d'exécution pour Esterel, qui permet de gérer plusieurs automates (opérateur **run** d'Esterel) à la fois, en plus de la gestion des tâches asynchrones (opérateur **exec** d'Esterel). En effet, pour un système d'exploitation temps réel multitâches, la machine d'exécution est une tâche asynchrone comme les autres. Les automates sont complètement gérés par la machine d'exécution et ils sont donc synchrones à l'intérieur d'une tâche asynchrone. Cette machine d'exécution est composée tout d'abord de canaux d'entrées représentés par des handlers qui recueillent les entrées dans des files FIFO. Ensuite, un générateur d'évènements gère ces données et engendre les instants logiques à partir de l'état de ces files d'attente. Puis en fonction d'une politique préalablement programmée dans le générateur, un ordonnanceur d'automates gère statiquement l'ordre de lancement de chaque automate en fonction de ses communications avec les autres automates. Enfin, un générateur de sorties permet d'émettre les signaux de sortie de différents automates vers l'environnement de façon cohérente pour l'environnement (voir figure 5.4). Cette machine d'exécution assure le fonctionnement des systèmes synchrones Esterel sur des systèmes d'exploitation temps-réel comme RTC.

Parmi les premiers travaux on trouve une machine d'exécution pour Esterel développée par Frédéric Boulanger et Charles André [ABG01, Bou93], qui s'est ensuite poursuivie par le développement du langage TESL pour combiner différentes notions du temps (discret, continu et périodique) [BJHP14].

On trouve aussi une autre machine d'exécution pour Esterel qui a été créée en

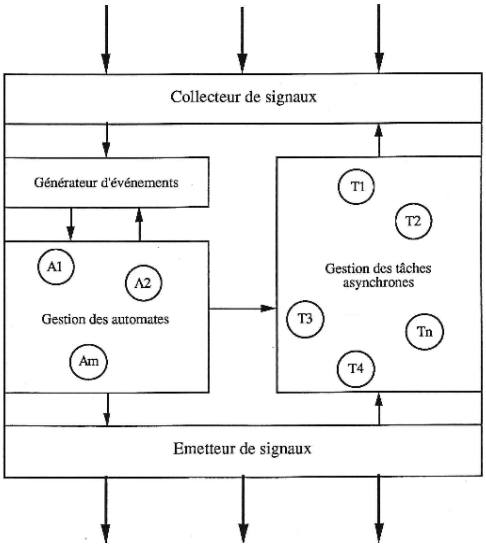


FIGURE 5.4 – Architecture de la machine d'exécution de D.Gaffé [Gaf91]

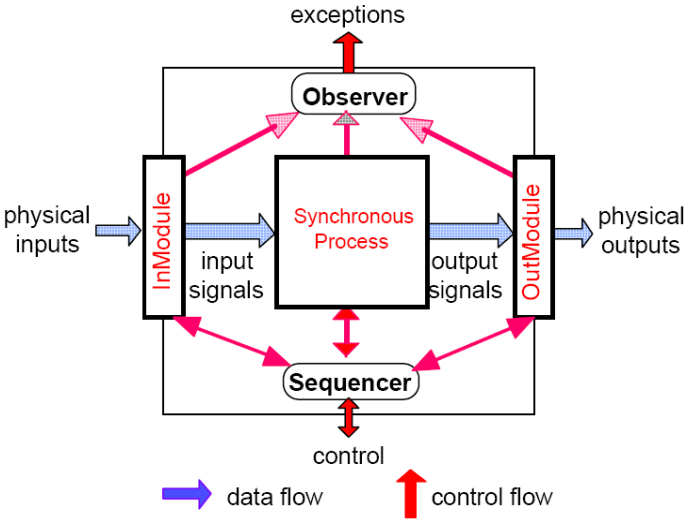


FIGURE 5.5 – Architecture de la machine d'exécution de C. André et H. Boufaied [AB00, Bou98]

1998 par Charles André et Hedi Boufaied [AB00, Bou98] pour combler l'écart entre le formalisme synchrone et sa mise en œuvre. Cette machine d'exécution (voir figure 5.5) présente une boite réactive qui englobe le programme/système synchrone, un module d'entrée qui transforme les données envoyées par le monde extérieur en données synchrones, un module de sortie qui transforme les sorties générées par le système synchrone en sorties physiques/asynchrones, et deux contrôleurs : un "séquenceur" et un "observateur".

Cette machine d'exécution a pour rôle d'exécuter des réactions pour que le comportement des entrées / sorties soit compatible avec celui décrit par le programme synchrone, de gérer les flux de données entrants et sortants en temps réel, de gérer les traitements qui durent sur plusieurs instants (l'opérateur **exec** d'Esterel) et qui sont gérés de la même façon que dans la machine d'exécution de Daniel Gaffé. L'idée est de préserver la sûreté apportée par l'approche synchrone.
Dans notre cas, puisque l'automate de reconnaissance fait déjà partie du système de reconnaissance, notre objectif est simplement de créer un module d'entrée pour ce

système.

Une autre façon de traiter la communication asynchrone/synchrone est d'utiliser le style de conception "globalement asynchrone localement synchrone" (GALS) [Cha85, HH03], le système est partitionné en blocs synchrones qui communiquent entre eux de manière asynchrone. L'architecture GALS permet un biais arbitraire entre les différentes horloges et utilise une forme de synchronisation pour la communication inter-bloc. Malheureusement, ces stratégies de synchronisation introduisent du non-déterminisme qui complique la validation, le débogage et le test.

Regrouper les états et les sorties d'un bloc synchrone dans un système GALS avec son horloge locale produira un système localement déterministe et une séquence de sortie en réponse à la séquence d'entrée donnée au bloc synchrone. Dans la plupart des méthodologies de GALS, chaque bloc synchrone possède son propre synchroniseur pour traiter ses entrées, mais il n'y a aucune synchronisation entre les blocs. Il y a donc un risque de non déterminisme. Un système GALS déterministe doit gérer la synchronisation des signaux à l'intérieur des blocs synchrones de sorte que la séquence d'entrée présentée au bloc synchrone soit unique malgré les variations de fréquence des signaux, les retards d'interconnexion, etc.

Les machines d'exécutions existantes sont capables de traiter les données de capteurs variés, mais leurs politiques de création d'instant logique sont fixes. C. André et H. Boufaied [AB00, Bou98] ont utilisé deux stratégies au niveau de l'acquisition de données (attente active ou par interruption) pour faire une mise à jour d'information. Dans notre cas, nous souhaitons créer un synchroniseur paramétrable et capable de suivre plusieurs politiques que ce soit au niveau de l'acquisition de données ou au niveau de la création d'instant. D'un autre côté, les machines d'exécution existantes n'offrent pas la garantie que l'instant créé est compatible avec le fonctionnement de l'automate synchrone. Plus spécifiquement, elles n'assurent pas que l'instant créé contient des évènements attendus par l'automate pour provoquer une transition. Pour palier ce problème, nous introduisons la notion d'*évènements attendus* par l'automate à chaque instant.

5.4.1 Notion d'évènements attendus

Définition

Les *évènements attendus* sont ceux qui sont nécessaires et suffisants pour provoquer la transition de l'automate d'activité à partir de l'instant courant. Ils servent donc au synchroniseur à déterminer des instants logiques compatibles avec ce qu'attend l'automate.

Définir des stratégies de création d'instant qui satisfont les évènements attendus de l'automate permet non seulement de garantir le bon fonctionnement de ce dernier, mais aussi d'optimiser le système de reconnaissance entier en évitant la création d'instants "fantômes" qui réveillent inutilement l'automate.

Satisfaction des évènements attendus

Le système d'équations associé à tout programme ADeL représente de façon implicite un automate synchrone. Un automate peut déclencher une transition sur la présence, l'absence d'un évènement d'entrée ou une conjonction ("et") de présence

ou d'absence d'évènements. Nous appellerons *monôme* une telle conjonction. Notons qu'un même évènement ne peut pas apparaître plusieurs fois dans un même monôme, que ce soit par sa présence, son absence ou les deux.

Les évènements attendus dans chaque instant sont donc structurés sous forme d'une disjonction ("ou") de monômes ("et"). Par exemple, si on a dans le code les instructions suivantes, **wait** a **seq if** b, les évènements attendus seront envoyés sous la forme de la formule logique : $a \vee (a \wedge b)$.

Soit m un monôme de la formule des évènements attendus. On note $\mathcal{E}_m$ (respectivement $\bar{\mathcal{E}_m}$) l'ensemble des évènements d'entrée référencés comme devant être présents (respectivement absents) dans le monôme m. Ainsi, si $m = a \wedge b \wedge \bar{c}$, $\mathcal{E}_m = \{a, b\}$ et $\bar{\mathcal{E}_m} = \{c\}$.

On dit qu'un ensemble $\mathcal{E}$ d'évènements d'entrée satisfait le monôme m si les deux conditions suivants sont vraies :

1. Tous les évènements indiqués comme devant être présents dans m sont aussi dans $\mathcal{E}$ ($\mathcal{E}_m \subseteq \mathcal{E}$).
2. Tous les évènements référencés comme absents dans m ne sont pas dans $\mathcal{E}$ ($\mathcal{E} \cap \bar{\mathcal{E}_m} = \varnothing$)

L'ensemble $\bigcup_{m\ satisfaits\ par\ \mathcal{E}} \mathcal{E}_m$ sera l'ensemble des évènements d'entrée transmis à l'automate, s'il n'est pas vide ; sinon le synchroniseur ne crée pas d'instant.

Le tableau suivant(tableau 5.1) présente un exemple d'évènements attendus et les résultats de différents cas où on les satisfait.

	Évènements attendus $a \vee (b \wedge a) \vee (b \wedge c) \vee (b \wedge \bar{a})$					Évènements envoyés au moteur de reconnaissance
Monomes à satisfaire		a	$(b \wedge a)$	$(b \wedge c)$	$(b \wedge \bar{a})$	
Évènements présents connus par le synchroniseur	a	✓				a
	a, b	✓	✓			a, b
	a, c	✓				a
	b				✓	b
	b, c			✓	✓	b, c
	c					
	a, b, c	✓	✓	✓		a, b, c

TABLEAU 5.1 – Exemples de satisfiabilité d'une formule des évènements attendus

Calcul de la formule des évènements attendus

Il y a deux manières de calculer la formule des évènements attendus. La première est statique (lors de la compilation). Elle consiste à calculer cette formule à partir du système d'équations généré par la sémantique opérationnelle. Cette méthode étudie, dans les états atteignables à partir de l'état initial de l'automate, toutes les combinaisons d'évènements d'entrée pour en déduire les états suivants et les évènements attendus dans cet état. Il s'agit donc d'une évaluation symbolique de l'automate. Cette méthode a l'avantage de fournir *a priori* l'ensemble des formules des évènements attendus pour chaque état de l'automate. Cependant, la combinatoire des évènements d'entrée peut être trop élevée et son temps de calcul s'est révélé rédhibitoire.

La deuxième méthode est dynamique. Elle consiste à calculer à partir de l'état courant la formule des évènements attendus pour l'état suivant. L'inconvénient c'est

qu'on ne découvre les formules qu'après avoir fait transiter l'automate. Donc les stratégies de création d'instant ne peuvent reposer que sur les propriétés de l'état courant, contrairement à la première méthode où il serait peut être possible d'avoir des stratégies qui anticipent sur plusieurs états. Nous avons choisi cette seconde méthode qui est plus réaliste en temps de calcul. Pour cela, nous avons étendu les règles de la sémantique opérationnelle pour effectuer le calcul de la formule des évènements attendus dans l'instant suivant (voir annexe C).

Pratiquement, pour chaque évènement d'entrée ou chaque combinaison d'évènements d'entrée apparaissant dans la description d'une activité, nous avons défini un évènement de sortie W_I (qui exprime le fait que l'évènement ou la combinaison I fait partie des évènements attendus ou non (W comme aWaited)). Nous avons ajouté dans la sémantique les équations opérationnelles pour calculer le statut de W_I.

Le calcul de W_I se fait selon les différentes situations suivantes :

1. W_I correspond à un évènement I de préemption déclarée "prioritaire" par l'utilisateur (cf. tableau 3.2), c'est à dire un évènement de préemption (d'un **Ptimeout** ou d'un **Pstop..when**) dont la présence nécessite un traitement particulier et doit être signalée le plus tôt possible à l'automate. Ces évènements correspondent à des situations d'urgence (début d'incendie, syncope du patient, appel au secours, etc). L'attente de I est transmise au synchroniseur avec l'information de préemption prioritaire. Notons que ces préemptions prioritaires doivent apparaître seules dans un monôme de la forme disjonctive de la formule des évènements attendus.
2. W_I correspond à un évènement d'entrée I non lié à une préemption prioritaire (par exemple, "le patient touche l'écran" dans le programme de la section 3.4.4). Dans ce cas, l'attente de I est transmise au synchroniseur.
3. W_I correspond à une combinaison I de *or*, *and* et *not* d'évènements d'entrée : les évènements de base constituant la condition sont envoyés au synchroniseur comme évènements attendus avec l'information relative à leur dépendance. Par exemple, si W_I est l'évènement associé à la condition S_1 *or* S_2 *and not* S_3, le synchroniseur reçoit la formule $S_1 \vee (S_2 \wedge \bar{S_3})$.

Pratiquement, pour calculer le statut de W_I, chaque opérateur est muni d'un évènement interne supplémentaire AWAITED, similaire à son évènement START pour démarrer le calcul des évènements attendus et des équations pour calculer W_I sont ajoutées. De plus, chaque évènement attendu mémorise, au cours de la compilation, la liste des évènements attendus qu'il implique. Par exemple, si I est le test d'un **if–then–else**, W_I a dans sa liste d'implications les évènements attendus des arguments.

5.5 Un synchroniseur paramétrable

Le synchroniseur est chargé de gérer l'interface avec l'environnement d'exécution de la reconnaissance d'activités, en particulier avec les capteurs, et de traiter les problèmes qu'une approche synchrone pure ne peut pas résoudre (retards, absence d'information, etc). Nous proposons de paramétrer notre synchroniseur par différentes heuristiques qui sont choisies par l'administrateur (ou par un utilisateur avancé) qui connait l'environnement asynchrone et l'activité décrite. Ces paramétrages sont fournis au synchroniseur via un fichier de configuration défini pour un environnement d'exécution. Rappelons que dans la première version développée, le synchroniseur ne traite qu'une seule activité.

Le synchroniseur doit communiquer d'une part avec les capteurs de son environnement et, d'autre part, avec le moteur de reconnaissance de notre système de reconnaissance (voir figure 5.2) :
Communication synchroniseur/moteur de reconnaissance :
Lors de l'exécution du système, le moteur de reconnaissance envoie au synchroniseur la formule des évènements attendus. C'est donc le moteur qui initie la communication avec le synchroniseur. Le synchroniseur communique avec le moteur de reconnaissance en lui retournant l'instant logique créé en fonction de cette formule. Le moteur de reconnaissance reçoit l'instant, le traite, et retourne ensuite la formule des évènements attendus à l'instant suivant.
Communication synchroniseur/environnement :
Le synchroniseur va chercher quelles données des capteurs peuvent correspondre à la formule des évènements attendus (la correspondance se fait sur le nom) envoyée par le moteur de reconnaissance pour les utiliser dans la création de l'instant logique.

5.5.1 Hypothèses

Nous avons posé plusieurs hypothèses qui déterminent la définition du synchroniseur. Nous les présentons ci-après ainsi qu'une spécification du diagramme de classes du synchroniseur. Le synchroniseur actuel se base sur six hypothèses :

Hypothèse 1

Nous considérons que nous recevons des données "normalisées". Le décodage des données brutes issues des capteurs est fait par ailleurs. Nous appelons ces données *SensorData*. Elles ont une structure générique d'évènement quel que soit le capteur qui a émis les données bas niveau.

Hypothèse 2

Dans cette version du synchroniseur, les stratégies et les tactiques sont choisies statiquement dès le début de l'exécution du système par l'utilisateur ou l'administrateur.

Hypothèse 3

On considère que les SensorData envoyés par les capteurs sont "impulsionnels", c'est à dire qu'un SensorData n'est pas conservé une fois pris en compte dans la construction d'un instant (même s'il n'appartient pas finalement à l'instant).

Hypothèse 4

Le synchroniseur impose son horodatage (timestamp) aux évènements créés de telle sorte que les SensorData aient une horloge unique.

Hypothèse 5

On peut avoir deux types de préemptions.

— Une préemption normale (non prioritaire) qui n'a pas *a priori* d'incidence sur la création de l'instant. Si un évènement de préemption non prioritaire arrive, le synchroniseur attend la fin de l'instant selon la stratégie choisie et continue à regrouper les évènements pour les envoyer au moteur de reconnaissance à la fin de l'instant.

— Une préemption prioritaire qui peut agir sur la création d'un instant : elle correspond aux cas urgents ou extrêmes. Dès qu'un évènement de préemption prioritaire arrive, le synchroniseur est prévenu pour éventuellement finir la création de l'instant et envoyer l'ensemble des évènements reçus au moteur de reconnaissance.

Hypothèse 6

On doit *satisfaire* les évènements attendus afin d'assurer qu'une transition va avoir lieu. Par exemple, pour la formule d'évènements attendus $a \vee (a \wedge b)$, on peut créer soit un instant qui contient "a" tout seul, soit un instant qui contient "a" et "b" ensemble.

5.5.2 Modèle de conception

Le diagramme de classes (voir figure 5.6) présente les acteurs internes intervenants dans le processus de transformation synchrone/asynchrone. Les classes les plus importantes dans ce diagramme sont les classes *Synchronizer*, *Strategy* et *Tactic*. Les stratégies et les tactiques assurent la généricité de notre synchroniseur car elles permettent par exemple à un utilisateur (ici administrateur) de rajouter de nouvelles stratégies ou de nouvelles tactiques adaptées à un domaine ou à un capteur particulier. Ces trois classes se basent sur d'autres classes qui représentent les entités qu'elles manipulent et avec lesquelles elles intéragissent. Nous commençons par présenter ces classes afin de rendre plus compréhensible le fonctionnement des trois classes principales. Nous avons cinq classes "secondaires" (en bleu dans la figure 5.6).

1. **Sensor**
   Cette classe représente une *abstraction* des capteurs. Les instances de cette classe vont envoyer des données de la classe SensorData. Ces instances sont capables de stocker les SensorData dans un buffer, et à partir de ces données, de créer un SensorData "final" selon une tactique de regroupement de données puis d'envoyer ce SensorData final au synchroniseur pour créer un évènement synchrone (de la classe Event).
2. **SensorData**
   Cette classe représente les flots de données typées ou les informations envoyées par les capteurs. Selon l'hypothèse 1, ces informations sont normalisées et ont une structure unique, générique pour tous les capteurs. Cette structure de données comporte un nom (généralement c'est le nom du capteur), une date (la date d'envoi de l'information), une fréquence (la fréquence d'envoi des données dans le temps) et le nombre de données nécessaires pour créer un évènement logique à partir de ce type de SensorData. C'est une classe mère dont plusieurs autres peuvent hériter. On peut trouver des SensorData purs (sans valeur), valués ou issus du traitement de données réalisé par la plateforme SUP.
3. **SensorDescriptors**
   Cette classe associe une description de chaque Sensor existant et des SensorData

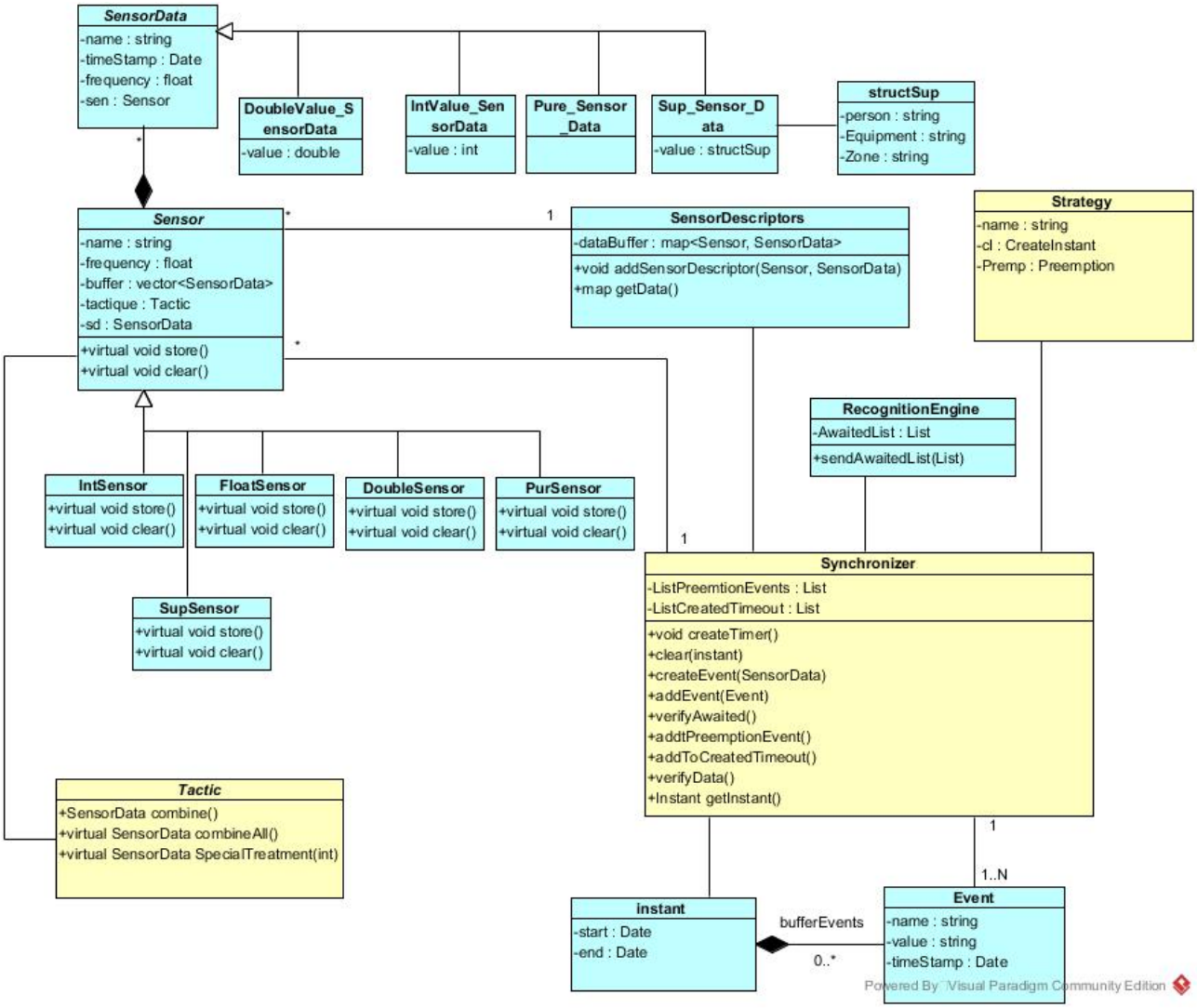


FIGURE 5.6 – Diagramme de classes du synchroniseur

qu'il peut fournir. Quand un nouveau Sensor est présent, sa description sera automatiquement ajoutée.

4. **Event**
   Event est la structure qui sera utilisée pour créer l'instant logique (un instant logique est composé d'une liste d'Event). Un évènement est créé par le synchroniseur, il contient un nom, une valeur, et un horodatage. La valeur est soit une valeur numérique, soit une structure de données contenant les objets reconnus par les capteurs. Par exemple, pour l'évènement "entrer(Person, Zone)" la structure de données de l'évènement créé par le synchroniseur à partir du capteur SUP (SupSensor) pourra contenir "Alex" de type Person et "salle de gym" de type Zone. Donc, le moteur de reconnaissance pourra apparier cet Event avec l'évènement attendu dans le programme ADeL.
5. **Instant**
   Instant représente l'instant logique qui sera envoyé par le synchroniseur au moteur de reconnaissance. Il contient la liste des évènements créés et regroupés par le synchroniseur.

Maintenant, nous présentons les trois classes principales (en jaune dans la figure 5.6) qui utilisent les classes présentées ci-dessus.

Synchronizer

La classe Synchronizer joue le rôle d'un "chef d'orchestre". Elle communique avec les classes Strategy et Sensor pour réaliser le traitement nécessaire à la création de l'instant logique selon la formule des événements attendus envoyée par le moteur de reconnaissance. Elle envoie cet instant au moteur de reconnaissance en lui demandant de lui retourner la formule des événements attendus de l'instant suivant. Dans notre version actuelle qui ne traite qu'une seule activité, nous n'avons qu'une seule instance de cette classe.

Strategy

La classe Strategy est associée à Synchronizer : il y a une Strategy par Synchronizer. Son rôle est de traiter l'instant logique, c'est-à-dire de décider (1) quand terminer la création de l'instant logique ; (2) et comment gérer les préemptions. Ces rôles sont définis par des heuristiques. Chaque heuristique est représentée par une classe de base abstraite contenant des méthodes virtuelles qui seront définies dans ses classes dérivées. Ceci permet à ces dernières de réaliser la tâche associée d'une façon différente. Nous proposons plusieurs classes dérivées qui spécialisent les méthodes de leur classe de base.

1. **Heuristique de création de l'instant logique** Cette heuristique (représentée par la classe CreateInstant de la figure 5.7) donne l'ordre au synchroniseur de créer l'instant logique tout en respectant les hypothèses de la section 5.5.1 ainsi que la satisfaction des évènements attendus. Nous avons implémenté trois façons de faire :

   (a) L'instant est crée dès qu'un monôme des évènements attendus est satisfait. Pour l'exemple $a \ \vee (a \ \wedge b)$, si on a reçu l'évènement "a" on crée l'instant et on l'envoie directement au moteur de reconnaissance. En revanche, si on reçoit d'abord l'évènement b, on doit attendre la réception de l'évènement a pour créer l'instant. Cette attente peut être infinie et si l'évènement a finit par arriver, on risque d'avoir un instant logique de longue durée physique ! Cependant, cette stratégie garantit la satisfaction des évènements attendus.

   (b) Si un monôme des évènements attendus est satisfait, on attend pendant une certaine durée avant de créer l'instant (et de l'envoyer au moteur de reconnaissance) afin de tenter de satisfaire un maximum de monômes des évènements attendus. Cette durée est un paramètre de configuration du synchroniseur, elle dépend de la fréquence avec laquelle les capteurs utilisés envoient leurs données. On peut prendre, par exemple, la valeur minimale ou la valeur moyenne des fréquences de ces capteurs.

   (c) Si on reçoit un élément d'un monôme des évènements attendus, on doit attendre la présence de tous les éléments de ce monôme pendant une durée définie à la configuration. Si cette durée est dépassée et qu'aucun monôme n'a pu être satisfait, on annule la création de l'instant et on n'envoie rien au moteur de reconnaissance.

   (d) On tente de créer un instant à chaque occurrence d'un évènement particulier ("tick"). Si les évènements connus du synchroniseur satisfont les évènements attendus par l'automate, on transmet l'instant, sinon on ignore l'instant et on attend le tick suivant. En général, cet évènement distingué sera fourni par une horloge mais il peut être un évènement quelconque du monde

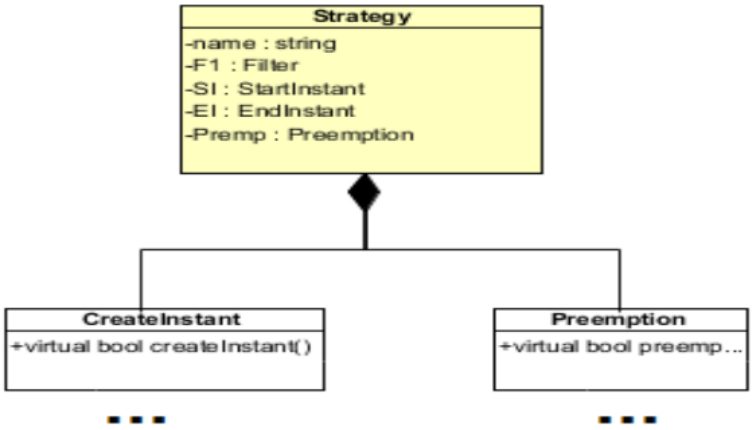


FIGURE 5.7 – Diagramme de classes de la classe Strategy avec ses heuristiques, les 3 points signifient qu'on peut avoir un héritage de ces classes

asynchrone. Notons que le tick lui même peut faire partie des évènements attendus, auquel cas il sera transmis dans l'instant.

Ces heuristiques peuvent être combinées pour créer de nouvelles heuristiques, tout en conservant l'hypothèse de base (satisfaire les évènements attendus). Par exemple, on peut combiner les heuristiques a et d ou c et d.

2. **Heuristique de gestion des préemptions**
   Cette heuristique (représentée par la classe Preemption) correspond aux méthodes qui permettent de gérer les préemptions prioritaires vues précédemment. Cette heuristique permet de finir directement la création de l'instant et de l'envoyer au moteur de reconnaissance. On rappelle que dans la formule des évènements attendus, un évènement de préemption prioritaire doit être seul dans son monôme et il suffit de le recevoir pour satisfaire les évènements attendus.

Tactic

Cette classe est responsable du traitement des données venues des capteurs. C'est pourquoi elle est liée à la classe Sensor. Il y a une instance de Tactic par instance de Sensor. Elle lui indique comment collecter et combiner ses données pour obtenir une donnée finale à envoyer au synchroniseur. Ce dernier peut ainsi créer l'évènement lui correspondant. La création de la donnée finale se fait selon plusieurs tactiques au choix. En effet, cette classe est une classe abstraite à partir de laquelle nous avons défini quatre classes filles en fonction des quatre types de capteurs que nous considérons dans ce synchroniseur.

Ces quatre classes peuvent elles-mêmes être encore dérivées pour raffiner les méthodes virtuelles de la classe Tactic. La classe Tactic a une méthode appelée *combine* dont le traitement est identique pour toutes les sous-classes. Elle indique au capteur d'envoyer au synchroniseur tout SensorData qui apparait, en le considérant comme un SensorData final. La classe Tactic propose aussi deux méthodes dont nous avons fourni une version dans les classes filles de premier niveau associées à chaque type de capteur.

1. La première méthode s'appelle *combineAll*, elle est redéfinissable et elle permet de combiner l'ensemble de SensorData stockés par le Sensor correspondant dès qu'on reçoit l'ordre du synchroniseur. Une autre tactique peut proposer de ne combiner qu'un nombre précis de SensorData (selon le choix de l'utilisateur/administrateur). L'implémentation de cette méthode diffère d'une tactique à une autre selon le type de capteur : si le capteur envoie des données pures, le SensorData final aura un nom, un horodatage identique à l'horodatage du dernier SensorData stocké (ou du premier, selon la tactique). Si le capteur a une valeur, la valeur du SensorData finale peut être la moyenne de toutes les valeurs des SensorData stockés, la valeur minimale ou la valeur maximale (selon la tactique).
2. La deuxième méthode s'appelle *SpecialTreatement*, cette méthode traite les cas spéciaux. Par exemple, pour un Sensor valué, une tactique peut indiquer que si ses SensorData atteignent un seuil donné, le Sensor doit combiner ses données et envoyer le résultat au synchroniseur. Une autre tactique peut proposer de ne combiner qu'un certain nombre d'occurrences de SensorData (donné par l'utilisateur). Une troisième tactique peut retourner seulement le $n^{ème}$ SensorData stocké, etc.

5.5.3 Scénario de fonctionnement du synchroniseur

Le scénario de fonctionnement du synchroniseur est illustré par le diagramme de séquence UML dans la figure 5.9. Il faut noter que dans ce diagramme nous considérons le moteur de reconnaissance comme un composant externe. Nous ne présentons aussi que la méthode *combine* de la classe Tactic.

Première étape : stockage de la description des Sensor existants

Le scénario commence par l'ajout de la description de chaque Sensor et des SensorData qu'il peut fournir dans le SensorDescriptors. En même temps, les Sensor stockent dans leur buffer les SensorData déjà fournis.

Deuxième étape : réception de la structure des évènements attendus et filtrage

Le synchroniseur reçoit la formule des évènements attendus envoyée par le moteur de reconnaisance. Il vérifie d'abord l'existence des évènements liés à des préemptions prioritaires, s'il trouve un évènement de préemption (*timeout* spécifique ou un évènement de danger), son nom est mis dans un tableau de préemptions prioritaires. Si l'évènement est lié à un *timeout*, le synchroniseur se charge de lancer un chronomètre ("timer") lié à ce *timeout*. Notons qu'un *timeout* peut généralement s'étendre sur plusieurs instants, c'est pourquoi le synchroniseur maintient une liste des *timeout* déjà lancés dans les instants précédents et qui ne sont pas finis. Si le *timeout* lié à l'évènement appartient déjà à cette liste, le synchroniseur vérifie juste que sa durée n'est pas dépassée. Sinon, il crée le chronomètre associé, et rajoute le *timeout* dans la liste. Lorsque la durée est écoulée, il crée l'évènement *timeout_elapsed* à envoyer à l'automate. Enfin, il filtre les données existantes dans le SensorDescriptors en ne choisissant que les Sensor dont les SensorData sont référencés dans l'ensemble des évènements qui satisfont les évènements attendus.

Troisième étape : réception de SensorData finaux et vérification de la satisfaction des évènements attendus

Sur demande du synchroniseur, les Sensor appliquent leurs tactiques pour créer un SensorData final. Le synchroniseur crée donc des évènements à partir des SensorData finaux en vérifiant au fur et à mesure que les évènements ainsi créés satisfont les évènements attendus par l'automate ou une préemption prioritaire dans les évènements attendus.

Quatrième étape : existence d'un monôme ou d'une préemption prioritaire

- **Cas d'une préemption :**
  Si un évènement de préemption prioritaire est reçu, le synchroniseur finit la création de l'instant en ne prenant en considération que les évènements satisfaisant les évènements attendus (monômes) existants, et l'envoie au moteur de reconnaissance.
- **Cas d'un monôme non lié à une préemption :**
  En l'absence de préemption prioritaire, le synchroniseur applique l'heuristique de la stratégie choisie pour la création de l'instant, ce qui lui permet de décider s'il doit créer l'instant et l'envoyer au moteur de reconnaissance.

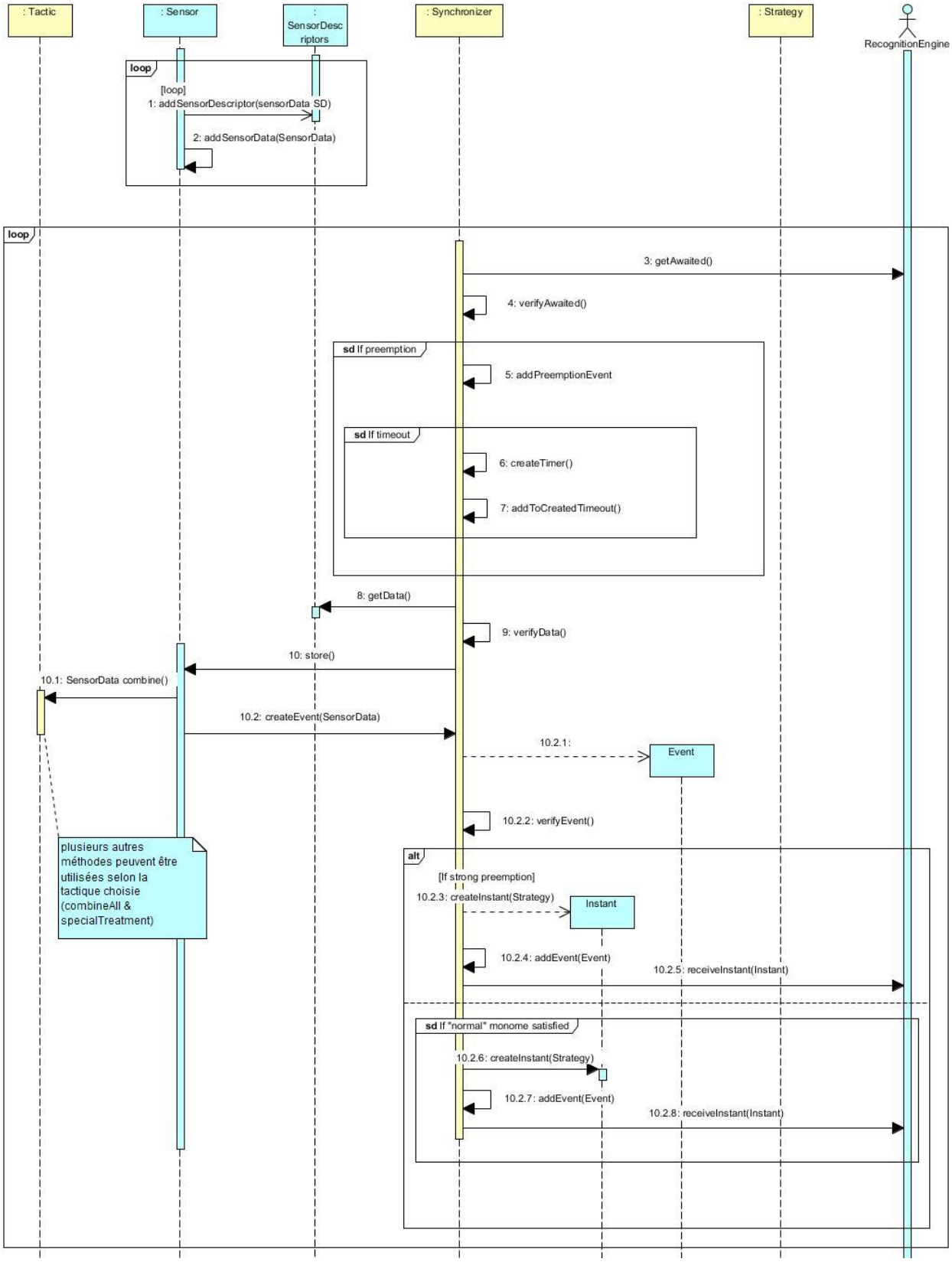


FIGURE 5.8 – Diagramme de séquence décrivant le fonctionnement normal typique du synchroniseur (et ses variantes).

5.6 Mise en œuvre et comparaison d'heuristiques

Dans cette section nous allons détailler un exemple de scénario pour illustrer le fonctionnement de notre synchroniseur. Cet exemple est lié au cas d'utilisation présenté dans la section 3.4.4 du chapitre 3. Dans chaque état d'exécution du programme, le synchroniseur reçoit la formule d'évènements attendus. Nous montrons dans le tableau 5.2 les évènements envoyés à un instant t (les lignes de code mentionnées dans le tableau correspondent au cas d'utilisation décrit dans la section 3.4.4) et la formule des évènements attendus pour l'instant t+1 pour chaque état du programme. Chaque évolution d'un état à un autre représente un instant. A chaque instant la satisfaction d'une formule d'évènements attendus conditionnera les instants suivants. Nous montrons dans la figure 5.9 les instants générés dans cet exemple.

Instant	**Evènements envoyés à l'automate à l'instant t**	**Formule d'évènements attendus pour l'instant t+1**
0		✓*inside_zone_patient_touchPad_gameZone*
1	✓inside_zone(patient, touchPad, gameZone) (ligne 1)	✓*touches_start_game_button_patient*
2	✓touches_start_game_button(patient) (ligne 3)	✓*Pstop_gets_out_of_zone_patient_gameZone* ∨ *displays_picture_touchPad*
3	✓displays_picture(touchPad) (ligne 7)	✓*Pstop_gets_out_of_zone_patient_gameZone* ∨ *asks_to_chose_right_picture_touchPad*
4	✓asks_to_chose_right_picture(touchPad (ligne 9)	✓*Pstop_gets_out_of_zone_patient_gameZone* ∨ *Ptimeout_5_min_touch_the_screen_patient* ∨ *timeout_8s_touches_the_screen_patient* ∨ *touches_the_screen_patient*
5.1	✓timeout_8s_touches_the_screen_patient (ligne 13) (absence de l'évènement *touches_the_screen_patient*)	✓*Pstop_gets_out_of_zone_patient_gameZone* ∨ *Ptimeout_5_min_touch_the_screen_patient* ∨ *asks_to_chose_right_picture_touchPad*
5.2	✓touches_screen(patient) (ligne 13)	✓*displays_happy_smiley_touchPad* ∨ *displays_unhappy_smiley_touchPad* ∨ *Pstop_gets_out_of_zone_patient_gameZone*
6.1	✓asks_to_chose_right_picture(touchPad) (ligne 9)	✓*Pstop_gets_out_of_zone_patient_gameZone* ∨ *Ptimeout_5_min_touch_the_screen_patient* ∨ *timeout_8s_touches_the_screen_patient* ∨ *touches_the_screen_patient*
6.2	✓displays_happy_smiley(touchPad) (ligne 21)	✓(*erases_picture_touchPad* ∧ *displays_picture_touchPad*) ∨ *Pstop_gets_out_of_zone_patient_gameZone*
7.2	✓erases_picture(touchPad) (ligne 32)	✓*Pstop_gets_out_of_zone_patient_gameZone* ∨ *displays_picture_touchPad*
8.2	✓displays_picture(touchPad) (ligne 32)	✓*Pstop_gets_out_of_zone_patient_gameZone*
6.3	✓displays_unhappy_smiley(touchPad)(else) (ligne 34)	✓*asks_to_chose_right_picture_touchPad* ∨ *Pstop_gets_out_of_zone_patient_gameZone*
7.3	✓ask_to_chose_right_picture(touchPad) (ligne 38)	✓*Pstop_gets_out_of_zone_patient_gameZone*

TABLEAU 5.2 – Tableau des évènements envoyés en fonction des évènements attendus pour chaque instant de l'exemple de 3.4.4

Ce cas d'utilisation ne contient que des évènements purs non valués provenant d'une Kinect (d'où l'utilisation de SensorData purs). Notre but est de vérifier les combinaisons de SensorData selon les différentes tactiques et différents types d'évènements. Nous souhaitons aussi vérifier comment fonctionnent les stratégies de création d'instant et, le cas d'une préemption. Dans cet exemple, les SensorData sont envoyés de façon

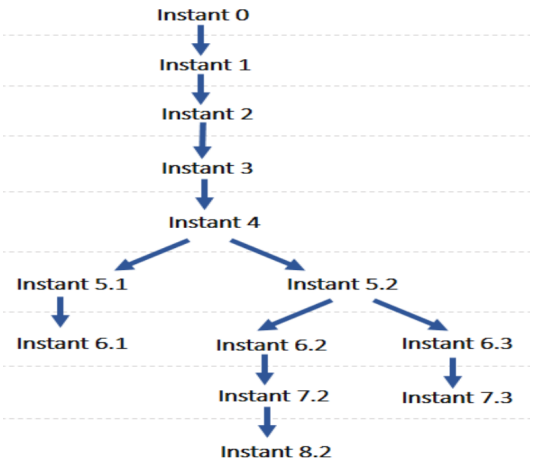


FIGURE 5.9 – Instants générés pour l'exemple du tableau 5.2.

automatique. Pour ce faire, nous avons développé un programme qui génère ces données au bon format et les envoie automatiquement selon une fréquence modifiable.

Cet exemple est composé de plusieurs sous-cas, chaque sous-cas applique une stratégie et une tactique différente pour chaque Sensor. Pour la liste des SensorData purs, et puisqu'ils proviennent tous du même capteur (Kinect), nous proposons que la tactique qui permet la création du SensorData final soit la plus simple, c'est à dire envoyer le SensorData dès qu'il est créé, on n'a pas besoin de tactique de combinaison dans ce cas.

Nous présentons la stratégie choisie pour chaque cas, nous rappelons qu'une stratégie contient deux heuristiques : heuristique de création d'instant et heuristique de préemption.

1. **Premier cas**
   - ***Stratégie***
     - Création d'instant L'instant est créé et envoyé dès qu'on satisfait un monôme des évènements attendus.
     - Préemption En recevant un évènement lié à une préemption prioritaire, on finit la création d'instant et on l'envoie au moteur de reconnaissance.
2. **Deuxième cas**
   - ***Stratégie***
     - Création d'instant Quand on satisfait au moins un monôme, on attend pendant une certaine durée avant de créer l'instant afin d'avoir la possibilité de satisfaire un maximum d'évènements attendus. Pour cet exemple, nous choisissons arbitrairement la durée supplémentaire d'une minute. Nous considérons que le synchroniseur a sa propre horloge lui permettant d'incrémenter son compteur interne pour atteindre cette minute.
     - Préemption On crée l'instant et on l'envoie dès qu'on reçoit un évènement lié à une préemption prioritaire.
3. **Troisième cas**
   - ***Stratégie***

— Création d'instant L'instant doit être créé toutes les 3 minutes (période choisie arbitrairement ici mais qui peut dépendre des caractéristique dynamique de l'application et des capteurs) sous condition de satisfaire au moins un monôme des évènements attendus.

— Préemption On crée et on envoie l'instant dès qu'on reçoit un évènement lié à une préemption prioritaire.

Nous souhaitons comparer dans ces exemples les différents instants créés à l'aide des différentes stratégies. Nous montrons les différentes manières de créer les instants à partir de l'instant 4 (instants 5.1 et 5.2) et à partir de l'instant 6 (instants 6.2 et 6.3) car c'est dans ces derniers qu'il y a le plus d'évènements attendus.

Instants créés à partir de l'instant 4 Pour traiter l'instant 4, nous commençons tout d'abord par les suppositions suivantes :

— l'évènement "touches_the_screen(patient)" arrive au bout de 30 secondes.

— l'évènement "gets_out_of_zone_patient_gameZone" arrive au bout de deux minutes et 30 secondes.

Pour l'instant 4, la formule d'évènements attendus sera de la forme :
$timeout\_8s\_touches\_the\_screen\_patient \vee timeout\_5min\_touches\_the\_screen\_patient \vee touches\_the\_screen(patient) \vee Ptimeout\_gets\_out\_of\_zone\_patient$.

Pour le premier cas, dès qu'on reçoit des évènements qui satisfont les évènements attendus, on crée l'instant et on l'envoie. Dans ce cas, on reçoit l'évènement lié au $timeout\_8s\_touches\_the\_screen(patient)$ à 8 secondes. Donc un monôme est satisfait, un instant qui contient l'évènement $timeout\_8s\_touches\_the\_screen(patient)$ sera créé et envoyé au moteur de reconnaissance.

Dans le deuxième cas, on satisfait les évènements attendus tout d'abord avec l'évènement $timeout\_8s\_touches\_the\_screen\_patient$, mais on attend pendant une minute avant de créer et d'envoyer l'instant afin de satisfaire un maximum de monômes. A 30 secondes, l'évènement $touches\_the\_screen(patient)$ arrive. Donc le monôme $touches\_the\_screen(patient)$ est satisfait. Une fois la minute supplémentaire écoulée, le synchroniseur crée un instant composé des évènements $timeout\_8s\_touches\_the\_screen(patient)$ et $touches\_the\_screen(patient)$ et l'envoie au moteur de reconnaissance.

Le troisième cas indique qu'un instant doit être créé toutes les trois minutes. Dans ce cas, l'instant sera fini avant la fin des trois minutes car une préemption prioritaire liée à l'évènement $gets\_out\_of\_zone\_patient\_gameZone$ arrive à deux minutes et 30 secondes. Donc l'instant sera créé et envoyé à 2 minutes et 30 secondes.

Le tableau 5.3 et le chronogramme de la figure 5.10 montrent les différentes façons de créer l'instant 4, leurs durées respectives selon les 3 cas ci-dessus et les moments de réception des SensorData finaux pour la création de leurs évènements correspondants.

Instants créés à partir de l'instant 5.2 Dans le tableau 5.4, nous montrons les différents instants obtenus à partir de l'instant 5.2 selon les différents cas. Notons que dans cet instant, les SensorData finaux n'arrivent pas au même moment qu'à l'instant 4 (voir le chronogramme de la figure 5.11). On suppose que les évènements de l'instant 6.2 arrivent dans les instants suivants :

— l'évènement *erases_picture(touchPad)* arrive au bout d'une minute et 30 secondes.

— l'évènement *displays_picture(patient)* arrive au bout de deux minutes.

— l'évènement *gets_out_of_zone_patient_gameZone* arrive au bout d'une minutes et 45 secondes.

Premier cas		
SensorData existants	**Monomes satisfaits**	**Instant final obtenu**
✓timeout_8s_touches_the_-screen_patient	✓*timeout_8s_touches_the_-screen_patient*	✓timeout_8s_touches_the_-screen_patient
Deuxième cas		
SensorData existants	**Monomes satisfaits**	**Instant final obtenu**
✓timeout_8s_touches_the_-screen_patient ✓touches_the_-screen(patient)	✓*timeout_8s_touches_the_-screen_patient* ✓*touches_the_-screen(patient)*	✓timeout_8s_touches_the_-screen_patient ✓touches_the_-screen(patient)
Troisième cas		
SensorData existants	**Monomes satisfaits**	**Instant final obtenu**
✓timeout_8s_touches_the_-screen(patient) ✓touches_the_-screen(patient) ✓gets_out_of_-zone(patient,gameZone)	✓*timeout_8s_touches_the_-screen_patient* ✓*touches_the_-screen(patient)* ✓*Pstop_gets_out_of_zone_-patient_gameZone*	✓timeout_8s_touches_the_-screen_patient ✓touches_the_-screen(patient) ✓Pstop_gets_out_of_-zone(patient,gameZone)

TABLEAU 5.3 – Tableau comparatif des différents instants à partir de l'instant 4 créés avec les différentes stratégies utilisées.

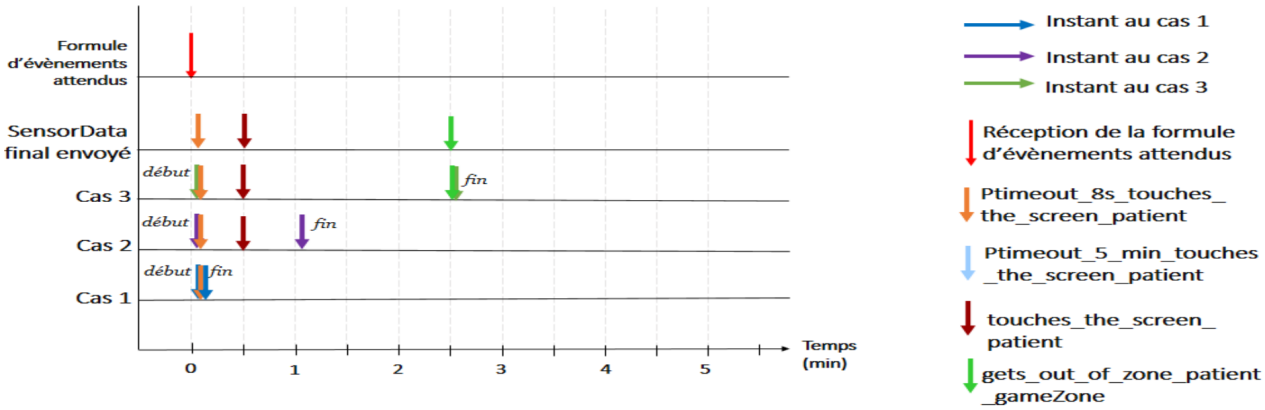


FIGURE 5.10 – Comparaison des instants créés à partir de l'instant 4 pour les trois cas proposés

Pour l'instant 5.2, la formule d'évènements attendus est de la forme :
$(erases\_picture(touchPad) \wedge displays\_picture(patient)) \vee Pstop\_gets\_out\_of\_zone(patient\_gameZone)$

De même que dans l'instant 4, nous considérons que les évènements arrivent au même moment pour les trois cas. Pour ces trois cas, le synchroniseur ne peut envoyer qu'un instant constitué de l'évènement $Pstop\_gets\_out\_of\_zone\_patient$ car dans les trois cas, même si on a l'évènement $erases\_picture(touchPad)$ qui arrive avant la préemption, on doit garantir la satisfaction des évènements attendus. Donc l'évènement $erases\_picture(touchPad)$ n'assure pas cette garantie et on ne le met pas dans l'instant.

Nous montrons de même les différentes créations d'instants et leurs durées à l'aide du chronogramme de la figure 5.11.

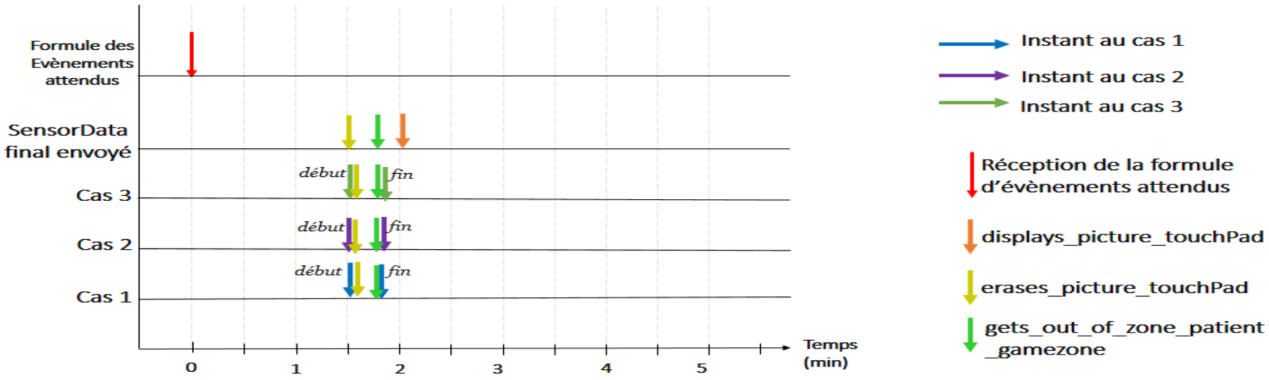


FIGURE 5.11 – Comparaison des instants créés à partir de l'instant 5.2 pour les trois cas proposés

Premier cas		
SensorData existants	**Monomes satisfaits**	**Instant final obtenu**
✓erases_picture(touchPad) ✓Pstop_gets_out_of_-zone(patient,gameZone)	✓*Ptimeout_gets_out_of_-zone_patient*	✓gets_out_of_-zone(patient,gameZone)
Deuxième cas		
SensorData existants	**Monomes satisfaits**	**Instant final obtenu**
✓erases_picture(touchPad) ✓Pstop_gets_out_of_-zone(patient,gameZone)	✓*Ptimeout_gets_out_of_-zone_patient*	✓gets_out_of_-zone(patient,gameZone)
Troisième cas		
SensorData existants	**Monomes satisfaits**	**Instant final obtenu**
✓erases_picture(touchPad) ✓Pstop_gets_out_of_-zone(patient,gameZone)	✓*Ptimeout_gets_out_of_-zone_patient*	✓gets_out_of_-zone(patient,gameZone)

TABLEAU 5.4 – Tableau comparatif de différents instants créés à partir de l'instant 5.2 avec les différentes tactiques et stratégies utilisées.

5.7 Conclusion

Dans ce chapitre nous avons tout d'abord présenté et comparé les synchroniseurs existants, en adoptant quelques unes de leurs idées.

Nous avons ensuite présenté notre dernière contribution de thèse : une première version d'un nouveau synchroniseur qui permet la transformation asynchrone/synchrone. Nous avons décrit son rôle dans notre système de reconnaissance global, sa composition ainsi que son fonctionnement. Contrairement aux autres approches, notre synchroniseur est paramétrable par différentes heuristiques (stratégies et tactiques), ce qui lui donne une certaine généricité. Les stratégies sont responsables de la construction de l'instant logique et les tactiques combinent les données provenant des capteurs pour former des évènements logiques. Le synchroniseur proposé ne traite

actuellement qu'une seule activité, dans le cas général où plusieurs activités sont à reconnaître, nous envisageons d'avoir plusieurs instances de synchroniseur, chacune paramétrée en fonction des besoins de l'activité concernée.

Pour illustrer les différentes stratégies actuellement implémentées, nous avons proposé une comparaison de la création d'instant avec différentes heuristiques. Il nous reste à présenter dans le chapitre suivant un cas d'utilisation global qui permet de décrire et de valider le fonctionnement de notre système de reconnaissance d'activités.

Chapitre 6

Expérimentations et tests

L'essentiel : Nous testons notre approche dans ce chapitre à l'aide de trois expérimentations. Ces expérimentations consistent à décrire des activités à partir de vidéos et à faire une simulation et des preuves pour valider le langage et son fonctionnement.

Sommaire

6.1 Introduction

Nous présentons dans ce chapitre trois cas d'utilisation. Chaque cas décrit des activités de personnes à partir de vidéos. Le premier cas d'utilisation décrit un protocole défini par les médecins de CoBTeK [1] dans le projet Demc@re [2]. Il consiste à reconnaître l'activité d'une personne âgée appelée à effectuer différentes tâches dans une période de temps déterminée [KJD+15]. Le deuxième cas provient d'une vidéo d'un ensemble de données traité par Toyota. En collaboration avec notre équipe de recherche, Toyota travaille aussi sur la reconnaissance d'activités à domicile. Ce cas d'utilisation consiste à reconnaître l'activité d'une personne en train de préparer à manger dans sa cuisine. Les vidéos des ensembles de données de Demc@re et de Toyota ne contiennent qu'une seule personne, nous souhaitons décrire des activités dans lesquelles plusieurs personnes participent. Pour ce faire, nous avons fait nous même une vidéo avec deux personnes dans une salle de gymnastique, qui est le troisième cas d'utilisation.

A partir de la description de ces activités avec le langage ADeL, notre compilateur a généré plusieurs formats, un format pour simuler le comportement des activités, un format pour la vérification formelle à l'aide du model-checker NuSMV et un format compatible avec notre moteur de reconnaissance.

6.2 Premier cas d'utilisation

Le premier cas d'utilisation consiste en la description d'activité de patients selon le protocole du projet européen Demc@re (**Dem**entia @mbient **care**) auquel notre équipe a participé. Ce projet vise à évaluer objectivement et à maintenir la capacité de personnes âgées ou atteintes de démence à mener de manière autonome des activités de la vie quotidienne.

L'objectif de ce projet est aussi de mettre au point un système complet fournissant

FIGURE 6.1 – Expérimentation sur l'aptitude d'une personne âgée de faire des activités quotidienne au sein du CMRR

des services de santé individualisés aux personnes atteintes de démence, ainsi

1. http://unice.fr/recherche/laboratoires/cobtek
2. http://www.demcare.eu/

qu'aux professionnels de la santé, en utilisant plusieurs capteurs pour la surveillance contextuelle et multi-paramétrique de l'environnement et des paramètres comme la mémoire par exemple. L'analyse de données multicapteurs, associée à des mécanismes décisionnels intelligents, permettra une représentation précise de l'état actuel de la personne et fournira le retour d'informations approprié, à la fois à la personne et aux soignants associés.

Les expérimentations se déroulent dans trois pays différents (France, Irlande, Suède) et dans des environnements différents : en laboratoire, en maison de retraite et à domicile.

Le premier cas d'utilisation consiste à reconnaître l'activité d'un patient dans une salle d'observation située dans le Centre Mémoire de Ressources et de Recherche (CMRR) du Centre Hospitalier Universitaire (C.H.U) de Nice (voir figure 6.1). Cette salle est équipée de plusieurs objets destinés aux activités de la vie quotidienne (Activity Daily Living (ADL)), par exemple un fauteuil, une table, un coin thé, une plante, un téléphone, etc.

Il est demandé au patient de réaliser des activités simples, chaque activité se déroulant dans une zone de la salle d'expérience. Dans cette expérience le patient doit effectuer six activités précises selon la zone dans laquelle il entre, dans une durée qui ne dépasse pas les 15 minutes au total pour les six activités [KJD+15].

— Arroser une plante
— Vérifier le compteur d'électricité et noter la consommation
— Préparer une enveloppe et la mettre dans une boîte à lettres
— Répondre au téléphone
— Ecrire une liste d'achats
— Préparer un thé

Pour la zone de thé, le patient prépare du thé en faisant bouillir l'eau, mettant l'eau dans un verre, puis le sachet de thé dans le verre et en buvant un peu de thé. Pour la zone plante, le patient doit prendre l'arrosoir, arroser la plante et mettre l'arrosoir sur la table. Dans la zone de compteur d'électricité, le patient doit vérifier le compteur d'électricité et écrire la valeur affichée sur un papier. Dans la zone d'écriture, le patient doit s'asseoir sur une chaise, prendre un papier, écrire (une liste d'achat) et poser le papier sur la table. Pour la zone de téléphone, le patient doit décrocher le téléphone, parler et raccrocher le téléphone.

Selon le protocole établi par les médecins de CoBTek, les activités réalisées peuvent l'être dans n'importe quel ordre et le patient peut passer d'une zone à une autre selon son choix (pas d'ordre obligatoire de tâches). Toutefois, ces activités doivent être toutes réalisées avant la fin de l'expérience.

6.2.1 Description de l'activité avec ADeL

Format textuel

La description du scénario de l'activité en mode textuel est comme suit. Tout d'abord on définit les types et les rôles des objets et acteurs dans l'activité. Nous avons trois types : Person, Equipment et Zone. Pour les rôles, nous avons un seul patient (Person), plusieurs équipement nécessaires au déroulement de l'activité (Equipment), comme une plante, un verre, un téléphone, une enveloppe, etc. Enfin, pour cette expérience nous avons défini 6 zones :

— une zone de thé

— une zone de plante

— une zone de compteur d'électricité

— une zone d'écriture

— une zone de préparation de lettre

— une zone de téléphone

```
Activity TestDemcare (patient : Person, table:Equipement, plant:Equipment,
                      watering_can:Equipment, paper:Equipment, envelope
                      :Equipment, electricity_meter:Equipment, stamp
                      :Equipment, letterBasket:Equipment, glass:Equipment,
                      chair:Equipment, door:Equipment, table2:Equipment,
                      paper2:Equipment, room:Zone, zone_plant:Zone,
                      zone_tea:Zone, zone_letter:Zone, zone_writing:Zone,
                      zone_electricity:Zone, zone_phone : Zone)
```

Ensuite nous définissons les évènements et les sous-activités qui interviennent dans le déroulement de l'activité. Notons que nous avons ici cinq activités car nous avons une activité imbriquée dans une autre (activité d'appel téléphonique).

```
Events
 inside_zone(Person, Zone);
 goes_to_zone(Person, Zone);
 opens(Person);
 gets_out_of_zone(Person,Zone);

SubActivities
 Watering_plant(Person,Equipment,Equipment,Equipment,Zone);
 Letter_and_phone(Person,Equipment,Equipment,Equipment,Equipment,Zone,Zone);
 Checking_electricity(Person,Equipment,Equipment,Zone);
 Writing(Person,Equipment,Equipment,Equipment,Zone);
 Preparing_tea(Person,Equipment,Equipment,Zone);
```

Finalement nous décrivons le déroulement du scénario de l'activité avec les évènements déjà définis, en utilisant les rôles.

```
1  InitialState : inside_zone(patient, room);
2  Start
3   {
4    {
5     wait goes_to_zone(patient, zone_plant)
6     seq
7      call Watering_plant(patient,watering_can,plant,table,zone_plant);
8    }
9    parallel
10   {
11    wait goes_to_zone(patient, zone_electricity)
12    seq
13    call Checking_electricity(patient,electricity_meter,paper,zone_electricity);
14   }
15   parallel
16   {
17    wait goes_to_zone(patient, zone_letter)
18    seq
19    call Letter_and_phone(patient,envelope,stamp,
20                          letterBasket,phone,zone_letter,zone_phone);
21   }
```

```
22    parallel
23    {
24     wait goes_to_zone(patient, writing_zone)
25     seq
26     call Writing(patient,chair,paper2,table2,writing_zone)
27    }
28    parallel
29    {
30     wait goes_to_zone(patient, zone_tea)
31     seq
32     call Preparing_tea(patient,tea_packet, glass,zone_tea);
33    }
34    parallel
35    {
36     wait opens(patient_door)
37     seq
38     wait gets_out_of_zone(patient,room)
39    }
40   } Ptimeout 15.0 min
41   {
42   emit Test_Ok
43   } alert Test_Failed
44   seq
45   emit test_demcare_over
46  End
```

Format graphique

La description de cette activité en format graphique est montrée dans la figure 6.2. Nous avons fait appel à plusieurs sous-activités qui décrivent chacune des tâches à faire. Ces sous-activités sont simples, nous montrons dans les figures 6.3 et 6.4 deux exemples.

6.2.2 Simulation

Nous allons tester le comportement de cette activité à l'aide d'une simulation. Pour cela, notre système génère le format *blif* à partir du programme ADeL et appelle le simulateur *blif_simul* qui nous permet de faire la simulation à travers son interface (voir figure 6.5). Dans cette interface de simulation, nous nous focalisons surtout sur la première et la dernière colonne. La première colonne indique la liste des évènements d'entrées du programme. La dernière colonne indique la liste des évènements attendus nécessaires pour l'instant suivant, pour pouvoir transiter de l'état actuel vers un état suivant ainsi que les alertes du programme. L'évènement *Timeout1* est un évènement qui correspond à l'attente de la durée de 15 minutes. Tant que le synchroniseur ne l'envoie pas ou que l'instruction de *timeout* n'est pas terminée, cet évènement sera présent dans la liste des évènements attendus, car il est préemptif. S'il existe, il va arrêter l'activité et envoyer *failure* avec une alerte *Test_Failed*, puisque c'est un **timeout prioritaire**. Le deuxième *timeout* (*Timeout0*) n'arrête pas l'automate puisque il n'est pas prioritaire. La présence de son signal va seulement générer une alerte *missed_call* et passer à l'instruction suivante. Dans la dernière colonne du simulateur, un carré jaune signifie absent et un carré bleu signifie présent. Pour éviter de mettre toutes les figures pour chaque instant, nous avons indiqué les résultats pour chaque évènement dans le tableau 6.1. Notons que ce tableau présente un seul cas (scénario),

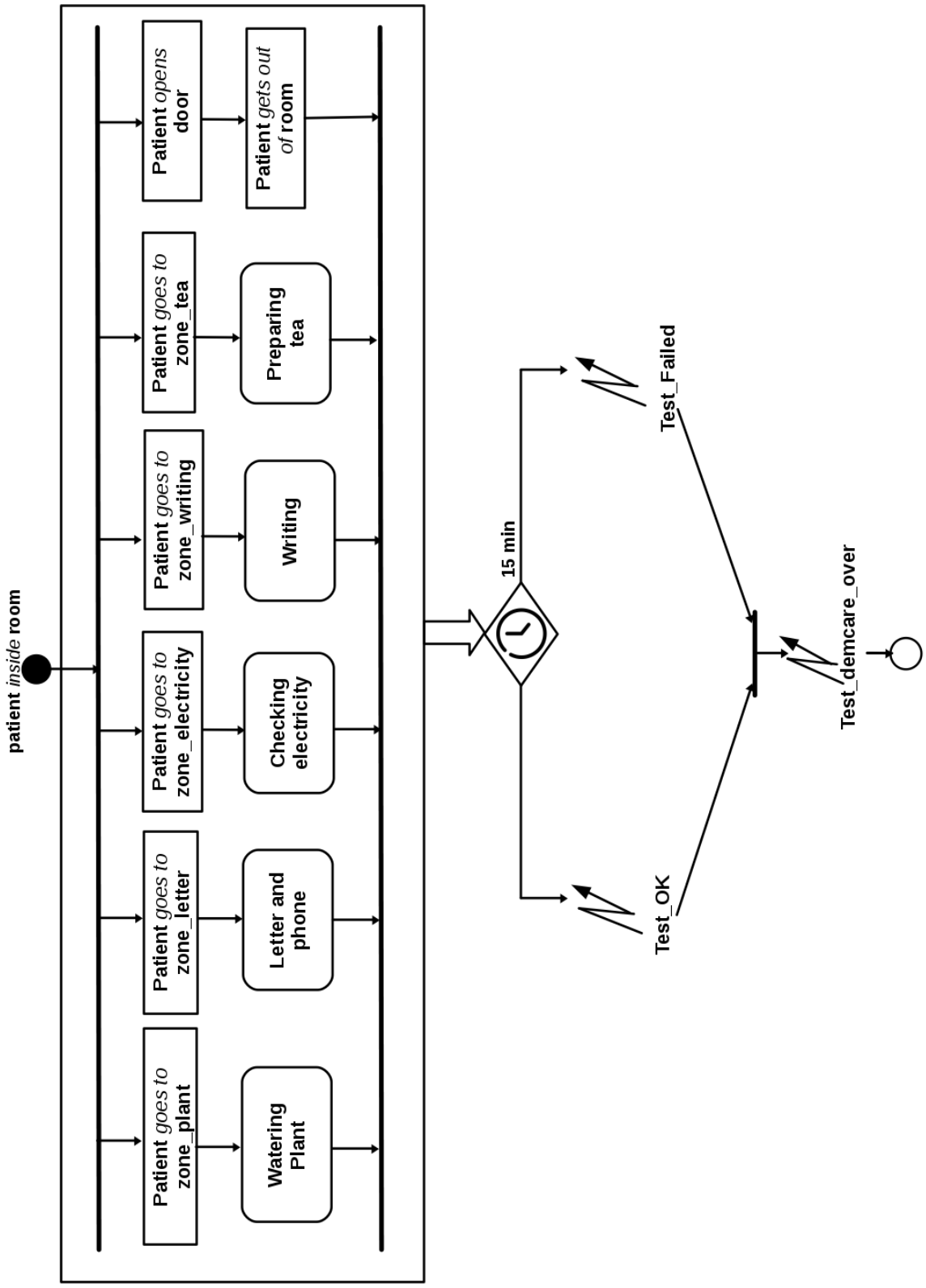


FIGURE 6.2 – Description graphique de l'activité TestDemcare. Le parallèle entre les sous-activités correspond à l'ordre non fixé pour la visite des zones.

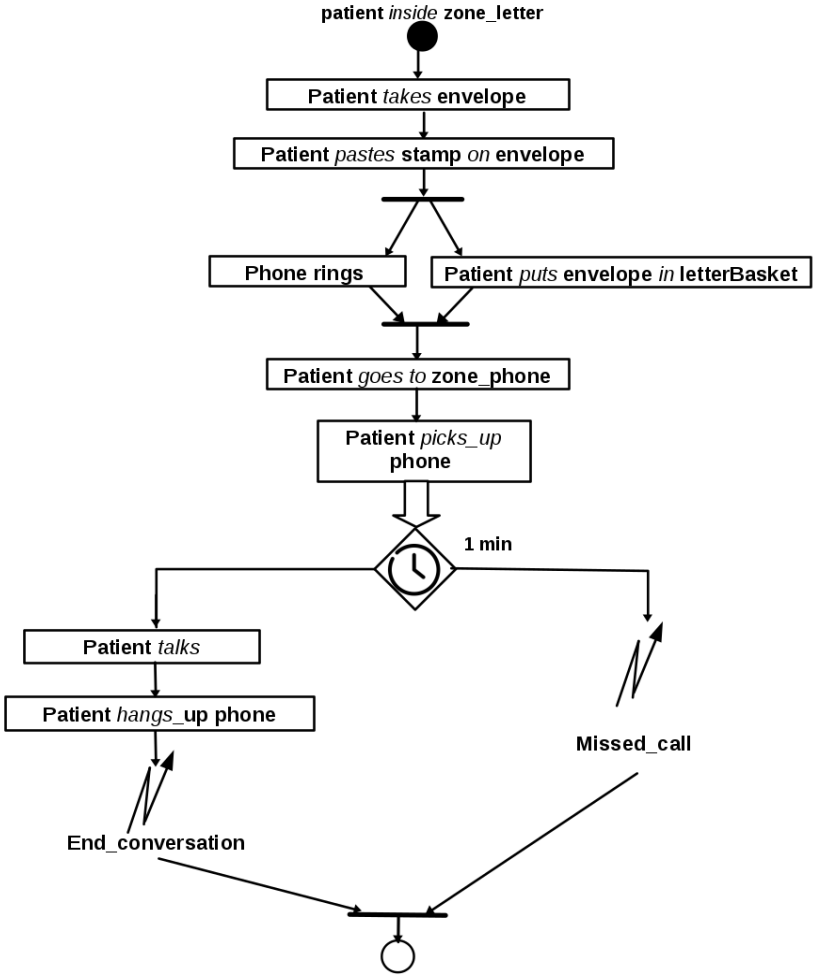


FIGURE 6.3 – Description graphique de la sous-activité *Letter and phone*

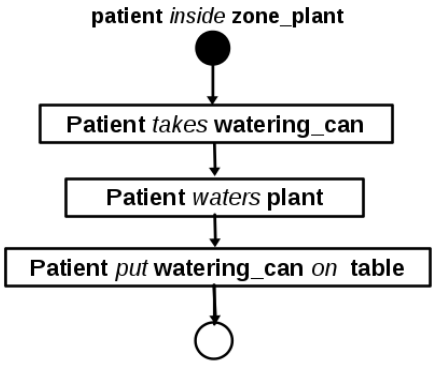


FIGURE 6.4 – Description graphique de la sous-activité *Watering plant*

les évènements attendus peuvent varier selon l'ordre d'entrée dans les zones. Dans ce tableau nous affichons la liste des évènements attendus car le simulateur n'affiche pas la formule d'évènements attendus comme le fait le moteur de reconnaissance pour le synchroniseur. Dans ce cas, la personne commence par arroser la plante, puis elle vérifie le compteur d'électricité, ensuite, elle prépare la lettre et elle répond au téléphone, puis elle écrit sa liste, puis elle prépare son thé et enfin elle sort de la salle.

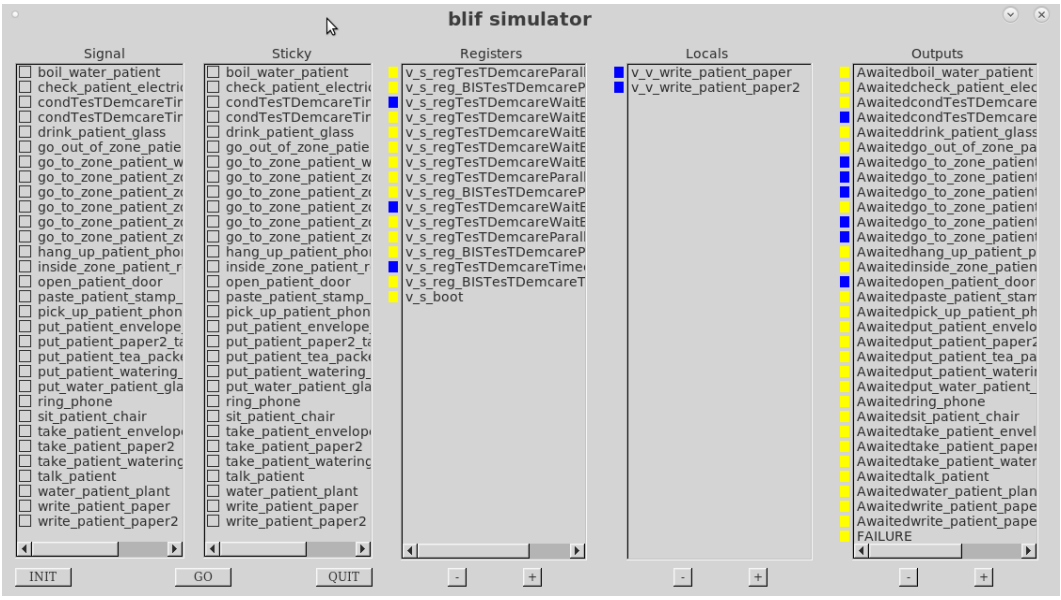


FIGURE 6.5 – Simulation du comportement de l'activité *TestDemcare* (affichage des évènements attendus du $2^{ème}$ instant)

Evènements envoyés à l'automate à l'instant t	**Evènements déclarés attendus par le simulateur pour l'instant t+1**	**Sorties générées**
Instant 0	Timeout1 inside_zone_patient_room	Aucune
inside_zone(patient, room)	Timeout1 go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_electricity go_to_zone_patient_zone_letter go_to_zone_patient_zone_plant go_to_zone_patient_zone_tea open_patient_door	Aucune
go_to_zone(patient, zone_plant)	Timeout1 take_patient_watering_can go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_electricity go_to_zone_patient_zone_letter go_to_zone_patient_zone_tea open_patient_door	Aucune
take(patient, watering_can)	Timeout1 water_patient_plant go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_electricity go_to_zone_patient_zone_letter go_to_zone_patient_zone_tea open_patient_door	Aucune

Evènements envoyés à l'automate à l'instant t	**Evènements attendus pour l'instant t+1**	**Sorties générées**
water(patient,plant)	Timeout1 put_patient_watering_can,table go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_electricity go_to_zone_patient_zone_letter go_to_zone_patient_zone_tea open_patient_door	Aucune
put(patient,watering_can,table)	Timeout1 go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_electricity go_to_zone_patient_zone_letter go_to_zone_patient_zone_tea open_patient_door	Aucune
go_to_zone(patient,zone_electricity)	Timeout1 check_patient_electricity_meter go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_letter go_to_zone_patient_zone_tea open_patient_door	Aucune
check(patient,electricity_meter)	Timeout1 write_patient_paper go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_letter go_to_zone_patient_zone_tea open_patient_door	Aucune
write(patient, paper)	Timeout1 go_to_zone_patient_zone_letter go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_tea open_patient_door	Aucune
go_to_zone(patient, zone_letter)	Timeout1 take_patient_envelope go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_tea open_patient_door	Aucune
take(patient,envelope)	Timeout1 paste_patient_stamp_envelope go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_tea open_patient_door	Aucune
paste(patient,stamp,envelope) .	Timeout1 put_patient_envelope_letterBasket ring_phone go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_tea open_patient_door	Aucune
put(patient,envelope,letterBasket) ring(phone)	Timeout1 go_to_zone_patient_zone_phone) go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_tea open_patient_door	Aucune

Evènements envoyés à l'automate à l'instant t	**Evènements attendus pour l'instant t+1**	**Sorties générées**
go_to_zone(patient, zone_phone)	Timeout1 Timeout0 pick_up_patient_phone) go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_tea open_patient_door	Aucune
pick_up(patient,phone)	Timeout1 talk_patient go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_tea open_patient_door	Aucune
talk(patient)	Timeout1 hang_up_patient_phone go_to_zone_patient_zone_writing_zone go_to_zone_patient_zone_tea open_patient_door	Aucune
hang_up(patient,phone)	Timeout1 go_to_zone_patient_writing_zone go_to_zone_patient_zone_tea open_patient_door	End_conversation
go_to_zone(patient, writing_zone)	Timeout1 sit_patient_chair go_to_zone_patient_zone_tea open_patient_door	Aucune
sit(patient,chair)	Timeout1 take_patient_paper2 go_to_zone_patient_zone_tea open_patient_door	Aucune
take(patient, paper2)	Timeout1 write_patient_paper2 go_to_zone_patient_zone_tea open_patient_door	Aucune
write(patient, paper2)	Timeout1 put_patient_paper2_table2 go_to_zone_patient_zone_tea open_patient_door	Aucune
put(patient,paper2,table2)	Timeout1 go_to_zone_patient_zone_tea open_patient_door	Aucune
go_to_zone(patient, zone_tea)	Timeout1 boil_water_patient open_patient_door	Aucune
boil_water(patient)	Timeout1 put_water_patient_glass open_patient_door	Aucune
put_water(patient,glass)	Timeout1 put_patient_tea_packet_glass open_patient_door	Aucune
put(patient, tea_packet, glass)	Timeout1 drink_patient_glass open_patient_door	Aucune
drink(patient, glass)	Timeout1 open_patient_door	Aucune

Evènements envoyés à l'automate à l'instant t	**Evènements attendus pour l'instant t+1**	**Sorties générées**
open(patient_door)	Timeout1 go_out_of_zone_patient_room	Aucune
go_out_of_zone(patient,room)	Timeout1	Test_OK Test_demcare_over
Timeout1 (15 min)		Test_Failed Test_demcare_over
Timeout0 (1 min)	go_to_zone_patient_writing_zone	Missed_call

TABLEAU 6.1 – Tableau de simulation de l'activité *TestDemcare*

6.2.3 Validation

Comme indiqué dans la section 4.6.2, pour faire des preuves sur cet exemple, nous créons tout d'abord un observateur qui permet de vérifier certains comportements. Pour cet exemple, nous souhaitons vérifier que si l'alerte *End_conversation* arrive après l'alerte *missed_call*, alors l'observateur génère une erreur (évènement *Error* dans l'observateur). Le code de l'observateur est comme suit :

```
Activity observerDemcare(patient:Person,phone:Equipment,room:Zone)
Events
 inside_zone(Person, Zone);
 Missed_call;
 End_conversation;
 Test_demcare_over;
InitialState : inside_zone(patient, room);
Start
 stop
 {
   wait Missed_call
  seq
   wait End_conversation
  seq
   emit Error
 }
 when Test_demcare_over alert TestDemcare_ok
End
```

Ce programme sera appelé et exécuté en parallèle avec le programme principal *TestDemcare* dans un programme global appelé *TestDemcareVerif*. Le code du programme global est comme suit :

```
Activity TestDemcareVerif(patient:Person,table: Equipement,watering_can:
                          Equipment,plant:Equipment,paper:Equipment,
                          electricity_meter:Equipment,envelope:Equipment,
                          stamp: Equipment,letterBasket:Equipment,
                          glass:Equipment,chair:Equipment,door:Equipment,
                          table2:Equipment,paper2:Equipment,room:Zone,
                          zone_plant:Zone,zone_tea:Zone,zone_letter:Zone,
                          zone_writing:Zone,zone_electricity:Zone,
                          zone_phone:Zone)
Events
goes_to_zone(Person, Zone);
opens(Person);
```

```
gets_out_of_zone(Person,Zone);

SubActivities
Watering_plant(Person,Equipment,Equipment,Equipment,Zone);
Letter_and_phone(Person,Equipment,Equipment,Equipment,Equipment,Zone,Zone);
Checking_electricity(Person,Equipment,Equipment,Zone);
Writing(Person,Equipment,Equipment,Equipment,Zone);
Preparing_tea(Person,Equipment,Equipment,Zone);
InitialState : inside_zone(patient,room);
Start
 local Missed_call, Test_demcare_over, End_conversation
 {
  call TestDemcare(patient,table,watering_can,plant,paper,
  electricity_meter,envelope,stamp,letterBasket,glass,chair,door,table2,
  paper2,room,zone_plant,zone_tea,zone_letter,zone_writing,
  zone_electricity,zone_phone)
  parallel
  call observerDemcareobserverDemcare(patient,phone,room)
 }
End
```

A partir du programme global, nous générons sa représentation dans le format d'entrée du model-checker NuSMV. Nous complétons la description en ajoutant les formules de logique temporelle représentant les propriétés que nous souhaitons prouver. Dans ce cas d'utilisation, les propriétés sont :

```
CTLSPEC AG ! mTestDemcareVerif.Error;
CTLSPEC EF mTestDemcareVerif.testDemcare_ok;
```

La première propriété signifie qu'il n'existe jamais un chemin où l'alerte *Error* est générée. La deuxième propriété signifie qu'il existe au moins un chemin où l'on arrive à finir l'exécution du programme avec succès. Notre programme vérifie ces deux propriétés. Le résultat de NuSMV est montré figure 6.6.

```
[isarray@elacrab sam_10_2018]$ ./NuSMV TestDemcareVerif.smv
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:36:56 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

-- specification AG !mTestDemcareVerif.Error  is true
-- specification EF mTestDemcareVerif.testDemcare_ok  is true
[isarray@elacrab sam_10_2018]$
```

FIGURE 6.6 – Résultats de vérification des propriétés du programme *TestDemcare* dans le model-checker NuSMV

6.2.4 Format pour le moteur de reconnaissance

Nous avons généré le format d'entrée pour le moteur de reconnaissance utilisé dans notre travail.Ce format correspond aux spécifications faites par les chercheurs créateurs de ce moteur de reconnaissance et il sera intégré très prochainement.

6.3 Deuxième cas d'utilisation

Dans le deuxième cas d'utilisation, l'objectif est de reconnaître d'autres activités quotidiennes d'une personne à partir d'une vidéo provenant de l'ensemble de données public CAD120 [DTS+19]. Dans cette vidéo, une personne est en train de préparer des céréales dans une zone (nous l'avons nommée Kitchen) (voir figure 6.7). Dans la vidéo

FIGURE 6.7 – Vidéo de préparation des céréales (activité à reconnaître)

de cette activité, le patient doit :

- être à côté de la table
- mettre le bol sur la table
- prendre la bouteille de lait et l'ouvrir
- mettre le lait dans le bol
- mettre la bouteille de lait sur la table
- fermer la bouteille de lait
- prendre le paquet des céréales
- mettre les céréales dans le bol
- mettre le paquet des céréales sur la table

Dans ce cas, le patient a le choix entre mettre le lait ou mettre les céréales en premier, donc nous avons mis ces deux séquences d'activités en parallèle.

6.3.1 Description de l'activité avec ADeL

Format textuel

Nous utilisons les mêmes types que dans le premier cas d'utilisation (Person, Equipment, Zone). La personne est supposée être dans une cuisine (Zone) et elle utilise

des équipements (Equipment) pour préparer ses céréales (bol, bouteille de lait, paquet de cérales, table).

```
Activity TestMeal (patient : Person,
                   table : Equipment,
                   bowl : Equipment,
                   cereal_packet : Equipment,
                   bottle_of_milk : Equipment,
                   kitchen : Zone)
```

```
Events
 inside_zone(Person, Zone);
 next_to(Person, Equipment);
 takes(Person, Equipment);
 opens(Person,Equipment);
 puts_milk(Person,Equipment);
 puts_cereals(Person,Equipment);
 puts(Person,Equipment,Equipment);
 closes(Person, Equipment);
```

Cette activité est simple, elle ne contient pas de sous-activités, la description du scénario de l'activité est comme suit :

```
InitialState : inside_zone(patient, kitchen)
Start
 wait next_to(patient, table)
 seq
 wait puts(patient, bowl,table)
 seq
  {
   wait takes(patient,bottle_of_milk)
   seq
   wait opens(patient,bottle_of_milk)
   seq
   wait puts_milk(patient,bowl)
   seq
   wait puts(patient, bottle_of_milk,table)
   seq
   if closes(patient, bottle_of_milk)
    then
    emit Go_ahead
    else
    emit Close_the_bottle
  }
  parallel
  {
   wait takes(patient,cereal_packet)
   seq
   wait puts_cereals(patient,bowl)
   seq
   wait puts(patient,cereal_packet,table)
  }
 seq
  emit TestMeal_over
 End
```

Format Graphique

Le format graphique de description de l'activité est présenté figure 6.8.

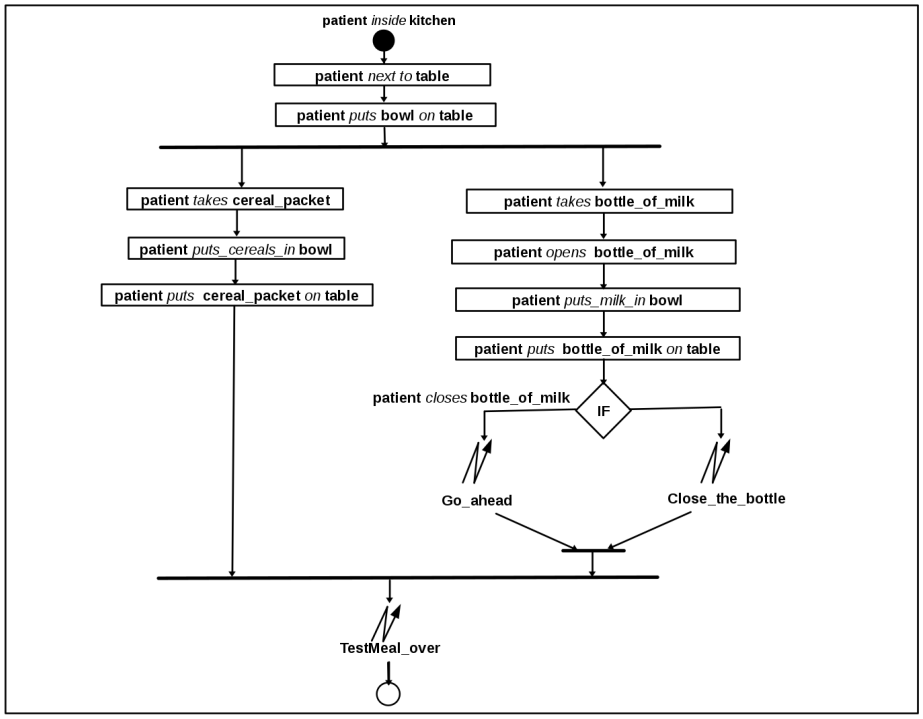


FIGURE 6.8 – Description graphique de l'activité *TestMeal*

6.3.2 Simulation

Nous avons fait une simulation d'un scénario de cette activité (voir figure 6.9) pour un scénario particulier. Dans cas, la vidéo commence par une personne à côté d'une table. La personne commence par mettre le bol sur la table, puis elle prend la bouteille de lait, l'ouvre et met le lait dans le bol, ensuite met la bouteille sur la table sans la fermer. Ensuite, elle prend le paquet de céréales et met les céréales dans le bol et enfin met le paquet sur la table. Les résultats sont affichés dans le tableau 6.2.

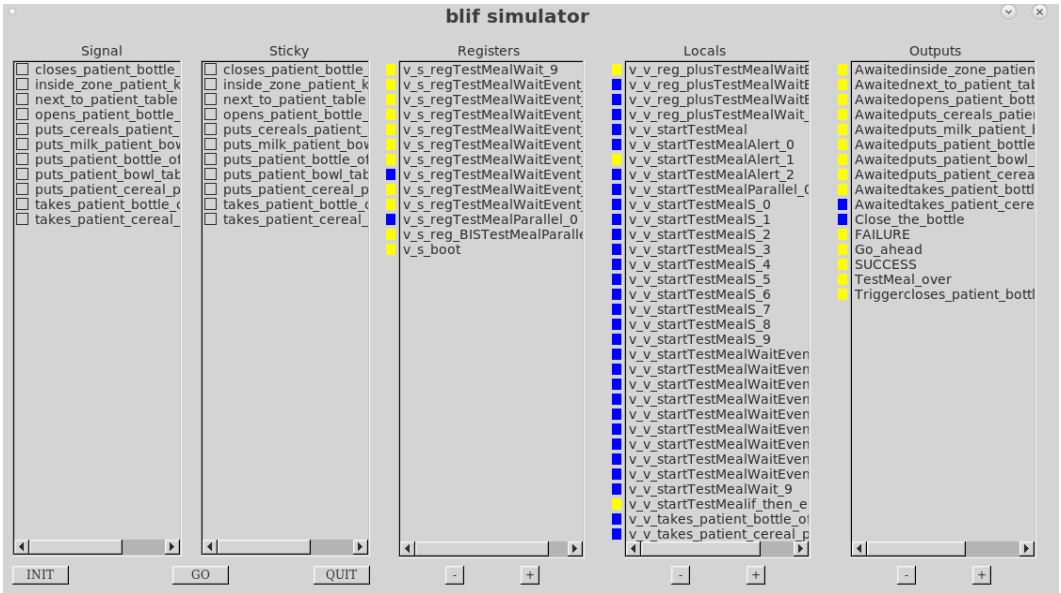


FIGURE 6.9 – Simulation du comportement de l'activité *TestMeal* (ici on voit que l'alerte *Close_the_bottle* a été déclenchée avec l'évènement attendu parce que l'évènement *close* n'a pas été sélectionné et on est passé à l'instant suivant)

Evènements envoyés à l'automate à l'instant t	**Evènements attendus pour l'instant t+1**	**Sorties générées**
Instant 0	inside_zone_patient_kitchen	Aucune
inside_zone(patient, kitchen)	next_to_patient_table	Aucune
next_to(patient, table)	put_patient_bowl_table	Aucune
put(patient, bowl,table)	take_patient_bottle_of_milk	Aucune
take(patient,bottle_of_milk)	open_patient_bottle_of_milk	Aucune
open(patient,bottle_of_milk)	put_milk_patient_bowl	Aucune
put_milk(patient,bowl)	close_patient_bottle_of_milk	Aucune
close(patient, bottle_of_milk	put_patient_bottle_of_milk_table	Go_ahead
put(patient, bottle_of_milk,table)	take_patient_cereal_packet	Close_the_bottle (en cas d'absence de l'évènement "close(patient, bottle_of_-milk")
take(patient,cereal_packet)	put_cereals_patient_bowl	Aucune
put_cereals(patient,bowl)	put_patient_cereal_packet_table	Aucune
put(patient,cereal_packet,table)		TestMeal_over

TABLEAU 6.2 – Tableau de simulation de l'activité *TestMeal* pour un scénario particulier

6.3.3 Validation

On veut vérifier dans ce cas d'utilisation la propriété suivante : on ne peut pas avoir l'évènement *close(patient,bottle_of_milk)* et l'alerte *Close_the_bottle* ensemble, sinon on génère une erreur. Le code de l'observateur qui décrit cette propriété est :

```
Activity observerTestMeal(patient:Person, table:Equipment,
                          bottle_of_milk:Equipment, kitchen:Zone)

Events
 inside_zone(Person, Zone);
 close(Person, Equipment);
 Close_the_bottle;
 TestMeal_over;

InitialState : inside_zone(patient, kitchen);
Start
 stop
 {
  wait (close(patient, bottle_of_milk) and close_the_bottle)
  seq
  emit Error
 }
 when TestMeal_over alert test_ok

End
```

Cet observateur sera appelé et exécuté en parallèle avec le programme à vérifier *TestMeal* dans un programme global appelé *TestMealVerif*. Le code de *TestMealVerif* est comme suit :

```
Activity TestMealVerif(patient: Person, table: Equipement, bowl: Equipment,
                       cereal_packet: Equipment, bottle_of_milk: Equipment,
                       kitchen: Zone)
Events
inside_zone(Person, Zone);
SubActivities
```

```
 TestMeal(Person,Equipment,Equipment,Equipment,Equipment,Zone);
 observerTestMeal(Person,Equipment,Equipment,Zone);

InitialState : inside_zone(patient, kitchen);
Start
 local TestMeal_over,Close_the_bottle
 {
  call TestMeal(patient,table,bowl,cereal_packet,bottle_of_milk,kitchen)
  parallel
  call observerTestMeal(patient,table,bottle_of_milk,kitchen)
 }
End
```

A partir de ce programme, on génère le code pour le model-checker NuSMV et on y intègre deux propriétés à vérifier en se basant sur notre observateur, pour faire la vérification en mode *batch*. Les propriétés sont :

```
CTLSPEC AG ! mTestMealVerif.Error;
CTLSPEC EF mTestMealVerif.test_ok;
```

Si ces deux propriétés sont vraies, ceci signifie que la propriété mentionnée au début de la section 6.3.3 à été vérifiée et que le programme est correct puisqu'il n'existe aucun chemin où on peut générer l'alerte *Error* et il existe au moins un chemin où le code s'exécute proprement. Les résultats de NuSMV sont montrés figure 6.10

```
[isarray@elacrab sam_10_2018]$ ./NuSMV TestMealVerif.smv
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:36:56 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

-- specification AG !mTestMealVerif.Error  is true
-- specification EF mTestMealVerif.test_ok  is true
[isarray@elacrab sam_10_2018]$
```

FIGURE 6.10 – Résultats de vérification des propriétés du programme TestMeal dans le model-checker NuSMV

6.4 Troisième cas d'utilisation

Les vidéos des activités dans nos ensembles de données (fournis par CoBTeK (Demc@re) ou Toyota) sont des activités simples et dans lesquelles on ne trouve qu'une seule personne. Nous avons fait nous même une autre vidéo où deux personnes se trouvent dans une salle de gymnastique et se rencontrent (voir figure 6.11). Le scénario de cette activité est comme suit :
La première personne (*firstPerson*) doit entrer dans la salle de gymnastique (*fitness_room*), elle se dirige vers la zone contenant un matelas (*zone_mattress*) et s'assoit sur une chaise (chair), pendant qu'elle est assise la deuxième personne (*secondPerson*)

FIGURE 6.11 – Vidéo de la troisième activité à reconnaître

entre dans la *zone_mattress* aussi. Puis la première personne se lève pour rencontrer la deuxième personne, les deux se serrent la main. Ensuite la première personne quitte la *zone_mattress*, la deuxième personne soit tombe et se relève dans un délais d'une minute (sinon on lance une alarme), puis quitte la salle de gymnastique, soit quitte la salle sans tomber.

6.4.1 Description de l'activité avec ADeL

.

Format textuel

Pour décrire cette activité, nous utilisons les mêmes types que les cas d'utilisations précédents : Person, Equipment et Zone. Dans cette activité, nous avons deux personnes (firstPerson et secondPerson), deux zones (fitness_room et zone_mattress) et deux équipements (chair et mattress). Pour décrire ce scénario d'activité avec ADeL, nous avons défini les évènements suivants :

```
Activity TestGym(firstPerson Person,secondPerson:Person,chair:Equipment,
                 mattress: Equipment,fitness_room:Zone,zone_mattress:Zone)

Events
 enters_zone(Person, Zone);
 goes_to_zone(Person, Zone);
 inside_zone(Person, Zone);
 sits(Person,Equipment);
 stands_up(Person);
 meets(Person, Person);
 shakes_hand(Person, Person);
 falls(Person,Equipment);
 goes_out_of_zone(Person,Zone);
```

Cette activité est simple, on n'utilise que des opérateurs de séquence et parallèle

```
InitialState : enters_zone(firstPerson, fitness_room);
Start
  wait goes_to_zone(firstPerson, zone_mattress)
 seq
  wait sits(firstPerson, chair)
 parallel
```

```
    wait goes_to_zone(secondPerson, zone_mattress)
 seq
  wait inside_zone(secondPerson,zone_mattress)
  parallel
  wait stands_up(firstPerson)
 seq
  wait meets(secondPerson, firstPerson)
 seq
  wait shakes_hand(secondPerson, firstPerson)
 seq
  wait goes_out_of_zone(firstPerson,zone_mattress)
 seq
  wait (falls(secondPerson,mattress)
        or goes_out_of_zone(secondPerson,fitness_room))
 seq
  if  falls(secondPerson,mattress)
  then
  {
   emit Fallen_secondPerson
    seq
    wait stands_up(secondPerson) timeout 1.0 min
    {
    wait goes_out_of_zone(secondPerson,fitness_room)
    } alert danger
   }
   else
   {
    emit Not_fallen
   }
    seq
  emit TestGym_over
End
```

Format graphique

La description de cette activité en format graphique est dans la figure 6.12.

6.4.2 Simulation

Les résultats de simulation d'un cas de cette activité dans *blif_simul* (voir figure 6.13) sont affichés dans le tableau 6.3. Dans ce cas la personne tombe sur le matelas.

Evènements envoyés à l'automate à l'instant t	**Evènements attendus à l'instant t+1**	**Sorties générées**
Instant 0	enters_zone(firstPerson, fitness_-room)	Aucune
enters_zone(firstPerson, fitness_-room)	goes_to_zone(firstPerson, zone_-mattress)	Aucune
goes_to_zone(firstPerson, zone_mattress)	sits(firstPerson, chair) goes_to_zone(secondPerson, zone_mattress)	Aucune
sits(firstPerson, chair) goes_to_zone(secondPerson, zone_mattress)	inside_zone(secondPerson, zone_mattress) stands_up(firstPerson)	Aucune
inside_zone(secondPerson,zone_-mattress) stands_up(firstPerson)	meets(secondPerson, firstPerson)	Aucune
meets(secondPerson, firstPerson)	shakes_hand(secondPerson, firstPerson)	Aucune
shakes_hand(secondPerson, firstPerson)	goes_out_of_-zone(firstPerson,zone_mattress)	Aucune
goes_out_of_-zone(firstPerson,zone_mattress)	falls(secondPerson, mattress)	Aucune
falls(secondPerson, mattress)	stands_up(secondPerson) Timeout_0	Fallen_secondPerson
stands_up(secondPerson)	goes_out_of_-zone(secondPerson,fitness_room)	Aucune
goes_out_of_-zone(secondPerson,fitness_room)		Not_fallen (en cas d'absence de l'évènement "falls(secondPerson, mattress)") testGym_over
Timeout_0		danger testGym_over

TABLEAU 6.3 – Tableau de simulation de l'activité TestGym pour un scénario particulier

Validation

Dans ce scénario d'activité, on peut par exemple définir une propriété qui génère *Error* comme erreur si elle reçoit l'évènement *fall(secondPerson,mattress)* et l'alerte *Not_fallen*. Le code de l'observateur est :

```
Activity observerTestGym(firstPerson:Person,secondPerson:Person,
                         mattress:Equipment,fitness_room:Zone)

Events
enters_zone(Person, Zone);
SubActivities
 falls(Person,Equipment);
 TestGym_over;
 Not_fallen;
InitialState : enters_zone(firstPerson, fitness_room);
Start
 stop
 {
   wait (falls(secondPerson,mattress) and Not_fallen)
  seq
   emit Error
 }when testGym_over alert test_ok
```

```
End
```

Le code du programme global est comme suit :

```
Activity TestGymVerif(firstPerson:Person,secondPerson:Person,
                      chair:Equipment,mattress: Equipment,
                      fitness_room:Zone,zone_mattress:Zone)
Events
enters_zone(Person, Zone);
SubActivities
 TestGym(Person,Person,Equipment,Equipment,Zone,Zone);
 observerTestGym(Person,Person,Equipment,Zone);
InitialState : enters_zone(firstPerson, fitness_room);
Start
 local TestGym_over,Not_fallen
 {
  call TestGym(firstPerson,secondPerson,chair,mattress,fitness_room,
     zone_mattress)
    parallel
    call observerTestGym(firstPerson,secondPerson,mattress,fitness_room)
 }
End
```

Les propriétés définies dans le format NuSMV à partir du code du programme global sont :

```
CTLSPEC AG ! mTestGymVerif.Error;
CTLSPEC EF mTestGymVerif.test_ok;
```

Si ces deux propriétés sont vraies, ceci signifie que le programme est correct vis à vis de ces dernières. Les résultats de NuSMV sont affichés dans la figure 6.14

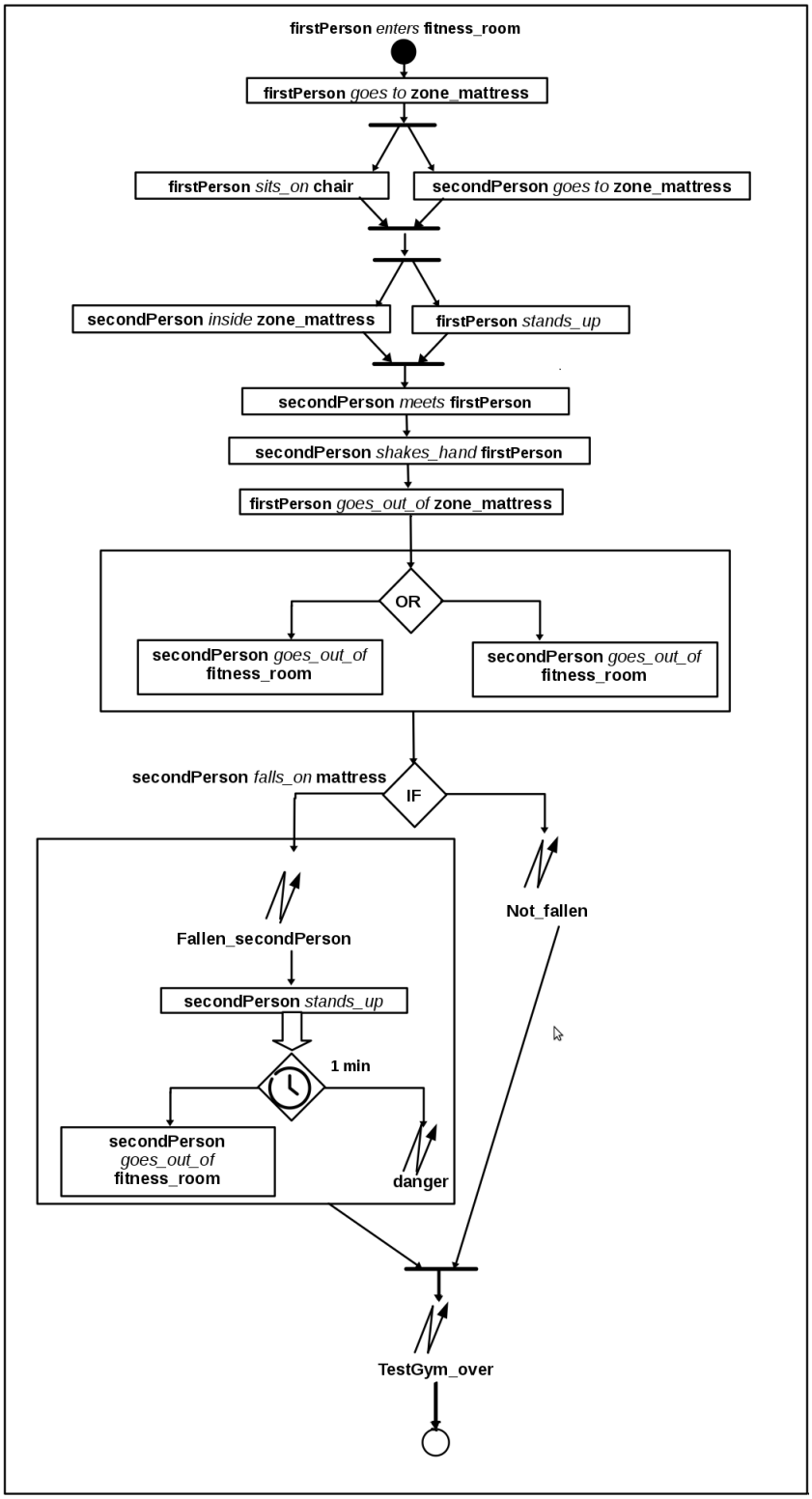


FIGURE 6.12 – Format graphique de la description de l'activité *TestGym*

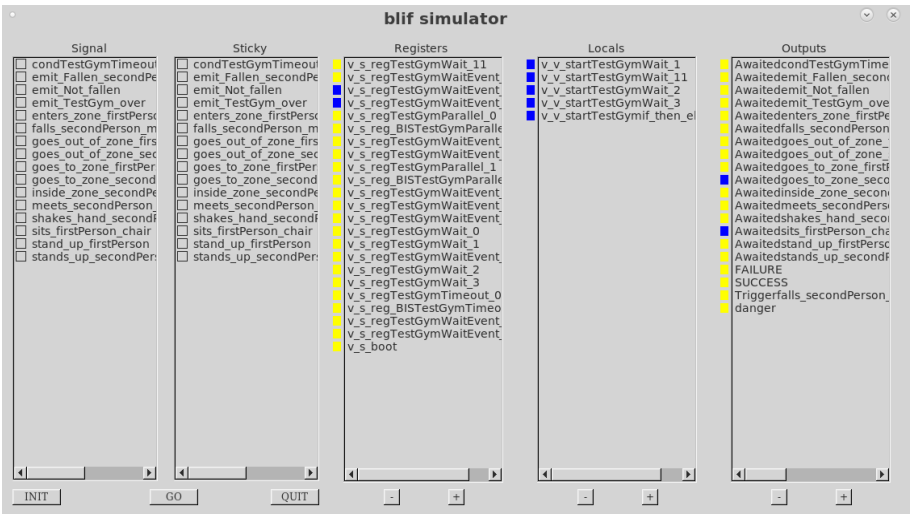


FIGURE 6.13 – Simulation du comportement de l'activité *TestGym* (instant2 dans le tableau 6.3)

```
[isarray@elacrab sam_10_2018]$ ./NuSMV TestGymVerif.smv
*** This is NuSMV 2.6.0 (compiled on Wed Oct 14 15:36:56 2015)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <Please report bugs to <nusmv-users@fbk.eu>>

*** Copyright (c) 2010-2014, Fondazione Bruno Kessler

*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://minisat.se/MiniSat.html
*** Copyright (c) 2003-2006, Niklas Een, Niklas Sorensson
*** Copyright (c) 2007-2010, Niklas Sorensson

-- specification AG !mTestGymVerif.Error  is true
-- specification EF mTestGymVerif.test_ok  is true
[isarray@elacrab sam_10_2018]$
```

FIGURE 6.14 – Résultats de vérification des propriétés du programme TestGym dans le model-checker NuSMV

6.5 Conclusion

Ces trois cas d'utilisation montrent que l'on peut vérifier des propriétés de programmes ADeL. Les simulations des activités ont montré que le comportement des programmes ADeL correspond à la description souhaitée. Les preuves avec NuSMV nous ont permis de faire la vérification et la validation des modèles d'activités spécifiés. Le format généré pour notre moteur de reconnaissance a été validé par ses créateurs, il ne reste que son intégration qui est en cours, ce qui nous permettra de faire des tests temps-réel complets.

Comme on a pu le constater dans ce chapitre, l'utilisation directe de NuSMV nécessite une bonne compréhension de la logique temporelle que nos utilisateurs n'ont pas en général. Il faudrait leur fournir un moyen plus facile d'exprimer des propriétés ou générer automatiquement de manière transparente des propriétés intéressantes à vérifier. De même, l'utilisation du simulateur pourrait être facilitée par une interface plus conviviale.

Chapitre 7

Conclusion et perspectives

Sommaire

7.1 Conclusion

Les systèmes de reconnaissance d'activités sont développés dans le cadre de la surveillance des comportements humains dans des domaines importants et critiques comme la santé ou la sécurité. Ils sont même utilisés aujourd'hui pour d'autres domaines comme le sport et le divertissement. Nous nous sommes intéressés plus particulièrement dans cette thèse au domaine médical, plus précisément à la reconnaissance d'activités de personnes âgées ou atteintes d'Alzheimer, en collaboration avec les médecins de l'Institut Claude Pompidou de Nice.

Aujourd'hui, la reconnaissance d'activités simples comme "se lever" ou "marcher" est facile pour la plupart des systèmes existants, mais la reconnaissance d'activités complexes et de longue durée reste toujours un challenge. Nous proposons dans cette thèse une approche générique pour concevoir des moteurs de reconnaissance capables de reconnaitre ce genre d'activités complexes et applicables dans une large variété de domaines d'applications.

7.1.1 Contributions

Dans cette thèse, nous avons choisi d'utiliser des modèles d'activités de haut-niveau intégrés dans un système de reconnaissance. Nous considérons les systèmes de reconnaissance d'activités comme des systèmes temps-réels réactifs, qui doivent être corrects et complets et qui nécessitent une haute fiabilité. Nous avons voulu tester la faisabilité de l'utilisation de l'approche synchrone pour ce problème. Cette approche assure deux propriétés qui sont le déterminisme et le parallélisme ; ces propriétés nous ont paru importantes car elles permettent de créer des systèmes sûrs, composés ou parallèles, en évitant plusieurs problèmes comme les courses critiques. D'autre part, cette approche s'appuie sur des bases formelles qui offrent un moyen pertinent pour la validation des systèmes de reconnaissance, d'autant que les simulations sont limitées dans le cas de systèmes complexes. Nous avons aussi adopté une approche modulaire pour pouvoir concevoir des modèles d'activités complexes en les divisant en plusieurs sous-modèles.

Langage synchrone de description d'activité destiné aux non-informaticiens : ADeL

Pour permettre aux utilisateurs non informaticiens de décrire leurs propres modèles d'activités, nous avons défini un langage (synchrone). Après une étude de langages existants comme Esterel, Lustre, Scade, Syncharts... nous avons décidé de créer notre propre langage synchrone qui est, selon les résultats d'un sondage (103 personnes), bien accepté par les non-informaticiens. Nous l'avons appelé ADeL (Activity Description Language) et nous avons défini ses concepts tout en insistant sur la possibilité du traitement du temps de la montre comme les autres évènements, et sur la nécessité d'un noyau minimal et complet à partir duquel tout utilisateur peut décrire une activité. Nous avons doté ce langage de deux formats proposés en collaboration avec les ergonomes de LudoTIC. Un format graphique permet de décrire les modèles d'activités à partir de bibliothèques d'acteurs, d'équipements, de zones et d'évènements. Ces bibliothèques sont prédéfinies par un administrateur selon les domaines d'utilisation et selon les informations qui peuvent provenir des capteurs. L'utilisateur peut décrire ses activités en plaçant les évènements sur une ligne de temps. Mais ce format peut

s'avérer inadapté pour décrire et modifier (ou maintenir) des activités très complexes, c'est pourquoi nous avons aussi proposé un format textuel.

Sémantiques formelles et compilation efficace

Pour créer des systèmes de reconnaissance fiables à partir de modèles d'activités sûrs, nous avons donné des bases formelles à notre langage. Nous avons doté ADeL d'une sémantique comportementale qui décrit les comportement de chaque opérateur à l'aide de règles de réécriture. Cette sémantique n'est pas efficace pour la compilation ni pour les preuves. Nous avons donc aussi défini une sémantique opérationnelle qui traduit un programme ADeL en un système d'équations booléennes, qui facilite sa compilation. Nous avons prouvé la relation entre ces deux sémantiques. Le compilateur ADeL peut facilement traduire ce système d'équations de la sémantique opérationnelle en un code efficace. Ce compilateur est capable de générer du code pour plusieurs cibles à partir du programme ADeL : un format pour la simulation (que nous avons intégré dans un simulateur), un format pour la vérification et la validation (intégrable dans un model-checker) et un format pour le moteur de reconnaissance de notre système.

Transformateur asynchrone/synchrone : un synchroniseur générique et souple

L'approche synchrone pose certains problèmes. L'un des plus importants est d'intégrer un système synchrone dans le monde asynchrone. Pour résoudre ce problème, nous avons créé un transformateur synchrone/asynchrone qui facilite la communication entre nos composants synchrones et les capteurs de l'environnement asynchrone. Nous avons appelé ce transformateur "synchroniseur". Nous avons voulu qu'il soit générique pour lui permettre d'être utilisable dans n'importe quel environnement : contrairement aux approches existantes, nous avons défini un synchroniseur paramétrable par différentes heuristiques (stratégies et tactiques). Les stratégies sont responsables de la construction de l'instant logique et les tactiques combinent les données provenant des capteurs pour former des évènements logiques. Nous avons montré (section 5.6 du chapitre 5) qu'un instant logique est différent d'une stratégie à une autre et d'une tactique à une autre. Ces heuristiques garantissent non seulement la généricité du synchroniseur mais aussi sa souplesse, en permettant à chaque administrateur de faire son choix et sa propre combinaison d'heuristiques pour créer une stratégie et des tactiques adaptées à son domaine. Notre synchroniseur est extensible, il permet le rajout de nouvelles heuristiques facilement (pour un administrateur informaticien).

7.1.2 Synthèse et limitations

Concernant le langage ADeL et le synchroniseur, les expérimentations et tests que nous avons effectués tout au long de notre travail et les cas d'utilisation que nous avons décrits dans ce mémoire ont validé qu'ADeL est bien adapté à la description d'activités, que sa compilation est efficace et que le synchroniseur fonctionne comme souhaité.

Les principaux problèmes rencontrés ont été : la création d'un noyau minimal d'opérateurs permettant la description de tout type d'activités, la définition de comportement de certains opérateurs de gestion du temps tout en respectant les règles du synchrone et la génération des évènements attendus.

Ces résultats sont certes satisfaisants et encourageants mais ils présentent aussi certaines limites qui nécessitent des améliorations mentionnées dans les perspectives

ci-après.

Trois limitations principales sont apparues dans ce travail :

Limitations du langage ADeL Le langage ADeL est capable de modéliser la plupart des activités que nous visons, surtout avec le mode textuel qui permet de décrire des activités complexes. Cependant, ce format n'est pas encore très "naturel", "intuitif" ,ni "élégant" pour ses utilisateurs, et certains peuvent éprouver des difficultés à l'utiliser et a en comprendre le comportement. Il est nécessaire d'utiliser des outils supplémentaires pour les aider à comprendre vraiment le fonctionnement de leurs automates, comme l'outil de simulation qui leur permet de vérifier le comportement d'un programme pas à pas. Ces difficultés sont réduites avec le format graphique mais ce dernier n'est pas adapté à décrire des activités d'une complexité aussi grande que le format textuel.

Limitations de la validation L'approche que nous avons utilisée pour faire des preuves est efficace, mais elle n'est pas adaptée à un utilisateur non-informaticien. Actuellement, la tâche de validation du langage est encore difficile pour ces utilisateurs : c'est nous qui générons le format compatible avec le model-checker (NuSMV), et nous l'introduisons dans ce dernier en utilisant des commandes spécifiques. Créer un système complet facile d'utilisation par un non informaticien reste un problème difficile à résoudre.

Limitations du synchroniseur Le synchroniseur actuel assure un fonctionnement correct du système de reconnaissance en garantissant la satisfaction des évènements attendus de l'automate généré à partir du programme ADeL et utilisé par le moteur de reconnaissance. Cependant, ses stratégies restent limitées par la nécessité de respecter cette condition. D'autre part, il peut causer de l'indéterminisme dans la composition des instants (les systèmes synchrones sont toujours déterministes) dans certains cas (présence de préemption). Nous sommes obligés d'accepter ce cas d'indéterminisme dans certains cas dangereux et d'urgence pour éviter des conséquences qui peuvent être graves, surtout sur le plan humain.

7.2 Perspectives

Le travail réalisé dans cette thèse et les limitations ouvrent de nombreuses perspectives pour des travaux futurs, à moyen et long terme.

7.2.1 Perspectives à moyen terme

Amélioration du langage ADeL

En collaboration avec des ergonomes, notre première perspective est d'améliorer et de simplifier le langage ADeL. Nous aimerions le rendre plus proche d'un langage naturel grâce à du sucre syntaxique et à l'introduction de "patterns" correspondants à des constructions récurrentes dans la description d'activités (comme "à chaque fois que" ou "tant que" l'évènement ne se produit pas,...). Nous travaillerons avec LudoTIC pour améliorer aussi l'interface graphique dans le cas d'activités complexes.

Amélioration du synchroniseur

Notre synchroniseur peut être amélioré en définissant plus de stratégies et de tactiques, pour qu'il soit utilisable dans un nombre plus grand de cas. Nous pouvons envisager de le doter lui aussi d'une sémantique pour permettre de prouver certaines propriétés de fonctionnement.

Validation et amélioration des tests

Notre système a été pour le moment testé à l'aide de plusieurs modèles d'activités, mais nous souhaitons faire des tests plus poussés sur plus de modèles plus compliqués et plus grands. Nous souhaitons aussi essayer de tester notre système dans des domaines autres que la médecine : par exemple, dans les banques ou dans les gares pour vérifier s'il fonctionne avec un grand nombre de personnes (acteurs) dans une scène.

7.2.2 Perspectives à long terme

Incertitude

Des informations fausses ou erronées provenant des capteurs peuvent engendrer des erreurs dans la reconnaissance d'activités. Une des perspectives à long terme porte sur la qualité des informations entrantes. En effet, la robustesse d'un système de reconnaissance est fortement liée à sa capacité de refuser les informations entrantes fausses. C'est pourquoi nous proposons d'améliorer la confiance apportée à un événement élémentaire. Une possibilité serait de tenir compte de l'incertitude [GSM17] physique des capteurs pour leur donner un degré de confiance et d'introduire cette notion dans notre système. Pour ce faire, il faut intégrer des calculs probabilistes au niveau du synchroniseur (tactiques) qui tiendront compte de cette incertitude. Des chercheurs de notre équipe ont commencé à travailler sur l'incertitude, en introduisant des probabilités de transition dans les modèles.

Intégrer l'Internet des objets

L'Internet des Objets est un domaine innovant et très intéressant. On peut intégrer dans notre travail quelques aspects comme la gestion de l'apparition et la disparition des objets de l'environnement. En effet, comme cela a été mentionné, c'est à l'administrateur de l'interface graphique de prendre en considération de nouveaux capteurs dans l'environnement et de rajouter les évènements qui leur sont liés dans les bibliothèques de l'interface graphique. Avec l'Internet des Objets, nous souhaitons réaliser ces opérations de façon automatique. De plus, un utiliseur ne doit pas sélectionner un évènement dans l'interface graphique qui correspond à un capteurs disparu ou en panne. On doit aussi traiter ces cas en n'affichant que la liste des évènements correspondants aux capteurs existants, l'Internet des Objets pourrait faciliter cette tâche.

Intégrer la charge mentale

La charge mentale [Cai07] est une des données importantes qui peut informer sur l'état psychique d'un patient. La charge mentale pourrait être définie par l'ensemble des sollicitations du cerveau pendant l'exécution d'une tâche. Elle diffère d'une personne à une autre, selon l'âge, l'état et la situation dans laquelle la personne est. Par exemple,

la charge mentale d'une personne âgée qui tombe peut être inférieure à celle d'une personne Alzheimer qui tombe. Cela pourrait avoir d'autres conséquences, par exemple, une personne Alzheimer tombe, elle est stressée, sa charge mentale augmente et elle peut devenir agressive. La charge mentale peut se mesurer à l'aide d'un ensemble de capteurs comme un "Eye tracker", des capteurs physiologiques (rythme cardiaque), etc. Le calcul de la charge mentale dans la reconnaissance d'activités peut aider à mieux cerner la situation et à agir de la meilleure façon (on parle ici de la génération de la sortie adéquate, exemple : envoyer une alarme au médecin, ou juste envoyer un signal pour lancer de la musique qui peut calmer le patient). L'une de nos perspective est de calculer et d'intégrer cette information dans le mécanisme de reconnaissance d'activités.

Améliorer l'ergonomie de la validation du langage pour les utilisateurs

Nous souhaitons rendre tâche de validation plus facile et transparente pour l'utilisateur, en rajoutant une interface pour la validation qui permettra aux utilisateurs de choisir les propriétés à vérifier et de faire la validation en un clic de bouton. Les exemples du chapitre 6 montrent que l'on a une certaine forme de généricité dans la formulation des propriétés de logique temporelle et nous pourrions ainsi générer ces formules automatiquement et intégrer la méthode de validation par observateur dans notre système.

Intégration d'ontologies

Les ontologies peuvent aider à décrire des activités d'une façon plus naturelle, en plus d'un analyseur syntaxique, en préparant un champ lexical des mots qui peuvent être utilisés et faisant la liaison avec nos concepts de base, ceci permettra à l'utilisateur de décrire son activité plus librement. Cependant il faut créer une ontologie pour chaque domaine d'utilisation. Il faut aussi vérifier si on peut lui rajouter des bases formelles avec des sémantiques.

7.3 Epilogue

Le travail de cette thèse a permis d'obtenir des logiciels opérationnels qui sont en cours d'intégration dans notre système de reconnaissance d'activités complet. Nous les avons validés sur des exemples du domaine médical à partir de vidéos, mais nous sommes confiants que leur généricité leur permet de s'appliquer dans d'autres domaines et avec d'autres capteurs. Notre système a des bases formelles, il permet de faire des preuves, ce qui assure son bon fonctionnement, il est adaptable à différents domaines d'applications autres que la médecine. La syntaxe générique du langage permet de décrire la plupart des activités et la souplesse du synchroniseur grâce aux stratégies de traitement d'instant et aux tactiques de traitement de données de l'environnement permettent de faciliter la reconnaissance de ces dernières.

Enfin l'approche synchrone s'est avérée efficace pour produire un système fiable et sûr, grâce aux bases formelles (sémantique) et aux méthodes de vérification formelles qu'elle offre. Cependant, elle est encore difficile à appréhender par les utilisateurs non-informaticiens à cause de certaines subtilités à respecter comme l'instantanéité de certains tests ou exécutions, la compréhension de la composition des instants, etc. Toutefois, il manque encore des outils pour rendre le synchrone plus accessible aux utilisateurs non informaticiens.

Bibliographie

[AB00] C. André and H. Boufaied. Execution machine for synchronous languages. In *IDPT'2000 (Integrated Design and Process Technology*, pages 144–149, Dallas (TX), June 2000. SDPS, (TX),colin.

[ABD98] C. André, H. Boufaïed, and S. Dissoubray. SyncCharts : un modèle graphique synchrone pour système réactifs complexes. In *Real-Time Systems (RTS'98)*, pages 175–196, Paris, France, January 1998. Teknea.

[ABG01] C. André, F. Boulanger, and A. Girault. Software implementation of synchronous programs. In *Proceedings of the 2nd International Conference on Application of Concurrency to System Design (ICACSD2001)*, pages 133–142, New Castle upon Tyne, UK, June 2001. IEEE Computer Society.

[ACM+08] M. Albanese, R. Chellappa, V. Moscato, A. Picariello, V. S. Subrahmanian, P. Turaga, and O. Udrea. A constrained probabilistic petri net framework for human activity detection in video. *IEEE Transactions on Multimedia*, 10(6) :982–996, Oct 2008.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[AK15] M. Awad and R. Khanna. *Machine Learning*, pages 1–18. Apress, Berkeley, CA, 2015.

[All83] J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11) :832–843, November 1983.

[AMP91] C. André, J-P. Marmorat, and J-P. Paris. An execution machine for ESTEREL. In *ECC'91*, volume 2, pages 1672–1677, Grenoble, France, July 1991. Hermès.

[And04] C. André. Computing SyncCharts reactions. *Electron. Notes Theor. Comput. Sci.*, 88 :3–19, October 2004.

[AR11] J.K. Aggarwal and M.S. Ryoo. Human activity analysis : A review. *ACM Comput. Surv.*, 43(3) :16 :1–16 :43, April 2011.

[AS80] Nicholas J. Aquilano and Dwight E. Smith. A formal set of algorithms for project scheduling with critical path scheduling/material requirements planning. *Journal of Operations Management*, 1(2) :57 – 67, 1980.

[AY99] R. Alur and M. Yannakakis. *Model Checking of Message Sequence Charts*, pages 114–129. Berlin, Heidelberg, 1999.

[BBSB12] J. Badie, S. Bak, S-T. Serban, and F. Bremond. Recovering people tracking errors using enhanced covariance-based signatures. In *Fourteenth IEEE International Workshop on Performance Evaluation of Tracking and Surveillance - 2012*, pages 487–493, Beijing, China, July 2012.

[BCP+85] Jean-Louis Bergerand, Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and Eric Pilaud. Outline of a real time data flow language. In *Proceedings of the 6th IEEE Real-Time Systems Symposium (RTSS '85), December 3-6, 1985, San Diego, California, USA*, pages 33–42, 1985.

[BDB16] P. Bilinski, A. Dantcheva, and F. Brémond. Can a smile reveal your gender ? In *15th International Conference of the Biometrics Special Interest Group (BIOSIG 2016)*, Darmstadt, Germany, September 2016.

[BDK+04] Matthias Brill, Werner Damm, Jochen Klose, Bernd Westphal, and Hartmut Wittke. *Live Sequence Charts*, pages 374–399. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos : a model-checking tool for real-time systems. In Moshe Y. Hu, Alan J. ; Vardi, editor, *Computer Aided Verification 10th International Conference, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–549, Vancouver, BC, Canada, June 1998. Springer.

[Bel77] Nuel D. Belnap. *A Useful Four-Valued Logic*, pages 5–37. Springer Netherlands, Dordrecht, 1977.

[Ber93] Gérard Berry. The semantics of pure esterel. *Series F : Computer and System Sciences*, 118, 01 1993.

[Ber96] G. Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, available at : http ://www.esterel-technologies.com 1996.

[Ber00] G. Berry. The Foundations of Esterel. In G. Plotkin, C. Stearling, and M. Tofte, editors, *Proof, Language, and Interaction, Essays in Honor of Robin Milner*. MIT Press, 2000.

[BGJ91] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations : the signal language and its semantics. *Science of computer programming*, 16(2) :103–149, 1991.

[BHS05] Y. Bontemps, P. Heymans, and P. . Schobbens. From live sequence charts to state machines and back : a guided tour. *IEEE Transactions on Software Engineering*, 31(12) :999–1014, Dec 2005.

[BJHP14] F. Boulanger, C. Jacquet, C. Hardebolle, and I. Prodan. Tesl : a language for reconciling heterogeneous execution traces. In *Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on*, pages 114–123, Lausanne, Switzerland, Oct 2014.

[BLPA98] M. Bayart, E. Lemaire, M-A Péraldi, and C André. External model and syncchartsdescription of an automobile cruise control. 31 :135–140, 03 1998.

[BMR83] G. Berry, S. Moisan, and J-P. Rigault. ESTEREL : Towards a Synchronous and Semantically Sound High Level Language for Real-Time Applications. In *IEEE Third International Real Time System Symposium*, Washington, USA, December 1983.

[BMS97] D. Pigozzi B. Mobasher and G. Slutzki. Multi-valued logic programming semantics : An algebraic approach. *Theoretical Computer Science*, 171 (1-2) :77–109, 1997.

[Bou93] F. Boulanger. *Intégration de modules synchrones dans la programmation par objets*. PhD thesis, Université Paris 11 Orsay, France, DEC 1993.

[Bou98] H. Boufaied. *Machines d'exécution pour langages synchrones*. PhD thesis, Université de Nice-Sophia Antipolis, November 1998.

[BRV04] Bernard Berthomieu, P.-O Ribet, and F Vernadat. The tool tina – construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research - INT J PROD RES*, 42 :2741–2756, 07 2004.

[Cai07] B. Cain. A review of the mental workload literature. page 35, 07 2007.

[CB04] T. Colombi and T. Baccino. Exploration visuelle et navigation dans les hypertextes : quelles stratégies. 01 2004.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2 : an OpenSource Tool for Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceeeding CAV*, number 2404 in LNCS, pages 359–364, Copenhagen, Danmark, July 2002. Springer-Verlag.

[CCM+03] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to scade/lustre to tta : A layered approach for distributed embedded applications. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, LCTES '03, pages 153–162, New York, NY, USA, 2003. ACM.

[Cha85] D. M. Chapiro. *Globally-asynchronous Locally-synchronous Systems (Performance, Reliability, Digital)*. PhD thesis, Stanford, CA, USA, 1985. AAI8506166.

[CN09] L. Chen and C. Nugent. Ontology-based activity recognition in intelligent pervasive environments. *International Journal of Web Information Systems*, 5(4) :410–430, 2009.

[CNW12] L. Chen, C. D. Nugent, and H. Wang. A knowledge-driven approach to activity recognition in smart homes. *IEEE Transactions on Knowledge and Data Engineering*, 24 :961–974, 2012.

[Cor18] R. Corazzon. *Theory and History of Ontology*. last update, 2018.

[CPP17] J. Colaço, B. Pagano, and M. Pouzet. Scade 6 : A formal language for embedded critical software development (invited paper). In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–11, Sept 2017.

[CV95] C. Cortes and V. Vapnik. Support-vector networks. In *Machine Learning*, pages 273–297, 1995.

[DA92] R. David and H. Alla. *Du Grafcet aux réseaux de Petri*. Série automatique. Hermès, 1992.

[DBB+16] A. Dantcheva, P. Bilinski, J. C. Broutart, P. Robert, and F. Bremond. Emotion facial recognition by the means of automatic video analysis. *Gerontechnology Journal*, September 2016.

[DH99] Werner Damm and David Harel. Lscs : Breathing life into message sequence charts. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 451–, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[DHC12] H. Dan, R. M. Hierons, and S. Counsell. A framework for pathologies of Message Sequence Charts. *Inf. Softw. Technol.*, 54(11) :1283–1295, nov 2012.

[DPBB07] S.A. Edwards D. Potop-Butucaru and G. Berry. *Compiling Esterel*. Springer, 2007.

[DTS+19] S. DAS, M. Thonnat, K. Sakhalkar, M. F Koperski, F. Bremond, and G. Francesca. A New Hybrid Architecture for Human Activity Recognition from RGB-D videos. In *25th International Conference on MultiMedia Modeling*, Thessaloniki, Greece, January 2019.

[FML15] Z. Feng, L. Mo, and M. Li. A random forest-based ensemble method for activity recognition. In *2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 5074–5077, Aug 2015.

[FO14] B. Finkbeiner and E. R. Olderog. Petri games : Synthesis of distributed systems with causal memory. In *Proceedings of Fifth International Symposium on Games, Automata, Logics and Formal Verification (GandALF 2014)*, January 2014. pub_id : 655 Bibtex : FiOl_14 :Petri URL date : None.

[Fon61] J.W. Fondahl. Non-computer approach to the critical path method for the construction industry. Technical Report 9, Stanford University, 1961.

[Gaf91] D. Gaffé. Conception d'un exécutif temps-réel pour esterel. Technical report, Ecole Doctorale SPI – DEA Informatique, Signaux et Systèmes, Septembre 1991. Rapport de stage de DEA.

[Gaf96] D. Gaffé. *Le modèle* GRAFCET *: réflexion et intégration dans une plate-forme multiformalisme synchrone*. PhD thesis, Université de Nice-Sophia Antipolis, Janvier 1996.

[GBA16] M. Gonza, L. Bitjoka, and H. Alla. Séparation des états du graphe de marquages d'un réseau de Petri pour la commande par supervision des systèmes à événements discrets. working paper or preprint, 2016.

[GBBG86] Paul Le Guernic, Albert Benveniste, Patricia Bournai, and Thierry Gautier. Signal-a data flow-oriented language for signal processing. *IEEE Trans. Acoustics, Speech, and Signal Processing*, 34(2) :362–374, 1986.

[GBC16] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[GDAB05] Karen Godary-Dejean and Isabelle Augé-Blum. Abstractions de modèles en automates temporisés pour la validation temporelle d'architectures embarquées. *Journal Européen des Systèmes Automatisés (JESA)*, 39(1/3) :63–78, 2005.

[Gen00] D. Gendreau. *Les sept facettes du grafcet*. Cépaduès Editions, 2000.

[GGH+07] T. Gazagnaire, B. Genest, L. Hélouët, P. S. Thiagarajan, S. Yang, and V. T. Vasconcelos. *Causal Message Sequence Charts*, pages 166–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[GGR93] J. Grabowski, P. Graubmann, and E. Rudolph. The standardization of message sequence charts. In *Proceedings 1993 Software Engineering Standards Symposium*, pages 48–63, Aug 1993.

[Gin88] M. Ginsberg. Multivalued logics : A uniform approach to inference in artificial intelligence. *Computational Intelligence*, 4 :265–316, 1988.

[GLMS11] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2010 : A toolbox for the construction and analysis of distributed processes. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 372–387, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[GMP04] Blaise Genest, Anca Muscholl, and Doron Peled. *Message Sequence Charts*, pages 537–558. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[GR08] D. Gaffé and A. Ressouche. The clem toolkit. In *Proceedings of 23rd IEEE/Aberry-bookCM International Conference on Automated Software Engineering(ASE 2008)*, L'aquila, Italy, September 2008.

[GR12] D. Gaffé and A. Ressouche. Algebras and Synchronous Language Semantics. Rapport de recherche RR-8138, INRIA, November 2012.

[GR13] D. Gaffé and A. Ressouche. Algebraic Framework for Synchronous Language Semantics. In Laviana Ferariu and Alina Patelli, editors, *2013 Symposium on Theoretical Aspects of Sofware Engineering*, pages 51–58, Birmingham, UK, July 2013. IEEE Computer Society.

[Gra16] Daniel Graupe. *Deep Learning Neural Networks : Design and Case Studies*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2016.

[GSM17] A. Gupta, S. Sarkar, and N. Mukherjee. On uncertainty determination in ehealth sensors. In *2017 IEEE Sensors Applications Symposium (SAS)*, pages 1–6, March 2017.

[GV04] Alain Griffault and Aymeric Vincent. The mec 5 model-checker. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification*, pages 488–491, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[GWVW08] J. Gao, M. Whalen, and E. Van Wyk. Extending lustre with timeout automata. *Electron. Notes Theor. Comput. Sci.*, 203(4) :111–124, June 2008.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic, 1993.

[Har87a] D. Harel. Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231 – 274, 1987.

[Har87b] D. Harel. Statecharts : A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3) :231–274, June 1987.

[Has95] M. H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, Cambridge, MA, USA, 1st edition, 1995.

[HB06] M. Hasan and F. Boris. SVM : Machines à Vecteurs de Support ou Séparateurs à Vastes Marges. Polycopié du cours BD Web, ISTY3, Université de Versailles St Quentin, 2006.

[HCHP18] C. Hu, Y. Chen, L. Hu, and X. Peng. A novel random forests based class incremental learning method for activity recognition. *Pattern Recognition*, 78 :277 – 290, 2018.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[HH03] M. Heath and I. Harris. A deterministic globally asynchronous locally synchronous microprocessor architecture. In *Proceedings. 4th International Workshop on Microprocessor Test and Verification - Common Challenges and Solutions*, pages 119–124, May 2003.

[HK01] David Harel and Hillel Kugler. Synthesizing state-based object systems from lsc specifications. In *Revised Papers from the 5th International Conference on Implementation and Application of Automata*, CIAA '00, pages 1–33, Berlin, Heidelberg, 2001. Springer-Verlag.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[HNT03] J. Hammarberg and S. Nadjm-Tehrani. Development of safety-critical reconfigurable hardware with esterel. 80 :219–234, 08 2003.

[HP72] Peter Hitchcock and David Michael Ritchie Park. Induction rules and termination proofs. In *ICALP*, 1972.

[IEC13] IEC, Genève (CH). *Grafcet specification language for sequential function charts*, 2002, revisited 2003, 2013. International standard IEC 60848.

[ITU] ITU-T TELECOMMUNICATION (03/93) STANDARDIZATION SECTOR OF ITU. *MESSAGE SEQUENCE CHART (MSC)*.

[JGP00] E. M. Clarke Jr., O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[JM99] L. C. Jain and L. R. Medsker. *Recurrent Neural Networks : Design and Applications*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1999.

[KD16] M. H. Kolekar and D. P. Dash. Hidden markov model based human activity recognition using shape and optical flow based features. In *2016 IEEE Region 10 Conference (TENCON)*, pages 393–397, Nov 2016.

[KHC10] E. Kim, S. Helal, and D. Cook. Human activity recognition and pattern discovery. *IEEE Pervasive Computing*, 9(1) :48–53, January 2010.

[KJD+15] A. König, C. F. Crispim Junior, A. Derreumaux, G. Bensadoun, P. D. Petit, F. Bremond, R. David, Frans R. J. Verhey, P. Aalten, and P. H. Robert. Validation of an automatic video monitoring system for the detection of instrumental activities of daily living in dementia patients. *Journal of Alzheimer's disease : JAD*, 44 2 :675–85, 2015.

[KJS18] B. Kamiński, M. Jakubczyk, and P. Szufel. A framework for sensitivity analysis of decision trees. *Central European Journal of Operations Research*, 26(1) :135–159, Mar 2018.

[KMB09] R. Kumar, E. G. Mercer, and A. Bunker. Improving translation of Live Sequence Charts to temporal logic. *Electron. Notes Theor. Comput. Sci.*, 250(1) :137–152, September 2009.

[LB98] Y. LeCun and Y. Bengio. The handbook of brain theory and neural networks. chapter Convolutional Networks for Images, Speech, and Time Series, pages 255–258. MIT Press, Cambridge, MA, USA, 1998.

[Lep94] P. Leparc. *Apports de la méthodologie synchrone pour la définition et l'utilisation du langage Grafcet.* PhD thesis, Université de Rennes 1, Janvier 1994.

[LGS16] L. Li, H. Gao, and T. Shan. An executable model and testing for Web software based on Live Sequence Charts. In *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pages 1–6, June 2016.

[LGTLL03] P. Le Guernic, J-P Talpin, and J-C Le Lann. Polychrony for system design. Research Report RR-4715, INRIA, 2003.

[L'H97] DOMINIQUE L'HER. *Modelisation du grafcet temporise et verification de proprietes temporelles.* PhD thesis, 1997. Thèse de doctorat dirigée par Marcé, Lionel Informatique Rennes 1 1997.

[Liu00] J. W. S. Liu. *Real-Time Systems.* Prentice Hall, Upper Saddle River, NJ, 2000.

[LLD07] F. Latfi, B. Lefebvre, and C. Descheneaux. Le rôle de l'ontologie de la tâche dans un habitat intelligent en télé-santé. 10 2007.

[Lou14] G. Louppe. *Understanding Random Forests : From Theory to Practice.* PhD thesis, 10 2014.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Int. J. Softw. Tools Technol. Transf.*, 1(1-2) :134–152, December 1997.

[LRR13] G. Lavee, M. Rudzsky, and E. Rivlin. Propagating certainty in petri nets for activity recognition. *IEEE Transactions on Circuits and Systems for Video Technology*, 23(2) :326–337, Feb 2013.

[Mar92] Florence Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In W.R. Cleaveland, editor, *CONCUR '92*, pages 550–564, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[Mea55] G. H. Mealy. A method for synthesizing sequential circuits. *Bell Sys. Tech. Journal*, 34 :1045–1080, September 1955.

[Mer74] Philip Meir Merlin. *A Study of the Recoverability of Computing Systems.* PhD thesis, 1974. AAI7511026.

[MG17] H. Alla M. Gonza, L. Bitjoka. Séparation des états du graphe de marquages d'un réseau de petri pour la commande par supervision des systèmes à événements discrets, 2017.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer, 1980.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3) :267 – 310, 1983.

[MMZ18] R. Mouhcine, A. Mustapha, and M. Zouhir. Recognition of cursive arabic handwritten text using embedded training based on hmms. *Journal of Electrical Systems and Information Technology*, 5(2) :245 – 251, 2018.

[MP95] K. L. McMillan and D. K. Probst. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1) :45–65, Jan 1995.

[MR01] F. Maraninchi and Y. Rémond. Argos : an automaton-based synchronous language. *Computer Languages*, 27(1) :61 – 92, 2001. Visual Formal Methods-VFM'99 Symposium.

[MR03] F. Maraninchi and Y. Rémond. Mode-automata : A new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46(3) :219–254, March 2003.

[Mur89] T. Murata. Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, 77(4) :541–580, April 1989.

[NAPS11] A. Navada, A. N. Ansari, S. Patil, and B. A. Sonkamble. Overview of use of decision tree algorithms in machine learning. In *2011 IEEE Control and System Graduate Research Colloquium*, pages 37–42, June 2011.

[NEM09] R. Nisbet, J. Elder, and G. Miner. *Handbook of Statistical Analysis and Data Mining Applications*. Academic Press, Inc., Orlando, FL, USA, 2009.

[NFP17] U. M. Nunes, D. R. Faria, and P. Peixoto. A human activity recognition framework using max-min features and key poses with differential evolution random forests classifier. *Pattern Recognition Letters*, 99 :21 – 31, 2017. User Profiling and Behavior Adaptation for Human-Robot Interaction.

[NHR92] F. Lagnier N. Halbwachs and C. Ratel. Programming and verifying critical systems by means of the synchronous data-flow programming language lustre. In *Special Issue on the Specification and Analysis of Real-Time Systems*. IEEE Transactions on Software Engineering, 1992.

[NKCN15] T. Nguyen, A. Khosravi, D. Creighton, and S. Nahavandi. Hidden markov models for cancer classification using gene expression profiles. *Information Sciences*, 316 :293 – 307, 2015. Nature-Inspired Algorithms for Large Scale Global Optimization.

[Nor97] J. R. Norris. *Markov Chains*. Cambridge University Press, Cambridge, UK, 1997.

[OCW14] G. Okeyo, L. Chen, and H. Wang. Combining ontological and temporal formalisms for composite activity modelling and recognition in smart homes. *Future Generation Computer Systems*, 39 :29 – 43, 2014. Special Issue on Ubiquitous Computing and Future Communication Systems.

[OSPI16] L. Onofri, P. Soda, M. Pechenizkiy, and G. Iannello. A survey on using domain and contextual knowledge for human activity recognition in video streams. *Expert Systems with Applications*, 63 :97 – 111, 2016.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical report, DAIMI FN-19, Aarhus University, Denmark, 1981.

[PRF11] J. Provost, J-M. Roussel, and J-M. Faure. A formal semantics for Grafcet specifications. In *7th IEEE Conference on Automation Science and Engineering (IEEE CASE 2011)*, pages 488–494, Trieste, Italy, August 2011.

[Pro08] S. Prochnow. *Efficient development of complex statecharts*. PhD thesis, University of Kiel, Germany, 2008.

[PT17] M. K. PHAN TRAN. *Maintaining the engagement of older adults with dementia while interacting with serious game*. Theses, Université Côte d'Azur, Apr 2017.

[RBD03] M. Richardson, J. Bilmes, and C. Diorio. Hidden-articulator markov models for speech recognition. *Speech Communication*, 41(2) :511 – 529, 2003.

[RG11] A. Ressouche and D. Gaffé. Compilation modulaire d'un langage synchrone. *Revue des sciences et technologies de l'information, série Théorie et Science Informatique*, 4(30) :441–471, June 2011.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch, editors. *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[Rus12] J. Rushby. The versatile synchronous observer. In *Proceedings of the 15th Brazilian Conference on Formal Methods : Foundations and Applications*, SBMF'12, pages 1–1, Berlin, Heidelberg, 2012. Springer-Verlag.

[SB16] K. Schneider and J. Brandt. Quartz : A synchronous language for model-based design of reactive embedded systems. pages 1–30, 01 2016.

[SBC+15] B. Selic, C. Bock, S. Cook, T. Rivett, P.and Rutt, E. Seidewitz, and D. Tolbert. *OMG Unified Modeling Language (Version 2.5)*. 03 2015.

[Sch09] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time Object-oriented Modeling*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[SM15] S. Shaily and V. Mangat. The hidden markov model and its application to human activity recognition. In *2015 2nd International Conference on Recent Advances in Engineering Computational Sciences (RAECS)*, pages 1–4, Dec 2015.

[SP97] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *Trans. Sig. Proc.*, 45(11) :2673–2681, November 1997.

[SRM+17a] I. Sarray, A. Ressouche, S. Moisan, J-P. Rigault, and D. Gaffé. An Activity Description Language for Activity Recognition. In *IINTEC 2017 - IEEE International Conference on Internet of Things, Embedded Systems and Communications*, page 6, Gafsa, Tunisia, October 2017.

[SRM+17b] I. Sarray, A. Ressouche, S. Moisan, J-P. Rigault, and D. Gaffé. Synchronous Automata For Activity Recognition. Research report, Inria Sophia Antipolis, April 2017.

[SRM+18] Ines Sarray, Annie Ressouche, Sabine Moisan, Jean-Paul Rigault, and Daniel Gaffé. Semantic Studies of a Synchronous Approach to Activity Recognition. In *International Conference on Software Engineering and Applications* , Dubaï, United Arab Emirates, January 2018.

[SRMG18] I. Sarray, J-P. Rigault, S. Moisan, and D. Gaffé. A synchronous approach to activity recognition. In *12th IEEE International Conference on Semantic Computing, ICSC 2018, Laguna Hills, CA, USA, January 31 - February 2, 2018*, pages 304–305, 2018.

[SSBD14] S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning : From Theory to Algorithms*. Cambridge University Press, New York, NY, USA, 2014.

[Sta88] J. A. Stankovic. Misconceptions about real-time computing : A serious problem for next-generation systems. *Computer*, 21(10) :10–19, October 1988.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2) :285–309, 1955.

[Tec] Esterel Technologies. Scade suite. http ://www.ansys.com/products/embedded-software/ansys-scade-suite.

[VA98] Jean Vareille and Mireille Arnoux. L'enseignement du langage Grafcet et ses interprétations. *Revue de l'EPI (Enseignement Public et Informatique)*, (90) :173–188, June 1998. Sommaire du numéro :
http ://archive-edutice.ccsd.cnrs.fr/edutice-00000878.

[VBT03] V-T. Vu, F. Bremond, and M. Thonnat. Automatic video interpretation : A novel algorithm for temporal scenario recognition. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 1295–1300, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[Zaf05] L. Zaffalon. *Programmation synchrone de systèmes réactifs avec Esterel et les SynChrats*. Presses Polytechniques et universitaires romandes, 2005.

[ZEV10] N. Zouba Ep Valentin. *Multisensor Fusion for Monitoring Elderly Activities at Home*. Theses, Université Nice Sophia Antipolis, January 2010.

Annexe A

Grammaire du langage ADeL

A.1 Grammaire BNF du langage ADeL

Les mots clés du langages sont indiqués en gras.

pnumber : := ”1” | ”2” | ”3” | ”4” | ”5” | ”6” | ”7” | ”8” | ”9”
zero : := ”0”
number : := zero | pnumber
natural : := zero | pnumber { number }
integer : := natural | ”-” pnumber {number}
real : := integer [”.” number {number}]
letter : := ”a” | ... | ”z”
cletter : := ”A” | ... | ”Z”
ident : := (letter | cletter){letter | cletter | ”0”..”9”}

activity_list : := program {program}
program : := **Activity** activity_name ”(”role” :”type_name
{”,”role” :”type_name} ”)”
[events_declaration]
[subActivities_declaration]
activity_body
activity_name : := ident
role : := ident
type_name : := ident

events_declaration : := **Events** event ”;” {event ”;”}
event : := ident [”(” ident_list ”)”]

ident_list : := ident {”,” ident}

subActivities_declaration : := **SubActivities** activity_name ”(”ident_list”)” ”;”
{activity_name ”(” ident_list ”)” ”;”}

activity_body : := initial_state_declaration [scene_declaration]

initial_state_declaration : := **initialState** ” :” event ”;”

scene_declaration : := **Start** {instruction} **End**

instruction : := nothing | wait | parallel | sequence | stop_when |pstop_when | if_then_else | while | local | call | emit | timeout |ptimeout

nothing : := **nothing**

wait : := [**wait**] quadrival_exp

parallel : := instruction **parallel** instruction

sequence : := instruction **seq** instruction

stop_when : := **stop** "{" instruction "}" **when** quadrival_exp [**alert** event]

pstop_when : := **Pstop** "{" instruction "}" **when** quadrival_exp [**alert** event]

if_then_else : := **if** quadrival_exp **then** instruction [**else** instruction]

while : := **while** quadrival_exp "{" instruction "}"

local : := **local** event {"," event} "{" instruction "}"

emit : := **emit** event

quadrival_exp : := quadrival_exp **or** quadrival_exp
| quadrival_exp **and** quadrival_exp
| **not** quadrival_exp
| event
| **"true"**
| **"false"**
| "(" quadrival_exp ")"

timeout : := instruction **timeout** duree_exp "{" instruction "}"
[**alert** event]

ptimeout : := instruction **Ptimeout** duree_exp "{" instruction "}"
[**alert** event]

duree_exp : := real units

units : := "msec" | "sec" | "min" | "hr" | "day" | "week"
call : := **call** activity_name [renaming]

renaming : := "[" event "/" event "]"

Annexe B

Preuve des théorèmes du chapitre 4

B.1 Preuve du théorème 1

Dans cette section, nous détaillons la preuve du théorème 1 de la section 4.3.2 du chapitre 4, établissant un isomorphisme entre les bitreillis $(\xi, \leq_B, \leq_K, \neg)$ et $\mathbb{B} \bigodot \mathbb{B}$.

1. $e(x \sqcup y) = e(x) \sqcup e(y)$:
   (a) $x = \bot$: $\forall y \in \xi, x \sqcup y = y$. Par ailleurs $e(x) = (ff, ff)$ ainsi $\forall z \in \mathbb{B} \times \mathbb{B}, e(x) \leq_K z$ et donc $e(x) \sqcup z = z$ et en particulier pour $e(y)$.
   (b) $x = 0$: nous distinguons 2 cas : (1) $y = \bot$ ou $y = 0$, $0 \sqcup y = 0$ et $e(0 \sqcup y) = (ff, tt)$, par ailleurs, $e(0) = (ff, tt)$ et $e(\bot) = (ff, ff)$ ainsi si $y = \bot$ ou $y = 0$, $e(0) \sqcup e(y) = (ff, tt)$; (2) $y = 1$ ou $y = \top$, $0 \sqcup y = \top$ et $e(0 \sqcup y) = (tt, tt)$, d'un autre coté, $e(1) = (tt, ff)$ et $e(\top) = (tt, tt)$ ainsi $e(0) \sqcup e(y) = (tt, tt)$.
   (c) $x = 1$ la preuve est semblable au cas précédent.
   (d) $x = \top$, $\forall y \in \xi, x \sqcup y = x$. D'autre part, $e(x) = (tt, tt)$ ainsi $\forall z \in \mathbb{B} \times \mathbb{B}, z \leq_K e(x)$ et en particulier pour $e(y)$ donc $e(x) \sqcup e(y) = e(x)$.
2. $e(x \sqcap y) = e(x) \sqcap e(y)$: si nous nous référons à la remarque ci-dessus, nous savons que $e(x) \sqcap e(y) = (x_h \cdot y_h, x_l \cdot y_l)$ (eg1).
   (a) $x = \bot$: $\bot$ est élément absorbant pour $\sqcap$, donc $x \sqcap y = \bot$ et $e(x \sqcap y) = (ff, ff)$. d'un autre coté, $e(x) = (ff, ff)$ donc en appliquant l'égalité (eg1), $e(x) \sqcap e(y) = (ff, ff)$.
   (b) $x = 0$: nous distinguons 2 cas : (1) si $y = 1$ ou $y = \bot$ alors $x \sqcap y = \bot$ et $e(x \sqcap y) = (ff, ff)$. Par ailleurs, $e(x) = (ff, tt)$ et $e(y) = (tt, tt)$ ou $e(y) = (ff, ff)$; toujours d'après (eg1) nous avons $e(x \sqcap y) = (ff, ff)$. (2) si $y = 0$ ou $y = \top$ alors $x \sqcap y = 0$ et $e(x \sqcap y) = (ff, tt)$. Par ailleurs, $e(x) = (ff, tt)$ et $e(y) = (ff, tt)$ ou $e(y) = (tt, tt)$; toujours d'après (eg1) nous avons $e(x \sqcap y) = (ff, tt)$.
   (c) $x = 1$: nous distinguons aussi 2 cas : (1) si $y = 1$ ou $y = \top$ alors $x \sqcap y = 1$ et $e(x \sqcap y) = (tt, ff)$. d'un autre coté, $e(x) = (tt, tt)$ et $e(y) = (tt, ff)$ ou $e(y) = (tt, tt)$; toujours d'après (eg1) nous avons $e(x \sqcap y) = (tt, ff)$. (2) si $y = 0$ ou $y = \bot$ alors $x \sqcap y = 0$ et $e(x \sqcap y) = (ff, tt)$. d'un autre coté, $e(x) = (tt, ff)$ et $e(y) = (ff, tt)$ ou $e(y) = (ff, ff)$; toujours d'après (eg1) nous avons $e(x \sqcap y) = (tt, ff)$.
   (d) $x = \top$: x étant élément neutre pour $\sqcap$, $e(x \sqcap y) = e(y)$. Par ailleurs, $e(x) = (tt, tt)$, l'égalité (eg1) $\implies$ $e(x \sqcap y) = e(y)$.

3. $e(x \boxplus y) = e(x) \boxplus e(y)$:

   (a) $x = \bot$: nous distinguons 2 cas : (1) $y = \bot$ or $y = 0$, $\bot \boxplus y = \bot$ ainsi $e(\bot \boxplus y) = (ff, ff)$, d'un autre coté, $e(\bot) = (ff, ff)$ et $e(0) = (ff, tt)$, on en déduit que $e(y) \leq_B e(\bot)$; donc $e(\bot) \boxplus e(y) = e(\bot) = (ff, ff)$.
   (2) $y = 1$ ou $y = \top$, $x \boxplus y = 1$ et $e(x \boxplus y) = (tt, ff)$. D'autre part, $e(1) = (tt, ff)$ et $e(\top) = (tt, tt)$, or $e(\bot) \boxplus e(1) = (tt, ff)$ et $e(\bot) \boxplus e(\top) = (tt, ff)$, si l'on se réfère au tableau ci dessus.

   (b) $x = 0$: $0 \boxplus y = y$, d'autre part dans $\mathbb{B} \odot \mathbb{B}$, nous avons : $(ff, tt) \leq_B (ff, tt) \leq_B (tt, ff)$ et $(ff, tt) \leq_B (tt, tt) \leq_B (tt, ff)$ ainsi $e(0) \boxplus e(y) = e(y)$.

   (c) $x = 1$: $1 \boxplus y = 1$ dans ξ. D'autre part, $(1, 0)$ est le plus grand élément de $\mathbb{B} \odot \mathbb{B}$, pour l'ordre $\leq_B$; donc $(tt, ff) \boxplus z = (tt, ff)$, ainsi l'égalité est vérifiée.

   (d) $x = \top$: la preuve est symétrique au cas (a).

4. $e(x \boxdot y) = e(x) \boxdot e(y)$: toujours en se référant à l'égalité ci dessus prouvée dans [BMS97] pour les bitreillis, nous savons que $e(x) \boxdot e(y) = (x_h \cdot y_h, x_l + y_l)$ (eg2).

   (a) $x = \bot$: nous distinguons 2 cas : (1) si $y = 1$ ou $y = \bot$, $x \boxdot y = \bot$, ainsi $e(x \boxdot y) = (ff, ff)$. d'un autre coté, $e(x) = (ff, ff)$ et $e(y) = (tt, ff)$ ou $e(y) = (ff, ff)$; donc conformément à l'égalité (eg2), $e(x) \boxdot e(y) = (ff, ff)$.
   (2) si $y = 0$ ou $y = \top$, $x \boxdot y = 0$, ainsi $e(x \boxdot y) = (ff, tt)$. d'un autre coté, $e(x) = (ff, ff)$ et $e(y) = (ff, tt)$ ou $e(y) = (tt, tt)$; donc conformément à l'égalité (eg2), $e(x) \boxdot e(y) = (ff, tt)$.

   (b) $x = 0$: 0 est absorbant pour $\boxdot$, donc $x \boxdot y = 0$ et $e(x \boxdot y) = (ff, tt)$. Par ailleurs, $e(x) = (ff, tt)$, quel que soit y, $e(y) = (y_h, y_l)$ et $ff \cdot y_y = ff$, $tt + y_l = tt$; ainsi $e(x) \boxdot e(y) = (ff, tt)$.

   (c) $x = 1$, 1 est élément neutre pour $\boxdot$ donc $x \boxdot y = y$ et $e(x \boxdot y) = e(y)$. Par ailleurs, $e(x) = (tt, ff)$ et nous avons $tt.y_h = y_h$ et $ff + y_l = y_l$, d'où $e(x) \boxdot e(y) = e(y)$.

   (d) $x = \top$: nous distinguons 2 cas : (1) si $y = 1$ ou $y = \top$, $x \boxdot y = \top$, ainsi $e(x \boxdot y) = (tt, tt)$. d'un autre coté, $e(x) = (tt, tt)$ et $e(y) = (tt, ff)$ ou $e(y) = (tt, tt)$; donc conformément à l'égalité (eg2), $e(x) \boxdot e(y) = (tt, tt)$.
   (2) si $y = 0$ ou $y = \bot$, $x \boxdot y = 0$, ainsi $e(x \boxdot y) = (ff, tt)$. d'un autre coté, $e(x) = (tt, tt)$ et $e(y) = (ff, tt)$ ou $e(y) = (ff, ff)$; donc conformément à l'égalité (eg2), $e(x) \boxdot e(y) = (ff, tt)$.

Finalement, $e(\neg x) = \neg(e(x))$: si $x = \bot$ ou $x = \top$ le résultat est immédiat car l'opérateur $\neg$ ne change pas ces valeurs. En effet, nous avons $e(x) = (x_h, x_l)$ avec $x_h = x_l$ donc $(x_h, x_l) = (x_l, x_h)$. $\neg 0 = 1$ ainsi $e(\neg 0) = (tt, ff) = \neg(ff, tt) = \neg(e(0))$. La preuve pour $x = 1$ est symétrique.

De plus, e est clairement une bijection. ■

Une première version de la preuve est dans [GR13]. La preuve ci-dessus la complète et en revoit quelques imprécisions .

B.2 Preuve du théorème 2

Nous rappelons l'énoncé du théorème (voir section 4.5.2) :

Théorème 2
Soit p une instruction ADeL et E un environnement d'entrée. Si $\langle p \rangle_E$ est l'environnement calculé par la sémantique opérationnelle tel qu'il ne contienne aucun évènement S avec le statut $\top$ ($\nexists S^{\top} \in \langle p \rangle_E$), alors la propriété suivante est vérifiée :

$\exists p'$ *tel que* $p \xrightarrow[E_C]{E', \text{FINISH}_{p_h}} p'$ *et* $\forall o \in E',\ o^1 \in \langle p \rangle_E$

Pour prouver ce théorème, nous introduisons la notion de taille d'une instruction. Intuitivement, cette notion correspond au nombre de nœuds de son arbre syntaxique. Nous utilisons cette taille pour faire une preuve par induction du théorème.

Définition 3 *La taille d'une instruction (notée [ ]) est définie comme suit :*

- $[\textbf{\textit{nothing}}] = 1$
- $[\textbf{\textit{wait}}\ S] = 1$
- $[\textbf{\textit{alert}}\ S] = 1$
- $[p_1 \parallel p_2] = max([p_1], [p_2]) + 1$
- $[p_1\ \textbf{\textit{seq}}\ p_2] = [p_1] + [p_2] + 1$
- $[\textbf{\textit{stop}}\ p\ \textbf{\textit{when}}\ S\ \textbf{\textit{alert}}(S_1)] = [p] + 1$
- $[\textbf{\textit{while}}\ cond\{p\}] = [p] + 1$
- $[\textbf{\textit{if}}\ cond\ \textbf{\textit{then}}\ p_1\ \textbf{\textit{else}}\ p_2] = max([p_1], [p_2]) + 1$
- $[p\ \textbf{\textit{timeout}}\ S\ \{p_1\}\textbf{\textit{alert}}\ S_1] = [p_1] + 1$

Soit P un programme ADeL et p l'instruction racine de son arbre syntaxique. Nous prouvons le théorème par induction sur la taille de p. Comme nous l'avons déjà précisé nous allons considérer la projection booléenne de la sémantique opérationnelle (finalisation (voir section 4.5.1)). Nous allons montrer que $\mathcal{F}(\mathcal{D}_p)$ calcule un environnement dans lequel $\mathcal{F}(\text{FINISH}_p)$ est égal au booléen de terminaison de la règle de la sémantique comportementale qui réécrit p et que tout événement de sortie présent dans une sémantique l'est aussi dans l'autre.

La sémantique opérationnelle construit l'automate implicite pour p. Dans cet automate, la première réaction est caractérisée par START = 1. En effet, la sémantique opérationnelle décrit l'évolution d'un programme quand il s'exécute. Dans la sémantique opérationnelle cela correspond à mettre l'événement START de son instruction racine à 1. Dans les opérateurs du langage, seul **if..then..else** peut mettre le START d'un de ses arguments à 0, les autres opérateurs transmettent la valeur de START à leurs arguments. Dans cette preuve, nous considérerons que START = 1, dans l'état initial, car s'il a été mis à 0, ce ne peut être que par une instruction englobante **if..then..else** et dans ce cas, aucune des deux sémantiques ne tient compte de la branche qui n'est pas prise et la terminaison et les environnements de sortie sont ceux de la branche choisie. Il en est de même pour l'événement KILL. Ce dernier est à 0 au début de l'évaluation, il est mis à 1 par les opérateurs de préemption afin d'arrêter l'exécution de leurs arguments en cas de préemption. Dans les autres opérateurs, il est juste transmis aux arguments. De plus pour les instructions ayant des registres, conformément à la représentation de Mealy, les combinaisons des valeurs de ceux-ci encodent les états de l'automate. Par convention, nous avons posé que l'état initial correspondait à la valuation qui met tous les registres à 0. Le système d'équations finalisé de p représente également un automate de Mealy mais avec des conditions booléennes sur les transitions. Nous avons vu dans la section 4.5.1 que la finalisation

consistait à poser $x_l = \overline{x_h}$ pour toute variable x du système dont l'encodage est (x_h, x_l). On élimine ainsi les équations définissant les secondes projections des variables et on peut remplacer les opérateurs de ξ par leurs correspondances dans $\mathbb{B}$.

Pour faire la démonstration, nous nous appuierons sur deux lemmes. Le premier permet de ne pas considérer le cas où KILL = 1 dans la démonstration du théorème 2.

B.2.1 Lemme 1

Lemme 1 *Pour toute instruction i, son évènement* KILL *ne peut pas être à* 1 *au premier instant (1) ; si* $\text{KILL}_i = 1$*, alors i est l'argument d'une instruction englobante ig et* $\text{FINISH}_{ig_h} = term_{ig}$ *(2).*

(1) Conformément au système d'équations associé à chaque opérateur, nous pouvons constater que les seuls opérateurs qui peuvent mettre l'évènement KILL de leur argument à 1 sont **stop**, **while** et **timeout**. Dans le cas d'un **stop**, l'équation qui définit le KILL de son argument p est :

$$\text{KILL}_p = (S \ \boxdot \ \neg\text{KILL} \ \boxdot \ \text{REG}) \boxplus \text{KILL} \quad (eq1)$$

Nous constatons que KILL_p est à 1 uniquement si au moins REG = 1, ce qui n'est pas le cas dans l'état initial puisque ce dernier est caractérisé par un registre à 0. Donc KILL_p ne peut pas être à 1 dans le premier instant.
Dans le cas d'un **while**, l'équation qui définit le KILL de son argument p est :

$$\text{KILL}_p = (\text{REG} \boxdot \ \neg cond) \ \boxplus \ \text{KILL}) \quad (eq2)$$

En conséquence, l'évènement KILL_p est à 1 si et seulement si REG = 1 et $cond = 0$
Dans le cas d'un **timeout**, l'équation qui définit le KILL de son argument p est :

$$\text{KILL}_p = (S \boxdot \neg\text{KILL} \boxdot \text{REG}_1 \boxdot \neg\text{REG}_2) \boxplus \text{KILL} \quad eq3-1$$

Cet opérateur a deux arguments, on a donc aussi :

$$\text{KILL}_{p_1} = (S \boxdot \neg\text{KILL} \boxdot \text{REG}_1 \boxdot \neg\text{REG}_2) \boxplus \text{KILL} \quad eq3-2$$

Là aussi, nous pouvons constater que $\text{KILL}_p = 1$ si $\text{REG}_1 = 1$ et si $\text{REG}_2 = 0$ Ce qui est faux dans l'état initial.

(2) D'après la remarque (1) ci-dessus, l'instruction englobante ig est soit un **abort**, soit un **while** ou un **timeout**. On a donc FINISH_{ig} défini par l'une des équations suivantes :

$\text{FINISH}_{ig} = \neg S \boxdot \text{FINISH}_p \boxdot \text{REG} \boxplus \text{FINISH}_p \boxdot \neg\text{REG} \boxplus \neg\text{FINISH}_p \boxdot S$ (stop-finish)

ou

$\text{FINISH}_{ig} = \neg cond$ (while-finish)

ou

$$\begin{aligned} \text{FINISH}_{ig} \ = \ & \text{REG}_1 \ \boxdot \ \neg\text{REG}_2 \ \boxdot \ S \boxplus \\ & \neg\text{REG}_1 \ \boxdot \ \text{REG}_2 \ \boxdot \ \text{FINISH}_{p_1} \boxplus \\ & \text{REG}_1 \ \boxdot \ \neg\text{REG}_2 \boxdot \text{FINISH}_p \boxdot \text{FINISH}_{p_1} \boxplus \\ & \neg\text{REG}_1 \ \boxdot \ \neg\text{REG}_2 \boxdot \text{START} \boxdot \text{FINISH}_p \boxdot \text{FINISH}_{p_1} \end{aligned} \quad \text{(timeout-finish)}$$

Dans le cas de **stop**, si $\text{KILL}_p = 1$, cela signifie que $S = 1$ (voir l'équation de stop-finish ci dessus) et donc $\text{FINISH}_{ig} = 1$. Par ailleurs, on ne peut pas être au premier instant car S n'est pas testé au premier instant. Cela signifie que dans la sémantique comportementale, on a déjà appliqué la règle *stop*2 et que l'on réécrit le

terme : **if** S **then emit** S_1 **else stop**(p', S, S_1) avec p' la dérivée de p dans la règle *stop*2. Comme S est vrai ($S = 1$), on a la réécriture :

$$\frac{\textbf{emit } S_1 \xrightarrow[E_C]{\{S_1\},\ tt} \textbf{nothing} \quad , \quad (S, E_C) \rightarrowtail tt}{\textbf{if } S \textbf{ then emit } S_1 \textbf{ else stop}(p,\ S,\ S_1) \xrightarrow[E_C]{\{S_1\},tt} \textbf{nothing}}$$

Donc $term_{ig} = tt = \text{FINISH}_{ig_h}$.

Dans le cas de **while**, conformément à l'équation while-finish, on peut déduire que $cond = 01$ et donc $\text{FINISH}_{ig} = 1$. Dans la sémantique comportementale, c'est la règle *while*2 qui s'applique et $term_{ig} = tt$.

Dans le cas de **timeout**, si $\text{KILL}_p = 1$ cela signifie que $S = 1$, $\text{REG}_1 = 1$ et $\text{REG}_2 = 0$. Dans ce cas $\text{KILL}_{p_1} = 1$ aussi. L'équation timeout-finish permet de déduire que $\text{FINISH}_{ig} = 1$. Comme pour l'opérateur **stop**, S n'est pas testé au premier instant ; cela signifie que dans la sémantique comportementale, la règle *timeout*1 a été appliquée dans l'instant précédent et que l'on réécrit le terme : **if** S **then emit** S_1 **else timeout**(p', S, p_1, S_1) avec p' la dérivée de p dans *timeout*1. Comme $S = 1$, on a la réécriture :

$$\frac{\textbf{emit } S_1 \xrightarrow[E_C]{\{S_1\},\ tt} \textbf{nothing} \quad , \quad (S, E_C) \rightarrowtail tt}{\textbf{if } S \textbf{ then emit } S_1 \textbf{ else timeout}(p,\ S,\ p_1,\ S_1) \xrightarrow[E_C]{\{S_1\},tt} \textbf{nothing}}$$

Donc $term_{ig} = tt = \text{FINISH}_{ig_h}$.

C'est la règle *timeout*2 qui s'applique dans la sémantique comportementale et $term_{ig} = tt$. Si $\text{KILL}_{p_1} = 1$, alors nous avons $\text{FINISH}_{p_1} = 1$, $\text{REG}_1 = 0$ et $\text{REG}_2 = 1$. L'équation $eq3 - 2$ permet de déduire que $\text{FINISH}_{ig} = 1$. Dans la sémantique comportementale, c'est la règle *timeout*1 qui s'applique et $term_{p_1} = tt$. On en déduit également que $term_{ig} = tt$.

Ce lemme montre que lorsque KILL est mis à 1, la terminaison de l'opérateur qui génère cette mise à 1 est à 1 dans la sémantique opérationnelle et à tt dans la sémantique comportementale. L'événement KILL est par défaut à 0, il sert uniquement à la sémantique opérationnelle pour désactiver les arguments des opérateurs qui ont une ξ expression préemptive. Ce lemme montre que si KILL a été mis à 1, les résultats dans la sémantique comportementale restent cohérents. Il permet ainsi de considérer KILL à 0 dans la preuve du théorème ; ce qui simplifie les équations à étudier. De plus, on ne regardera pas non plus les équations définissant les KILL des arguments des autres opérateurs qui ne font que transmettre la valeur de leur KILL.

B.2.2 Lemme 2

Le second lemme stipule que si un opérateur termine (son événement FINISH = 1) alors c'est que son événement START a été à 1 dans le premier instant.

Lemme 2 *Pour toute instruction i, si* $\text{START}_i \neq 1$ *alors* $\text{FINISH}_i \neq 1$.

Nous détaillons aussi la preuve de ce lemme en annexe B. La preuve est une induction sur la taille des instructions.

$[p] = \mathbf{1}$

— **nothing**
L'équation de l'opérateur donne le résultat immédiatement.

— **wait** S
REG $= 0$ dans la première réaction,et dans les réactions suivantes il est égal à 0 ou 1, suivant. le statut de S. Conformément à la définition de $\boxdot$, FINISH $= 0$ ou $\bot$.

— **emit** S
Immédiat à partir de l'équation.

$[p] = \mathbf{n > 1}$

— $p_1 \parallel p_2$
Si START $\neq 1$, alors $\text{START}_1 \neq 1$ et $\text{START}_2 \neq 1$. Par induction, nous savons que $\text{FINISH}_1 \neq 1$ et $\text{FINISH}_1 \neq 1$. REG_1 et REG_2 sont à 0 au premier instant par convention, dans les réactions suivantes, ils seront soit 0 soit $\bot$. L'équation du FINISH de l'opérateur montre que ce dernier est lui aussi soit 0 soit $\bot$.

— p_1 **seq** p_2
Le système d'équations nous permet de déduire le résultat immédiatement, en appliquant l'hypothèse d'induction.

— **stop** p **when** S **alert** S_1
Le registre REG de l'opérateur est 0 ou $\bot$ dans toutes les réactions et conformément à la définition des opérateurs $\boxdot$ et $\boxplus$, on en déduit que FINISH $\neq 1$.

— **while** $cond\{p\}$
Comme pour l'opérateur précédent, en appliquant l'hypothèse d'induction, les équations de l'opérateur montrent que son registre REG $\neq 1$ et par conséquence FINISH aussi.

— **if** $cond$ **then** p_1 **else** p_2
Si START $\neq 1$ il en est de même pour START_{p_1} et START_{p_2} (les START respectifs de p_1 et p_2). Donc en appliquant l'hypothèse d'induction, le résultat est immédiat.

— **local**(S) {p}
Le résultat découle immédiatement du système d'équations en appliquant l'hypothèse d'induction.

— p **timeout** S $\{p_1\}$ **alert** S_1
La valeur des registres dépend des FINISH respectifs des arguments et de START. Ils ne sont donc jamais égaux à 1. L'équation de FINISH nous permet de déduire qu'il n'est lui non plus jamais égal à 1.

■

B.2.3 Preuve du théorème 2

Maintenant, nous commençons la démonstration du théorème 2, en étudiant le premier cas de l'induction. Dans cette démonstration, l'environnement de départ de l'opérateur considéré est E et celui des règles de la sémantique comportementale qui correspondent aux divers cas que l'on traite est $E_{\mathcal{C}}$.

Instructions de taille 1

Tout d'abord nous considérons les instructions de taille 1 ($[p] = 1$), i.e. **nothing**, **wait** et **emit**. Le but est de montrer que dans chaque réaction, la sémantique opérationnelle "finalisée" et la sémantique comportementale calculent la même valeur de terminaison et les mêmes valeurs pour les évènements de sortie.

nothing
Pour cet opérateur, la finalisation nous donne :

$$\mathcal{F}(\mathcal{D}_{nothing}) = \left[\begin{array}{lcl} \text{FINISH}_h & = & \text{START}_h \end{array} \right]$$

$\text{FINISH}_h = \text{START}_h = tt$ car nous considérons que START vaut 1 dans l'état initial. Par ailleurs, la règle de sémantique comportementale pour cet opérateur est

$$\textbf{nothing} \xrightarrow[\emptyset]{\emptyset,\ tt} \textbf{nothing}$$

D'une part, nous avons $term_{nothing} = tt = \text{FINISH}_h$. Par ailleurs, l'environnement de sortie de la sémantique comportementale ne contient aucun événement. Ainsi la propriété est vérifiée pour **nothing**.

wait S
Dans la finalisation du système de l'opérateur, nous n'avons pas tenu compte de l'événement KILL car dans le système d'équations de l'opérateur il intervient uniquement avec l'opérateur $\boxdot$ et avec $\neg$KILL. Grâce au lemme 1 nous pouvons supposer qu'il est à 0.

$$\mathcal{F}(\mathcal{D}_{wait}) = \left[\begin{array}{lcl} \text{REG}_h^+ & = & \text{START}_h + \text{REG}_h.\overline{S_h} \\ \text{FINISH}_h & = & S.\text{REG} \end{array} \right]$$

Considérons la première réaction, où $\text{START}_h = tt$ et $\text{REG}_h = ff$. En appliquant les équations de cet opérateur on en déduit que $\text{FINISH}_h = ff$. Dans la sémantique comportementale, c'est la règle *wait* qui s'applique et donc $term_p = ff$.
Si l'on considère l'environnement de sortie résultant de l'application de la règle *wait*, dans la sémantique comportementale, il est vide.

Si l'on considère les réactions suivantes, il est clair que $\text{REG}_h^+ = tt$ dans la première réaction et donc $\text{REG}_h = tt$. La preuve dépend du statut de S :

1. $S_h = tt$
   $\text{REG}_h = tt$ et donc $\text{FINISH}_h = tt$. Dans la sémantique comportementale, la règle $iwait1$ s'applique en conséquence de la règle *wait* qui caractérise la première réaction ; ainsi $term_p = tt = finish_h$. En fait, la règle de la sémantique comportementale qui s'applique est :

   $$\frac{S \in E_{\mathcal{C}}}{\textbf{iwait S} \xrightarrow[E_c]{\emptyset,tt} \textbf{nothing}}$$

   S est l'unique événement de la sorte de l'opérateur, si $S_h = tt$ alors $S^1 \in E$ (on a écarté le cas où un événement a un statut $\top$) et donc $S \in E_{\mathcal{C}}$. L'environnement de sortie de la sémantique comportementale est vide, donc la propriété est vérifiée.

2. $S_h = ff$:
   $\text{REG}_h = 1$ puisque nous ne sommes plus dans la première réaction. Donc $\text{FINISH}_h = ff$. Dans la sémantique comportementale, c'est la règle $iwait2$ qui s'applique et nous avons $term_{iwait} = ff = \text{FINISH}_h$. Comme dans l'item 1, l'environnement de sortie dans la règle $iwait2$ est vide.

Ainsi, la propriété est vérifiée pour l'opérateur **wait**.

emit S

$$\mathcal{F}(\mathcal{D}_{emit}) = \begin{bmatrix} \text{FINISH}_h & = & \text{START}_h \\ S_h & = & \text{FINISH}_h \end{bmatrix}$$

Comme **nothing**, cet opérateur est instantané et nous avons : $\text{FINISH}_h = \text{START}_h = tt$. La règle de la sémantique comportementale qui correspond est :

$$\textbf{emit S} \xrightarrow[E_C]{\{S\},\ tt} \textbf{nothing}$$

Ainsi, nous avons $term_{emit} = tt$. L'environnement de sortie de la sémantique comportementale est $\{S\}$. Par ailleurs, dans le système d'équations finalisé, nous avons $S_h = \text{FINISH}_h = tt$. Donc $S_h^{tt} \in \langle \textbf{emit} \rangle_{\mathcal{F}(E)}$, grâce à la propriété 1, nous savons que $S_h^{tt} \in \mathcal{F}(\langle \textbf{emit} \rangle_E)$. Par définition de la finalisation appliquée aux environnements quadri valués, on a donc $S^x \in \langle \textbf{emit} \rangle_E$ et $\mathcal{F}(x) = tt$. x est donc soit 1 soit $\top$ (voir le tableau de la section 4.5.1). Par hypothèse nous avons écarté le cas où un événement a $\top$ pour statut. Donc $S^1 \in \langle \textbf{emit S} \rangle_E$ (S^1 signifie que S est présent).

B.2.4 Instructions de taille n

Dans le cas où les opérateurs sont de taille $n > 1$ ($[p] = n$), nous ne présentons ici que deux des opérateurs du langage : **parallel** car c'est l'opérateur principal des langages synchrones et **timeout** car c'est un opérateur spécifique à ADeL. La démonstration pour les autres opérateurs du langage est détaillée en annexe B dans la section **??**.

p_1 **seq** p_2
Son système finalisé est :

$$\mathcal{F}(\mathcal{D}_{seq}) = \begin{bmatrix} \text{START}_{p_{1h}} & = & \text{START}_h & \\ \text{START}_{p_{2h}} & = & \text{FINISH}_{p_{1h}} & (1) \\ \text{FINISH}_h & = & \text{FINISH}_{p_{2h}} & (2) \end{bmatrix}$$

L'exécution de la séquence dépend du statut de $\text{FINISH}_{p_{1h}}$. Nous distinguons donc deux cas possibles :

1. $\text{FINISH}_{p_{1h}} = ff$:

   Nous avons $\text{START}_{p_{2h}} = ff$ et donc $\text{FINISH}_h = \text{FINISH}_{p_{2h}} = ff$ grâce au lemme 2. Dans la sémantique comportementale, c'est la règle $seq1$ qui s'applique :

$$\frac{p_1 \xrightarrow[E_C]{E_1,\ ff} p_1'}{p_1 \textbf{ seq } p_2 \xrightarrow[E_C]{E_1,\ ff} p_1' \textbf{ seq } p_2}$$

Le booléen de terminaison est ff.

L'environnement $\langle p_1 \textbf{ seq } p_1\rangle_E$ est calculé en appliquant $\mathcal{D}_{seq}$ à $\langle p_2\rangle_{\langle p_1\rangle_E}$. L'environnement de sortie de la sémantique comportementale est E_1, obtenu lors de la réécriture de la règle $seq1$ à partir de $E_{\mathcal{C}}$. Par récurrence nous savons que pour toute sortie o dans E_1, $o^1 \in \langle p_1\rangle_E$ (c'est-à-dire que o est à 1 dans $\langle p_1\rangle_E$). $\langle p_2\rangle_{\langle p_1\rangle_E}$ est obtenu en appliquant les équations de p_2 à $\langle p_1\rangle_E$. Les équations des opérateurs font grossir l'information sur les statuts des événements selon l'ordre de connaissance. Comme $o^1 \in \langle p_1\rangle_E$, en appliquant les équations de p_2, le statut de o ne peut que rester à 1 ou grossir vers $\top$; ce qui est écarté par hypothèse. Donc $o^1 \in \langle p_2\rangle_{\langle p_1\rangle_E}$. Les équations de $\mathcal{D}_{seq}$ ne changent pas la valeur des sorties, ainsi $o^1 \in \langle p_1 \textbf{ seq } p_1\rangle_E$.

2. $\text{FINISH}_{p_{1_h}} = tt$:
Dans la sémantique opérationnelle, $\text{FINISH}_h = \text{FINISH}_{p_{2_h}}$. Dans la sémantique comportementale, c'est la règle $seq2$ qui s'applique :

$$\frac{p_1 \xrightarrow[E_{\mathcal{C}}]{E_1,\ tt} \textbf{nothing} \quad , \quad p_2 \xrightarrow[E_{\mathcal{C}}]{E_2, term_{p_2}} p_2'}{p_1 \textbf{ seq } p_2 \xrightarrow[E_{\mathcal{C}}]{E_1 \cup E_2, term_{p_2}} p_2'}$$

Le booléen de terminaison de l'opérateur est $term_{p_2}$ et par induction nous avons : $term_{p_2} = \text{FINISH}_{p_{2_h}} = \text{FINISH}_h$.

La sémantique comportementale a pour environnement de sortie $E_1 \cup E_2$, l'union des environnements de sortie respectifs de p_1 et de p_2 calculés à partir de $E_{\mathcal{C}}$ (voir la règle ci dessus). Considérons une sortie o dans $E_1 \cup E_2$. Si $o \in E_1$, par induction nous savons que $o^1 \in \langle p_1\rangle_E$ et aussi $o^1 \in \langle p_2\rangle_{\langle p_1\rangle_E}$ et $o^1 \in \langle p_1 \textbf{ seq } p_1\rangle_E$ (même raisonnement que pour $\text{FINISH}_{p_{1_h}} = ff$ étudié ci dessus). Si $o \notin E_1$ et $o \in E_2$, par induction nous savons que $o^1 \in \langle p_2\rangle_E$. Soit $o^1 \in E$ et $o^1 \in \langle p_2\rangle_{\langle p_1\rangle_E}$ et donc $o^1 \in \langle p_1 \textbf{ seq } p_1\rangle_E$. Soit $o^\perp \in E$; dans ce cas, soit le statut de o a été mis à 1 dans $\langle p_1\rangle_E$, soit il est resté à $\perp$. Dans les deux cas, on a $o^1 \in \langle p_2\rangle_{\langle p_1\rangle_E}$ et comme les équations de $\mathcal{D}_{seq}$ ne changent pas les statuts des sorties, on a bien $o^1 \in \langle p_1 \textbf{ seq } p_1\rangle_E$.

Parallel : $p_1 \parallel p_2$
Comme pour **wait**, nous avons ignorer l'événement KILL, car nous pouvons le considérer à 0 grâce au lemme 1.

$$\mathcal{F}(\mathcal{D}_{\parallel}) = \left[\begin{array}{lcl}
\text{REG}^+_{1_h} & = & \text{REG}_{1_h}.\overline{\text{FINISH}_{p_{2_h}}} \ + \ \overline{\text{REG}_{2_h}}.\text{FINISH}_{p_{1_h}}.\overline{\text{FINISH}_{p_{2_h}}} \\
\text{REG}^+_{2_h} & = & \text{REG}_{2_h}.\overline{\text{FINISH}_{p_{1_h}}} \ + \ \overline{\text{REG}_{1_h}} \ .\overline{\text{FINISH}_{p_{1_h}}} \ .\text{FINISH}_{p_{2_h}} \\
\text{START}_{p_{1_h}} & = & \text{START}_h \\
\text{START}_{p_{2_h}} & = & \text{START}_h \\
\text{FINISH}_h & = & \text{REG}_{1_h} \ .\overline{\text{REG}_{2_h}}.\text{FINISH}_{p_{2_h}} \ + \text{REG}_{2_h}.\overline{\text{REG}_{1_h}}.\text{FINISH}_{p_{1_h}} \ + \\
 & & \overline{\text{REG}_{1_h}}.\overline{\text{REG}_{2_h}}.\text{FINISH}_{p_{1_h}}.\text{FINISH}_{p_{2_h}}
\end{array}\right]$$

L'état initial est caractérisé par $\text{REG}_{1_h} = ff$ et $\text{REG}_{2_h} = ff$. Dans cet état, ce sont les valeurs respectives de $\text{FINISH}_{p_{1_h}}$ et de $\text{FINISH}_{p_{2_h}}$ qui pilotent les nouvelles valeurs des registres et déterminent celle de FINISH_h, conformément au système d'équations de l'opérateur. Il y a 4 cas :

1. $\text{FINISH}_{p_{1_h}} = ff$ et $\text{FINISH}_{p_{2_h}} = ff$:
   $\text{REG}^+_{1_h}$ et $\text{REG}^+_{2_h}$ valent tous les deux ff ; donc, dans ce cas, on ne change pas d'état. Nous calculons $\text{FINISH}_h = ff$. Dans la sémantique comportementale, la règle *parallel* devient :

$$\frac{p_1 \xrightarrow[E_c]{E_1, ff} p'1 \quad , \quad p_2 \xrightarrow[E_c]{E_2, ff} p'_2}{p_1 \| p_2 \xrightarrow[E_c]{E_1 \cup E_2,\ ff} p'1 \| p'_2}$$

   Nous avons bien $term_{\|} = ff$.

2. $\text{FINISH}_{p_{1_h}} = tt$ et $\text{FINISH}_{p_{2_h}} = ff$:
   $\text{REG}^+_{1_h} = tt$ et $\text{REG}^+_{2_h} = ff$. Dans ce cas on change d'état, l'état futur sera (tt, ff) et $\text{FINISH}_h = ff$. Voici la règle de la sémantique comportementale appliquée dans ce cas :

$$\frac{p_1 \xrightarrow[E_c]{E_1, tt} \textbf{nothing} \quad , \quad p_2 \xrightarrow[E_c]{E_2, ff} p'_2}{p_1 \| p_2 \xrightarrow[E_c]{E_1 \cup E_2,\ ff} \textbf{nothing} \| p'_2}$$

   Par induction, nous savons que $term_{p_1} = \text{FINISH}_{p_{1_h}} = tt$ et que $term_{p_2} = \text{FINISH}_{p_{2_h}}$. En conséquence, nous avons bien $term_{\|} = ff$.

3. $\text{FINISH}_{p_{1_h}} = ff$ et $\text{FINISH}_{p_{2_h}} = tt$:
   $\text{REG}^+_{1_h} = ff$ et $\text{REG}^+_{2_h} = tt$; Dans ce cas on change d'état, l'état futur sera (ff, tt) et $\text{FINISH}_h = ff$; toujours en appliquant l'hypothèse d'induction, on en déduit que $term_{\|} = ff$. Toutefois, on a la réécriture suivante dans la sémantique comportementale :

$$\frac{p_1 \xrightarrow[E_c]{E_1, ff} p'1 \quad , \quad p_2 \xrightarrow[E_c]{E_2, tt} \textbf{nothing}}{p_1 \| p_2 \xrightarrow[E_c]{E_1 \cup E_2,\ ff} p'1 \| \textbf{nothing}}$$

4. $\text{FINISH}_{p_{1_h}} = tt$ et $\text{FINISH}_{p_{2_h}} = tt$:
   $\text{REG}^+_{1_h}$ et $\text{REG}^+_{2_h}$ valent tous les deux ff ; dans ce cas encore on ne change pas d'état mais $\text{FINISH}_h = tt$. Dans la sémantique comportementale, la règle appliquée est :

$$\frac{p_1 \xrightarrow[E_c]{E_1, tt} \textbf{nothing} \quad , \quad p_2 \xrightarrow[E_c]{E_2, tt} \textbf{nothing}}{p_1 \| p_2 \xrightarrow[E_c]{E_1 \cup E_2,\ tt} \textbf{nothing} \| \textbf{nothing}}$$

   Donc $term_{\|} = tt$.

A partir de l'état initial, on crée deux nouveaux états ($\text{REG}_{1_h} = tt$ et $\text{REG}_{2_h} = ff$; $\text{REG}_{1_h} = ff$ et $\text{REG}_{2_h} = tt$). Dans chacun de ces états, ce sont toujours les valeurs respectives de $\text{FINISH}_{p_{1_h}}$ et de $\text{FINISH}_{p_{2_h}}$ qui pilotent les nouvelles valeurs des registres et déterminent celle de FINISH_h, conformément au système d'équations de l'opérateur.

1. $\text{REG}_{1_h} = tt$ et $\text{REG}_{2_h} = ff$: dans cet état, nous avons :
   $\text{REG}^+_{1_h} = \overline{\text{FINISH}_{p_{2_h}}} + \text{FINISH}_{p_{1_h}}.\overline{\text{FINISH}_{p_{2_h}}} = \overline{\text{FINISH}_{p_{2_h}}}$
   $\text{REG}^2_{2_h} = ff$
   Nous avons aussi $\text{FINISH}_p = \text{FINISH}_{p_{2_h}}$. Ainsi nous ne regarderons que la valeur de $\text{FINISH}_{p_{2_h}}$.

(a) $\text{FINISH}_{p_{2_h}} = ff$: $\text{REG}^+_{1_h} = ff$ et $\text{REG}^+_{2_h} = tt$; dans ce cas on ne change pas d'état. Nous calculons $\text{FINISH}_h = ff$. Dans la sémantique comportementale, c'est la règle suivante qui est appliquée :

$$\frac{\mathbf{nothing} \xrightarrow[E_C]{\emptyset,tt} \mathbf{nothing} \quad , \quad p_2 \xrightarrow[E_C]{E_2,ff} p'2}{\mathbf{nothing} \| p_2 \xrightarrow[E_C]{E_2,\ ff} \mathbf{nothing} \| p'2}$$

car $p1$ a déjà été réécrit en **nothing** dans la première application de la règle de la sémantique comportementale. Nous avons bien $term_{\|} = ff$.

(b) $\text{FINISH}_{p_{2_h}} = tt$: $\text{REG}^+_{1_h} = ff$ et $\text{REG}^+_{2_h} = ff$; dans ce cas, on retourne dans l'état initial. l'équation de FINISH nous permet de calculer : $\text{FINISH}_h = tt$. Dans la sémantique comportementale, la réécriture est la suivante :

$$\frac{\mathbf{nothing} \xrightarrow[E_C]{\emptyset,tt} \mathbf{nothing} \quad , \quad p_2 \xrightarrow[E_C]{E_2,tt} \mathbf{nothing}}{\mathbf{nothing} \| p_2 \xrightarrow[E_C]{E_2,\ tt} \mathbf{nothing} \| \mathbf{nothing}}$$

En conséquence, $term_{\|} = tt$.

2. $\text{REG}_{1_h} = ff$ et $\text{REG}_{2_h} = tt$: cet état est atteint quand $\text{FINISH}_{p_{1_h}} = ff$ et $\text{FINISH}_{p_{2_h}} = tt$ dans la réaction précédente ; de façon duale au cas précédent, nous avons :
$\text{REG}^+_{1_h} = ff$
$\text{REG}^+_{2_h} = \overline{\text{FINISH}_{p_{1_h}}} + \text{FINISH}_{p_{2_h}}.\overline{\text{FINISH}_{p_{1_h}}} = \overline{\text{FINISH}_{p_{1_h}}}$
$\text{FINISH}_p = \text{FINISH}_{p_{1_h}}$
Nous n'avons donc que deux cas à étudier :

(a) $\text{FINISH}_{p_{1_h}} = ff$: $\text{REG}^+_{1_h} = ff$ et $\text{REG}^+_{2_h} = tt$; dans ce cas on ne change pas d'état. Nous calculons $\text{FINISH}_h = ff$. Dans la sémantique comportementale, c'est la règle suivante qui est appliquée :

$$\frac{p_1 \xrightarrow[E_C]{E_1,ff} p'_1 \quad , \quad \mathbf{nothing} \xrightarrow[E_C]{\emptyset,tt} \mathbf{nothing}}{p_1 \| \mathbf{nothing} \xrightarrow[E_C]{E_1,\ ff} p'_1 \| \mathbf{nothing}}$$

car p_2 a déjà été réécrit en **nothing** dans la première application de la règle de la sémantique comportementale (voir l'étude de l'état initial). Nous avons bien $term_{\|} = ff$.

(b) $\text{FINISH}_{p_{1_h}} = tt$: $\text{REG}^+_{1_h} = ff$ et $\text{REG}^+_{2_h} = ff$; dans ce cas, on retourne dans l'état initial. L'équation de FINISH_h nous permet de calculer $\text{FINISH}_h = tt$. Dans la sémantique comportementale, la réécriture est la suivante :

$$\frac{p_1 \xrightarrow[E_C]{E_1,tt} \mathbf{nothing} \quad , \quad \mathbf{nothing} \xrightarrow[E_C]{\emptyset,tt} \mathbf{nothing}}{p_1 \| \mathbf{nothing} \xrightarrow[E_C]{E_1,\ tt} \mathbf{nothing} \| \mathbf{nothing}}$$

En conséquence, $term_{\|} = tt$.

Aucun autre état n'a été créé, donc la démonstration est terminée en ce qui concerne la concordance de la terminaison dans les deux sémantiques.

Concernant les valeurs des événements de sortie dans les deux sémantiques, dans la sémantique comportementale, il n'y a qu'une règle générale pour le parallèle que nous avons interprétée suivant les différents cas. Mais l'environnement de sortie est toujours $E_1 \cup E_2$ avec l'un des ensembles possiblement vide. L'hypothèse de récurrence permet d'affirmer que : $\forall o \in E_1, o^1 \in \langle p_1 \rangle_E$ et $\forall o \in E_2, o^1 \in \langle p_2 \rangle_E$. Dans la sémantique opérationnelle, nous faisons l'unification des environnements des arguments ($\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_E$) et donc $\forall o^x \in \langle p_1 \rangle_E \sqcup \langle p_2 \rangle_E \; x = 1$. L'environnement $\langle p_1 \| p_2 \rangle_E$ est obtenu en appliquant $\mathcal{D}_\|$ à $\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_E$, et aucune équation de $\mathcal{D}_\|$ ne change le statut des événements de sortie, donc $o^1 \in \langle p_1 \| p_2 \rangle_E$.

stop p **when** S **alert** S_1

Le système finalisé est le suivant :

$$\mathcal{F}(\mathcal{D}_{stop}) = \left[\begin{array}{lcl} \text{REG}_h^+ & = & \text{REG}_h.\overline{\text{FINISH}_{p_h}}.\overline{S_h} \; + \; \overline{\text{REG}_h}.\text{START}_h.\overline{\text{FINISH}_{p_h}} \\ \text{START}_{p_h} & = & \text{START}_h \\ \text{FINISH}_h & = & S_h.\text{REG}_h \; + \; \text{FINISH}_{p_h} \\ S_{1_h} & = & \text{REG}_h.S_h \end{array} \right]$$

Dans la réaction initiale, $\text{REG}_h = ff$ et $\text{START}_h = tt$. On a donc :
$\text{REG}_h^+ = \text{FINISH}_{p_h}$
$\text{FINISH}_h = \text{FINISH}_{p_h}$
$S_{1_h} = ff$
Le comportement dépend de la terminaison de p dans le premier instant :

1. $\text{FINISH}_{p_h} = tt$: l'état futur est tt ($\text{REG}_h^+ = tt$) et $\text{FINISH}_h = tt$. Dans la sémantique comportementale, c'est la règle *stop*1 qui s'applique :

$$\frac{p \xrightarrow[E_C]{E_1, tt} \textbf{nothing}}{\textbf{stop}(p,\ S,\ S_1) \xrightarrow[E_C]{E_1,\ tt} \textbf{nothing}}$$

   Le booléen de terminaison est aussi tt.

2. $\text{FINISH}_{p_h} = ff$: l'état futur est ff, on reste dans l'état initial ; $\text{FINISH}_h = ff$. Dans la sémantique comportementale c'est la règle *stop*2 qui s'applique :

$$\frac{p \xrightarrow[E_C]{E_1,\ ff} p'}{\textbf{stop}(p,\ S,\ S_1) \xrightarrow[E_C]{E_1,\ ff} \textbf{if } S \textbf{ then emit } S_1 \textbf{ else stop}(p',\ S,\ S_1)}$$

   Le booléen de terminaison de la règle est ff.

Quelles que soient les règles de la sémantique comportementale appliquées, l'environnement de sortie est E_1, l'environnement de sortie de p calculé à partir de E_C. Par induction, nous savons que pour tout $o \in E_1$ $o^1 \in \langle p \rangle_E$. L'environnement de l'opérateur est calculé à partir de $\langle p \rangle_E$ en appliquant les équations dans $\mathcal{D}_{stop}$. La seule sortie que les équations peuvent modifier c'est S_1. Dans cet état, S_1 n'appartient pas à E_1 et $S_{1_h} = ff$, donc $S_1^x \in \langle \textbf{stop}(p',\ S,\ S_1) \rangle_E$ et $x \neq 1$.

Maintenant, nous étudions le second état de l'opérateur $\text{REG}_h = tt$. Dans cet état
$\text{START}_h = ff$
$\text{REG}_h^+ = \overline{\text{FINISH}_{p_h}}.\overline{S_h}$

$\text{FINISH}_h = \text{FINISH}_{p_h} \ + \ S_h$
$S_{1_h} = S_h$
Nous voyons que dans cet état ce sont les valeurs de S_h et de FINISH_{p_h} qui déterminent le comportement de l'opérateur.

1. $S_h = ff$: on obtient $S_{1_h} = ff$ et l'état suivant et la terminaison dépendent de la terminaison de p :
   (a) $\text{FINISH}_{p_h} = ff$: $\text{REG}_h^+ = tt$; on reste dans l'état courant et $\text{FINISH}_h = ff$. Dans la sémantique comportementale, comme nous sommes arrivés dans cet état en appliquant la règle *stop*2, on doit dériver le terme d'arrivée de la règle *stop*2 : **if** S **then emit** S_1 **else stop**$(p',\ S,\ S_1)$. Ici c'est la règle $if2$ qui s'applique :

$$\frac{\textbf{emit}\ S_1 \xrightarrow[E_C]{\{S_1\},tt} \textbf{nothing} \quad , \quad p \xrightarrow[E_C]{E_1,term_p} p' \quad , \quad (S, E_C) \rightarrowtail ff}{\textbf{if}(S, \textbf{emit}\ S_1, p) \xrightarrow[E_C]{E_1,term_p} p'}$$

   Alors, le booléen de terminaison de l'opérateur est $term_p = \text{FINISH}_{p_h}$ par induction. Donc $term_p = \text{FINISH}_h$.
   (b) $\text{FINISH}_{p_h} = tt$: $\text{REG}_h^+ = ff$; on retourne dans l'état initial et l'évaluation est terminée. $\text{FINISH}_h = tt$. Comme nous ne sommes plus dans l'état initial et $S_h = ff$, c'est encore la règle $if2$ qui s'applique et le booléen de terminaison de l'opérateur est $term_p$. Par induction nous avons $term_p = \text{FINISH}_{p_h} = tt$.

   Dans les deux cas, l'environnement de sortie de la sémantique comportementale pour l'opérateur est E_1, l'environnement de sortie de p calculé à partir de E_C et $S_h = ff$ donc $S_1 \neq 1$. Nous sommes dans le même contexte que pour l'état initial et un raisonnement similaire prouve que $\forall o \in E_1, o^1 \in \langle \textbf{stop}(p,\ S,\ S_1)\rangle_E$.
2. $S_h = tt$: on obtient $S_{1_h} = tt$; $\text{REG}_h = ff$ et $\text{FINISH}_h = tt$ quelle que soit la valeur de de FINISH_{p_h}. Comme précédemment, c'est une des règles de réécriture de **if** qui s'applique dans la sémantique comportementale, mais cette fois c'est la règle $if1$ car $S_h = tt$:

$$\frac{\textbf{emit}\ S_1 \xrightarrow[E_C]{\{S_1\},tt} \textbf{nothing} \quad , \quad p \xrightarrow[E_C]{E_1,term_p} p' \quad , \quad (S, E_C) \rightarrowtail tt}{\textbf{if}(S, \textbf{emit}\ S_1, p) \xrightarrow[E_C]{\{S_1\},tt} \textbf{nothing}}$$

   Dans ce cas, l'environnement de sortie de la sémantique comportementale est $\{S_1\}$, d'une part. D'autre part dans la sémantique opérationnelle, $S_{1_h} = tt$. Avec un raisonnement similaire à celui de l'opérateur **emit**, nous pouvons en déduire que $S_{1_h}^{tt} \in \mathcal{F}(\langle \textbf{stop}(p,\ S,\ S_1)\rangle_E)$. Par définition de la finalisation nous avons $S_1^x \in \langle \textbf{stop}(p,\ S,\ S_1)\rangle_E$ et $x = 1$ ou $x = \top$. Comme nous avons fait l'hypothèse qu'aucun événement dans l'environnement de sortie de cette sémantique ne valait $\top$, on a $S_1^1 \in \langle \textbf{stop}(p,\ S,\ S_1)\rangle_E$.

while $cond\{p\}$

Voici son système d'équations finalisé :

$$\mathcal{F}(\mathcal{D}_{while}) = \begin{bmatrix} \text{REG}_h^+ & = & \text{REG}_h \ + \ \text{START}_h \\ \text{START}_{p_h} & = & \text{START}_h \ + \ \text{FINISH}_{p_h} \\ \text{FINISH}_h & = & \overline{cond} \end{bmatrix}$$

Comme pour les autres opérateurs, dans le premier instant nous avons $\text{START}_h = tt$ et $\text{REG}_h = ff$. Ici la valeur du registre n'influe pas sur la terminaison ; il sert juste pour tuer l'argument uniquement lorsque le **while** a démarré (c'est à dire que START et *cond* aient été vrais au premier instant) quand la condition n'est plus vraie. Comme nous ne tenons pas compte des événements KILL, grâce au lemme 1, il est clair que l'on pourrait le supprimer du système $\mathcal{F}(\mathcal{D}_{while})$.

1. $cond_h = tt$: $\text{FINISH}_h = ff$. Dans la sémantique comportementale, c'est la règle $while1$ qui s'applique :

$$\frac{p \xrightarrow[E_C]{E_1, ff} p' \ , \ (cond, E_C) \rightarrowtail tt}{\textbf{while}(cond, p) \xrightarrow[E_C]{E_1, \ ff} p' \ \textbf{seq while}(cond, p)}$$

   Le booléen de terminaison ff est est égal à FINISH_h.

2. $cond_h = ff$: $\text{FINISH}_h = tt$. Dans la sémantique comportementale, c'est la règle $while2$ qui s'applique :

$$\frac{p \xrightarrow[E_C]{E_1, term_p} p' \ , \ (cond, E_C) \rightarrowtail \ ff}{\textbf{while}(cond, p) \xrightarrow[E_C]{E_1, \ tt} \textbf{nothing}}$$

   Le booléen de terminaison est tt et est aussi égal à FINISH_h.

Dans les deux règles de la sémantique comportementale, l'environnement de sortie est E_1 l'environnement de sortie de p calculé avec E_C comme environnement d'entrée. Par induction, nous savons que pour tout $o \in E_1$, alors $o^1 \in \langle p \rangle_E$. Les équations de $\mathcal{D}_{while}$ ne modifient pas les statuts des événements de sortie, donc $o^1 \in \langle \textbf{while}(p.cond) \rangle_E$.

If *cond* **then** p_1 **else** p_2

$$\mathcal{F}(\mathcal{D}_{if}) = \begin{bmatrix} \text{START}_{p_{1_h}} & = & \text{START}_h.cond_h \\ \text{START}_{p_{2_h}} & = & \text{START}_h.\overline{cond} \\ \text{FINISH}_h & = & \text{FINISH}_{p_{1_h}} \ + \ \text{FINISH}_{p_{2_h}} \end{bmatrix}$$

Comme pour tous les opérateurs, nous ne considérons pas KILL_h dans le système d'équations finalisé de l'opérateur.
Comme pour l'opérateur précédent, la démonstration distingue 2 cas :

1. $cond_h = tt$
   Le système d'équation de la sémantique opérationnelle permet de déduire $\text{START}_{p_{1_h}} = tt$ et $\text{START}_{p_{2_h}} = ff$ Comme $\text{START}_{p_{2_h}} = ff$, nous savons grâce au lemme 2 que $\text{FINISH}_{p_{2_h}} = ff$; donc $\text{FINISH}_h = \text{FINISH}_{p_{1_h}}$. Dans la sémantique comportementale, c'est la règle $if1$ qui est appliquée et le booléen de terminaison est $term_{p_1}$. Par induction, nous savons que $\text{FINISH}_{p_{1_h}} = term_{p_1}$.

2. $cond_h \neq ff$
   De façon similaire au cas précédent, nous déduisons que $\text{FINISH}_h = \text{FINISH}_{p_{2_h}}$. Dans la sémantique comportementale, c'est la règle $if2$ qui est appliquée et le booléen de terminaison est $term_{p_2}$. Par induction, nous savons que

$\text{FINISH}_{p_{2_h}} = term_{p_2}$.

L'environnement de sortie de l'opérateur dans la sémantique opérationnelle est calculé à partir de $\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_S$. Si $cond_h = tt$ alors $\text{START}_{p_{2_h}} = ff$. Donc $\text{START}_{p_2} \neq 1$. Considérons une sortie $o^x \in \langle p_2 \rangle_S$, seulement trois opérateurs changent les statuts des locaux et des sorties de l'environnement : **emit**, **stop**, **timeout**. Si p_2 a une sous instruction i composée de l'un de ces trois opérateurs, avec le lemme 2, nous savons que le START de cette sous instruction (START_i) n'est pas égal à 1.

1. si $i = \textbf{timeout}(p, S, p', o)$ alors $o = \text{REG}_1 \boxdot \text{REG}_2 \boxdot S$. Mais si START_i n'est pas 1 dans le premier instant, alors les deux registres restent à 0 et donc $o^0 \in \langle p_2 \rangle_S$.
2. si $i = \textbf{stop}(p, S, o)$ alors $o = \text{REG} \boxdot S$. Comme précédemment, si START_i n'est pas 1 alors le registre reste à 0 et $o^0 \in\in \langle p_2 \rangle_S$.
3. si $i = \textbf{emit}\ o$ alors $o = \text{FINISH}_i$. Comme START_i n'est pas 1, $\text{FINISH}_i \neq 1$.

Si le statut de o est x_1 (resp. x_2) dans $\langle p_1 \rangle_S$ (resp. $\langle p_2 \rangle_S$), dans $\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_S$ son statut sera $x_1 \sqcup x_2$ et dans $\langle p_1 \rangle_E \sqcup \langle p_2 \rangle_S$ le statut de o sera $x'_1 \sqcup x'2$; x'_1 résultant de l'application des équations de $\mathcal{D}_{if}$ à x_1. Nous sommes dans le cas où $cond_h = tt$, donc $cond = 1$ et $\text{START}_{p_1} = 1$. Ainsi, les équations ne modifie pas x_1. En revanche, si $\text{START}_{p_{2_h}} = ff$, $x'2 = 0$ ou $x'2 = \bot$ (voir ci dessus l'étude des trois cas possibles). Si $x_1 = \bot$ alors $x_1 \sqcup x'_2 = x_1$ car $\bot$ est élément neutre pour $\sqcup$. Si $x_1 = 0$ on a toujours $x_1 \sqcup x'_2 = x_1$ (car nous ne pouvons pas avoir $\top$) et si $x_1 = 1$ alors $x_1 \sqcup x'_2 = x_1$ aussi car nous éliminons $\top$. Alors $o^{x_1} \in \langle \textbf{if}(cond, p_1, p_2) \rangle_E$. Dans la sémantique comportementale, c'est la règle $if1$ qui est appliquée, l'environnement de sortie est donc E_1, l'environnement de sortie de p_1 calculé à partir de $E_\mathcal{C}$. Par récurrence, nous savons que si $o \in E_1$ alors $o^1 \in \langle p_1 \rangle_E$. Donc si $x_1 = 1$ alors $o^1 \in \langle p_1 \rangle_E \sqcup \langle p_2 \rangle_S$.
Si $cond_h = ff$, le raisonnement est similaire.

Timeout : p **timeout** S $\{p_1\}$ **alert** S_1
Le comportement de cet opérateur dépend du statut de S et des valeurs de terminaison de p et de p_1. Nous définissons en premier son système d'équations finalisé :

$$\mathcal{F}(\mathcal{D}_{timeout}) = \left[\begin{array}{lcl}
\text{REG}^+_{1_h} & = & \text{REG}_{1_h}.\overline{S_h}.\overline{\text{FINISH}_{p_h}} \ + \ \overline{\text{REG}_{1_h}}.\overline{\text{REG}_{2_h}}.\text{START}_h.\overline{\text{FINISH}_{p_h}} \\
\text{REG}^+_{2_h} & = & \text{REG}_{1_h}.\overline{\text{REG}_{2_h}}.\overline{S_h}.\text{FINISH}_{p_h}.\overline{\text{FINISH}_{p_{1_h}}} \ + \\
 & & \overline{\text{REG}_{1_h}}.\text{REG}_{2_h}.\overline{\text{FINISH}_{p_{1_h}}} \ + \\
 & & \overline{\text{REG}_{1_h}}.\overline{\text{REG}_{2_h}}.\text{START}_h.\text{FINISH}_{p_h}.\overline{\text{FINISH}_{p_{1_h}}} \\
\text{START}_{p_h} & = & \overline{\text{REG}_{1_h}}.\overline{\text{REG}_{2_h}}.\text{START}_h.\text{FINISH}_{p_h}.\text{FINISH}_{p_{1_h}} + \\
 & & \overline{\text{REG}_{1_h}}.\overline{\text{REG}_{2_h}}.\text{START}_h.\overline{\text{FINISH}_{p_h}} \\
\text{START}_{p_{1_h}} & = & \text{REG}_{1_h}.\overline{\text{REG}_{2_h}}.\overline{S_h}.\text{FINISH}_{p_h}.\overline{\text{FINISH}_{p_{1_h}}} + \\
 & & \overline{\text{REG}_{1_h}}.\overline{\text{REG}_{2_h}}.\text{START}_h.\text{FINISH}_{p_h} \\
\text{FINISH}_h & = & \text{REG}_{1_h}.\overline{\text{REG}_{2_h}}.S_h \ + \ \overline{\text{REG}_{1_h}}.\ \text{REG}_{2_h}.\text{FINISH}_{p_{1_h}} \ + \\
 & & \text{REG}_{1_h}.\overline{\text{REG}_{2_h}}.\text{FINISH}_p.\text{FINISH}_{p_{1_h}} \ + \\
 & & \overline{\text{REG}_{1_h}}.\overline{\text{REG}_{2_h}}.\text{START}_h.\text{FINISH}_{p_h}.\text{FINISH}_{p_{1_h}} \\
S_{1_h} & = & \text{REG}_1.\overline{\text{REG}_{2_h}}.S_h
\end{array}\right]$$

Dans l'expression de $\mathcal{F}(\mathcal{D}_{timeout})$ nous avons supposé $\text{KILL} = 0$ grâce au lemme 1. Cet opérateur a deux registres qui déterminent les états de l'automate, nous allons étudier son comportement dans les différents états atteignables de l'automate. Tout d'abord, nous étudions l'état initial :

1. $\text{REG}_{1_h} = ff$ et $\text{REG}_{2_h} = ff$.
   C'est, par définition, l'état de départ de l'évaluation de **timeout**. Dans cet état, $\text{START}_h = tt$. Nous calculons $\text{REG}^+_{1_h} = \overline{\text{FINISH}_{p_h}}$ et $\text{REG}^+_{2_h} = \text{FINISH}_{p_h}.\overline{\text{FINISH}_{p_{1_h}}}$, ce qui détermine l'état suivant. De plus, $\text{FINISH}_h = \text{FINISH}_{p_h}.\text{FINISH}_{p_{1_h}}$. On a donc deux cas, suivant que p termine instantanément ou non.

   (a) $\text{FINISH}_{p_h} = tt$: si p_1 est lui aussi instantané ($\text{FINISH}_{p_{1_h}} = tt$) alors l'état futur est (ff, ff) on ne change pas d'état et $\text{FINISH}_h = tt$. Dans la sémantique comportementale, c'est la règle *timeout*1 qui s'applique :

$$\frac{p \xrightarrow[E_C]{E_1, tt} \textbf{nothing}, p_1 \xrightarrow[E_C]{E_2, tt} \textbf{nothing}}{\textbf{timeout}(p,\ S,\ p_1, S_1) \xrightarrow[E_C]{E_1 \cup E_2, tt} \textbf{nothing}}$$

   Si p_1 n'est pas instantané ($\text{FINISH}_{p_{1_h}} = ff$) l'état futur est (ff, tt) et $\text{FINISH}_h = ff$. Dans la sémantique comportementale, nous avons :

$$\frac{p \xrightarrow[E_C]{E_1, tt} \textbf{nothing}, p_1 \xrightarrow[E_C]{E_2, ff} p_1'}{\textbf{timeout}(p,\ S,\ p_1, S_1) \xrightarrow[E_C]{E_1 \cup E_2, ff} p_1'}$$

   Dans tous les cas, on a bien $term_{timeout} = \text{FINISH}_h$.

   (b) $\text{FINISH}_{p_h} = ff$, alors quelle que soit la valeur de $\text{FINISH}_{p_{1_h}}$ l'état futur est (tt, ff) et $\text{FINISH}_h = ff$. Dans la sémantique comportementale, c'est la règle *timeout*2 qui s'applique et $term_{timeout} = ff$.

   Dans cet état, l'environnement de sortie de la sémantique comportementale, calculé à partir de E_C est soit $E_1 \cup E_2$ ou bien E_1, les environnements respectifs des arguments de l'opérateur calculés à partir de E_C. Par récurrence, nous savons que pour toute sortie o dans E_1 (resp. E_2) $o^1 \in \langle p \rangle_E$ (resp. $\langle p_1 \rangle_E$). L'environnement de sortie de l'opérateur, dans la sémantique opérationnelle, est $\langle p_1 \rangle_{\langle p \rangle_E}$, il résulte de l'application de $\mathcal{D}_{p_1}$ à $\langle p \rangle_E$. La sémantique opérationnelle est constructive et aucun événement d'un environnement d'entrée ne peut avoir un statut plus petit (au sens de l'ordre de connaissance) dans l'environnement de sortie d'une instruction. Donc, dans $\langle p_1 \rangle_{\langle p \rangle_E}$, il y a un élément o^x avec $1 \leq_K x$. Comme nous avons écarté les cas où un événement a pour statut $\top$ dans un environnement de sortie, o^1 est dans $\langle p_1 \rangle_{\langle p \rangle_E}$. Par ailleurs, $\mathcal{D}_{timeout}$ ne change le statut d'aucune sortie sauf S_1. Dans cet état, ni la règle *timeout*1 ni *timeout*2 ne mettent S_1 dans l'environnement de sortie et dans la sémantique opérationnelle, $S_{1_h} = ff$ donc soit S^0 ou $S^\perp$ est dans $\langle timeout \rangle_E$.
   Depuis cet état initial, nous avons deux états atteignables. Nous étudions ces deux états.

2. $\text{REG}_{1_h} = tt$ et $\text{REG}_{2_h} = ff$.
   Cet état correspond à l'état s_1 dans la figure 4.2. Dans cet état, nous ne sommes plus dans l'état initial et $\text{START}_h = ff$. On a
   $\text{REG}^+_{1_h} = \overline{s_h}.\overline{\text{FINISH}_{p_h}}$
   $\text{REG}^+_{2_h} = \overline{s_h}.\text{FINISH}_{p_h}.\overline{\text{FINISH}_{p_{1_h}}}$
   $\text{FINISH}_h = S_h + \text{FINISH}_{p_h}.\text{FINISH}_{p_{1_h}}$.
   La terminaison dépend de S_h et des terminaison de p et p_1 :

   (a) $S_h = ff$: la valeur de FINISH_h dépend des terminaison de p et de p_1 :

i. $\text{FINISH}_{p_h} = ff$: quelle que soit la valeur de $\text{FINISH}_{p_{1_h}}$ on a $\text{REG}^+_{1_h} = tt$ et $\text{REG}^+_{2_h} = ff$; on ne change pas d'état et $\text{FINISH}_h = ff$. Pour arriver dans cet état, il a fallu que $\text{FINISH}_{p_h} = ff$ dans l'instant précédent ; ceci signifie qu'à l'instant précédent, on a appliqué la règle *timeout*2 dans la sémantique comportementale. Nous appliquons une des règles de l'opérateur **if..then..else**. Comme $S_h = ff$, nous appliquons la règle $if2$ à **if** S_h **then emit** S_{1_h} **else timeout**$(p,\ S_h,\ p_1, S_{1_h})$. Dans la règle $if2$, l'opérateur **if..then..else** se réécrit en son second argument et son booléen de terminaison est celui de ce dernier qui est aussi celui de p. On a donc $term_{timeout} = ff$ dans ce cas.

ii. $\text{FINISH}_{p_h} = tt$: alors $\text{REG}^+_{1_h} = ff$ et $\text{REG}^+_{2_h} = \overline{\text{FINISH}_{p_{1_h}}}$; de plus $\text{FINISH}_h = \text{FINISH}_{p_{1_h}}$. On doit donc regarder les deux valeurs possibles de la terminaison de p_1 :

A. $\text{FINISH}_{p_{1_h}} = ff$: l'état futur est (ff, tt) et $\text{FINISH}_h = ff$. Nous sommes toujours dans l'état (tt, ff) et donc dans l'instant précédent c'est la règle *timeou*2 qui a été appliquée. Donc comme précédemment, c'est la règle $if2$ qui est appliquée à **if** S_h **then emit** S_{1_h} **else timeout**$(p,\ S_h,\ p_1, S_{1_h})$. Dans ce cas, on a la réécriture suivante :

$$\frac{\dfrac{p \xrightarrow[E_C]{E_1, tt} \textbf{nothing} \quad , \quad p_1 \xrightarrow[E_C]{E_2, ff} p'_1}{\textbf{timeout}(p,\ S,\ p_1, S_1) \xrightarrow[E_C]{E_1 \cup E_2, ff} p'1 \quad , \quad (S, E_C) \rightarrowtail ff}}{\textbf{if } S \textbf{ then emit } S_1 \textbf{ else timeout}(p,\ S,\ p_1, S_1) \xrightarrow[E_C]{E_1 \cup E_2, ff} p'1}$$

Ainsi,on a bien $term_{timeout} = ff = \text{FINISH}_h$.

B. $\text{FINISH}_{p_{1_h}} = tt$: l'état futur est (ff, ff) ; on transite vers l'état initial et l'évaluation est terminée. On est dans la même configuration que précédemment avec p_1 qui termine. La réécriture dans la sémantique comportementale devient donc :

$$\frac{\dfrac{p \xrightarrow[E_C]{E_1, tt} \textbf{nothing} \quad , \quad p_1 \xrightarrow[E_C]{E_2, tt} \textbf{nothing}}{\textbf{timeout}(p,\ S,\ p_1, S_1) \xrightarrow[E_C]{E_1 \cup E_2, tt} \textbf{nothing} \quad , \quad (S, E_C) \rightarrowtail ff}}{\textbf{if } S \textbf{ then emit } S_1 \textbf{ else timeout}(p,\ S,\ p_1, S_1) \xrightarrow[E_C]{E_1 \cup E_2, tt} \textbf{nothing}}$$

Dans ce cas on a toujours $term_{timeout} = tt = \text{FINISH}_h$.

(b) $S_h = tt$: nous avons $\text{REG}^+_{1_h} = ff$ et $\text{REG}^+_{2_h} = ff$; on transite donc vers l'état initial et l'exécution est terminée. De plus nous avons $\text{FINISH}_p = tt$ et $S_{1_h} = tt$. Dans la sémantique comportementale, nous sommes toujours dans le même état et dans l'instant précédent nous avions $\text{FINISH}_{p_h} = ff$ et nous avons donc appliqué la règle *timeout*2. La réécriture pour cette état est donc :

$$\frac{\textbf{emit } S_1 \xrightarrow[E_C]{\{S_1\},\ tt} \textbf{nothing} \quad , \quad (S, E_C) \rightarrowtail tt}{\textbf{if } S \textbf{ then emit } S_1 \textbf{ else timeout}(p,\ S,\ p_1, S_1) \xrightarrow[E_C]{\{S_1\}, tt} \textbf{nothing}}$$

Ainsi $term_{timeout} = tt = \text{FINISH}_p$

Concernant les événements dans l'environnement de sortie des différentes règles de la sémantique comportementale appliquées dans cet état, nous distinguons le cas où $S_h = tt$ et celui où $S_h = ff$. Dans ce dernier cas, nous voyons que l'environnement de sortie est $E_1 \cup E_2$ calculé à partir de E. L'environnement d'entrée de la sémantique opérationnelle est E. Nous sommes dans une situation similaire à celle de l'état initial et nous pouvons faire le même raisonnement pour prouver le théorème. En revanche, si $S_h = tt$, dans la règle de la sémantique comportementale, l'environnement de sortie est $\{S_1\}$. Dans la sémantique opérationnelle, on a $S_{1_h} = tt$, avec un raisonnement similaire à celui de l'opérateur **emit**, nous savons que $S^{tt}_{1_h} \in \mathcal{F}(\langle \mathbf{timeout}(p, S, p_1, S_1)\rangle_E)$. Ainsi par définition de la finalisation, nous avons $S^x \in \langle \mathbf{timeout}(p, S, p_1, S_1)\rangle_E$. Donc $x = 1$ ou $x = \top$ (voir section 4.5.1). Par hypothèse, S^1_1 est dans l'environnement de sortie de l'opérateur.

3. $\text{REG}_{1_h} = ff$ et $\text{REG}_{2_h} = tt$.
Dans cet état, les équations deviennent :
$\text{REG}_{1_h} = ff$
$\text{REG}_{2_h} = \overline{\text{FINISH}_{p_{1_h}}}$
$\text{FINISH}_h = \text{FINISH}_{p_{1_h}}$
$S_{1_h} = ff$
Cet état correspond à l'état s_2 dans la figure 4.2. On y arrive depuis l'état initial avec $\text{FINISH}_{p_h} = tt$ et $\text{FINISH}_{p_{1_h}} = ff$. Il correspond à l'état où p a terminé son exécution sans être préempté et donc on commence l'exécution de p_1. D'ailleurs, on peut vérifier que $\text{START}_{p_{1_h}} = tt$. Dans la sémantique comportementale, la règle suivante a été appliquée dans l'instant précédent (cf.l'état initial) :

$$\frac{p \xrightarrow[E_c]{E_1, tt} \mathbf{nothing}, p_1 \xrightarrow[E_c]{E_2, ff} p'_1}{\mathbf{timeout}(p,\ S,\ p_1, S_1) \xrightarrow[E_c]{E_1 \cup E_2, ff} p'_1}$$

Dans cet état, la règle qui correspond est donc celle de la réécriture de p_1 :

$$p_1 \xrightarrow[E_c]{E_2, term_{p_1}} p'1$$

Par induction, nous savons que $term_{p_1} = \text{FINISH}_{p_{1_h}}$. On a donc $term_{timeout} = term_{p_1} = \text{FINISH}_{p_{1_h}} = \text{FINISH}_h$. Ainsi, si $\text{FINISH}_{p_{1_h}} = tt$, l'état futur est (ff, ff), on retourne dans l'état initial ; l'exécution est terminée et si $\text{FINISH}_{p_{1_h}} = ff$, on reste dans cet état jusqu'à la terminaison de p_1.
Concernant les événements de sortie appartenant à l'environnement de sortie de la sémantique comportementale, par induction, nous savons que pour tout $o \in E_2$, alors $o^1 \in \langle p_1\rangle_E$. Notre sémantique opérationnelle étant constructive, si $o^x \in E$ alors $o^y \in \langle p\rangle_E$ et $x \leq_K y$. Si $x = y$ alors $o^1 \in \langle p_1\rangle_{\langle p\rangle_E}$ en appliquant l'induction ; sinon soit $x = \bot$ et $y = 1$ et nous avons aussi $o^1 \in \langle p_1\rangle_{\langle p\rangle_E}$ puisque nous avons écarté le cas où $o^\top \in \langle p_1\rangle_{\langle p\rangle_E}$; soit $x = \bot$ et $y = 0$, alors le statut de x grossirait de $\bot$ à 0 dans $\langle p\rangle_E$et de $\bot$ à 1 dans $\langle p_1\rangle_E$; ce qui donnerait $\top$ dans $\langle p_1\rangle_{\langle p\rangle_E}$; soit $y = \top$ mais cela est exclu par hypothèse.

local(S) **{p}**

$$\mathcal{F}(\mathcal{D}_{local}) = \left[\begin{array}{lcl} \text{START}_{p_h} & = & \text{START}_h \\ \text{FINISH}_h & = & \text{FINISH}_{p_h} \end{array} \right]$$

Dans la sémantique opérationnelle, nous avons $\text{FINISH}_h = \text{FINISH}_{p_h}$. Par induction, nous savons que $\text{FINISH}_{p_h} = term_p$. Dans la sémantique comportementale, Aussi bien dans $local1$ que dans $local2$ le booléen de terminaison est $term_p$.

Concernant les statuts des événements de sortie dans les deux sémantiques, l' environnement de départ pour appliquer $\mathcal{D}_{local}$ est $\langle p \rangle_E$. Si $S^1 \in \langle \mathbf{local}(p, S) \rangle_E$, alors l'événement S est émis et c'est la règle $local1$ qui est appliquée. Par récurrence, nous savons que : $\forall o \in E', o^1 \in \langle p \rangle_E$, E' étant l'environnement de sortie de la sémantique comportementale de p. Les équations de $\mathcal{D}_{local}$ ne changent pas le statut de o donc $o^1 \in \langle \mathbf{local}(p, S) \rangle_E$. Si $S^x \in \langle \mathbf{local}(p, S) \rangle_E$ et $x \neq 1$, alors $S \notin E'$ et c'est la règle $local2$ qui s'applique. On déduit le résultat de l'hypothèse de récurrence comme pour l'autre cas.

■

Annexe C

Règles de la sémantique opérationnelle pour les évènements attendus

Nous décrivons maintenant les ajouts faits aux règles de la sémantique opérationnelle de chaque opérateur pour permettre le calcul des évènements attendus à chaque instant. Nous gardons les mêmes conventions de nommage que dans la section 4.4 du chapitre 4. Dans la suite, AWAITED dénote l'évènement de l'opérateur correspondant aux évènements attendus dans l'instant, et les évènements AWAITED de ses arguments sont paramétrés par les noms des arguments. Comme déjà mentionné dans la section 5.4.1, l'évènement I_a est l'évènement émis quand l'évènement d'entrée I est attendu.

Opérateur nothing

L'opérateur **nothing** n'a pas d'équation supplémentaire car il ne génère pas d'évènements attendus.

Opérateur wait S

Dans cet opérateur, nous ajoutons l'équation :

$$S_a = \text{AWAITED} \boxplus \text{REG}\boxdot \sim S \boxdot \neg\text{KILL}$$

Notons que la formule des évènements attendus de l'opérateur peut ne pas être vide, mais elle peut dépendre du second argument d'un **seq**, dont le premier argument est **wait**.
Pour tous les éléments w_a de la formule des évènements attendus de l'opérateur, nous ajoutons l'équation :

$$w_a = \text{REG}\boxdot \sim S \boxdot \neg\text{KILL}$$

De plus, nous ajoutons w_a à la liste des évènements attendus impliqués par S_a.

Opérateur p_1 seq p_2

Cet opérateur passe l'évènement AWAITED à ses arguments. Nous ajoutons les équations suivantes :

$$\begin{aligned} \text{AWAITED}_{p_1} &= \text{AWAITED} \\ \text{AWAITED}_{p_2} &= \text{FINISH}_{p_1} \boxdot \sim \text{FINISH}_{p_2} \end{aligned}$$

AWAITED_{p_2} dépend de FINISH_{p_1}, car le second argument de **seq** commence lorsque le premier a terminé.

Opérateur parallel ($p_1 \| p_2$)

L'opérateur **parallel** ne fait que transmettre son AWAITED à ses arguments, les équations suivantes sont ajoutées :

$$\text{AWAITED}_{p_1} = \text{AWAITED}$$
$$\text{AWAITED}_{p_2} = \text{AWAITED}$$

Opérateur while *cond* {p}

Voici les ajouts pour **while** :

$$\text{AWAITED}_p = cond \boxdot \text{START} \boxplus cond \boxdot \neg\text{START} \boxdot \text{FINISH}_p$$
$$cond_a = cond \boxplus \text{AWAITED}$$

AWAITED_p dépend de la condition *cond* puisque qu'il cesse de s'exécuter quand elle n'est plus 1. Il est mis à 1, dans le premier instant quand START est à 1 et ensuite à chaque fois que p termine. L'évènement $cond_a$ est émis quand **while** reçoit l'évènement AWAITED et tant que *cond* est à 1.

Opérateur emit S

Cet opérateur ne calcule le statut d'aucun évènement attendu, son système d'équations ne bouge donc pas.

Opérateur stop {p} when S

L'opérateur de préemption transmet son évènement AWAITED à son argument et il calcule aussi l'évènement attendu associé à son évènement de préemption. Son système d'équations a un registre REG. Voici les équations ajoutées :

$$\text{AWAITED}_p = \text{AWAITED} \boxplus \text{REG} \boxdot \sim S \boxdot \sim \text{FINISH}_p$$
$$S_a = \text{AWAITED} \boxplus \text{REG} \boxdot \sim S \boxdot \sim \text{FINISH}_p$$

AWAITED_p et S_a ont la même équation. Ils sont à 1 quand l'opérateur reçoit son AWAITED, dans le premier instant. Ensuite, ils sont à 1 quand l'automate de l'opérateur n'est plus dans son instant initial et que les conditions de terminaison de l'opérateur ne sont pas à 1.

Opérateur if *cond* then {p_1} else {p_2}

Concernant les évènements attendus, **if..then..else** gère à la fois les AWAITED de ses arguments et l'évènement qui correspond à l'attente de sa condition. Ainsi, l'ajout est le suivant :

$$\text{AWAITED}_{p_1} = \text{AWAITED} \boxdot cond$$
$$\text{AWAITED}_{p_2} = \text{AWAITED} \boxdot \sim cond$$
$$cond_a = \text{AWAITED}$$

La transmission de l'évènement AWAITED aux arguments est naturelle. Si la conditon est à 1, on transmet à p_1, sinon on transmet à p_2. Comme cet opérateur est instantané, $cond_a$ est attendu au premier instant.

Opérateur p timeout S $\{p_1\}$

timeout gère les AWAITED de ses deux arguments et aussi l'évènement créé pour représenter l'attente de son évènement de préemption. Son système d'équations possède deux registres qui encodent les différents états de son automate. Les ajouts à son système d'équations sont :

$$\begin{aligned}
&\text{AWAITED}_p = \neg\text{REG} \boxdot \neg\text{REG}_1 \boxdot \text{AWAITED} \boxplus \text{REG} \boxdot \neg\text{REG}_1 \boxdot \sim S \boxdot \sim \text{FINISH}_p \\
&\text{AWAITED}_{p_1} = \text{REG} \boxdot \neg\text{REG}_1 \boxdot \sim S \boxdot \text{FINISH}_p \\
&S_a = \text{AWAITED} \boxplus \text{REG} \neg\text{REG}_1 \boxdot \sim S \boxdot \sim \text{FINISH}_p
\end{aligned}$$

Dans l'instant initial de l'automate de l'opérateur, l'évènement AWAITED du premier argument est activé par l'AWAITED de l'opérateur. Dans un instant suivant (caractérisé par $\text{REG} \boxdot \neg\text{REG}_1$) , il est activé tant que S n'est pas à 1 et que ce premier argument ne termine pas. Dans ce même instant, l'AWAITED du second argument est activé lorsque S n'est pas à 1 et que le premier argument termine. Enfin, l'évènement représentant l'attente de S est activé dans le premier instant avec l'AWAITED de l'opérateur ou bien, toujours dans l'état $\text{REG} \boxdot \neg\text{REG}_1$, si S n'est pas à 1 et le premier argument ne termine pas.

Opérateur local (S) $\{p\}$

Cet opérateur transmet juste son évènement AWAITED à son argument. S étant un évènement local, aucun évènement d'attente n'est généré pour lui. Les ajouts à son système d'équations sont :

$$\text{AWAITED}_p = \text{AWAITED}$$

Annexe D

Questionnaire sur le langage ADeL

Nous montrons dans cette annexe toutes les questions posées au public pour évaluer l'acceptabilité du langage en ses deux formats. La première partie des questions nous a permis de déterminer le profil et le niveau d'expertise en informatique du public concerné. La deuxième partie est dédiée à *valider* notre approche concernant le langage ADeL.

Questionnaire pour la conception d’un outil

Dans le cadre de la création d’un outil logiciel de description de scenarios pour le suivi de patients, nous avons besoin de vos réponses à notre questionnaire pour mieux définir de vos besoins.
Cela vous prendra moins de 10 minutes, et vos réponses nous aiderons à améliorer l’outil en cours de conception. Nous vous remercions pour votre participation.

Profession:
Pays:
Age: ☐ entre 20 ans et 40 ans
☐ entre 41 ans et 55 ans
☐ entre 56 ans et 65 ans
☐ plus que 65 ans

Utilisation des nouvelles technologies

1/ Quels équipements technologiques utilisez-vous?
☐ PC (mobile/fixe) - Si oui, depuis quand?……...
☐ Tablette - Si oui depuis quand?……..…
☐ Smartphone - Si oui depuis quand?……...
☐ Bracelet/Montre connecté(e) - Si oui depuis quand?…….
☐ Autres (merci de préciser) - Si oui depuis quand?…….

2/ Connaissez-vous le système d’exploitation de votre PC?
☐ Oui, si oui quel est-il?……...
☐ Non

3/En moyenne, combien de temps par jour passez-vous à utiliser votre PC?
☐ Moins d'1 fois par jour
☐ Quelques minutes
☐ Quelques heures
☐ "En permanence"

4/ Est-ce que vous vous sentez à l'aise avec l’utilisation de votre PC?
☐ Très à l'aise
☐ Plutôt à l'aise
☐ Cela m'est indifférent
☐ Pas vraiment à l'aise
☐ Pas du tout à l'aise

5/ Etes-vous intéressé(e) par les nouvelles technologies et l'informatique en général ?
☐ Très intéressé
☐ Plutôt intéressé
☐ Cela m'est indifférent
☐ Pas vraiment intéressé
☐ Pas du tout intéressé

6/ Avez-vous des connaissances en langages de programmation informatique?
☐ Oui, si oui quels langages?……
Niveau d’expertise: débutant, intermédiaire ou avancé?……
☐ Non

7/ Est-ce que vous préférez utiliser vous-même des outils informatiques de manière autonome sans vous faire aider par un informaticien ?

☐ Oui
☐ Non

Description de scénarios d'activités

8/ Supposons qu'on souhaite décrire le scénario d'activité suivant à l'aide d'un langage textuel:
Une personne agée tombe dans son salon, si elle ne se relève pas dans les 3 minutes, on lance une alerte de danger.
Ci-jointes plusieurs descriptions de ce scénario en plusieurs langages, veuillez choisir qui vous parait le plus facile à comprendre:

1/ Langage1

☐
```
Activity exemple (patient : Person, salon : Zone)
Events
tombe(Person, Zone)
se_releve(Person)
 bouge(Person)
InitialState : dans_zone(patient, salon);
start
        tombe(patient, salon)
        seq
        se_releve(patient) timeout 3.0 min
        {
          bouge(patient)
        } alert danger
end
```

Pourquoi?……

☐ ***2/ Langage2***

```
module exemple:
  input: tombe_patient_zone, se_leve_patient, bouge_patient, timeout_3_minutes;
  output: danger;
      await tombe_patient_zone;
        abort
            await se_leve_patient
        when timeout_3_minutes ;
        present timeout_3_minutes then emit danger else await bouge_patient
       end
```

Pourquoi?……

☐ ***3/ Langage3***

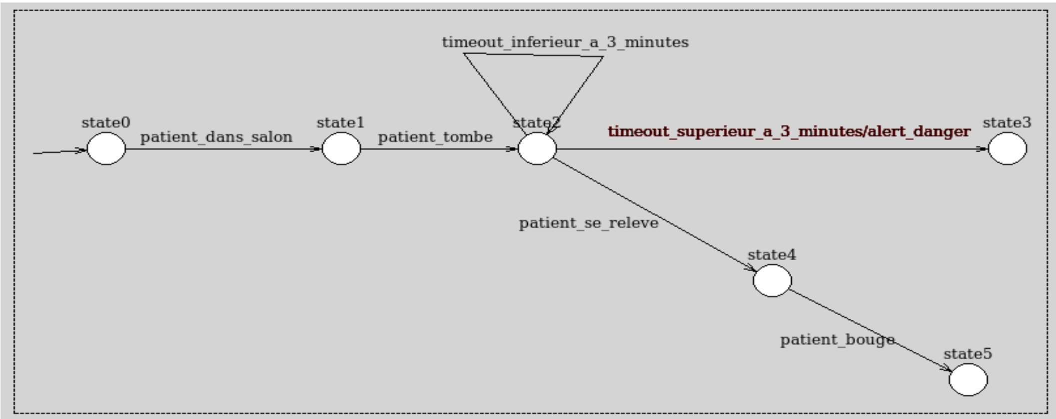
```
node exemple (tombe_patient_zone, se_leve_patient,  bouge_patient, timeout_3_minutes: bool)
returns (danger, ok :bool);
  var est_tombe : bool;
    let
      est_tombe = false -> if (tombe_patient_zone) then true else pre(est_tombe);
       danger = est_tombe and not se_leve_patient and timeout_3_minutes;
       ok = est_tombe and se_leve_patient and not timeout_3_minutes and bouge_patient;
    tel
```

Pourquoi?……

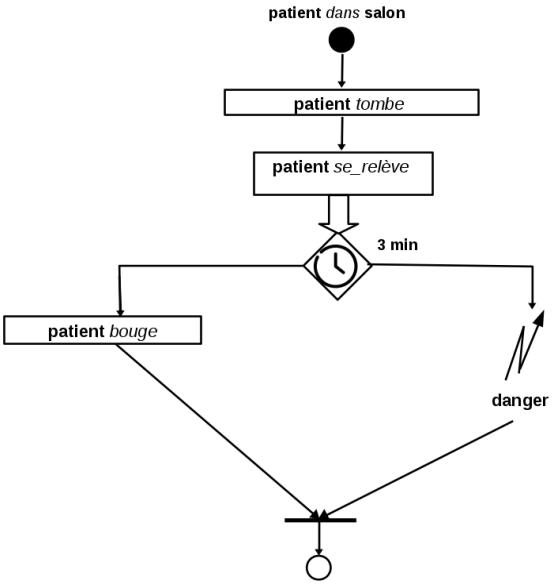
☐ **4/** Si ces langages vous paraissent compliqués, pouvez-vous proposer un exemple de formalisme qui vous semblerait plus simple:
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
……
…….

9/ On peut décrire graphiquement la même activité sous deux formes différentes, choisissez celle qui vous parait plus facile à comprendre et utiliser:

1/ 1er format:



2/ 2ème format

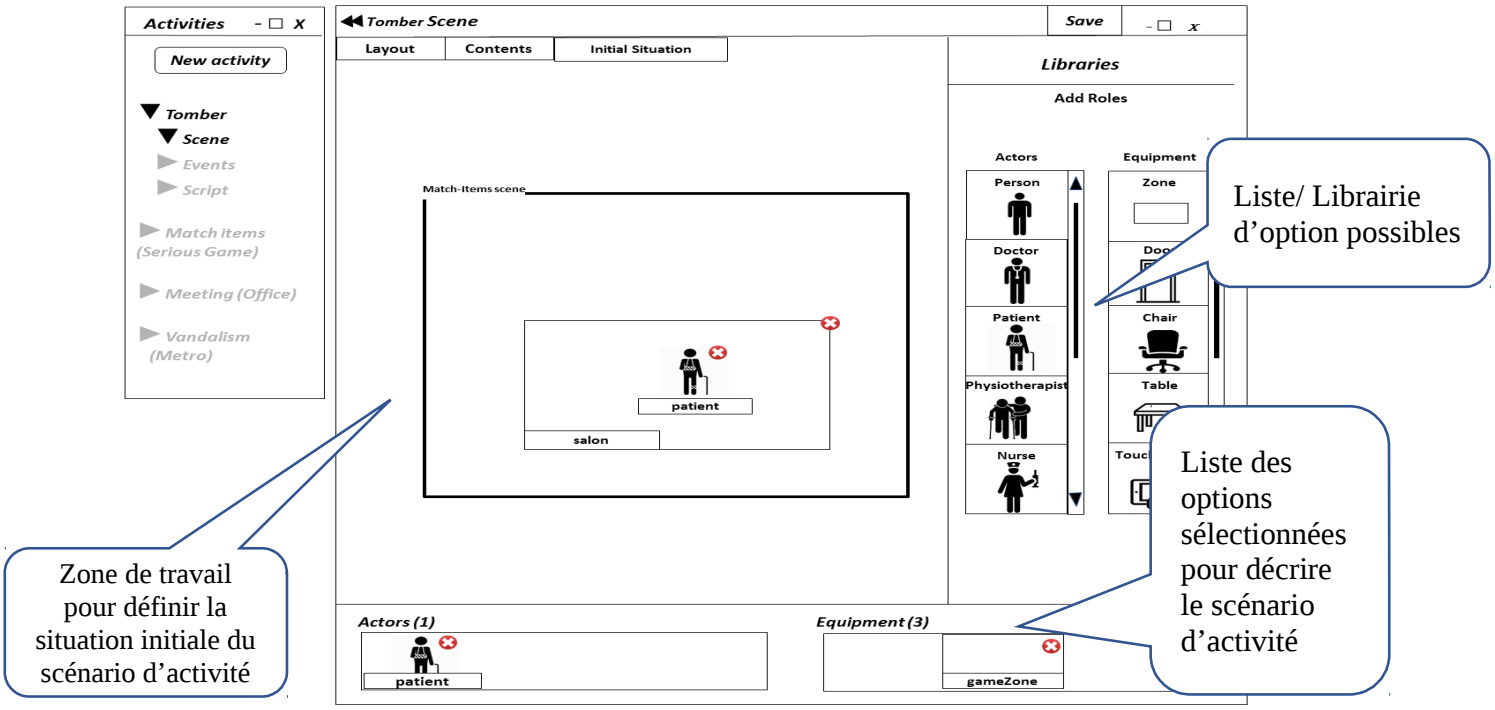


Pourquoi?……

10/ Si on vous donne un outil pour définir les scénario, est-ce que vous préférez que cela soit avec un langage graphique ou textuel ?

☐ Graphique, Pourquoi?……
☐ Textuel, Pourquoi?……
☐ Les deux, Pourqoui?……

11/ Dans un scénario, pour exprimer le fait qu'un patient est dans le salon par exemple, nous pouvons vous proposer cette interface:



Dans cette interface, on peut définir les rôles qui vont participer au scénario de l'activité, les équipements nécessaires et les zones, l'ajout se fait à l'aide d'un "drag and drop".

Cette interface vous parait elle facile à utiliser ? Veuillez donner une note entre 0 et 5 pour représenter la facilité de cette interface en justifiant votre note: (0 : pas facile / 5: très facile)
……
……
……

12/ Est ce qu'un logiciel capable de dessiner et de représenter graphiquement des scénario comme ceux des questions précédentes, peuvent vous aider dans votre travail ?

- ☐ Oui, Pourquoi?…….
- ☐ Non, Pourquoi?…….
- ☐ Peut-être, Pourqoui?……..

13/ Avez-vous des commentaires ?
……
……
……
……
……
……
……
……
……
……
……
……
……

Merci!