Test fonctionnel de propriétés hybrides

Yves Grasland

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Yves Grasland

Thèse dirigée par **Ioannis Parissis**
et codirigée par **Roland Groz**

préparée au sein des laboratoires **LCIS et LIG**
et de l'école doctorale **MSTII**

Test fonctionnel de propriétés hybrides

Thèse soutenue publiquement le **15(?)/02/2013**,
devant le jury composé de :

M. Ioannis Parissis
Professeur, Grenoble INP, Directeur de thèse
M. Roland Groz
Professeur, Grenoble INP, Co-Directeur de thèse
Mme Lydie du Bousquet
Professeur, Université Joseph Fourier, Encadrante de thèse
M. Frédéric Boniol
Professeur, Université de Toulouse, Rapporteur
Mme Fatiha Zaïdi
Professeur, Université Paris-Sud XI, Rapporteur
M. Yves Le Traon
Professeur, Université de Luxembourg, Examinateur

“Soyons fermes, purs et fidèles ; au bout de nos peines, il y a la plus grande gloire du monde, celle des hommes qui n’ont pas cédé. " *(C. de Gaulle)*

Remerciements

Tout au long de la préparation de cette thèse, j'ai pu compter sur une équipe d'encadrement qui m'a énormément soutenu. Mes amis m'ont souvent demandé si avoir trois encadrants, ce n'était pas trop compliqué. Si les divergences de point de vue ne créaient pas de difficultés. Au terme de cette expérience, je réponds sans hésiter : non. Les qualités de Ioannis, Roland et Lydie se sont complétées à merveille pour former une équipe avec qui travailler pendant trois ans a été une chance et un honneur. Tous trois m'ont apporté énormément sur le plan scientifique par leurs conseils précis et rigoureux, et si leurs points de vue ont été différents par moments, cela était source de richesse, non de conflits. Je vous remercie tous trois chaleureusement pour m'avoir apporté tout cela.

Cette thèse fut également enrichissante sur le plan relationnel et humain ; a ce titre je voudrais remercier en particulier Lydie pour sa franchise, sa générosité et ses conseils plein de bon sens qui, avouons-le, m'ont souvent été plus utiles que je ne l'aurais souhaité !

Je remercie également Ioannis pour savoir si bien allier un caractère ouvert et une inaltérable bonne humeur à un sens de l'organisation sans faille. Les moyens qu'il a pu mettre à ma disposition pour me permettre de m'intégrer à Valence notamment m'ont permis de réaliser mes travaux dans des conditions matérielles et humaines excellentes.

Je remercie Frédéric Boniol, Yves Le Traon et Fatiha Zaïdi, pour l'intérêt qu'ils ont porté à ces travaux et leur participation à mon jury.

J'ai aussi passé trois ans mémorables grâce à l'équipe du département informatique de l'IUT de Valence où j'ai effectué mon service d'enseignement, et en particulier grâce à mon tuteur d'enseignement Damien ; je les remercie pour la confiance et l'accueil si chaleureux qu'ils m'ont réservé. J'ai pu mener à l'IUT mes activités d'enseignement impliquant des sujets passionnants et des responsabilités qui sont assez rarement offertes aux doctorants, dans une ambiance solidaire et très agréable.

Tout ce travail n'aurait pas pu se faire sans l'aide des personnels techniques et administratifs de Grenoble et Valence ; grâce aux secrétaires des laboratoires et de l'école doctorale, en particulier Ghislaine, Pascale, Béatrice, Carole, Jennifer, Dominique et Alice, toujours aimables et efficaces, aux ingénieurs du LIG et du LCIS, et bien sûr à Laurent Perge de Grenoble INP qui a habilement réécrit l'histoire d'un trait de plume.

J'ai eu aussi beaucoup de plaisir à travailler dans deux laboratoires vivants que sont le LIG et le LCIS, où tous les moments passés autour d'un café, d'un déjeuner ou simplement quelques mots et sourires échangés dans les couloirs créent une ambiance qui font de ces lieux des endroits où on a plaisir à venir travailler et échanger. J'ai eu beaucoup de plaisir à côtoyer mes voisins, permanents et non-permantents, et à établir ici ou là une relation privilégiée avec certains d'entre vous. Ces trois ans passés à Grenoble et Valence me laissent de ce point de vue de très bons souvenirs.

Je remercie enfin ma famille et mes amis qui, en-dehors de ces travaux de thèse somme toute relativements prenants, ont toujours été présents, compréhensifs et d'un grand secours pour conserver le cap lorsque les préoccupations s'amoncelaient.

Enfin, j'ai une pensée pour le Alice au Pays des Merveilles de Tim Burton, que je ne pourrai plus jamais revoir de la même façon. Et je dois énormément à la capricieuse mais irremplaçable D.M. qui a toujours été présente dans les périodes difficiles. Ce travail n'aurait jamais abouti sans elle.

Table des matières

Chapitre 1

Introduction

On nomme systèmes hybrides les systèmes qui intègrent des comportements continus et discrets. Leur comportement est en partie déterminé par des contraintes temporelles et/ou leur interaction avec un environnement physique via l'utilisation de capteurs et d'actionneurs. Il peut s'agir par exemple d'équipements de contrôle de procédés industriels [39] ou de services domotiques [3].

La validation de ces systèmes est un problème crucial, car ils sont parfois utilisés au sein d'applications où la sûreté de fonctionnement est critique (un système critique est un système dont la défaillance peut entraîner des blessures ou la mort d'êtres humains). Cette validation soulève plusieurs problèmes ; en particulier, une description détaillée du système à valider n'est pas toujours disponible, ou celle-ci est trop complexe pour qu'il soit possible en pratique d'établir avec certitude sa correction (par exemple parce que cette vérification exhaustive nécéssite des ressources en temps ou en matériel trop importantes).

Dans ces conditions, nous nous intéressons à la validation de systèmes hybrides spécifiés par un ensemble de propriétés de sûreté. Une propriété de sûreté est une exigence portant sur le comportement du système qui ne contraint pas entièrement ce comportement, mais en spécifie des aspects particuliers. Elle peut ainsi porter sur la réaction attendue du système en réponse à une suite d'événements donnée, ou imposer des limites à l'évolution des grandeurs environnementales qu'il contrôle.

Le travail présenté dans ce document consiste en une approche de test fonctionnel, destinée à permettre la validation de systèmes hybrides pour lesquels on dispose d'une spécification composée de propriétés de sûreté. La validation consiste à vérifier que le système satisfait à sa spécification, et donc aux besoins qui motivent sa création (sous réserve que la spécification traduise correctement les besoins). Ce chapitre introductif présente les systèmes hybrides, et les propriétés portant sur ces systèmes que nous considérons. Il présente ensuite l'activité de test logiciel dans le contexte hybride, les problèmes qui se posent pour sa mise en pratique et la contribution que nous apportons. L'organisation de la suite du document est finalement présentée.

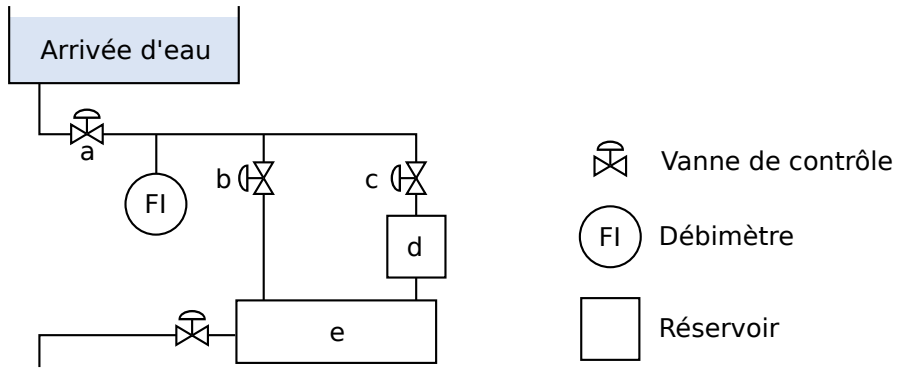


FIG. 1.1 – Capteurs et actionneurs d'un système hybride contrôlant un flux d'eau dans un lave-linge

1.1 Présentation des systèmes hybrides

Les systèmes hybrides sont ainsi nommés car leur comportement allie des éléments discrets et continus [33]. Considérons un exemple, illustré par le schéma en Fig. 1.1 : un sous-système intervenant dans un lave-linge gère l'alimentation en eau du tambour (le réservoir e sur le schéma). Le remplissage du tambour peut avoir lieu pour deux raisons ; lorsque le lavage démarre, le sous-système reçoit un événement l'en informant et il active les vannes a (par laquelle arrive l'eau depuis le réseau de distribution) et c ; l'eau traverse alors le réservoir d qui contient le produit destiné au lavage avant d'arriver dans le tambour. En phase de rinçage, un événement différent est envoyé au sous-système, qui entraîne l'ouverture des vannes a et b, et l'eau arrive directement dans le tambour.

Le sous-système de contrôle vérifie par ailleurs que le remplissage se déroule correctement. Il ferme les vannes et envoie un signal d'erreur si l'eau n'arrive pas dans la machine ou si le remplissage est anormalement long. Il ferme les vannes et envoie un signal d'acquittement si le remplissage s'est déroulé correctement et que le bon volume d'eau a été introduit (il est à noter que cette propriété porte sur le débit introduit, et pas sur un niveau d'eau atteint dans le tambour, et n'est donc pas conçue pour détecter d'éventuelles fuites). Pour évaluer laquelle de ces deux situations survient, il est nécessaire de connaître l'évolution du volume d'eau introduit. Le système la calcule en exploitant les données remontées par un débimètre FI situé derrière la vanne d'admission d'eau.

Le système de contrôle d'alimentation en eau possède une logique de commande partiellement discrète. Dans notre exemple il y a ainsi plusieurs modes de fonctionnement possibles pour ce système : en attente, interrompu pour cause de défaillance du remplissage, ou en cours de remplissage. Il existe en outre des transitions entre ces modes, qui sont pour certaines déclenchées par la réception d'événements discrets, ou qui en produisent. Cette partie du système peut être formalisée par un formalisme de machine à états finis classique.

D'autre part, l'évolution du volume d'eau introduit est égal à l'intégrale au cours du temps des valeurs de débit mesurées sur l'environnement par le débimètre. Elle peut être modélisée au sein d'un système dynamique, que nous qualifions de continu car il exprime l'évolution d'une grandeur réelle et continue presque partout au fil du temps. La présence de ce comportement continu lié au comportement discret mentionné au paragraphe précédent confère à ce système un caractère hybride.

1.2 Spécification et propriétés

La spécification d'un logiciel regroupe un ensemble d'exigences qu'il doit satisfaire, et raffine avec précision un ensemble de besoins sous-jacents qui étaient initialement exprimés de manière informelle. Les activités de *vérification* et de *validation* visent à s'assurer qu'un logiciel respecte cette spécification et produit bien le comportement attendu. Nous exposons ici notre positionnement par rapport à ces deux problématiques.

1.2.1 Vérification et validation de logiciels

La vérification d'un logiciel ou d'une partie de logiciel est *le processus consistant à évaluer un système ou un composant pour savoir si les produits d'une phase de développement logiciel donnée satisfont les exigences établies au début de cette phase* [1] (cette satisfaction peut être établie avec certitude ou relativement à un degré de confiance minimal admissible, suivant la technique employée). Cette évaluation permet de s'assurer qu'on a "bien construit le système".

L'activité de validation consiste par contraste à *évaluer un système ou un composant pendant le processus de développement ou à la fin de celui-ci pour déterminer s'il satisfait les exigences spécifiées* [1]. Il est généralement admis [8] que ces exigences sont avant tout liées ici aux besoins sous-jacents à la création du logiciel et à son usage prévu, et indirectement à la spécification, qui détaille ces besoins. Il est en effet possible que les besoins initiaux aient été décrits de façon imprécise ou incomplète dans la spécification et que le logiciel, quoique conforme à cette spécifcation, ne remplisse pas sa fonction désirée. Il s'agit donc de s'assurer qu'on a "construit le bon système".

Les présents travaux traitent de la *vérification* de propriétés. Toutefois nous ne posons pas la question de l'existence de possibles écarts entre les propriétés et les besoins associés ; nous admettons que les deux coïncident et utiliserons donc parfois également le terme de validation.

Ces problèmes sont cruciaux pour les systèmes hybrides, car ils sont parfois utilisés au sein d'applications où la sûreté de fonctionnement est critique. De telles applications incluent le contrôle de réactions chimiques industrielles, les Smart Grids (c'est-à-dire les applications de gestion décentralisée des réseaux de distribution d'énergie) ou encore des applications de maintien à domicile de personnes âgées. Une défaillance du système peut y entraîner des coûts humains ou financiers inacceptables. Dans ce contexte hybride, les activités de vérification et de validation sont soumises à plusieurs contraintes spécifiques ; la première d'entre elles concerne l'établissement de la spécification.

1.2.2 Influence de la précision de la spécification

L'établissement d'une spécification précise d'un système est une étape essentielle : elle conditionne ce qu'il est possible d'attendre de ce système une fois celui-ci vérifié. Elle contraint le service rendu par le système, qui doit répondre au besoin sous-jacent motivant sa création. Le niveau de détail de cette spécification est donc une question importante. Si elle est trop lâche ou imprécise elle peut conduire à l'acceptation de systèmes qui ne fourniront pas le service attendu. Trop précise, elle peut conduire au rejet de systèmes conformes aux besoins qui ont motivé leur création.

Considérons l'exemple du contrôleur d'eau ; notre description informelle en 1.1 mentionne une durée maximale de remplissage. Nous la noterons t_M, fixée à 5 minutes, avec une tolérance. Si la spécification fixe cette tolérance à plus ou moins 4 minutes, et que la durée totale du cycle doit être connue à 3 minutes près pour satisfaire les utilisateurs, alors un système peut être valide au regard de la spécification mais ne pas fournir le service escompté. Au contraire, si cette tolérance est de plus ou moins 1s, alors une machine précise à 5s près pourra être déclarée invalide alors qu'elle rend un service utile.

De façon similaire si, dans notre exemple, la spécification omet de mentionner le temps maximal de remplissage (une telle omission résultant par exemple d'un oubli), alors elle ne capture plus tous les éléments du besoin qui a motivé la conception du système. La vérification perd alors de sa pertinence car elle n'est pas basée sur une spécification complète et ne porte donc pas sur toutes les fonctions attendues du système. Au contraire, écrire dans la spécification que les composants physiques du système doivent être de couleur noire alors que ce point n'a aucune importance pour les utilisateurs du système peut conduire au rejet d'un système au contrôleur d'eau de couleur différente, alors qu'il répond en tous points aux besoins de ses utilisateurs.

Ces deux exemples ont montré que le niveau de détail d'une spécification devait être soigneusement choisi tant sur le plan quantitatif (les tolérances numériques) que qualitatif (ce sur quoi portent ces tolérances). Dans certains contextes applicatifs, ce niveau de détail est contraint et ne peut pas être choisi arbitrairement.

A titre d'exemple, on peut considérer les architectures à services qui peuvent être utilisés pour implémenter des systèmes hybrides, notamment domotiques [51, 23]. Ces architectures sont constituées de composants remplaçables, dont les détails d'implémentation ne sont en général pas connus. Ils sont spécifiés par des contrats qui décrivent leur fonctionnalité, et donc un besoin sous-jacent. Leurs spécifications sont génériques, ce qui laisse une grande liberté d'implémentation et facilite donc le développement de composants interchangeables pour une même fonction.

Une collection de *propriétés*, associées chacune à un besoin particulier avec un niveau de détail adapté, permet de spécifier ce type de système. Nous allons présenter ces propriétés et présenter la façon dont nous les employons pour spécifier les systèmes hybrides.

1.2.3 Notion de propriété de sûreté

Les propriétés peuvent être classifiées en deux types [4, 43]. Les propriétés de *vivacité* expriment le fait qu'un événement souhaitable advient dans le futur (sans limitation de durée). Les propriété de *sûreté* caractérisent le fait qu'un événement indésirable ne survient jamais. Une propriété qui exprime le fait qu'un événement souhaitable doit arriver dans un temps *limité* est ainsi un cas particulier de propriété de sûreté (on peut en effet la formuler autrement : il n'arrive jamais qu'on ait dépassé *la date limite* d'apparition de l'événement sans l'avoir observé).

Nous définissons pour nos travaux une spécification comme un ensemble d'une ou plusieurs propriétés de sûreté indépendantes. Nous les formalisons en utilisant la théorie des automates hybrides à entrées/sorties de Lynch [40]. Pour illustrer ceci et motiver notre choix de formalisme, nous allons présenter la spécification de l'exemple du système contrôleur d'eau ; elle contient une unique propriété de sûreté qui regroupe et précise les éléments fonctionnels décrits informellement en 1.1.

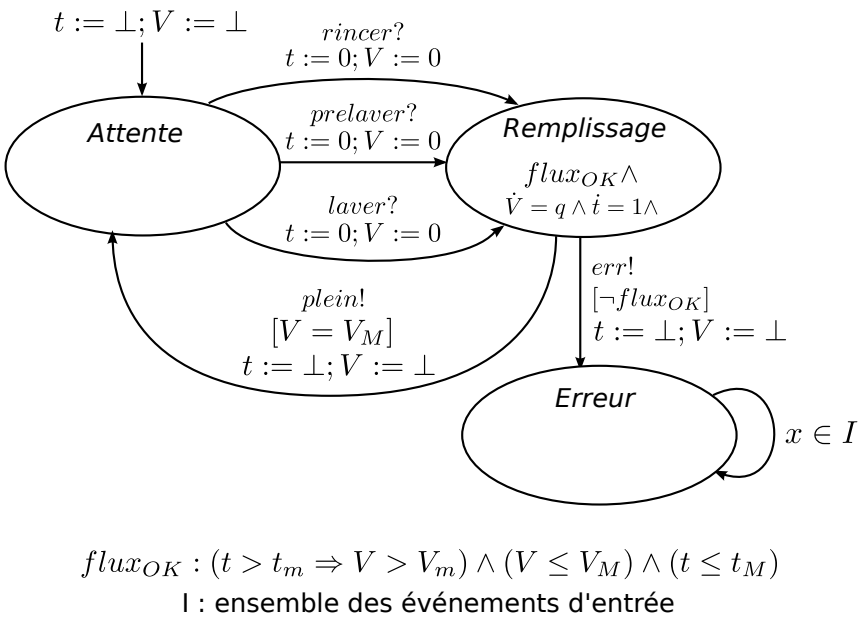


FIG. 1.2 – Propriété de sûreté hybride spécifiant le système de contrôle de l'eau.

1.3 Les automates hybrides comme formalisme pour les propriétés de sûreté

Nous avons mentionné le fait que le comportement d'un système hybride mêle des aspects continus et discrets. Le système dans son ensemble ne peut pas être décrit complètement par l'un ou l'autre de ces éléments seulement. Il existe en effet des interactions entre ces deux dynamiques ; une de ces interactions dans notre exemple survient à la fin d'une période de remplissage : le volume est une variable dont l'évolution est donnée par une équation continue, or lorsque le volume dépasse un seuil prédéterminé cela provoque un changement de mode discret. Les changements de mode discrets déterminent quand à eux l'activation ou la désactivation des vannes, et fixent par conséquent les paramètres de l'équation d'évolution continue du volume.

Les automates hybrides à entrées/sorties permettent de formaliser l'évolution de ce type de système en spécifiant conjointement les aspects continus et discrets, et leurs interactions [40]. Nous donnons ici une première intuition des possibilités de ce formalisme, en l'appliquant à la spécification du système de contrôle de débit d'eau.

La propriété de sûreté qui compose cette spécification regroupe les éléments présentés en 1.1 et s'exprime informellement ainsi : *Un volume d'eau V_M est introduit dans le tambour de la machine en un temps inférieur à t_M en réponse à un ordre de lavage, prélavage ou rinçage, puis un événement d'acquittement est émis et le remplissage cesse ; un volume V_m a en outre été introduit avant la date t_m.* Nous illustrons en Fig. 1.3 l'automate hybride qui la formalise. Les définitions de ces automates, ainsi que la sémantique de la représentation graphique que nous utilisons dans ce manuscrit seront détaillées par la suite ; cette illustration a pour objectif d'en donner une première intuition.

Les ellipses du schéma figurent classiquement les différents *modes* de fonctionnement du système. Mais l'état d'un automate hybride ne se limite pas à ce mode : il est défini par les valeurs que prennent un ensemble de *variables*, qui évoluent au cours du temps (dans ces travaux, nous faisons l'hypothèse qu'une de ces variables représente le mode). La variable t par exemple est un chronomètre qui mesure le temps passé dans le mode $Remplissage$ (elle n'a pas de signification ailleurs). V représente le volume d'eau introduit, et q la mesure fournie par le débimètre. Toutes ces variables prennent leurs valeurs dans $\mathbb{R}$, les variables V et t peuvent en outre prendre une valeur spéciale $\perp$, traduisant le fait qu'elles sont indéfinies.

Pour chaque mode, nous précisons les relations existant entre ces variables, et les contraintes qui portent sur leur évolution temporelle. Dans le mode $Remplissage$ par exemple, t mesure le temps passé afin de pouvoir décrire la limite de temps de remplissage. V représente le volume d'eau introduit, défini comme l'intégrale temporelle des valeurs de q. On remarque que les informations représentées par t et V sont des artefacts introduits par la spécification, qui servent à encoder l'exigence de sûreté, et ne correspondent pas à un signal d'entrée ou de sortie physique du système.

Les possibilités de changements de mode sont formalisées par des transitions. Chaque flèche sur le schéma représente un ensemble d'une ou plusieurs de ces transitions. Ces groupes de transitions sont décorés par l'événement discret, la garde, ainsi qu'une réinitialisation qui leur sont associés. Considérons par exemple une transition entre les modes $Remplissage$ et $Attente$. Lorsqu'elle est empruntée, l'événement $plein$ est émis (le signe ! dénote un événement émis, et ? un événement reçu). Elle ne peut survenir que si la garde (notée entre crochets) est active, c'est-à-dire lorsque le volume V d'eau introduit est égal au seuil V_M à atteindre. Et pour illustrer enfin le concept de réinitialisation, lors d'une transition d'$Attente$ vers $Remplissage$ la valeur de la variable t change et prend la valeur 0 après la transition (afin de pouvoir jouer ensuite son rôle de chronomètre).

On peut remarquer que nulle mention n'est faite des signaux de commande des vannes. En effet la présence de ces vannes est un détail d'implémentation du système sans lien avec l'exigence encodée par cette propriété (elle auraient pu être remplacées par un équipement issu d'une technologie différente et fournir le même service). Réaliser le système en utilisant spécifiquement des vannes n'est pas nécessaire pour réaliser la fonction de remplissage du tambour, par conséquent la spécification ne les mentionne pas. Si la spécification que nous décrivons est utilisée par un intégrateur dans le cadre d'un système composé de services, une seconde spécification, dédiée à *une implémentation particulière*, peut avoir été vérifiée en amont par le concepteur du sous-système de contrôle de l'eau pour valider le comportement des vannes. Nous illustrons ainsi le principe de généricité des propriétés évoqué précédemment.

1.4 Validation des propriétés hybrides par le test

L'objectif de nos travaux est de déterminer comment valider des propriétés hybrides sur un système. Ces propriétés, formalisées par des automates hybrides, expriment un comportement combinant des aspects continus et discrets.

Pour cela, nous employons une technique de test fonctionnel [49]. Le test est une méthode très largement utilisée pour la validation de systèmes dans l'industrie, en particulier en l'absence de modèle explicite et simple du système à valider. Tester un système (dit *système sous test*) par rapport à une spécification consiste à soumettre à ce système des entrées couvrant un ensemble de scénarios « pertinents » vis-à-vis de la spécification, puis à observer les réactions du système afin de vérifier qu'elles sont

conformes aux prédictions de la spécification. L'emploi d'une telle technique pose plusieurs problèmes difficiles que nous allons présenter ici. Nous les précisons ensuite dans le context hybride que nous avons fixé.

1.4.1 Problèmes classiques du test

Une question qui se pose de façon évidente lorsqu'on applique une démarche de test est celle de la *génération des tests*. Il faut choisir un ensemble de cas de tests (i.e. d'entrées) qui seront soumis au système pour en exercer les fonctionnalités. Ces entrées peuvent être choisies en s'appuyant sur des informations disponibles sur le système, par exemple en exploitant un modèle du système ou une spécification. Elles peuvent également être choisies à partir d'un ensemble de tests préexistant, et on parle alors de *sélection* de tests plutôt que de *génération*. Ceci est en particulier utilisé lorsqu'on souhaite re-valider des fonctionnalités existantes du logiciel suite à une modification, on parle alors de *test de non-régression*.

La génération des tests s'interrompt lorsque le système à valider a passé avec succès un jeu de tests *adéquat*. Cette notion d'adéquation caractérise informellement la «qualité» des tests. Cette adéquation et les bases sur lesquelles elle est évaluée doivent être soigneusement définies pour que le processus de test réduise effectivement le risque d'apparition de défaillance et permette d'acquérir une confiance suffisante dans le logiciel une fois celui-ci validé. Cette problématique de l'évaluation de l'adéquation d'un jeu de test en vue d'estimer sa qualité, mais aussi et surtout de décider de l'arrêt du processus de test est dénommée *problème de l'arrêt*. Dans ces travaux, nous ne proposons pas de méthode de génération de tests applicable à n'importe quelle propriété formalisée par un automate hybride en général ; nous présentons des approches existantes de ce problème. En revanche nous posons la question de l'évaluation de l'adéquation d'un jeu de tests appliqué à une telle propriété.

Lors de l'exécution du système sous test, des observations sont faites sur son comportement. Sur la base de ces observations, il faut être capable d'évaluer l'exécution et de délivrer un verdict sur l'issue de l'exécution du test. Ce problème est dénommé *problème de l'oracle*, l'oracle étant le composant ou l'entité qui délivre le verdict. Ceci peut être empêché ou rendu difficile par des facteurs tels qu'une observabilité limitée du système ou une complexité importante de la procédure de décision. Dans ces travaux, nous posons un problème consistant à décider si une exécution du système sous test satisfait aux propriétés qui le spécifient à partir d'observations faites sur les signaux que le système échange avec son environnement.

Enfin, un dernier problème est celui de *l'exécution* des tests. Suivant les contextes différentes difficultés peuvent se présenter liées par exemple à l'instantiation de cas concrets (i.e. exécutables) de tests à partir de cas de test abstraits (définis par rapport à un modèle de haut niveau), la contrôlabilité du système sous test (l'implémentation peut ne pas permettre d'observer tous les comportements que prédit son modèle, utilisé pour décrire les tests) ou encore le coût de cette exécution (dans le cas d'un système réel qui exploite des ressources réelles elles aussi). Les cas d'application présentés dans ce travail n'ont pas nécessité de développements particuliers sur ce point, autres que techniques.

1.4.2 Évaluation des propriétés et test de conformité

Nous avons dans ce travail identifié et traité des problèmes liés à l'évaluation des tests, et plus particulièrement à l'observabilité de l'état de la propriété. Dans le cas des systèmes hybrides ce problème

d'observabilité se pose car certaines variables ou actions dites *internes*, qui sont des artefacts de la spécification, ne peuvent être mesurées directement sur le système sous test car elles ne correspondent à aucun phénomène physique. Ainsi dans l'exemple du contrôleur d'alimentation en eau, la variable V qui caractérise l'intégrale temporelle du débit de l'eau admise n'est pas observable. Ici, la solution est simple puisqu'il suffit d'appliquer une méthode d'intégration numérique à la séquence de valeurs observées pour q afin d'inférer celles de V. Sur des spécifications plus complexes, cette inférence peut être plus difficile voir indécidable.

Une autre difficulté est que la spécification peut faire référence comme dans notre exemple à des concepts tels qu'une dérivée qui impliquent que la spécification utilise un modèle de temps dense, alors que les observations réalisées sont échantillonnées à un nombre de dates fini. L'influence de cette incomplétude des observations sur l'évaluation de la correction d'une exécution est une seconde difficulté dans le cadre du test fonctionnel d'un système concret.

1.4.3 Génération et adéquation des tests

Lorsque la correction d'une trace observée est établie, la ou les exécutions possibles du système correspondant à cette trace sont utilisées pour estimer l'adéquation des tests, c'est-à-dire la qualité de l'exploration par le processus de test de l'espace d'état. Or les propriétés hybrides font intervenir des variables continues dont les domaines peuvent être de taille infinie. Il n'est donc pas possible en général de tester exhaustivement une propriété. Choisir un ensemble d'entrées de taille raisonnable mais malgré tout susceptible de détecter un nombre important de défauts est un problème délicat (un défaut est une différence entre la réalisation du logiciel et ce qui est prévu par sa spécification). Il faut en effet définir sur des bases rigoureuses et non ambiguës la notion informelle de *pertinence* des tests que nous avons utilisée plus haut. Ceci passe par la définition d'un *critère d'adéquation* [64], qui pourra fournir suivant les cas l'assurance que les tests ont permis de valider le système à un niveau minimal asmissible (on parlera alors de *critère d'arrêt*), ou délivrer une évaluation quantitative de la qualité des tests effectués (auquel cas on parlera de *critère de couverture*).

1.4.4 État interne et contrôlabilité des systèmes

Il convient enfin de rappeler que les propriétés hybrides peuvent être génériques. Il est ainsi possible que la spécification par propriétés autorise plusieurs évolutions pour une entrée donnée. Un système qui la respecte peut ne pas être capable de les exhiber toutes. Par suite, il est possible lors du test que certains objectifs requis par le critère d'adéquation considéré ne soient pas atteints. On ne dispose pas forcément d'un moyen d'établir si cela est lié à un processus de génération inadéquat (ou qui doit être exécuté plus longuement), ou bien si cet objectif est inatteignable par l'implémentation testée.

Il est alors tentant de « supprimer » ces objectifs du calcul du critère d'adéquation et de les supposer irréalisables. Ainsi, on évite une cause d'inapplicabilité [63] possible du critère (l'inapplicabilité traduit le fait que le critère ne peut être atteint). Toutefois en supprimant ces objectifs on peut créer de faux négatifs, c'est-à-dire éliminer des objectifs réalisables. Ceci conduit à une surévaluation de l'adéquation des tests, et compromet donc la validité des conclusions de la validation. Savoir si, et dans quelle mesure, on peut évaluer la contrôlabilité du système (c'est-à-dire déterminer les comportements qu'il est effectivement *possible* de le voir exhiber) est donc une question importante.

1.5 Contributions proposées et organisation du document

La contribution proposée par ces travaux se décline en deux points principaux, destinés à fournir les composants essentiels d'une approche de test fonctionnel des propriétés de sûreté hybrides :

- Une méthodologie d'évaluation de propriété hybride définie en temps continu à partir d'observations en temps discret
- Une méthode d'évaluation de l'adéquation d'un jeu de test vis-à-vis d'objectifs prédéfinis, incluant :
  - un formalisme inspiré des profils opérationnels pour formaliser ces objectifs en définissant un critère d'adéquation
  - une mesure de ce critère pour un jeu de tests donné.

Le document est organisé de la manière suivante après la présente introduction : le chapitre 2 formalise précisément le concept d'automate hybride et présente un exemple conducteur un peu plus complexe que celui qui est utilisé dans ce chapitre, qui sera utilisé dans le reste du manuscrit pour illustrer le discours. Différentes contributions existantes en lien avec les travaux sont présentées au chapitre 3. Notre contribution est détaillée dans les chapitres 4 et 5, et validée sur l'exemple conducteur au chapitre 6. Nous terminons sur une conclusion et des perspectives présentées au chapitre 7.

Chapitre 2

Formalisation des automates hybrides et exemple conducteur

Ce chapitre présente le formalisme des automates hybrides à entrées/sorties de Lynch [40] que nous utilisons pour décrire les propriétés hybrides, et dont nous avons donné une intuition au chapitre précédent. Nous y introduisons également un nouvel exemple qui nous servira de fil conducteur par la suite.

Ce chapitre est organisé en deux parties : la première présente l'exemple de manière informelle mais détaillée. Dans la seconde partie, nous présentons le formalisme d'automates hybrides que nous utilisons [40] en l'illustrant à l'aide de l'exemple. Nous précisons également la sémantique des notations graphiques employées dans ce document (par exemple en Fig. 1.3) en la reliant à ce formalisme.

2.1 Exemple conducteur

2.1.1 Un système de maintien de la température anticipatif et écologique

Le système que nous considérons est un système de climatisation dit «intelligent». Sa fonction est proche de celle des systèmes de chauffage à thermostat programmable qui équipent depuis longtemps les habitations : il maintient une température de consigne définie par l'utilisateur, qui change au fil du temps. Il en diffère toutefois sur deux points. Premièrement, il anticipe les changements de température, de manière à ce que la température effective de l'habitat atteigne le niveau décidé par l'utilisateur à une date choisie par lui (là où les systèmes actuels *commencent* à poursuivre une température de consigne *à partir* d'une date choisie par l'utilisateur). Supposons par exemple qu'une température de consigne soit initialement fixée à 18°C. À 15h l'utilisateur impose que la consigne monte à 20°C à 18h. Dans ce cas, le système n'entreprend aucune action immédiatement, conservant sa consigne de 18°C. Il modifie sa commande un peu avant 18h, de sorte que la température de consigne qu'il suit évolue progressivement pour atteindre 20°C à 18h, et s'y maintienne ensuite (la température ambiante minimale suit donc une évolution semblable, à une tolérance près).

La deuxième fonction de ce système qui le distingue d'un système classique est qu'il gère sa consommation d'énergie de façon à consommer préférentiellement l'énergie lorsqu'elle est produite par des

sources renouvelables. Pour cela, le système exploite des données actualisées régulièrement (c'est-à-dire avec un intervalle de rafraîchissement de l'ordre de quelques secondes ou quelques minutes) portant sur la quantité de ce type d'énergie disponible au sein de l'alimentation globale. Les sources d'énergie non renouvelables ne sont sollicitées que lorsque la température ambiante approche d'un seuil dit critique (défini par rapport à la température de consigne). Lorsque la température ambiante est supérieure à ce seuil et en est suffisamment éloignée, seules les sources renouvelables peuvent être exploitées. L'utilisateur échange ainsi une partie de son confort contre des économies d'énergies au coût écologique important.

2.1.2 Gestion des contrats, illustration sur un scénario

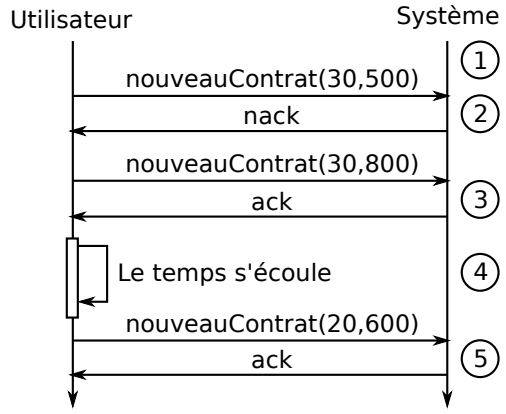
Nous détaillons ici la façon dont la température de consigne suivie à chaque instant par le système est calculée. Cette température est fixée par l'utilisateur via l'émission de contrats ; un contrat est une paire de valeurs comportant une température de consigne et un délai d'exécution. Envoyés à des instants arbitraires par l'utilisateur, les contrats informent le système d'un changement à venir de la consigne ; ils précisent la nouvelle valeur de celle-ci et la date (future) à laquelle elle devra être atteinte. Pour donner l'intuition de la façon dont ces contrats sont traités par le système, nous présentons ici un scénario d'utilisation type.

Ce scénario est illustré en figure 2.1.a. Le diagramme de séquence inclus dans la figure présente la séquence de messages (contrats et acquittements) qui sont échangés par l'utilisateur et le système au cours du temps. Les graphes représentent la consigne de température que le système projette de suivre à différents instants du scénario. Ils ne représentent pas en revanche l'évolution de la température *ambiante* effective ; celle-ci est déterminée par les actions du système et de l'environnement physique.

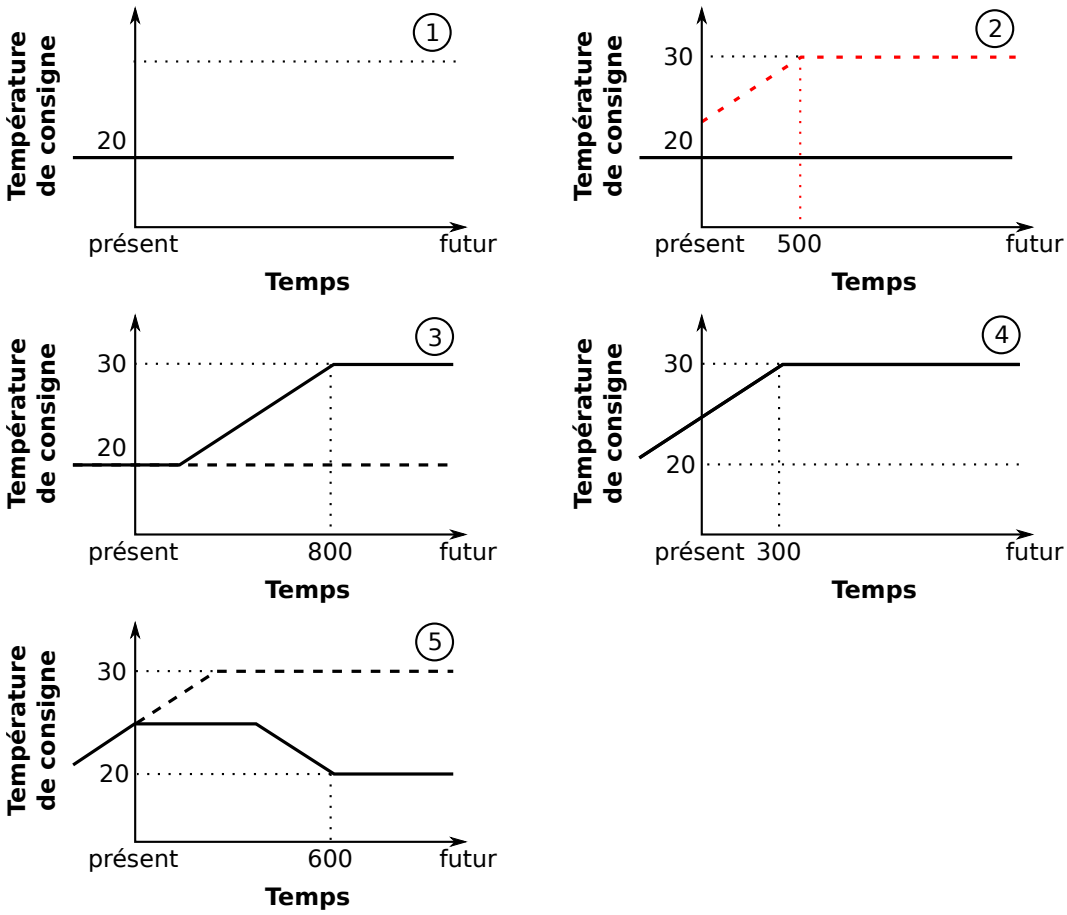
Initialement (cas 1), le système est actif et cherche à maintenir une température minimale de 20°C. L'utilisateur soumet alors un contrat pour une température de consigne de 30°C, température qui devra être atteinte 500 unités de temps plus tard. Pour évaluer la faisabilité de ce changement, le système exploite une connaissance de l'environnement qui lui permet de prévoir sa capacité à influer sur la température de la maison. Ceci n'est raisonnable que si les perturbations de l'environnement physique, telles que celles induites par une fenêtre ouverte, restent dans des limites prédéfinies. Cette supposition est légitime dans un environnent contrôlé, tel qu'une maison à haute qualité environnementale.

La stratégie de chauffe du système consiste à suivre une température de consigne initialement stable, qui croît ensuite linéairement jusqu'à atteindre la valeur fixée par le contrat à son échéance (le modèle exploité par le système lui permet de déterminer quelle est la vitesse de changement maximale qu'il peut garantir). Le contrat peut être soumis trop tard pour qu'une telle stratégie puisse être établie, c'est ce qui se produit pour le premier contrat de notre scénario, ceci est illustré sur le cas 2 de la figure 2.1 : la ligne en pointillés illustre le fait que pour atteindre 30°C 500 unités de temps plus tard, il faudrait que la consigne actuelle soit plus élevée, ou que le contrat ait été envoyé plus tôt. Le système dans ce cas de figure refuse le contrat et conserve sa consigne initiale de 20°C.

L'utilisateur révise alors sa demande et soumet un nouveau contrat, avec un délai de réalisation plus important de 800 unités de temps. Le système calcule une stratégie satisfaisante, illustrée sur le cas 3 ; la ligne en pointillés représente sa stratégie précédente, qui est abandonnée. Si l'utilisateur laisse le temps s'écouler, le système va suivre cette stratégie. Au bout de 500 unités de temps, la consigne qu'il se fixe aura donc changé et se situera entre 20 et 30°C (voir cas 4 de la figure 2.1). Si aucun nouveau contrat supplémentaire n'est envoyé, le système prévoit d'atteindre la consigne de 30 °C 300 unités de temps



a. Diagramme de séquence décrivant le scénario



b. Évolution de la température de consigne au cours du scénario

FIG. 2.1 – Scénario d'exemple pour le système de chauffage anticipatif

plus tard.

Le dernier cas qui peut se produire est qu'un contrat soit reçu alors que la consigne courante du système est en train de changer pour passer d'une valeur à une autre. Le scénario se termine par une situation de ce type : l'utilisateur a changé d'avis et souhaite que la consigne repasse à 20°C, au bout de 600 unités de temps. Le système évalue alors la faisabilité du contrat sur la base de la température de consigne actuelle. Ainsi, comme illustré dans le cas 5 de la figure 2.1, il adopte une nouvelle stratégie consistant à conserver sa consigne actuelle (environ 25°C), puis à la faire décroître afin qu'elle revienne à 20°C au bout de 600 unités de temps au total.

2.1.3 Gestion de l'énergie

Nous avons expliqué ci-dessus comment le système calculait sa stratégie afin de déterminer à tout instant une température de consigne. Nous expliquons ici le traitement qui est fait de cette consigne par le système pour déterminer la consommation énergétique utilisée pour le chauffage.

Comme de nombreux systèmes de chauffage, le système que nous considérons doit maintenir la température ambiante proche de la consigne, avec une tolérance. Cependant, la définition de cette tolérance est ici plus complexe qu'à l'accoutumée car elle doit prendre en compte les exigences exposées en 2.1.1 sur l'utilisation prioritaire des énergies renouvelables. Nous illustrons les différentes situations possibles en figure 2.2 ; cette figure représente une évolution conjointe possible de la température ambiante et de la température de consigne au cours du temps. Chaque période temporelle correspond à une situation particulière, identifiée par un numéro.

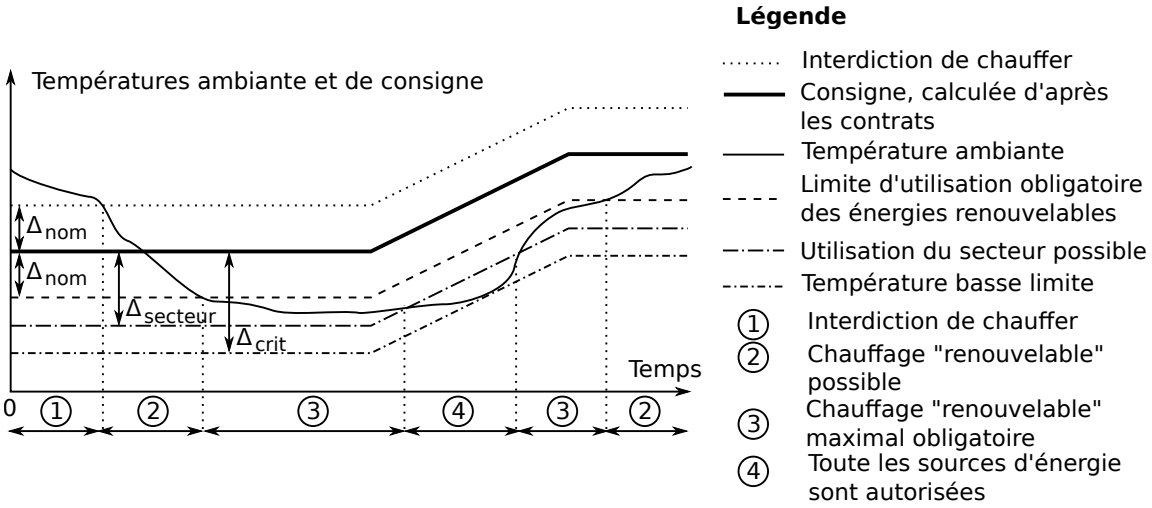


FIG. 2.2 – Traitement de la température de consigne par le système de chauffage

Dans la situation 1, la température ambiante est supérieure à la température de consigne et présente avec elle un écart supérieur à une grandeur Δ_{nom} prédéfinie : le système ne doit pas chauffer dans ce cas. Dans le cas 2, lorsque la température est proche de la consigne (écart inférieur à Δ_{nom}), le système peut chauffer, ou ne pas chauffer. Lorsqu'il chauffe, il ne peut consommer plus que les sources d'énergie renouvelable ne fournissent.

Lorsque l'écart $\Delta = C - T$ entre la consigne C et la température ambiante T est supérieur à un

certain seuil Δ_{nom} et inférieur à $\Delta_{secteur}$ (Cas 3), le système doit chauffer en utilisant la totalité de la puissance renouvelable disponible. Si Δ excède un seuil $\Delta_{secteur}$ (Cas 4), le système devra utiliser au minimum l'énergie renouvelable disponible (et pourra excéder cette limite). Il devra dans tous les cas éviter que Δ ne franchisse le seuil Δ_{crit}.

2.1.4 Spécification considérée

La spécification que nous allons considérer comme exemple conducteur incluera une propriété. Cette propriété est l'union des éléments fonctionnels que nous avons présentés dans la présente section et se formule intuitivement ainsi : *Le système reçoit et évalue les contrats des utilisateurs portant sur la consigne de température à suivre, et accepte ceux qui sont faisables au regard des règles présentées plus haut. Il établit des plans d'évolution de cette consigne pour le futur et les met à jour au fil des soumissions de nouveaux contrats. A chaque instant, il commande le chauffage du bâtiment de manière à respecter les règles précédemment énoncées d'utilisation des énergies renouvelable, en fonction des températures ambiante et de consigne*. Nous allons à présent l'utiliser pour illustrer le formalisme des automates hybrides à entrées/sorties.

2.2 Formalisation générale des automates

2.2.1 Introduction

Pour formaliser les propriétés de sûreté hybrides nous utilisons les automates hybrides à entrées/sorties de Lynch [40]. Nous donnons tout d'abord dans cette section une définition formelle générale des automates. Nous en illustrons ensuite chaque point en formalisant la propriété dont nous avons donné l'intuition en 2.1.4. La sémantique des notations graphiques qui sont utilisées dans ce document pour rendre plus compactes les définitions des automates (par exemple en Fig. 1.3) est également définie. Enfin, nous formulons plusieurs hypothèses de travail qui restreignent la catégorie d'automates que nous considérons.

Nous allons présenter ici les points essentiels du formalisme uniquement. L'article de référence [40] est plus complet et présente notamment de façon plus détaillée des hypothèses de régularité portant sur les intervalles temporels et les trajectoires, ainsi qu'une caractérisation de la notion de composabilité de deux automates. L'exposition détaillée de ces éléments n'étant pas indispensable à la présentation des présents travaux, nous les avons pas rapportés ici.

La définition générale d'un automate hybride un automate hybride, ainsi que celle d'un automate hybride à entrées/sorties (IOHA) sont données ci-dessous :

Définition 1 (Automate hybride)**.** *Un automate hybride est un tuple $\mathcal{A} = (W, X, \Theta, E, H, D, \mathcal{T})$ composé des éléments suivants :*

- *Un ensemble W de variables externes.*
- *Un ensemble X de variables internes disjoint de W. On note $V = W \cup X$.*
- *Un ensemble Θ non vide d'états initiaux.*
- *Un ensemble E d'actions externes.*

- *Un ensemble H d'actions internes disjoint de E. On note $A = E \cup H$.*
- *Un ensemble $D \subseteq val(X) \times A \times val(X)$ de transitions discrètes.*
- *Un ensemble $\mathcal{T}$ de trajectoires dans V.*

Définition 2 (Automate hybride à entrées/sorties)**.** *Un automate hybride à entrées/sorties (IOHA) $\mathcal{A}$ est un tuple (H, U, Y, I, O) où $H = (W, X, \Theta, E, H, D, \mathcal{T})$ est un automate hybride, et U et Y d'une part, ainsi que I et O d'autre part, partitionnent W et E respectivement en variables ou actions d'entrée et de sortie.*

2.2.2 Variables

Le premier composant d'un IOHA que nous décrivons est l'ensemble de ses *variables*. Notre propriété conductrice fait référence à pluieurs grandeurs qui évoluent au fil du temps, telles que la température ambiante. La notion de variable formalise ces grandeurs et leurs caractéristiques au sein d'un IOHA. Une variable v est un élément auquel est associé un ensemble de valeurs appelé *type* et noté $type(v)$, qui caractérise les valeurs qu'elle peut prendre. Soit $Vars$ un ensemble de variables, on nommera *valuation* de $Vars$ toute fonction qui associe à $v \in Vars$ une valeur de $type(v)$. $Val(Vars)$ dénote l'ensemble des valuations de $Vars$ et pour toute valuation x, $x \downarrow \{V_1 \dots V_n\}$ représente sa restriction aux variables $V_1 \dots V_n$.

La propriété que nous utilisons fait référence à des grandeurs de trois natures différentes. La température ambiante par exemple est une grandeur physique mesurée par le système et qui est une entrée. La puissance fournie pour le chauffage est une grandeur physique de sortie. Enfin la température de consigne contrairement aux deux premières n'est liée à aucune quantité physique, c'est un artefact de la spécification que nous avons introduit pour exprimer un besoin. Attacher cette information aux variables est important car elle conditionne le rôle que chacune des variables occupe dans le processus de test. Nous la représentons en exploitant la répartition des variables au sein d'un IOHA en différents ensembles : le premier, noté X, regroupe les variables dites *internes*. Le second, noté U, regroupe les variables d'*entrée*. Le troisième, noté Y, regroupe les variables de *sortie*. Enfin, un ensemble W de variables *externes* est défini comme l'union de U et Y. L'ensemble V des variables de l'automate est égal à l'union de W et X. Une valuation de V, qui associe une valeur à chaque variable de l'automate, sera nommée *état* de l'automate.

La table ci-dessous donne la liste des variables qui sont utilisées dans la formalisation de notre propriété conductrice, assortie d'une description de leur rôle et de leur catégorie :

Variable	Rôle	Catégorie
T	Température ambiante	Entrée
P_{renew}	Puissance renouvelable disponible	Entrée
P_{out}	Puissance utilisée pour le chauffage	Sortie
inD	Nouvelle consigne (paramètre des contrats)	Entrée
$inDL$	Délai accordé (paramètre des contrats)	Entrée
$oldD$	Consigne la plus ancienne	Interne
$hasNextContract$	Présence d'un contrat qui n'est pas encore arrivé à échance	Interne
$newD$	Consigne à venir à l'échéance du contrat courant	Interne
$timeRemaining$	Délai restant pour le contrat courant	Interne
$rampDuration$	Durée de la période de changement de la consigne	Interne
loc	Mode de la propriété	Interne

Toutes ces variables ont un *type* égal à $\mathbb{R}$, à plusieurs exceptions près : $hasNextContract$ est booléenne et prend ses valeurs dans $\{true, false\}$. Les variables $newD$, $timeRemaining$ et $rampDuration$ quand à elles peuvent être indéfinies. En effet, elles caractérisent les paramètres du contrat traité par le système lorsqu'il n'est pas encore arrivé à échance. Aux instants où il n'existe aucun contrat dans cet état, cette information n'existe pas. Nous formalisons cela en assignant pour type à ces variables l'ensemble $\mathbb{R} \cup \{\bot\}$, où $\bot$ formalise une valeur indéfinie. Enfin, une variable loc encode le *mode* de la propriété, c'est-à-dire une phase particulière dans le cycle de fonctionnement du système, parmi un ensemble de phases identifiées par l'écrivain de la spécification. Son type est un ensemble composé des éléments suivants : $\{waitContract, steady, followRamp, validContract, invalidContract\}$. Leur interprétation est la suivante :

$waitContract$: Caractérise le mode actif initialement, lorsque le système n'a encore reçu aucun contrat.

$steady$: Caractérise un mode est de fonctionnement où la consigne reste constante (soit parce qu'aucun contrat à venir n'est défini, soit parce que la transition entre l'ancienne valeur de consigne et la nouvelle n'a pas encore commencé).

$followRamp$: Caractérise le mode où la température de consigne est en train de varier entre son ancienne valeur et la nouvelle.

$validContract$: Caractérise un mode actif lorsqu'un contrat valide (i.e., réalisable dans le temps imparti) vient d'être reçu. C'est un mode où le temps ne progresse pas, il est immédiatement quitté pour le mode $steady$; il a été introduit pour rendre la spécification plus explicite.

$invalidContract$: Caractérise un mode semblable au mode précédent, mais pour les contrats invalides.

La variable loc tient ici un rôle particulier. Elle n'est pas imposée par le formalisme et permet d'introduire dans la spécification la notion de *mode*, qui est comparable à l'état d'une machine à état fini[1]. La présentation que nous avons faite des automates hybrides comme la réunion d'une machine à états finis et d'un système dynamique repose sur la présence de cette variable loc. Il s'agit d'une hypothèse de travail que nous faisons dans ce document, qui n'entraîne pas de perte de généralité et permet à l'usage d'écrire plus facilement des spécifications. En outre, comme le souligne l'article de référence [40], ceci permet de se rapprocher de formalismes hybrides usuels [5] qui mentionnent explicitement plusieurs états discrets.

[1] Nous employons le terme de *mode* au lieu des termes d'*état* ou d'*état discret*, pour ne pas créer de confusion avec la notion d'état des IOHA qui caractérise une valuation de l'ensemble V des variables de l'automate.

2.2.3 Évolutions possibles des variables

Nous avons exposé la notion de *variable* et la notion de *type* qui caractérise le domaine d'une variable. Nous présentons ici la formalisation par les IOHA de l'évolution temporelle des variables, considérées individuellement. Dans notre exemple, les variables P_{out} (qui représente la puissance de chauffe fournie à l'habitation par le système) et T (qui représente la température ambiante du bâtiment) par exemple n'évoluent pas de la même façon : la température représentée par T est une grandeur physique qui évolue continument avec une variation limitée (fonction de l'environnement physique). La puissance de chauffe en revanche peut être discontinue et suivre une évolution quelconque. Les IOHA capturent ces caractéristiques via une notion de *type dynamique* associé à chaque variable, qui caractérise ses évolutions possibles au fil du temps.

Le temps est modélisé au sein d'un automate comme un sous-groupe compact T de $(\mathbb{R}, +)$. On peut ainsi utiliser entre autres les ensembles $\mathbb{Z}$ ou $\mathbb{R}$ pour instancier des automates en temps discret ou dense. Nous nous intéressons dans ce travail aux propriétés formalisées par des automates hybrides utilisant un temps dense ; cela permet d'utiliser pour la spécification des concepts usuels tels que dérivée ou continuité des fonctions. Ce choix d'un temps dense est également présent dans d'autres contributions existantes [5, 20, 24].

Le type dynamique $dtype(v)$ d'une variable v caractérise les évolutions qu'elle est susceptible de suivre au fil du temps, c'est donc une fonction (éventuellement partielle) de T dans $type(v)$. Lynch [40] pose un certain nombre de conditions pour la définition des types dynamiques, en particulier leur invariance temporelle : si, étant donné une variable v, il existe une fonction f de T dans $type(v)$, alors pour tout $t \in \mathsf{T}$ on a $(f + t) \in dtype(v)$, où $(f + t)(t') = f(t + t')$ pour tout $t' \in \mathsf{T}$. Étant donné que ce terme de *type dynamique* peut prêter à confusion, à cause de la signification différente qu'il adopte dans d'autres domaines de l'informatique, nous prendrons la liberté de le remplacer par celui d'*évolutions possibles* d'une variable.

Nous illustrons cette notion en définissant les évolutions possibles des variables de notre exemple conducteur. La variable $timeRemaining$ peut soit décroître linéairement avec une dérivée égale à -1 (lorsqu'un contrat est actif, elle mesure le temps restant avant sa réalisation), soit être constamment indéfinie (si aucun contrat n'est actif). L'ensemble des évolutions possibles de cette variable est donc l'union de l'ensemble des fonctions de dérivée égale à -1 et de la fonction constante égale à $\perp$. Formellement on a $dtype(timeRemaining) = decr \cup \{undef\}$ avec

$$decr = \{g : \mathbb{R} \to \mathbb{R}, \exists g_0 \in \mathbb{R}\ g(0) = g_0 \wedge \dot{g}(x) = -1\ \forall x \in \mathbb{R}\}$$

et

$$undef : \mathbb{R} \to \{\perp\}, \forall x \in \mathbb{R}\ undef(x) = \perp$$

L'ensemble des évolutions possibles de P_{out} regroupe toutes les fonctions de $\mathbb{R}$ dans $[0, P_{Max}]$ avec $P_{Max} \in \mathbb{R}$ puisque comme nous n'avons expliqué précédemment il n'existe pas de condition de régularité particulière sur son évolution (mais cette puissance est nécessairement positive et limitée). Celui de T contient les fonctions continues et k-lipschitziennes de $\mathbb{R}$ dans $[-10; 40]$ avec $k \in \mathbb{R}$, considérant que la température ambiante ne connaît pas de discontinuité, varie lentement et reste dans un intervalle limité. La variable loc enfin, qui représente le mode, est toujours constante. Les autres variables ont des évolutions possibles définies de la même façon que celles qui sont présentées ici.

2.2.4 Trajectoires

La notion d'évolutions possibles concerne *une unique variable*, considérée en isolation. Le formalisme des IOHA permet également de spécifier les conjonctions d'évolutions qui sont autorisées pour *l'ensemble de ses variables*. Ces conjonctions sont appelées *trajectoires*. La formalisation de notre propriété nécessite ce degré d'expressivité pour exprimer les relations existant entre les variables. Par exemple, les variables $newD$, $timeRemaining$ et $rampDuration$ sont toujours soit toutes définies (et s'interprètent comme les paramètres d'un contrat en cours de traitement), soit toutes indéfinies (lorsqu'aucun contrat à venir n'est mémorisé), mais la sémantique de la propriété exclut qu'une partie seulement d'entre elles soient définies. De la même façon, les contraintes portant sur la consommation énergétique du système interdisent certaines combinaisons de valeurs de la variable P_{out} et de la consigne (calculée à partir des valeurs de $oldD$, $hasNextContract$, $newD$, $timeRemaining$ et $rampDuration$), ce qui contraint les trajectoires autorisées au sein de l'automate.

Un automate hybride formalise ceci sous la forme d'un de ses composants qui est un ensemble de trajectoires $\mathcal{T}$; chacune d'entre elles est une fonction définie sur un intervalle de T fermé à gauche dont la borne inférieure et 0, et à valeur dans $Val(V)$. La projection $tr \downarrow v$ d'une trajectoire tr sur une variable $v \in V$ est une des fonction de $dtype(v)$.

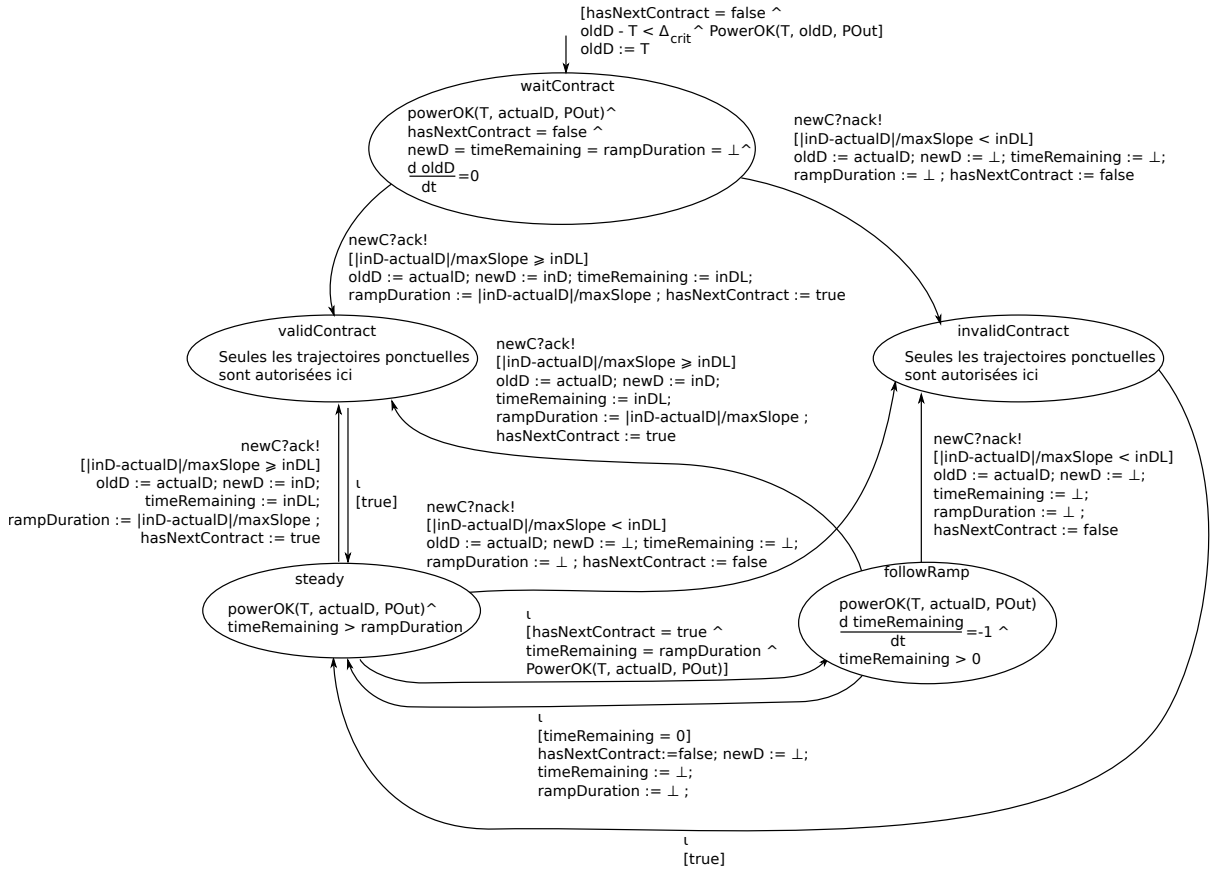
Remarquons que, sous notre hypothèse d'existence de la variable interne spéciale loc (qui représente un mode de la propriété), ces trajectoires peuvent être étudiées indépendamment pour chaque mode. En effet, les évolutions possibles de loc sont un ensemble de fonctions constantes. Au cours d'une trajectoire, loc est donc toujours constante, ce qui permet de définir un ensemble de trajectoires propre à chaque mode, en partitionnant $\mathcal{T}$ suivant la valeur constante prise par loc dans chaque trajectoire. Nous nommerons ainsi *trajectoires du mode* m pour tout $m \in type(loc)$ les trajectoires tr de $\mathcal{T}$ telles que $(tr \downarrow loc)(t) = m \; \forall t \in \mathsf{T}$.

Cette partition nous permet de décrire les automates hybrides avec une notation visuelle compacte. Nous en avions donné un premier exemple en Fig. 1.3 ; la Fig. 2.3 en donne un autre, qui s'applique à notre propriété d'étude. Les concepts que nous avons déjà présentés nous permettent d'en définir certains éléments. Chaque ellipse définit un ensemble de trajectoires qui sont associé à un mode de la propriété. Le nom figuré en haut de chaque ellipse fait référence à une des valeurs de $type(loc)$, l'ensemble des modes de l'automate. Le contenu de l'ellipse définit les trajectoires autorisées dans ce mode.

Les flèches portées sur le schéma figurent les derniers éléments du formalisme que nous n'avons pas encore présentés : les événements et les transitions discrètes. Nous allons les définir à présent.

2.2.5 Actions

Nous avons exposé jusqu'ici les éléments continus du formalisme, et en particulier les trajectoires. Nous avons expliqué que la valeur de la variable loc, qui représente le mode de l'automate, restait constante au cours du temps au sein d'une même trajectoire. La description informelle de la propriété stipule pourtant que ce mode change, par exemple au moment où le système entre dans une phase de changement de la température de consigne. Ce type de changement discret est formalisé au sein d'un automate hybride par des transitions. Une transition survient à un instant donné et réinitialise l'état de l'automate (c'est-à-dire que tout ou partie des valeurs des variables change) en interrompant une trajectoire pour en débuter une nouvelle.



Dans tous les modes, les contraintes suivantes s'appliquent :
T ∈ [-5;40], PRenew ∈ [0;2500], inD ∈ [10;30], inDL ∈ [0;7200],
oldD ∈ [10;30], newD ∈ [10;30], timeRemaining ∈ {true, false},rampDuration ∈ [0;7200]

actualD est définie comme suit :
Si hasNextContract = true alors actualD = oldD
Sinon
  Si timeRemaining < rampDuration alors
    ratio = timeRemaining/rampDuration
    actualD = ratio*oldD + (1-ratio)*newD
  Sinon
    actualD = oldD
  Fin si
Fin si

FIG. 2.3 – Automate formalisant la propriété de sûreté conductrice

Le formalisme associe à chaque transition une *action*. Les actions, comme les variables sont réparties dans trois ensembles dont l'un regroupe des actions dites *internes* (il est noté H), le second des actions dites *d'entrée* (il est noté I) et le dernier des actions dites *externes* (il est noté O). On définit $E = I \cup O$ l'ensemble des actions externes, et $A = E \cup H$ dénote l'ensemble de toutes les actions de l'automate.

Lors de la formalisation de propriétés hybrides par des automates, nous interprétons les actions d'entrée comme des événements reçus à des instants déterminés par le système, les actions de sortie par des événements qui sont produits par le système à destination de son environnement, et les actions internes comme des artefacts de la spécification destinés à pouvoir formaliser les transitions spontanées entre modes, c'est-à-dire celles qui ne sont pas effectuées en réponse à un événement extérieur ponctuel et qui ne provoquent pas l'émission d'un événement à destination de l'environnement.

Notons que Lynch et al. imposent l'existence de deux actions particulières pour tout automate : l'action interne ι, qui représente une transition interne (non observable) produite de façon autonome par le système, et e qui représente une transition de l'environnement. En pratique, l'ajout d'autres actions internes pour formaliser des propriétés durant dans le cadre des présents travaux ne s'est pas avéré nécessaire.

La formalisation de notre propriété conductrice nous a amenés à identifier les actions suivantes :

Action	**Rôle**	**Catégorie**
$newC$	Réception d'un contrat	Entrée
ack	Acceptation d'un contrat	Sortie
$nack$	Refus d'un contrat	Sortie
e	Action de l'environnement	Interne
ι	Action interne autonome du système	Interne

2.2.6 Transitions discrètes

Nous avons exposé le concept d'action, qui permet de formaliser les échanges de signaux discrets du système avec son environnement lors de transitions. Il nous reste à définir la notion de transition. Comme nous l'avons exposé précédemment, une transition permet de modifier instantanément l'état de l'automate. Elle se formalise comme un élément de $Val(V) \times A \times Val(V)$ et nous l'interprétons comme l'association d'un état dit *initial*, d'une action et d'un état dit *final* ou d'*arrivée*.

Illustrons ce concept à l'aide de notre exemple conducteur ; considérons le cas où le système entame une phase de changement de sa température de consigne. Dans l'automate que nous avons défini pour formaliser la propriété, l'état initial de la transition vérifie les prédicats suivants :

- $hasNextContract = true$: s'il n'y a pas de contrat à venir, aucun changement de la température de consigne ne survient
- $timeRemaining = rampDuration$: ceci caractérise le moment où la transition doit avoir lieu
- $PowerOK(T, oldD, P_{out})$: la relation $PowerOK$ fictive exprime le fait que la puissance produite est compatible avec la température ambiante et la consigne au vu des exigences de la propriété
- $loc = steady$: le système était dans un mode où la température de consigne est constante.

L'état final est le même que l'état initial, à l'exception de la valeur de loc, qui est réinitialisée pour prendre la valeur $followRamp$. L'action associée à cette transition est l'action interne ι, puisqu'aucun signal n'est envoyé à l'environnement pour signaler le début du changement de consigne.

On constate que la conjonction de prédicats sur l'état initial que nous avons établie est valide pour plus d'un état. Ils décrivent en quelques lignes une infinité de transitions de l'automate. De la même façon, nous représentons sur nos schémas des ensembles de transitions. Sur la figure 2.3 par exemple, entre les ellipses spécifiant les trajectoires autorisées dans les modes $steady$ et $followRamp$ nous avons figuré par une flèche l'ensemble des transitions que nous avons décrites ci-dessus. La première ligne de l'étiquette de la flèche spécifie l'action associée à la transition (c'est une action d'entrée si elle est suivie d'un signe ?, une action de sortie si elle est suivie d'un signe !, une action interne sinon), la seconde entre crochet spécifie ce que nous nommerons la *garde* de cet ensemble de transitions (un prédicat caractérisant les états initiaux des transitions de l'ensemble) et la dernière les réinitialisations des variables qui permettent de calculer l'état final des transitions.

Dans le cas où certains éléments ne sont pas spécifiés, nous supposons les valeurs suivantes par défaut : l'action par défaut est l'action interne ι, la garde est le prédicat $true$ (tout état est un état initial admissible), et les variables pour lesquelles aucune valeur après réinitialisation n'est fournie sont réputées ne pas être modifiées. Enfin, dans le cas où la réception d'un événement par l'automate entraîne la production en réaction d'un autre événement sans délai (c'est-à-dire que l'automate effectue une transition, une trajectoire ponctuelle de longueur nulle puis une seconde transition), nous représentons les deux ensembles de transitions sur la même flèche en l'étiquetant par les deux événements.

2.2.7 États initiaux

Nous avons exposé les différents éléments qui décrivent les évolutions possibles de l'état d'un automate, mais nous n'avons pas encore évoqué la question de l'initialisation de cette évolution. Notre propriété par exemple traite du comportement du système dans le cas où celui-ci n'a initialement aucun contrat en cours (ceci est figuré sur le schéma 2.3 par une flèche entrante qui ne provient d'aucun autre mode dans le mode $waitContract$). Cette notion se formalise au sein d'un automate hybride à entrées/-sorties par un ensemble $\Theta \subset Val(V)$. Cet ensemble d'états dans le cas de notre propritété est caractérisé par la conjonction des prédicats suivants :

- $loc = waitContract$: initialement le système n'a pas de contrat en cours, le mode $waitContract$ caractérise cette situation.
- $hasNextContract = false$: aucun contrat actif initialement
- $oldD - T < \Delta_{crit}$: La température doit être au-dessus de sa limite basse, sans quoi la propriété est immédiatement violée.
- $PowerOK(T, oldD, P_{out})$: La puissance consommée est compatible avec les valeurs de la température et de la consigne.

2.2.8 Exécutions hybrides

La notion d'exécution hybride s'interprète intuitivement comme l'historique complet de l'évolution de l'état d'un automate, correspondant à une exécution possible du système concret. Dans notre exemple, une telle exécution décrit l'évolution de toutes les variables (température de consigne, tempé-

rature ambiante, puissance de chauffe, etc ...) au cours du temps ainsi que les dates et caractéristiques des transitions discrètes.

Cette notion se formalise en une séquence alternée $\tau_0 a_0 \tau_1 a_1 \tau_2 a_3 \tau_3 \ldots$ de trajectoires et d'actions de l'automate. Chaque τ_i est un élément de $\mathcal{T}$, et Chaque a_i est un élément de A. En notant $\tau.fstate$ le premier état d'une trajectoire τ et $\tau.lstate$ son dernier état (dans le cas où elle est fermée), la séquence doit de plus vérifier ces conditions supplméentaires :

- $\tau_0.fstate \in \Theta$: le premier état de l'exécution est un état initial
- pour tout i tel que τ_i n'est pas la dernière trajectoire de l'exécution, $(\tau_{i-1}.lstate, a_i, \tau_i.fstate) \in D$; D représente l'ensemble des transitions de l'automate.
- Si l'exécution contient un nombre fini d'éléments, le dernier élément de l'exécution est une trajectoire.

2.2.9 Traces hybrides

La dernière notion du formalisme que nous présentons est celle de trace hybride. Elle formalise les observations qui sont faites sur le système sous test lors de son exécution, à partir desquelles l'issue des tests et leur couverture seront évaluées. C'est donc une notion importante, que nous définissons d'une manière un peu différente de celle de Lynch [40] afin de l'adapter au contexte du test. Nous présentons la définition originale, suivie de notre définition modifiée.

Une trace hybride au sens de Lynch est informellement la projection d'une exécution hybride sur les variables et actions externes de l'automate (nous utilisons le terme *signal* pour désigner une variable ou une action) :

Définition 3 (Trace hybride, au sens de Lynch)**.** *Soit une exécution $e = \tau_0 a_0 \tau_1 a_1 \tau_2 \ldots$ d'un automate hybride $\mathcal{A} = (W, X, \Theta, E, H, D, \mathcal{T}$. La trace $htrace(e)$ correspondante est obtenue à partir de e en effectuant les transformations suivantes :*

- *Pour toute trajectoire $\tau_i \in e$, τ_i est remplacée par sa restriction $\tau_i' = (\tau_i \downarrow W)$ aux variables externes de $\mathcal{A}$.*
- *Pour toute action $a_i \in e$, si a_i est interne alors elle est remplacée par l'action interne ι.*
- *Pour toute action a_i ainsi remplacée, si les variables externes ne sont pas modifiée par la transition associée (i.e. si $\tau_i'.lstate = \tau_{i+1}'.fstate$), alors l'action a_i n'est pas simplement remplacée par ι, elle est complètement supprimée et les trajectoires voisines τ_i' et τ_{i+1}' sont concaténées.*

Nous souhaitons formaliser les observations faites sur le système sous test. Cette notion de trace hybride est en première approximation une façon naturelle de le faire, car elle contient presque exclusivement des signaux externes, que nous identifions aux signaux qui peuvent être relevés sur le système sous test concret. Toutefois, une trace au sens de Lynch peut contenir des actions internes ι qui dans notre interprétation ne sont pas observables. Cette définition ne convient donc pas à nos besoins, et nous en utilisons une autre plus restrictive. Toutes les actions internes sans exception y sont supprimées :

Définition 4 (Trace hybride, dans le cadre des présents travaux)**.** *Soit une exécution $e = \tau_0 a_0 \tau_1 a_1 \tau_2 \ldots$ d'un automate hybride $\mathcal{A} = (W, X, \Theta, E, H, D, \mathcal{T}$. La trace $htrace(e)$ correspondante est obtenue à partir de e en effectuant les transformations suivantes :*

- *Pour toute trajectoire $\tau_i \in e$, τ_i est remplacée par sa restriction $\tau_i' = (\tau_i \downarrow W)$ aux variables externes de $\mathcal{A}$.*

– *Pour toute action $a_i \in e$, si a_i est interne alors elle est* supprimée.
– *Pour toute action a_i ainsi supprimée, les trajectoires voisines τ'_i et τ'_{i+1} sont concaténées.*

Dans les deux définitions, la trace $htrace(e) = \tau'_0 a_0 \tau'_1 a_1 \ldots$ si elle est finie ne contient pas forcément le même nombre d'éléments que l'exécution e dont elle est tirée, car les concaténations de trajectoires réduisent le nombre de ses éléments. Par ailleurs, les trajectoires τ'_i incluses dans la trace ne sont pas nécessairement les projections de trajectoires τ_i de l'automate $\mathcal{A}$ auquel appartient e ; intuitivement, les concaténations peuvent accoler des trajectoires qui dans la définition de $\mathcal{A}$ ne peuvent pas s'enchaîner sans transition discrète.

Dans notre exemple du système de chauffage, une telle trace contiendrait l'évolution des variables externes telles que la température ambiante, mais les variables internes telles que la consigne à suivre ou le mode en seraient exclues.

Chapitre 3

Validation des automates et systèmes hybrides

La validation des systèmes hybrides et des spécifications hybrides est un problème qui a reçu une attention soutenue depuis une quinzaine d'années. Nous présentons ici les principales approches qui existent dans ce domaine à notre connaissance. Dans la première section 3.1 de ce chapitre, nous présentons des approches qui visent à valider la correction d'un modèle en explorant son espace d'états de manière exhaustive. La section 3.2 présente de manière générale le test et les problèmes de recherche classiques qu'il pose. La section 3.3 décrit plus spécifiquement des approches de test structurel, qui exploitent un modèle du système (pour évaluer la qualité des jeux de tests par exemple). Enfin, nous présentons en 3.4 des techniques de test fonctionnel, qui ont pour objectif de s'assurer qu'une implémentation respecte les exigences exprimées par sa spécification, sans cette fois disposer d'un modèle de cette implémentation.

3.1 Approches de vérification applicables aux automates hybrides

3.1.1 Model checking

Certains modèles formels, tels que les machines à états finis, permettent de modéliser les exécutions possibles d'un système. Par ailleurs, des propriétés portant sur ce modèle peuvent être exprimées, qui caractérisent le respect par le modèle d'une exigence sous-jacente. Dans le cas des machines à états finis, une telle propriété peut être exprimée dans une logique temporelle telle que LTL [19] et exprimer une contrainte sur une succession d'états interdite par exemple. Le problème se pose alors de savoir si le modèle autorise cette séquence d'états. Les approches de *model checking* [18] répondent à ce type de question en proposant des méthodes d'exploration systématique de l'espace d'état du modèle, pour y rechercher d'éventuelles contradictions avec la propriété à vérifier.

Plus généralement, une approche de model checking consiste à calculer l'ensemble des états atteignables du système, et à montrer que son intersection avec un ensemble d'états dits non sûrs (correspondant à un comportement non désiré) est vide. Cette analyse peut être conduite en partant des états initiaux du système et en calculant les états atteignables lorsque le temps progresse (on parle alors de

forward reachability). On peut également calculer l'ensemble des prédécesseurs des états non sûrs pour y rechercher des états initiaux (on parle de *backward reachability* [2]). Certaines approches proposent des algorithmes qui alternent des phases d'analyse en avant et en arrière (une étude de 2006 [11] en donne quelques exemples, dans le cas des systèmes hybrides).

3.1.2 Analyse d'atteignabilité pour les systèmes hybrides

Les premières approches de model checking ont été développées pour des modèles de machine à états finis. D'autres méthodes spécifiquement destinées aux systèmes hybrides ont été introduites par la suite, que nous détaillons ici.

Dans le cas des systèmes hybrides, les propriétés à vérifier consistent dans plusieurs contributions à ne pas visiter un ensemble d'états dits *non sûrs* ; ces approches sont généralement regroupées sous le terme d'*analyse d'atteignabilité*. Un travail précurseur de ce type a été proposé par Alur et Dill [7] dans le cadre du formalisme des automates temporisés. Ce formalisme est une des formes les plus restrictives d'automates hybrides (du point de vue de l'expressivité qu'il autorise), puisque les variables d'un automate temporisé ne peuvent que progresser linéairement (avec une dérivée temporelle égale à 1) et être éventuellement réinitialisées à une valeur de 0 lors d'une transition discrète, qui ne peut survenir qu'à des dates entières (c'est-à-dire pour $t = 0, 1, 2, 3, etc \dots$). L'idée centrale dans cette contribution a été de remarquer qu'il est possible de partitionner l'ensemble des états initiaux en *régions*. Tous les états d'une telle région suivent nécéssairement la même séquence de transitions dans l'automate. il est donc possible de vérifier un tel modèle en construisant un graphe de régions (une machine à état fini) décrivant le comportement observé pour chacune, et d'y appliquer un algorithme existant de model checking pour vérifier une propriété exprimée en logique temporelle. Des outils comme UPPAAL [15] par exemple implémentent cette technique.

D'autres contributions ont été construites par la suite pour des formalismes hybrides plus expressifs, s'inspirant dans une certaine mesure de cette première approche. Elles reprennent en effet l'idée de manipuler des ensembles d'états ; le concept de région disparaît cependant de ces contributions, car il n'a plus de sens hors du formalisme des automates temporisés. Plus spécifiquement, des propositions ont été formulées pour des formalismes tels que les automates hybrides linéaires [34] (dans ce formalisme, les équations d'évolution continue des variables sont de la forme $\dot{x} = A$ où A est un vecteur de constantes, ou $\dot{x} \in [a, b], (a, b) \in \mathbb{R}^2$). Ces approches ont par la suite été prolongées pour traiter les automates hybrides affines [26] (dont les équations d'évolution sont de la forme $\dot{x} = A.x + B$ où x représente l'état).

La décidabilité des problèmes et la complexité des algorithmes varient suivant les formalismes et les problèmes considérés [35]. La complexité des algorithmes en particulier est une des difficultés auxquelles sont confrontées les approches d'analyse d'atteignabilité. La taille de la représentation des ensembles d'états manipulés, ainsi que le coût du calcul des états atteignables sont en effet importants et peuvent empêcher l'applicabilité en pratique de ces méthodes à des cas d'étude de taille courante. L'espace d'état atteignable d'un système hybride est en effet classiquement représenté par un ou plusieurs polyèdres mis à jour itérativement au fur et à mesure de l'exploration du modèle. Le nombre de ces ensembles d'états ainsi que le nombre de contraintes nécessaire à leur représentation exacte croît parfois trop rapidement pour être calculé en pratique. Ceci a motivé le développement de plusieurs contributions portant sur la manière de représenter ces ensembles (par exemple en les sur-approximant, pour réduire la taille de leur représentation) et les algorithmes de vérification associés [11, 60, 27].

3.1.3 Hybridisation et partitionnement

Les contributions récentes ont considéré des formalismes de plus en plus expressifs au fil du temps, et s'intéressent en particulier au formalisme des automates hybrides affines. La dynamique complexe de ces systèmes (où les variables peuvent suivre des trajectoires exponentielles) ne permet pas la réutilisation directe de techniques appliquées aux formalismes plus simples (tels que les automates hybrides linéaires) pour le calcul de l'ensemble des états atteignables. Il est possible toutefois de s'y ramener en effectuant une *linéarisation locale* des équations différentielles du modèle. Ceci consiste à approximer la dynamique affine par une dynamique linéaire sur des intervalles de temps courts et une partition de l'espace d'état du modèle. Les techniques de vérification issues des automates hybrides linéaires peuvent alors être adaptées. Cette technique est désignée dans la littérature par le terme d'*hybridisation* [12].

Les contributions et les outils en lien avec ces techniques tels que CheckMate [57], PHAVer [26] ou d/dt [13] effectuent donc une analyse approximative, en sur-approximant l'espace d'état atteignable par le modèle. L'approximation réalisée est *complète* ; cela signifie que si l'approximation du système est jugée sûre, alors le modèle original l'est aussi. Il est toutefois possible que l'approximation mène à la découverte de "fautes" qui n'existent pas dans le modèle d'origine.

Le *raffinement* de l'approximation (qui consiste à re-partitionner les zones de l'espace d'état initial suspectées de conduire à une faute) est une technique utilisée au sein de ces approches pour limiter ou éliminer les faux positifs. L'apport d'une analyse à la fois en avant et en arrière peut être également utilisée pour améliorer l'approximation dans les zones de violations potentielles [26].

3.1.4 Exploration de modèle par abstraction de prédicat

Pour tenter de réduire la complexité des algorithmes d'analyse d'atteignabilité, la notion d'abstraction de prédicat [29] a également été inventée et appliquée aux systèmes hybrides [6]. Cette technique consiste à partitionner l'espace d'état d'un système en fonction des valeurs que prennent un ensemble de prédicats, qui portent sur les états. Les éléments de cette partition sont nommés *états abstraits*. On étudie ensuite les transitions (continues ou discrètes) existant entre des états concrets qui appartiennent à des états abstraits différents. Lorsque de telles transitions existent, on crée des transitions entre les états abstraits concernés ; ceci mène à la définition d'un graphe abstrait du système.

Le principe de construction de ce graphe abstrait est illustré en Fig. 3.1 : soit un automate hybride avec un espace de dimension 2 tel que représenté en haut à gauche de la figure. Le cadre en haut à droite figure son espace d'état $S = [-1; 1]^2$, et les flèches grises le champ de vecteurs caractérisant l'évolution continue de l'état. On peut partitionner cet espace d'états suivant les valeurs de prédicats, que nous choisissons ici arbitrairement : $\varphi_1 : x > -0.5$, $\varphi_2 : y > 0.3$, $\varphi_3 : y < -0.2$ et $\varphi_4 : \sqrt{(x-1)^2 + (y+1)^2} < 0.5$. En analysant l'existence de transitions (uniquement continues dans cet exemple) entre des états appartenant aux différents éléments de la partition, on peut construire le graphe de transition abstrait représenté en bas à gauche de la figure ($\varphi_{p1\ldots p_n}$ caractérise l'état abstrait qui représente l'ensemble des états concrets où seuls les prédicats φ_{p1} à φ_{pn} sont vrais).

En identifiant les états abstraits qui contiennent des états initiaux du système concret, et ceux qui contiennent des états non sûrs du système concret, on peut rechercher au niveau du système abstrait un chemin menant des uns aux autres. Si un tel chemin n'existe pas, le système est sûr. S'il en existe un, alors il existe peut-être une exécution du modèle concret qui mène à un état non sûr. Dans l'exemple

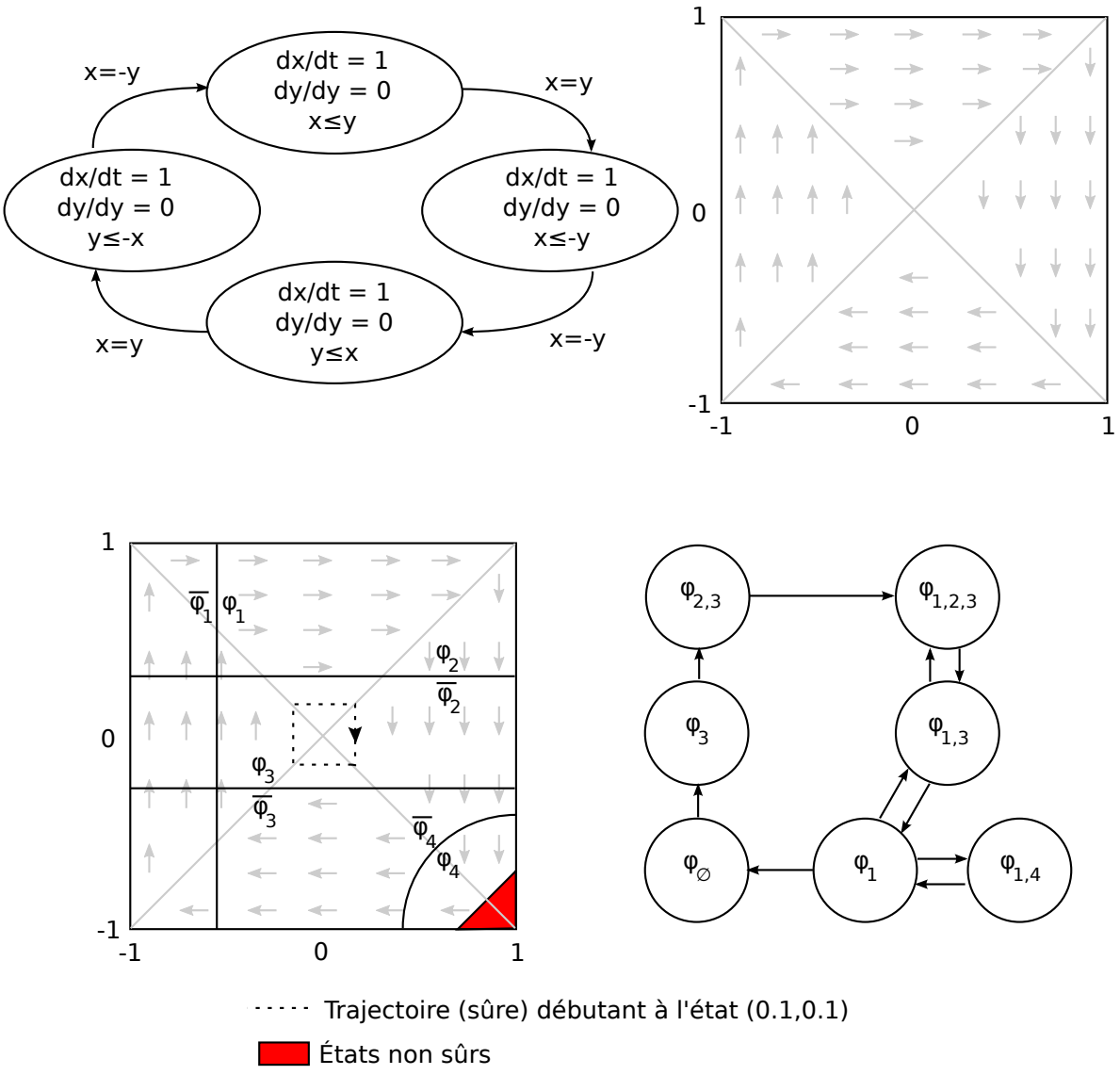


FIG. 3.1 – Construction d'un graphe abstrait dans le cadre d'une technique d'abstraction de prédicat.

précédent, si l'automate contient comme unique état initial l'état $(0.1, 0.1)$ (appartenant à l'état abstrait $\varphi_{1,3}$) et que les états tels que $y - x + 1.9 < 0$ (appartenant à l'état abstrait $\varphi_{1,4}$) sont réputés non sûrs, le chemin $(\varphi_{1,3}, \varphi_1, \varphi_{1,4})$ indique qu'il existe peut-être une trajectoire non sûre. Une telle exécution n'existe pas toujours dans le système concret (notre exemple le montre : la trajectoire illustrée en pointillée, périodique et qui débute en $(0.1, 0.1)$, ne rencontre jamais d'états non sûrs).

Une technique d'analyse d'atteignabilité classique (c'est-à-dire, une de celles que nous avons présentées dans les sous-sections précédentes) est alors utilisée pour tenter de confirmer ce fait ; elle ne considère cette fois que les états initiaux caractérisés par l'état abstrait au début du chemin suspect identifié précédemment, et non l'ensemble des états initiaux du système concret, ce qui réduit la complexité de cette analyse. Si l'analyse confirme l'existence d'une violation, l'algortihme s'arrête en revoyant les états initiaux trouvés qui mènent à un état non sûr. Si l'analyse échoue (comme ce serait le cas dans notre exemple), un nouveau prédicat est recherché et ajouté à l'ensemble des prédicats précédents pour raffiner l'abstraction. Une nouvelle itération de recherche de chemins dans le nouveau graphe abstrait commence alors.

L'utilisation de cette technique par Alur et al. [6] appliquée aux système hybrides linéaires (dont l'évolution continue est régie par des équations de la forme $\dot{x} = A$) a ainsi permis de développer des algorithmes de vérification plus rapides.

3.2 Test des systèmes hybrides

3.2.1 Preuve ou test des logiciels

Les approches de validation par model checking et analyse d'atteignabilité que nous avons présentées apportent lorsqu'elles aboutissent une preuve de la correction d'un modèle vis-à-vis d'une spécification. Sous réserve que la spécification corresponde aux besoins auxquels doit répondre le logiciel et que l'implémentation du modèle soit effectuée sans que des fautes n'y soient introduites, le logiciel produit remplira donc toujours sa fonction. L'applicabilité de ces méthodes peut toutefois être limitée ou rendue impossible par des facteurs tels que la taille du modèle ou son expressivité (qui peuvent rendre les algorithmes trop complexes ou indécidables), voire son indisponibilité.

Les approches par le test que nous allons maintenant présenter permettent de valider des modèles ou des implantations concrètes de modèles dans certains cas où les techniques de preuve ne sont ainsi pas applicables, au prix d'une assurance moindre sur le respect de la spécification (ce n'est plus une preuve de correction qui est apportée mais une *amélioration de la fiabilité*). Une définition classique du test qui illustre ceci est ainsi donnée par Myers [49] en ces termes : *Testing is the process of executing a program with the intent of finding errors*.

L'application de techniques de test à la validation des systèmes pose un certain nombre de problèmes que nous présentons ici : génération et sélection de tests, exécution des tests et interprétation des résultats, et enfin adéquation des tests. Nous présentons ici ces problèmes et quelques solutions qui leur sont apportées par des contributions existantes.

3.2.2 Génération et sélection de tests

Une approche de test ne peut explorer qu'une quantité limitée d'états d'un modèle ou d'une spécification. Ceci pose par conséquent deux problèmes qui sont la génération et la sélection des tests. Le problème de la génération de tests est celui du choix d'entrées qui vont tester l'artefact testé (modèle ou implémentation) de façon suffisamment diverse pour être considérée *adéquate*. Une telle génération peut être faite par exemple en échantillonnant aléatoirement des données de test (on parle de test aléatoire, un *random testing*). L'adéquation d'une suite de tests caractérise le fait qu'elle contient un ensemble de données de tests minimum indispensable pour pouvoir valider l'artefact testé (dans le domaine de l'avionique par exemple, le critère d'adéquation MC/DC [45] est un standard reconnu). Ceci est formalisé par des *critères d'adéquation*, sur lesquels nous reviendrons par la suite. Le problème réciproque de la *sélection* des tests consiste à choisir au sein d'un ensemble de tests qui satisfont un tel critère un ensemble le plus petit possible qui continue de le satisfaire, ceci afin de réduire le coût (en temps et en mémoire par exemple) de l'exécution des tests, tout en conservant un ensemble de test adéquat.

3.2.3 Évaluation des tests et simulation des automates hybrides

Quelque soit le type de test qui est employé pour la validation d'un système hybride (structuel ou fonctionnel), on dispose suivant les cas d'un modèle ou d'une spécification (hybrides) vis-à-vis desquels il convient d'évaluer une trace d'exécution. Cela suppose deux choses : premièrement il faut disposer d'une méthode qui transforme la trace pour la rendre compatible avec l'objet formel vis à vis duquel on souhaite l'évaluer. Tan et al. [58] par exemple formalisent en utilisant un langage adapté les événements utiles qui peuvent survenir au cours d'une trace d'exécution. Cette formalisation est utilisée pour extraire de la trace observée ces événements et leur date d'apparition.

La séquence d'événements inférée est ensuite évaluée vis-à-vis d'une formule de logique temporelle pour décider de la *correction* de l'exécution. Un système (ou un modèle) dont toutes les exécutions sont correctes est dit *conforme* à sa spécification. Les notions de correction d'une exécution et de conformité d'un artefact (système ou modèle) se caractérisent par la définition d'une *relation de conformité*. Tretmans [61] donne ainsi la définition d'une telle relation (nommée $ioco$) applicable aux systèmes de transitions étiquetées ; elle est définie comme une relation définie entre un ensemble d'implémentations $\mathcal{I}$ et un ensemble de spécifications $\mathcal{S}$. Van Osch [62] propose une relation de conformité baptisée $hioco$ inspirée de la relation $ioco$ et étendue aux modèles hybrides.

On appelle *test de conformité* l'activité consistant à valider par le test la relation de conformité entre une implémentation et sa spécification. Ceci est effectué en exécutant suffisamment de tests (au sens d'un certain critère d'adéquation) et en évaluant leur correction. Dans les travaux de Dang et al. [20], la correction d'une exécution est ainsi évaluée en construisant itérativement l'exécution (unique, en raison du formalisme sous-jacent employé) du modèle correspondant à une trace observée sur le système, au cours de l'exécution d'un test.

Cette reconstruction itérative d'une exécution à partir d'une trace (ou, de façon équivalente, à partir de données remontées par des capteurs) est semblable à celles qui sont employés dans le domaine de la simulation numérique. L'objectif alors est de simuler l'évolution de l'état d'un système (éventuellement en présence de données incontrôlables acquises en temps réel depuis des capteurs). Des outils courants comme Matlab [47] ou Modelica [41] peuvent ainsi s'interfacer avec toutes sortes de sources de données externes (capteurs, cartes d'acquisition, autre programme de simulation) à condition qu'un module

logiciel adéquat soit disponible. L'état du système à un instant donné de la simulation est toujours complètement déterminé de façon unique par les étapes de simulation précédentes et les entrées soumises au système.

On peut remarquer que la sémantique des modèles manipulés par ces outils est différente de celle des automates hybrides : en simulation, les changements de mode sont spécifiés par des fonctions numériques dont le passage à zéro déclenche une transition discrète. Dans le formalisme des automates hybrides, l'activation de ces transitions est en revanche contrôlée par un prédicat sur l'espace d'état (appelé garde). Des contributions ont été proposées pour transformer un modèle exprimé dans un formalisme dédié à la simulation en un modèle exprimé dans un formalisme d'automate hybride [55] ou inversement [50]. De telles transformations posent cependant des problèmes de passage à l'échelle car la taille des modèles obtenus est importante.

3.2.4 Adéquation des tests et profils opérationnels

Comme nous l'avons exposé précédemment, un critère d'adéquation peut être utilisé pour formaliser la notion de qualité d'une suite de tests (on parle de critère de couverture), ou spécifier un objectif minimal à atteindre (on parle alors de critère d'arrêt) [64]. Il n'existe pas à notre connaissance de tel critère spécifiquement développé pour les automates hybrides. De nombreuses propositions existent en revanche pour caractériser la couverture des machines à états [52], ou des nuages de points (qui peuvent être vus comme la discrétisation des traces d'exécution de systèmes dynamiques). Cependant, certains critères (qui sont des critères de couverture) ont ainsi été adaptés depuis d'autres domaines ou formalismes et appliqués aux systèmes hybrides [24, 53, 20]. Nous les présenterons en détails dans la suite de ce chapitre au sein de l'approche de test dont ils sont tirés.

Une caractéristique remarquable de ces critères existants mérite toutefois d'être exposée immédiatement ; ils sont définis à l'aide de la spécification hybride uniquement. Dans notre exemple conducteur, un tel critère de couverture existant serait défini à l'aide de l'automate présenté en Fig. 2.3 p.20. Les modes $waitContract$ et $steady$, qui regroupent chacun une fraction comparable de l'espace d'état total de l'automate, contribueraient donc de façon comparable à la définition du critère d'adéquation. Des travaux existants portant sur la définition et l'utilisation de *profils opérationnels* posent toutefois la question du bien-fondé d'une telle définition basée uniquement sur la spécification fonctionnelle d'un système.

Le profil opérationnel original introduit par Musa [48] est une décomposition hiérarchique du sytème à plusieurs niveaux de plus en plus fins, qui associe à chaque niveau un coefficient représentant sa fréquence d'utilisation (dans la proposition originale de Musa, cette fréquence d'utilisation est identifiée à la proportion des tests qui devront être appliqués). Musa a montré que la fiabilité des logiciels pouvait être améliorée en faisait porter les tests en priorité sur les fonctions qui sont les plus susceptibles d'être utilisées. En effet, en exerçant en particulier les opérations les plus fréquentes, il est plus vraisemblable de détecter les erreurs qui s'y trouvent. Le code le plus utilisé sera donc également celui qui a été le mieux testé, ce qui réduit la probabilité d'apparition de fautes comparée au cas d'un test uniforme de chaque opération. Ceci a motivé l'examen dans ces travaux de l'opportunité de définir un critère d'adéquation utilisant conjointement une spécification fonctionnelle (sous la forme de propriétés de sûreté hybrides) et un profil opérationnel.

3.3 Test structurel des systèmes hybrides

Cette section présente les approches de test *structurel* pour les systèmes hybrides, c'est-à-dire celles qui exploitent un modèle du système et valident sa correction par rapport à une spécification[1]. Ceci par opposition aux approches de test *fonctionnel* que nous présenterons par la suite et qui s'appliquent aux systèmes dont seules une implémentation et une spécification sont disponibles.

3.3.1 Approches par trajectoires robustes

Nous avons vu en 3.1.4 qu'une partition de l'ensemble des états initiaux pouvait être utilisée pour conduire des analyses d'atteignabilité plus simples, en élaguant des ensembles d'états initiaux dont on peut prouver la sûreté sans calculer explicitement tous leurs successeurs. Les techniques utilisant des trajectoires robustes [37, 28] utilisent un principe similaire ; mais au lieu de définir une partition à priori et d'étudier les successeurs des états contenus dans les éléments de cette partition, des états initiaux sont échantillonnés et on cherche ensuite à trouver des voisins de ces états qui ont un comportement similaire.

Intuitivement, la robustesse d'une trajectoire caractérise la distance à laquelle elle passe de l'ensemble des états non sûrs. Cette notion de robustesse permet d'identifier, autour d'un état initial qui est le point de départ d'une exécution correcte (c'est-à-dire qui n'inclut aucun état non sûr), un voisinage d'autres états initiaux qui initient des trajectoires visitant la même séquence d'états discrets, et n'intersectant pas non plus l'ensemble des états non sûrs (la Fig. 3.2 illustre le principe de cette méthode).

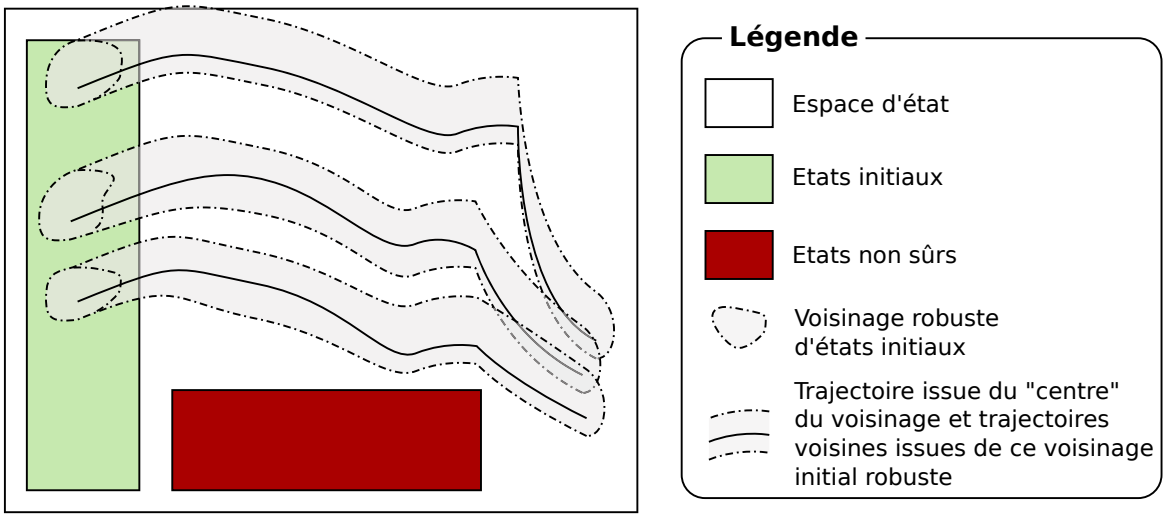


FIG. 3.2 – Pavage partiel de l'espace d'états initiaux par des voisinages robustes

Pour valider le modèle entier, on répète cette analyse pour un ensemble fini d'états initiaux X ; pour tout $x \in X$ on détermine un voisinage $V(x)$ autour de x, tel que tous les points de $V(x)$ visitent la même suite d'états discrets et restent en-dehors de l'ensemble des états non-sûrs. Si on choisit X tel que l'union des voisinages $V(x)$ explorés recouvre l'ensemble des états initiaux du système, on montre par conséquent que chaque état initial conduit à une trajectoire sûre ; le modèle considéré est donc valide.

[1] Ces approches vont parfois au-delà de la définition donnée par Myers dans la mesure où le test est parfois pratiqué uniquement sur un modèle sans qu'une implémentation ne soit utilisée.

Il peut toutefois arriver qu'on ne puisse trouver un ensemble X qui mène à la couverture complète de l'espace des états initiaux (pour des raisons de coût des calculs et donc d'épuisement du budget alloué au test par exemple). Cette méthode dans certains cas n'apportera donc pas de garantie de validité du modèle, afin de mieux passer à l'échelle en contrepartie.

Ce concept de robustesse des trajectoires a été exploité d'une manière différente dans les travaux de Sankaranarayanan et al. [54] où il est utilisé pour guider la génération de cas de test (c'est-à-dire, de trajectoires robustes) pour chercher à violer une propriété de sûreté. Cette génération exploite la méthode de l'entropie croisée [21] pour optimiser les paramètres d'un processus de génération de test ; l'objectif est de déterminer une séquence de signaux d'entrée applicable à un système hybride, telle que la trace résultante produite par le système falsifie une propriété de logique temporelle MTL donnée. La notion de robustesse est ici définie comme une *distance* (signée) séparant une trace de la violation de la propriété (elle est positive si la trace vérifie la propriété, négative si elle la viole). La méthode d'entropie croisée génère des trajectoires suivant une densité de probabilité dont les paramètres sont optimisés itérativement de manière à ce que les trajectoires produites soient le moins robustes possibles. L'algorithme proposé termine au bout d'un temps prédéfini en retournant la trace la moins robuste trouvée, qui correspond à une violation de la propriété si cette robustesse est négative. Cette approche a été implémentée dans un outil nommé S-TALIRO [9] applicable à des modèles Simulink.

3.3.2 Test de modèles par exploration arborescente

Les approches par trajectoires robustes nécessitent une analyse du modèle pour évaluer la robustesse d'une trajectoire. Cette analyse possède un coût, et impose des contraintes sur l'expressivité du formalisme. En contrepartie, elle apporte des garanties sur la sûreté d'*ensembles* d'états initiaux, c'est-à-dire sur le fait qu'ils n'ont pas de sucesseur non sûr au cours d'une exécution. D'autres contributions [53, 24, 20] adoptent une stratégie différente : elles analysent la sûreté d'un ensemble fini d'états uniquement (et ne peuvent donc jamais prétendre à une exploration exhaustive dès lors que l'espace d'état est infini), mais cette analyse est moins coûteuse et moins exigeante vis-à-vis du formalisme du modèle. En effet, elle nécessite seulement que le modèle puisse prédire les états successeurs d'un état donné suite à une évolution discrète ou continue (pour une entrée et une durée déterminées).

3.3.2.1 Génération des tests

Ces techniques exploitent un algorithme d'exploration issu du domaine de la robotique et de la planification de mouvements nommé RRT [38] (pour *Rapidly Expanding Random Trees*), illustré en Fig. 3.3. Cet algorithme construit itérativement un arbre des états que peut atteindre le système ; à chaque itération, un état but s_{goal} est généré aléatoirement. Ensuite, l'état le plus proche déjà visité dans l'arbre courant s_{near} est calculé, puis une combinaison d'entrées est choisie, qui amène le système de l'état s_{near} à un nouvel état s_{new} en un pas de simulation. Ce nouvel état est alors ajouté à l'arbre. En explorant ainsi itérativement et par simulation l'espace d'état du modèle, l'algorithme vise à établir l'atteignabilité éventuelle d'états non sûrs (autrement dit, détecter des fautes).

Remarquons que l'algorithme RRT original échantillonne uniformément l'espace d'état pour choisir s_{goal}. Si cet espace contient des états *inatteignables*, c'est-à-dire qui ne sont visités lors d'aucune exécution de l'automate, le processus d'échantillonnage peut les sélectionner malgré tout et provoquer une exploration plus importante des états atteignables les plus proches (au sens de la mesure utilisée par

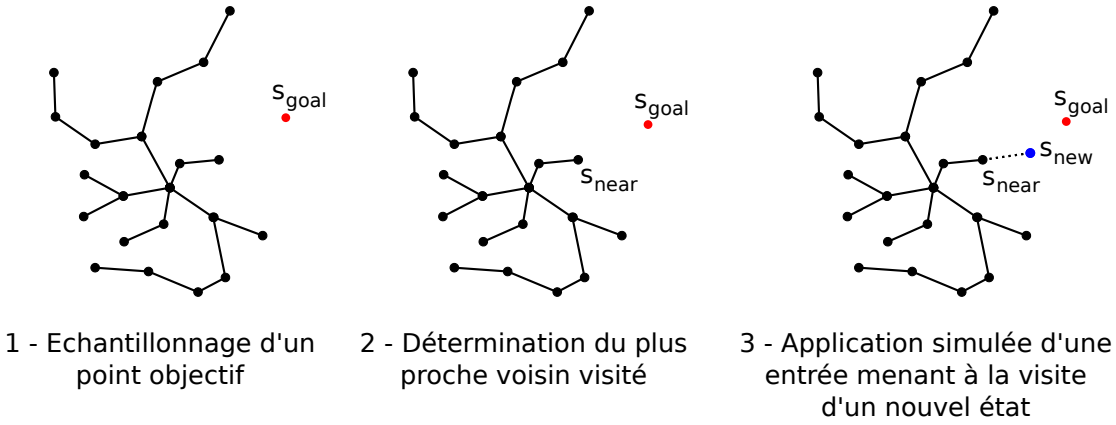


FIG. 3.3 – Algorithme d'exploration de l'espace d'états RRT

RRT pour déterminer s_{near}). Ce problème est nommé problème de *contrôlabilité* du modèle. Informellement, cela va provoquer l'accumulation d'états visités sur les « bords » de l'espace atteignable. Les travaux de Dang et al. [20] proposent une version modifiée de cet algorithme baptisée agRRT qui utilise une stratégie d'échantillonnage limitant cet effet ; l'algorithme utilise alternativement deux méthodes d'échantillonnage pour déterminer s_{goal} à chaque itération. Dans l'une d'elles, il échantillonne l'espace d'état en favorisant les zones peu visitées (en faisant un choix entre plusieurs états candidats pour s_{goal}). Cette stratégie a l'avantage d'explorer rapidement l'espace d'état initialement, mais se heurte au problème de contrôlabilité que nous avons mentionné. Dans l'autre phase, l'algorithme échantillonne préférentiellement des états qui vont davantage couvrir les zones déjà explorées (ceci est calculé à l'aide d'une mesure qui caractérise l'impact de l'ajout d'un ppoint sur la couverture du modèle). Ceci favorise l'exploration des états atteignables lorsque ceux-ci ont déjà été « densément » occupés par l'arbre que construit l'algorithme.

Enfin, pour guider la progression de l'exploration vers les états qui semblent davantage susceptibles de mener à des fautes et atteindre plus rapidement des états de l'espace d'état qui sont éloignés (relativement à la dynamique du système) des états intiaux, une approche proposée par Plaku et al. [53] exploite une notion de *piste*. Une piste est une séquence de sous-ensembles de l'espace d'état que l'exploration arborescente cherche à traverser successivement, suivant un principe similaire à celui des techniques d'exploration par abstraction de prédicats que nous avons présentées en 3.1.4.

3.3.2.2 Adéquation des tests

Un critère de couverture applicable aux systèmes hybrides a été proposé dans une contribution d'Esposito [24]. Cette mesure est basée sur une notion de plus proche voisin : sur l'ensemble des états qui correspondent à un mode particulier, on superpose une grille de référence ayant les mêmes limites que l'espace d'état du système. Lors de l'exécution d'un test, les différents états visités (plus précisément, un ensemble de points résultant de l'échantillonnage des trajectoires suivies lors de l'exécution) sont recensés au sein de l'espace d'état. Chaque point de la grille de référence se voit alors attribuer un *score* de couverture entre 0 et 1, fonction de la distance à laquelle il se trouve d'un point visité lors du test. Si tel un point est situé à proximité du point de la grille de référence (dans une maille voisine), ce point de référence se voit affecter un score inférieur à 1 (avec un minimum de 0 si les deux points sont confondus). Sinon, il se voit affecter le score de maximal de 1 (un score bas correspond donc à une meilleure

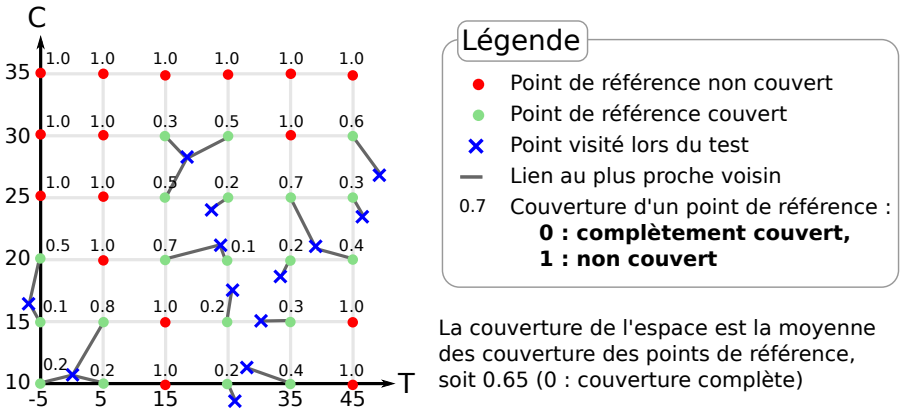


FIG. 3.4 – Exemple de calcul d'une couverture au sens d'Esposito et al.

couverture). La couverture totale est alors la moyenne des couvertures de chaque point de la grille de référence (cf. Fig. 3.4).

Cette mesure de couverture présente l'avantage de fournir un critère d'arrêt immédiat : si la couverture parvient à zéro, tous les points de la grille de référence ont été « testés », au sens où un point proche a été visité. Mais à moins que les états visités échantillonnés ne coïncident parfaitement avec chaque point de référence, cette limite n'est jamais atteinte. Par ailleurs, il est possible de calculer la couverture localement (sur un sous-ensemble de la grille) afin d'identifier les zones où la couverture est bonne ou insuffisante.

En revanche, l'espace est traité uniformément, il n'est pas prévu d'affecter des coefficients différents à différentes zones en fonction des recommandations d'un profil opérationnel par exemple. En outre, l'indicateur statistique retenu (la moyenne) n'apporte pas d'information utile sur le nombre ou la répartition des états visités lors du test : il peut traiter de la même façon des jeux de tests dont la répartition peut être très différente, notamment en matière de dispersion. Nous illustrons ceci sur la Fig. 3.5 : les deux jeux de tests ont des caractéristiques très différentes en termes de dispersion et de nombre d'états visités, mais leur adéquation est dans les deux cas évaluée à une valeur de 0.5.

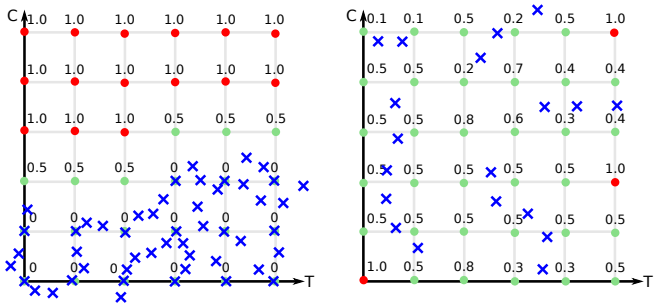


FIG. 3.5 – Deux jeux de test présentant une couverture de 0.5 au sens d'Esposito

Une autre mesure de couverture est utilisée dans [20] par Dang et al. Elles s'appuie sur la notion de *star discrepancy* [59]. Cette mesure caractérise la dispersion d'un ensemble P de k points au sein d'un hyperrectangle B. Elle est est définie comme le maximum sur toutes les sous-boîtes J incluses dans B

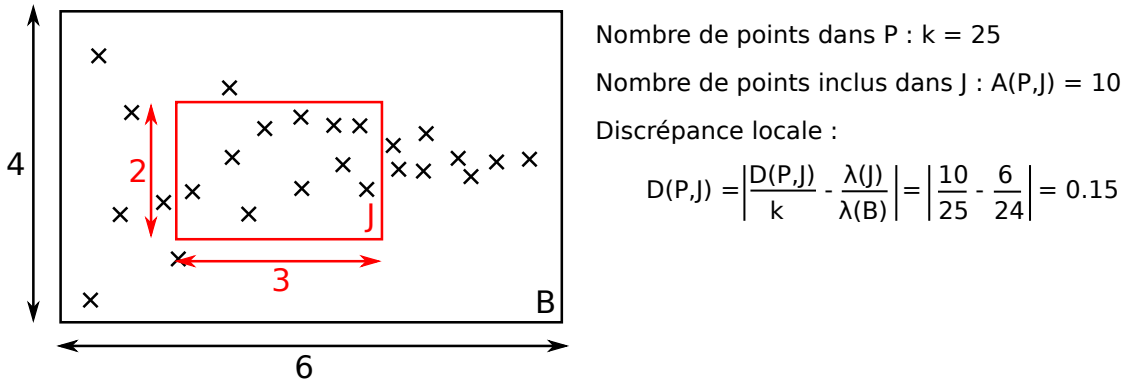


FIG. 3.6 – Exemple de calcul d'une discrépance locale

d'une quantité $D(P, J)$ dite *discrepance locale* (en anglais, *local discrepancy)* définie comme suit ($\lambda(V)$ dénote le volume d'une boîte V) :

$$D(P, J) = \left| \frac{A(P, J)}{k} - \frac{\lambda(J)}{\lambda(B)} \right|$$

Nous illustrons le calcul de la discrépance locale pour une boîte d'exemple sur le schéma 3.6. Cette quantité caractérise l'écart entre la proportion de points de P et la proportion de l'espace d'état qui sont capturés par la boîte. Un ensemble de points de taille infinie uniformément réparti dans un espace aurait donc une discrépance de 0.

Cette mesure de dispersion d'un ensemble de points est utilisée par Dang et al. pour guider la génération de tests (c'est-à-dire, l'échantillonnage des états-buts qui servent à faire progresser l'arbre des états explorés). Toutefois, malgré son nom elle ne constitue pas un critère d'adéquation pertinent : elle ne peut comparer efficacement que des ensembles de points de même taille ; l'ajout d'un nouvel état dans une zone déjà « densément » visitée conduit à une évaluation d'une dispersion plus élevée. Cette mesure constitue donc un critère d'adéquation non monotone. Par conséquent il n'est pas possible de fixer rationnellement une valeur suffisante que doit atteindre un jeu de test pour ce critère (et d'en faire ainsi un critère d'arrêt), ni de comparer deux jeux de tests à l'aide de cette mesure (et d'en faire ainsi un critère de couverture), à moins qu'ils n'aient exactement la même taille. Nos contributions s'attachent tout particulièrement à ce problème de définition et d'évaluation de l'adéquation.

3.4 Test fonctionnel des systèmes hybrides

3.4.1 Test aléatoire

Des approches existantes traitent la question de la génération des données de test en générant aléatoirement des séquences d'entrées représentatives de ce que le système est susceptible de recevoir (selon un modèle d'environnement prédéfini), qui lui sont ensuite soumises. Les travaux de Tan et al. [58] en fournissent un exemple ; les cas de test sont générés à partir d'un automate hybride non-déterministe qui représente l'environnement. Cet automate est animé en le déterminisant de différentes manières, et en

projetant la trajectoire obtenue sur les signaux d'entrée du système sous test. De même, pour les systèmes non hybrides mais temps-réel il existe des exemples d'approches de ce type [36], qui utilisent une mesure de couverture de l'environnement pour le guidage et l'arrêt de la génération, si un modèle du système n'est pas disponible.

3.4.2 Limites des stratégies de génération aléatoire uniforme

A l'opposé de la méthode précédente de test aléatoire, qui échantillonne des données uniformément, le test anti-aléatoire [17], ou *antirandom testing*, consiste à choisir des données de test les plus éloignés possibles les unes des autres (au sens d'une mesure qui paramètre la méthode). Malaiya [42] propose une méthode de génération incrémentale où chaque test généré est choisi le plus éloigné possible de tous les autres. Remarquons que cette approche suppose qu'on dispose d'une métrique associée ou d'en proposer une, ce qui n'est pas toujours simple en particulier lorsqu'on traite de variables réelles et de structures de données complexes. L'article de référence propose un certain nombre de pistes et de propositions à ce sujet. Ici, il n'existe pas de critère d'adéquation explicite, mais les tests précédemment générés permettent de choisir les suivants. Des travaux récents [32] ont exploré empiriquement les propriétés du test anti-aléatoire dans le cadre de la sélection de tests et montré que cette méthode devenait moins efficace pour détecter des fautes lorsque quelques cas de test sont "isolés" des autres.

L'applicabilité du test anti-aléatoire trouve également ses limites lorsqu'un critère d'adéquation est employé pour formaliser les exigences portant sur la suite de tests à générer : il peut être nécessaire au contraire de chercher à atteindre un ou plusieurs états bien définis de la spécification ou du modèle au cours de l'exécution d'un test, pour satisfaire le critère. Le modèle du système peut alors être exploité pour guider la génération, en plus de servir de base à l'établissement d'un critère d'adéquation.

Dans cette optique, des méthodes existantes visent ainsi à atteindre des objectifs de test individuels choisis de façon à améliorer l'adéquation des tests. On parle alors d'algorithmes de recherche ; une étude récente présente les contributions de ce type et leur applications à différents formalismes [46].

3.4.3 Test de systèmes hybrides utilisant des algorithmes de recherche

Des méthodes utilisant des méthodes de recherche ont été proposées pour les systèmes hybrides par exemple par Baresel et al. [14] ainsi que Hänsel et al. [31], exploitant plus particulièrement des *algorithmes génétiques*. Un tel algorithme cherche à atteindre un objectif de test prédéfini, qui doit permettre d'améliorer l'adéquation. Une première donnée de test (c'est-à-dire une séquence de signaux d'entrée) est ensuite générée aléatoirement. L'algorithme cherche alors à optimiser itérativement le choix des fragments qui composent cette séquence et/ou les paramètres qui déterminent sa création, de manière à ce que l'exécution du test utilisant la donnée de test calculée s'approche de l'objectif de test (par rapport aux exécutions précédentes) et idéalement l'atteigne. Cette itération est ensuite répété avec d'autres objectifs jusqu'à atteindre une adéquation satisfaisante des tests. Remarquons que les approches mentionnées nécessitent de déterminer complètement la séquence d'entrée à appliquer au système avant son exécution (elle n'est pas recalculée ou modifiée au cours de l'exécution, en réponse aux réaciton du système sous test).

3.5 Conclusion

Les propriétés de sûreté hybrides que nous considérons dans ce travail sont formalisées par des automates hybrides au sein desquels les variables connaissent des évolutions qui ne respectent pas de propriétés de régularité particulières. Nous avons vu par exemple dans notre exemple conducteur que la variable P_{out} qui caractérise la puissance de chauffe émise par le système n'est pas continue. Elle n'est donc pas dérivable, et ne possède pas a fortiori de dérivée constante, bornée ni même combinaison linéaire des composantes de l'état du système. Ceci ne permet pas d'appliquer les techniques de preuves applicables aux automates hybrides que nous avons présentées en 3.1. En outre, le non-déterminisme inhérent à la sémantique des propriétés de sûreté (qui formalisent des besoins généralistes, idéalement applicables à de nombreux systèmes) permet en général l'existence de trajectoires au sein de l'automate qui mènent à des violations de la propriété (c'est-à-dire à un blocage de l'automate). La validation que nous pouvons chercher à atteindre est donc celle d'*implémentations particulières*, afin de valider leur respect des exigences de la spécification.

La validation de ces implémentations sera donc étudiée à partir des traces qui seront relevées lors de l'exécution du système. Ceci implique que seuls les signaux discrets seront disponibles, et qu'ils seront observés à des dates discrètes. L'identification des exécutions possibles de l'automate qui correspondent à cette trace échantillonnée pose un problème similaire à celui de la simulation des systèmes hybrides que nous avons exposé en 3.2.3 pour l'évaluation de la correction des exécutions concrètes du système. Nous examinons ce problème dans le chapitre suivant.

Enfin, certaines questions non résolues que nous avons identifiées en 3.3.2 et 3.2.4, relatives à la définition de critères d'adéquation applicables aux systèmes hybrides et à l'opportunité d'utiliser des profils opérationnels pour définir de tels critères, motivent également les propositions que nous développons dans le chapitre 5.

Chapitre 4

Évaluabilité des propriétés

L'exécution d'un cas de test entraîne la production d'une *trace d'exécution*, qui regroupe les comportements observés du système. Pour décider de la correction de l'exécution par rapport à une propriété, on cherche à montrer que cette trace d'exécution correspond à une exécution de l'automate qui formalise la propriété. Le début de ce chapitre définit les notions pertinentes liées à cette évaluation. Nous identifions ensuite et discutons deux problèmes qui se posent pour l'effectuer.

Le premier problème est lié à la non-observabilité sur l'implémentation d'une partie des signaux de l'automate (nous utilisons le terme de *signal* pour parler indifféremment de variable ou d'action). C'est un aspect du *problème de l'oracle* que nous avons présenté dans l'introduction de ce manuscrit, et qui se pose plus généralement pour toute approche de test.

Le second problème concerne l'évaluation de la correction d'une trace à partir d'observations *discrétisées dans le temps* des signaux du système ; nous avons vu que la spécification par automate hybride que nous utilisons est définie dans le cas général avec un modèle de temps dense (les raisons de ce choix ont été exposées en 2.2.3). Les observations disponibles en revanche sont remontées en pratique par des capteurs physiques qui fournissent un échantillonnage de ce que nous interprétons comme les signaux de l'automate. La décidabilité de la correction peut être influencée par cet échantillonnage temporel également.

Nous proposons une solution à ce problème qui permet de vérifier de façon approchée la correction d'une exécution à partir d'une trace, sous certaines hypothèses. Nous qualifions cette vérification d'approchée car elle se base sur l'instanciation d'une spécification approchée, dérivée de la spécification originale et adaptée à un modèle de trace échantillonnée dans le temps.

4.1 Définitions

4.1.1 Trace d'exécution et système sous test

Du point de vue du test, le système sous test est caractérisé par les comportements qu'il peut produire. Dans le contexte hybride, ces comportements sont continus (évolution de grandeurs physiques contrôlées

par le système) et discrets (émissions d'événements) et interviennent en réponse à des entrées de même nature. Cette notion de comportement rappelle la notion d'exécution pour les automates hybrides en première approximation, mais elle en diffère sur un point essentiel : les automates hybrides possèdent des signaux internes, et leurs exécutions fixent les évolutions ou les occurrences de ces signaux. Les comportements exhibés par les systèmes concrets ont donc une sémantique plus proche de celle des traces hybrides des automates, où seuls figurent les aspects externes du comportement. Nous définissons donc un système comme un ensemble de traces d'exécutions :

Définition 5 (Système sous test)**.** *Un système sous test est un tuple $\mathcal{S} = (U, Y, I, O, Tr)$ où :*
- *$V = U \cup Y$ est un ensemble de variables partitionné entre un ensemble U de variables d'entrée et un ensemble Y de variables de sortie.*
- *$A = I \cup O$ est un ensemble d'actions partitionné en un ensemble I d'actions d'entrée et un ensemble O d'actions de sortie.*
- *Tr est un ensemble de traces d'exécution.*

L'ensemble $\mathcal{S}.W$ des variables de $\mathcal{S}$ s'interprète comme l'ensemble des variables évoluant au cours du temps qu'il est possible d'observer sur le système (en entrée ou en sortie). E représente de même les actions dont il est possible d'observer les occurrences.

Définition 6 (Trace d'exécution)**.** *Une trace d'exécution d'un système $\mathcal{S}$ est une séquence alternée $\tau_0 a_0 \tau_1 a_1 \ldots$ où :*
- *Chaque τ_i est une fonction, définie sur un intervalle temporel fermé à gauche, à valeurs dans l'ensemble des valuations de $\mathcal{S}.W$ (le temps est un sous-groupe de $\mathbb{R}^+$, comme pour les automates hybrides).*
- *Chaque ai est un élément de $\mathcal{S}.E$.*
- *La borne gauche de l'intervalle de définition de τ_0 est 0.*
- *La séquence ne se termine pas par une action, et si τ_i n'est pas le dernier élément de la séquence, la borne gauche de l'intervalle de définition de τ_{i+1} est égale à la borne droite de l'intervalle de définition de τ_i .*

Une trace d'exécution est donc une séquence alternée d'évolution des variables du système dans laquelle s'intercalent des actions. Intuitivement, le dernier point de la définition requiert que les morceaux "continus" de la trace se suivent sans interruption ni recouvrement. Contrairement à la définition de trace hybride donnée en 2.2.9, une trace d'exécution n'est pas directement liée à une exécution connue d'un automate, dont elle serait la projection. On souhaite au contraire déterminer si une telle exécution existe.

4.1.2 Compatiblité et conformité d'un système par rapport à une propriété

La relation de conformité que nous allons présenter à présent repose sur le principe de l'inclusion de traces : l'ensemble des traces produites par le système doit être inclus dans l'ensemble des traces de l'automate qui le spécifie. Il est nécessaire pour évaluer cela que le système possède les variables externes de l'automate qui le spécifie. Nous formalisons cela par une notion de compatibilité définie comme suit :

Définition 7 (Compatibilité d'un système et d'un automate)**.** *Un système $\mathcal{S} = (U_\mathcal{S}, Y_\mathcal{S}, I_\mathcal{S}, O_\mathcal{S}, Tr_\mathcal{S})$ est compatible avec un automate hybride à entrées-sorties $\mathcal{A} = (U_\mathcal{A}, Y_\mathcal{A}, I_\mathcal{A}, O_\mathcal{A})$ ssi*
- *$U_\mathcal{A} \subset U_\mathcal{S}$*
- *$Y_\mathcal{A} \subset Y_\mathcal{S}$*

– $I_{\mathcal{A}} \subset I_{\mathcal{S}}$
– $O_{\mathcal{A}} \subset O_{\mathcal{S}}$

Munis de cette notion de compatibilité, nous pouvons définir la correction d'une exécution :

Définition 8 (Correction d'une exécution)**.** *Soit un système $\mathcal{S}$ compatible avec un automate $\mathcal{A}$ qui formalise une propriété P. Une trace d'exécution tr de $\mathcal{S}$ est correcte par rapport à P ssi il existe une exécution e de $\mathcal{A}$ dont la trace hybride ht est égale à la projection de tr sur les signaux externes de $\mathcal{A}$.*

Intuitivement, cette définition stipule qu'un système est correct si lors de ses exécutions il respecte une des évolutions possibles qu'autorise P. L'évolution des éventuels signaux du système qui n'appartiennent pas à l'automate spécifiant P n'est pas contrainte ; de tels signaux supplémentaires n'influencent pas la correction de ses exécutions.

Définissons enfin la conformité d'une implémentation (un système) $\mathcal{S}$ à une propriété P formalisée par une automate $\mathcal{A}$. Les relations de conformité telles que celles que nous avons présentées au chapitre 3 admettent classiquement qu'un système qui produit le comportement attendu par sa spécification lorsqu'il est soumis à une entrée prévue par cette spécification, et n'importe quel comportement lorsqu'ils sont soumis à une entrée qu'elle ne spécifie pas, sont conformes. Nous proposons une définition semblable pour une conformité d'une implémentation à une propriété hybride. Les automates de Lynch [40] que nous avons présentés au chapitre 2 spécifient complètement leur comportement via leurs trajectoires et leurs transitions (bien que ce comportement puisse être non-déterministe). Toute entrée soumise à un système *modélisé* par un automate hybride est donc complètement spécifiée. Il est toutefois possible que ce système ait d'autres signaux d'entrée/sortie étrangers à l'automate, qui en est donc un modèle partiel. Nous n'imposons pas de condition sur ces signaux supplémentaires dans notre définition. Notons $htraces(\mathcal{A})$ l'ensemble des traces hybrides de $\mathcal{A}$ et $\mathcal{S}.Tr_{\mathcal{A}}$ l'ensemble des traces d'exécution de $\mathcal{S}$ projetées sur les signaux externes de $\mathcal{A}$. La conformité est définie comme suit :

Définition 9. *Soit un système $\mathcal{S}$ compatible avec un automate $\mathcal{A}$ qui formalise une propriété P. $\mathcal{S}$ est conforme à P ssi toutes ses exécutions sont correctes. Autrement dit, ssi*

$$\mathcal{S}.Tr_{\mathcal{A}} \subset htraces(\mathcal{A})$$

Nous utilisons cette définition dans la suite de ce chapitre, et la mettons en perspective en la comparant à des contributions existantes pertinentes en 4.4.3.

4.2 Test de conformité et évaluabilité des propriétés en temps dense

Dans cette section, nous identifions des exigences de déterminisme portant sur les propriétés ; le respect de ces exigences conditionne l'usage qu'il est possible d'en faire au sein de notre approche, car il n'est pas possible de tester la conformité de toutes les propriétés hybrides. Nous montrons que des problèmes se posent au niveau de l'évaluation de la correction d'une trace, et de l'évaluation de l'adéquation d'un jeu de tests pour certaines propriétés.

4.2.1 Présentation du problème

L'estimation de la conformité d'une implémentation passe comme nous l'avons mentionné en 3.2.3 par l'évaluation de la correction des exécutions du système. Pour un système hybride, il s'agit connaissant une trace d'exécution tr de décider s'il existe une exécution e de l'automate dont la trace est égale à tr. Toutefois, pour que le processus de test que nous proposons soit praticable il est nécessaire que cette trace soit *unique* pour une trace d'exécution donnée. En effet, nous employons un formalisme d'automates généraliste qui ne fait pas d'hypothèses particulières (de linéarité par exemple) sur les trajectoires des automates. Faire ce type d'hypothèse permet aux approches de preuve que nous avons présentées en 3.1 de représenter par des structures aux descriptions finies (telles que des polyèdres) des ensembles de points infinis, et de calculer sur ces ensembles des opérations mathématiques (union, intersection, etc ...) afin d'estimer l'espace d'état atteignable. Bien que cet espace comprenne un nombre infini de trajectoires, sa représentation est finie.

Dans notre approche, il n'est pas possible de garantir ceci à cause de la généralité du formalisme considéré. Par conséquent, si une trace hybride observée correspond à une infinité de trajectoires de l'automate, cet ensemble n'aura pas forcément de représentation finie. La même absence d'hypothèse particulière sur la dynamique de l'automate empêche de chercher une approximation finie de l'ensemble des trajectoires correspondant à une trace. Il est donc indispensable pour pouvoir évaluer une propriété que le nombre d'exécutions correspondant à une trace soit fini.

Par ailleurs, nous nous intéresserons dans le chapitre suivant à l'évaluation de l'adéquation de jeux de tests. Un critère d'adéquation est nécessaire pour évaluer rigoureusement la qualité des tests et décider du moment où il devient acceptable d'arrêter la génération. Un tel critère est évalué en utilisant l'ensemble des états de la spécification visités lors du test. Aucun critère existant à notre connaissance pour l'évaluation de jeux de tests hybrides ne considère que certains états pourraient *éventuellement* avoir été visités.

Pour ces raisons nous nous intéressons uniquement aux propriétés formalisées par des automates qui ne possèdent qu'une seule exécution au plus correspondant à la trace observée lors d'un test. Nous qualifions de tels automates d'*évaluables*, et ce caractère d'évaluabilité pourra être défini par rapport à une trace ou de manière générale :

Définition 10 (Évaluabilité sur une trace)**.** *Soit une propriété de sûreté formalisée par un automate hybride A. Soit une trace hybride tr composée des mêmes signaux que les signaux externes de A. A est évaluable sur tr s'il existe au plus une exécution e de A dont tr est la trace.*

Définition 11 (Évaluabilité)**.** *Soit une propriété de sûreté formalisée par un automate hybride A. A est évaluable si pour toute trace hybride tr composée des mêmes signaux que les signaux externes de A, il existe au plus une exécution e de A dont tr est la trace.*

4.2.2 Causes de non-évaluabilité dues à la non-observabilité des signaux internes

La non-observabilité de certains signaux de l'automate peut limiter son évaluabilité pour au moins deux raisons, que nous illustrons ici. Premièrement, la spécification peut autoriser plusieurs exécutions correspondant à une trace. Deuxièmement, le calcul des exécutions correspondant à une trace observée peut être décidable, mener théoriquement à la détermination d'une unique trace, mais ne pas être

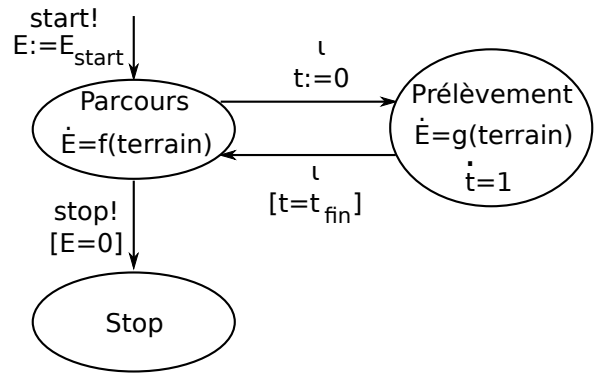


FIG. 4.1 – Ex. d'automate spécifiant la consommation d'un robot collecteur d'échantillons

réalisable en pratique pour des raisons de complexité algorithmique induites par la non-observabilité. Considérons à titre d'ilustration l'exemple suivant :

Exemple 1 (Problème du robot autonome)**.** *Un robot autonome est chargé de faire des prélèvements tant qu'il dispose d'énergie. Il se déplace le long d'un trajet prédéterminé sur un terrain complètement connu ; les signaux observables sur le système sont les deux actions de sortie $start$ et $stop$, correspondant respectivement à l'activation du cycle du robot et à son arrêt lorsqu'il tombe à court d'énergie. Se déplacer et faire un prélèvement consomment des quantités d'énergie différentes, dépendant du terrain sur lequel se déplace le robot. On veut tester sur ce système une propriété qui décrit l'usure attendue des batteries, telle que celle de la figure 4.1. f et g caractérisent la consommation d'énergie attendue du robot dans chaque situation.*

Lorsque le robot s'arrête, on peut mesurer le temps t_{total} qui s'est écoulé. Évaluer la conformité revient donc à savoir s'il existe une séquence temporelle de longueur t_{total} représentant une alternance de périodes de déplacement et de récolte, pour laquelle la prévision de la consommation d'énergie est égale à la charge initiale des batteries du robot au début de la séquence, et nulle à la fin. Ce problème a une combinatoire très importante, même si on considère que les changements de mode ne peuvent se produire qu'à certains instants (ex : des multiples d'une période d'échantillonnage T) et que le temps est discret. Il faut en effet explorer un espace de grande taille, une fois la trace complète collectée. Au contraire, si on avait accès à l'information sur la vitesse de décharge du robot, on pourrait la comparer aux fonctions f et g prédites (puisque le terrain traversé est connu), et par conséquent connaître le mode dans lequel opère le robot (sous réserve que $f(x) \neq g(x)$ pour tout terrain x). On pourrait enfin évaluer l'état de l'automate à chaque instant t en utilisant uniquement les observations antérieures à t et donc vérifier la propriété avec une complexité faible.

Par ailleurs, si f et q coïncident en certains points, plusieurs exécutions peuvent correspondre à une trace donnée, où le robot visite suivant les cas une séquence de mode différente (i.e. à une date t dans une de ces exécutions l'automate est dans le mode correspondant à une activité de déplacement, alors que dans une autre exécution il sera dans le mode correspondant à une récolte). Même en connaissant l'évolution effective de la charge du robot, on ne peut alors pas déduire une unique valeur de loc pour compléter la trace. Le nombre d'états possibles de l'automate peut donc exploser, et en l'absence de contraintes sur l'expressivité de cet automate il n'est pas possible d'assurer qu'il existe une représentation finie de cet ensemble d'états (comme c'est le cas quand on s'intéresse à la vérification d'automates hybrides linéaires par exemple). La complexité engendrée par ce non-déterminisme peut donc empêcher

d'évaluer la correction d'une trace d'exécution.

Cet exemple montre que le caractère interne de certaines variables peut rendre impossible le fait de discerner l'état de l'automate à partir d'une trace. On en conclut que les automates utilisés dans le cadre du test doivent posséder une forme de déterminisme : la connaissance de la trace d'exécution complète à évaluer (incluant à la fois les entrées soumises au système et ses réactions) doit permettre de déterminer l'état de l'automate (ou bien un nombre raisonnable d'états possibles s'il en existe plusieurs). Il est nécessaire en pratique de disposer d'un algorithme d'une complexité raisonnable pour cela.

Nous nous intéressons dans la suite à la caractérisation des automates pour lesquels on peut déterminer un *unique* état possible à partir d'une trace ; ce sont en effet ceux-ci que nous proposerons de considérer pour le calcul de l'adéquation des jeux de test, au chapitre suivant.

4.3 Test de conformité en temps discret

Nous avons jusqu'ici examiné le problème de l'évaluabilité des propriétés en supposant que la trace est définie au sein d'un modèle de temps dense, de même que la propriété. Ceci est cohérent avec les travaux existants qui considèrent souvent un tel modèle de temps. Nous allons à présent compléter cette première réflexion en présentant des questions additionnelles qui se posent lorsque les observations de la trace sont échantillonnées au cours du temps. Cette question est pertinente dans le cadre d'une approche de test applicable à des systèmes réels, qui ne peuvent être observés qu'à certains instants uniquement (les capteurs physiques possédant une période d'échantillonnage non nulle).

4.3.1 Difficultés introduites par la discrétisation du temps

4.3.1.1 Occurrences proches ou simultanées d'actions

La première question qui se pose concerne la date des transitions discrètes et leur possible occurrence simultanée au sein de la trace discrétisée. Considérons à ce propos l'automate A de la figure 4.2. Il doit émettre les signaux a et b, aux dates 2.5 et 2.7 respectivement. On souhaite valider la propriété qu'il formalise sur un système concret sur lequel on dispose d'observations effectuées à chaque unité de temps (c'est-à-dire pour $T = 1, 2, 3 \ldots$). Le premier problème qui se pose est que les dates des événements produits par le système sont connues avec une précision limitée. Ainsi, si le système est conforme, les observations réalisées mentionneront uniquement que les événements a et b se sont produits entre les dates 2 et 3. Si le système n'est pas conforme à la propriété associée à l'automate A, mais plutôt à une autre propriété telle que celle qui est formalisée par l'automate B, il produira une trace qui ne correspond pas à une exécution de A (l'événement a arrive trop tard). Cependant, une observation réalisée toutes les unités de temps de cette trace livrera les mêmes informations que celle de la trace correcte. De la même façon, une inversion de l'ordre des événements telle que peut en produire une implémentation conforme à la propriété associée à l'automate C ne pourra pas être détectée.

Pour résoudre ce problème, on peut définir et utiliser une propriété approchée, adaptée à la nature échantillonnée des observations, en utilisant une démarche de spécification inspirée de celle des logiciels synchrones [16]. Ces systèmes sont réputés voir leur sortie changer instantanément en réponse à une variation de leur environnement, et ces variations surviennent à des instants discrets prédéfinis. Dans

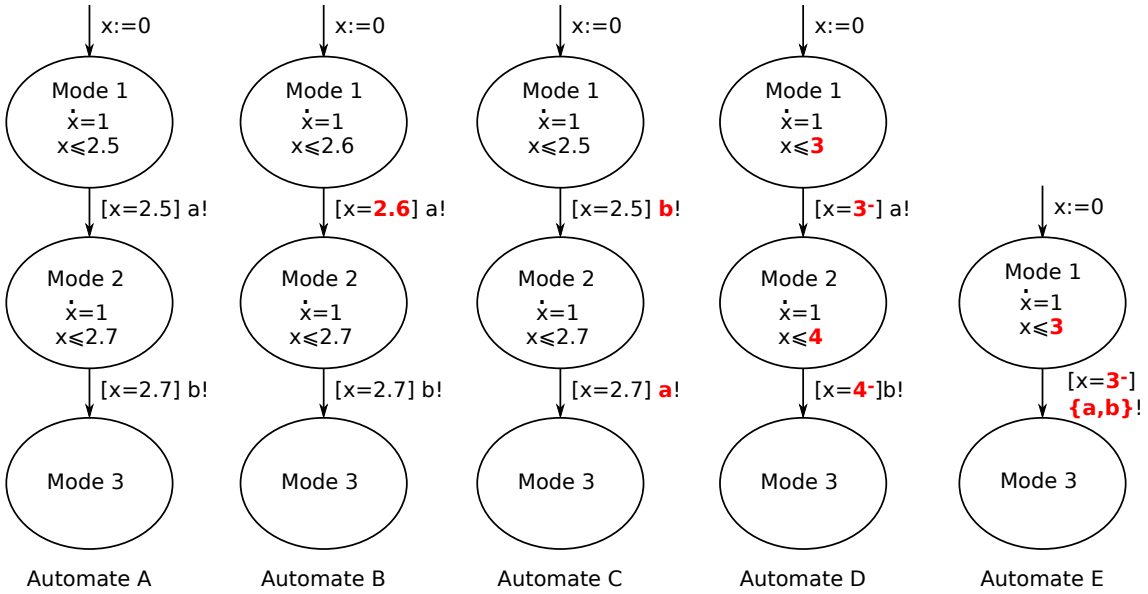


FIG. 4.2 – illustration des problèmes liés à la discrétisation du temps

l'automate D par exemple, les transitions discrètes sont obligatoirement séparées d'au moins une période et surviennent en fin de période. Dans l'automate E, les transitions peuvent en outre survenir simultanément, mais leur ordre d'apparition n'est pas différencié. En définissant les propriétés sur ce principe, on peut faire l'hypothèse au sein de l'oracle que les observations qui seront faites sont des observations *exactes* de la trace et des dates d'occurrence des événements.

Faire cette hypothèse crée naturellement la possibilité de juger correctes les discrétisations de traces produites par des systèmes non conformes (comme dans l'exemple des automates A, B et C), mais ceci est inévitable en temps discret puisque ces exécutions pruidsent les mêmes traces observées. En contrepartie, les questions de non-déterminisme ou d'ordre d'occurrence des événements mentionnées précédemment sont explicitement résolues lors de l'écriture de la spécification auxiliaire, qui peut ainsi devenir évaluable tout en possèdant une sémantique explicite et "proche" de celle de la propriété originale.

4.3.1.2 Indétectabilité de certains phénomènes "courts"

Une question similaire à la précédente se pose au niveau de la dynamique continue de l'automate. Considérons l'automate A de la Fig. 4.3 : il spécifie un système comportant une variable externe E dont la dérivée doit rester contenue dans l'intervalle $[-8; 8]$ et qui peut avoir une unique discontinuité, qui est un saut de -3 unités de sa valeur. L'automate B est une version modifiée de cet automate, définie sur une structure de temps discrète que nous interprétons comme un échantillonnage du temps dense avec une période T. On constate trivialement que si le signal du système qui est identifié avec la variable E de l'automate est continu, alors les traces conformes à A observées à une période T coïncident avec les traces reconnues par B lorsque T tend vers 0. Cependant, lorsque T est suffisamment importante cela n'est pas le cas. Sur la même figure 4.3 sont représentées trois traces pour lesquelles l'évaluation de la conformité de la trace aux automates A et B donne des résultats qui peuvent ainsi différer.

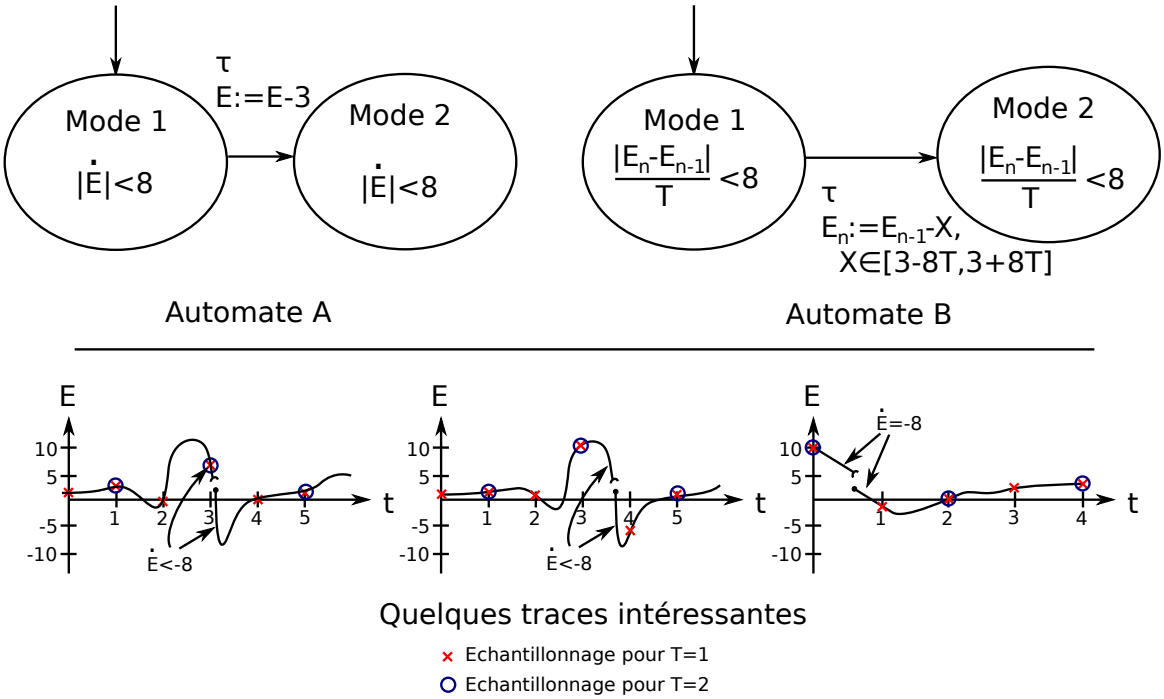


FIG. 4.3 – Un autre exemple de problème lié au test en temps discret

Sur la première trace par exemple, une discontinuité est présente et autour de ce point la dérivée de E est plus importante que ce qui est autorisé par l'automate A. Mais l'échantillonnage temporel masque ce phénomène dans le cas de l'automate B, pour $T = 1$ ou $T = 2$ par exemple : les points effectivement observés sont compatibles avec B. Si on échantillonne différemment, comme sur le second exemple, les observations ne sont plus conformes à ce que spécifie l'automate B. On en conclue donc trivialement que la période d'échantillonnage doit être plus faible que la durée des phénomènes que l'on souhaite pouvoir détecter.

4.3.1.3 Influence de la période d'échantillonnage sur le déterminisme

Un autre point intéressant qui concerne cette fois le pas de discrétisation est mis en évidence sur la dernière trace de la Fig. 4.3 : une discrétisation du temps avec $T = 1$ permet de détecter que la discontinuité prévue par l'automate a eu lieu de façon certaine puisque l'écart entre les deux observations n'est pas compatible avec l'invariant du premier mode. En revanche, pour $T = 2$ les deux observations auraient pu avoir lieu pour des exécutions qui ne changent pas de mode, et pour des exécutions qui changent de mode. Ceci correspond à un non-déterminisme qui apparaît dans la spécification de l'automate B lorsque $T > 3/16$: l'espace d'états atteignable au sein du mode 1 après un pas de temps à partir d'un état s, et l'espace d'état du mode 2 atteignable à partir de s après une transition discrète et une évolution d'un pas de temps ont alors une intersection non nulle. On en conclue, encore une fois très logiquement, que la durée de la période temporelle entre deux observations limite ce qu'il est possible de spécifier par l'automate, si on souhaite conserver la propriété d'évaluabilité.

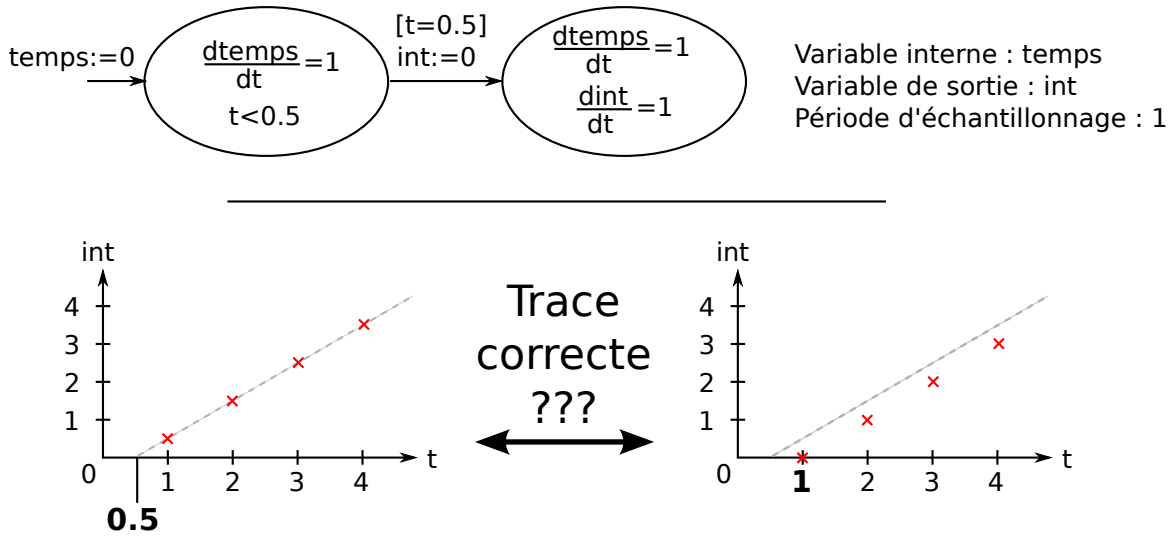


FIG. 4.4 – Un problème de traduction de propriété du temps continu vers le temps discret

4.3.1.4 Le problème des réinitialisations entre deux échantillonnages

Le dernier problème que nous présentons est lié à la sémantique des réinitialisations lors du passage au temps discret. Sur la figure 4.4 on a représenté un automate défini en temps continu que l'on souhaiterait évaluer sur une trace observée en temps discret. Il spécifie le comportement d'une variable de sortie int du système, qui doit intégrer le temps à partir de la date 0.5, avec comme valeur initiale à cette date 0. La ligne en pointillé sur les graphiques présente ce comportement de référence.

Pour cela on se propose de définir un automate auxiliaire adapté au temps discret tel que discuté en 4.3.1.1. La période d'observation visée est de 1 unité de temps. Normalement, la variable int est réinitialisée à la date 0.5 à zéro. Mais la propriété en temps discret ne peut spécifier que ce qui est attendu à la date 1 (et aux dates suivantes à période fixe). On peut écrire cette propriété différemment suivant la façon dont on fait la traduction. Par exemple :

- On peut considérer que la transition dans la spécification discrète est déplacée à la date 1. l'avantage est que cette méthode est sa simplicité : la réinitialisation associée est un simple passage à 0 de la variable, et on peut appliquer cette même transformation quelle que soit la période d'échantillonnage. Son désavantage est qu'elle diffère de la sémantique de la propriété originale : on voit sur le graphique en bas à droite de la figure que les valeurs prédites par cette spécification modifiée (figurées par des croix) ne coïncident pas avec les valeurs de la propriété continue que l'on souhaitait tester initialement.
- On peut considérer qu'à la date 1, l'automate a dû effectuer à la date 0.5 sa transition dans le nouveau mode ; la transition prenant en compte ceci, elle réinitialise int à 0.5 au lieu de 0. L'avantage de cette méthode est qu'elle coïncide avec la sémantique de la propriété initiale en temps continu (Cf. graphique en bas à gauche). En particulier, elle n'introduit pas d'erreur dépendant de la période d'échantillonnage. Son désavantage est que la valeur réinitialisée dépend, elle, de la période d'échantillonnage, et qu'il faut donc réécrire la propriété si elle change. De plus, cela complexifie le calcul de la valeur réinitialisée, ce qui peut poser problème si la dynamique dans le nouveau mode est complexe. Enfin, il faut déterminer à quelle date *aurait dû* se produire la transition, ce qui est également source de complexité.

L'arbitrage entre ces deux solutions dépend bien évidemment du problème considéré, de sa complexité et du degré de précision numérique nécessaire, pour que la propriété en temps discret effectivement testée ait une sémantique "suffisamment proche" de la propriété originale en temps continu. Quelle que soit l'option choisie, il est important de conserver une cohérence dans la démarche, sans quoi de faux positifs peuvent être trouvés lors du test. Nous avons rencontré un problème de ce genre pendant les premières expérimentations que nous avons menées : Le système sous test respectait la propriété en temps continu alors que la propriété en temps discret effectivement testée que nous utilisions se basait sur la forme la plus simple de la réinitialisation, sensible à la longueur du pas de simulation. Ceci conduisait à détecter des "erreurs" qui étaient en fait dues à des variations dans la sémantique de la propriété testée, introduites lors du passage au temps discret.

Ces faux positifs peuvent également être supprimés en relaxant légèrement les tests numériques des invariants : ainsi, les petits écarts à la spécification (aussi bien dûs à des dépassements du système qu'aux artefacts précités) sont traités comme des comportements corrects. Ceci ne permet pas de détecter les violations de propriétés de faible amplitude, mais dans le cas où elles sont bénignes, cela permet de résoudre le problème précédent. Enfin, on peut jouer sur le pas de temps de la vérification en échantillonnant si possible plus fréquemment les variables externes. Ceci permet de réduire la tolérance à ajouter à la spécification, et d'éviter les problèmes de non-déterminisme discutés en 4.3.1.2.

4.4 Une solution au problème de l'oracle

4.4.1 Transformation de propriétés pour le test en temps discret

Nous avons proposé une caractérisation des propriétés hybrides évaluables et discuté intuitivement une démarche d'adaptation des propriétés hybrides au test en temps discret et les difficultés qui surviennent lors de cette adaptation. Pour définir les propriétés auxiliaires (basées sur un modèle de temps discret) en traitant ces problèmes, nous proposons de les formaliser par un ensemble de *fonctions* qui caractérisent leur dynamique concrete, associé à une représentation abstraite de leur dynamique continue.

Nous spécifions les automates en temps discret en deux parties. Une *partie discrète* regroupe la description des signaux (variables et actions) et leur type, les modes, et les transitions (regroupées en un nombre fini d'ensemble dénommés *ensembles de transitions*. Elle s'inspire de la définition d'IOHA de Lynch, avec une différence notable : il est possible d'associer plusieurs actions à une transition. En effet, si une action de sortie du système typiquement doit être émise en réponse à une action de l'environnement, ces deux actions sont observées simultanément sur une trace discrétisée, sans que leur ordre ne soit connu. Nous regroupons ensemble ces transitions en cascade. La partie discrète de la spécification se formalise ainsi :

Définition 12 (Partie discrète d'une spécification hybride)**.** *La partie discrète de la spécification d'un automate hybride à entrées sorties en temps discret $\mathcal{A}$ est un tuple $\mathcal{A}_D = (W, X, E, H, D)$ composé des éléments suivants :*

- *Un ensemble W de variables externes.*
- *Un ensemble X de variables internes disjoint de W. On note $V = W \cup X$.*
- *Un ensemble $locs$ de modes.*
- *Un ensemble E d'actions externes.*
- *Un ensemble H d'actions internes disjoint de E. On note $A = E \cup H$.*

– *Un ensemble D d'ensembles de transitions discrètes, qui sont des éléments de $loc \times 2^A \times loc$.*

Les ensembles U et Y d'une part, ainsi que I et O d'autre part, partitionnent W et E respectivement en variables ou actions d'entrées et de sortie.

Une seconde partie dite *partie continue* de la spécification vient compléter la première en précisant tous les éléments quantitatifs qui déterminent les exécutions possibles de l'automate : évolution continue entre deux instants d'observation, prédicats caractérisant les gardes et les états initiaux, réinitialisations. Ces éléments sont formalisés par des fonctions :

Définition 13 (Partie continue d'une spécification hybride)**.** *Soit $\mathcal{A}_D$ la partie discrète de la spécification d'un automate hybride à entrées sorties en temps discret. Une partie continue de spécification compatible avec $\mathcal{A}_D$ est un tuple $\mathcal{A}_C = (Init, Inv, Succ, Garde, Reinit)$ tel que :*

- *$Init$ est une fonction définie sur $locs$ qui associe à toute valeur $l \in locs$ une fonction $Init_l$ de $Val(W)$ dans $Val(V) \cup \perp$. Cette fonction renvoie un initial complet de l'automate dans le mode l, correspondant à l'état externe (partiel) qui lui est fourni.*
- *Inv associe à toute valeur $l \in locs$ un prédicat Inv_l portant sur $Val(V)$ qui caractérise l'invariant du mode l.*
- *$Succ$ associe à toute valeur $l \in locs$ une fonction $Succ_l$ de $Val(V) \times Val(W) \times \mathbb{R}_+^*$ dans $Val(X) \cup \perp$. qui caractérise le successeur continu d'un état du mode l.*
- *$Garde$ associe à toute valeur $ts \in D$ un prédicat $Garde_{ts}$ portant sur $Val(V)$ qui caractérise la garde de l'ensemble de transitions ts.*
- *$Reinit$ associe à toute valeur $ts \in D$ une fonction $Reinit_{ts}$ de $Val(V)$ dans $Val(X)$ qui caractérise la réinitialisation des variables internes qui intervient lorsque qu'une transition de ts est empruntée.*

Cette formalisation permet de décrire de façon compacte un automate en temps discret. La paramétrisation des fonctions qui calculent le successeur continu d'un état par le temps séparant deux observations permet d'écrire une spécification générique qui peut s'adapater à différents scénarii d'observation. Ces fonctions peuvent renvoyer une valeur spéciale $\perp$, ce qui permet d'exprimer dans la spécification l'impossibilité pour l'automate de progresser, si la propriété est violée ou n'est plus évaluable à partir de l'état courant. Le choix de la stratégie de spécification des réinitialisations que nous discutions en 4.3.1.4 est spécifié par l'intermédiaire des fonctions $Reinit_{ts}$ associées à chaque ensemble de transitions discrètes.

La forme de cette spécification, et en particulier la définition des fonctions Succ qui calculent le successeur continu d'un état, évoquent l'hypothèse dite de *synchronisme* [16]. Sous cette hypothèse, un système réagit à ses entrées et à son environnement instantanément à chertaines dates. Ceci permet d'abstraire de la spécification le temps physique et les problèmes d'événements courts. Notre travail en diffère toutefois car la spécification que nous faisons conserve de nombreux éléments propres aux automates hybrides tels que les gardes et les transitions. De plus, le temps physique n'est pas abstrait de la spécification : celle-ci de par sa définition peut s'adapter à des observations réalisées à des dates quelconques, où l'intervalle de temps entre deux observations n'est pas nécessairement constant.

4.4.2 Un algorithme de vérification dynamique des propriétés hybrides

Nous avons proposé une représentation d'automates hybrides en temps discret sous la forme d'une collection de fonctions. Nous proposons d'utiliser cette représentation pour adapter en temps discret

des automates hybrides définis initialement en temps dense, et effectuer une vérification dynamique des propriétés de sûreté hybride qu'ils représentent, c'est-à-dire évaluer la correction d'observations qui sont réalisées sur le système auquel s'aplliquent ces propriétés.

Pour effectuer cette vérification, nous proposons un algorithme, présenté en figures 4.1 (algorithme principal) et 4.2 (sous-fonction $essayerTransitions$), qui réalise un oracle de test. Cet algorithme évalue itérativement l'état de l'automate correspondant à chaque point d'observation (cet état est unique, si une seule exécution correspond aux observations qui sont faites). Si l'automate devient bloqué (parce que ni évolution discrète ni évolution continue ne sont disponibles) ou peut suivre plusieurs exécutions différentes, une erreur est renvoyée. Ceci permet de tester la conformité de l'implémentation.

L'entrée de l'automate est une séquence d'observations $o_0 \dots o_n$. A une observation o_i sont associés une durée $o_i.l \in \mathbb{R}_+^*$, une valuation des variables externes $o_i.val$, et un ensemble d'actions externes $o_i.acts$. À la première observation o_0 n'est associée aucune action, c'est-à-dire que $o_0.acts = \varnothing$. Sa durée est également nulle (c'est-à-dire que $o_0.l = 0$).

Pour chaque observation, l'algorithme calcule tout d'abord l'évolution continue de l'état interne au cours de la période écoulée. Les actions recensées au sein d'une observation individuelle sont réputées s'être produites à la fin de la période d'observation et leur influence sur l'état interne de l'automate est donc évalué seulement ensuite. Finalement, le nouvel état après avoir traité l'observation est vérifié par rapport à l'invariant du nouveau mode. Il est à noter que cette vérification peut être redondante car un test de l'invariant est effectué après chaque transition discrète. Mais il est nécessaire de faire ce test même au cours des périodes d'observation au cours desquels aucune transition discrète ne s'est produite.

4.4.3 Discussion

Plusieurs relations de conformité [44, 62, 61, 20] ont été proposées pour les systèmes hybrides, basées sur la même idée générale d'inclusion des traces observées dans les traces de l'automate. Un système hybride (qu'il s'agisse d'un système concret ou d'un modèle) est ainsi déclaré conforme à une spécification fournie sous la forme d'un automate hybride si toutes les traces (ou trajectoires, si toutes les variables sont observables) que peut produire ce système en réponse à une séquence de contrôle font partie des traces (ou exécutions) que l'automate est susceptible de produire lorsqu'on lui soumet les mêmes entrées.

La question de la discrétisation du temps et donc de l'observabilité des variables à certains instants seulement, ainsi que celle de la non-observabilité de certaines variables (dites internes) ont été traitées de façon inégales suivant les cas. Une des premières relations de conformité pour les systèmes hybrides a par exemple été proposée par Manna et Pnuelli [44][1]. Elle introduit la notion de *sampling computation*, qui est une séquence d'états indexés par $\mathbb{N}$ et associés à des dates extraites de $\mathbb{R}^+$. Ces états peuvent être interprétés comme les différents instants d'observation au cours d'une exécution. Pour prendre en compte le fait que des événements notables qui doivent figurer dans la trace (ex : changement de signe d'une variable, transition discrète ...) pourraient survenir entre deux observations, la notion d'événement important est également introduite. Les événements importants sont des prédicats, et incluent au minimum les prédicats d'activation des transitions discrètes. Une *sampling computation* bien formée doit comporter des observations pour chaque date ou un événement important se produit (ex : activation ou désactivation d'une transition discrète). Les *sampling computation* bien formées permettent donc en

[1] pour être précis, la relation s'applique aux systèmes à transition de phase, qui sont des cas particuliers de systèmes hybrides

Algorithm 4.1 Algorithme de vérification dynamique : fonction principale

```
1: /* Recherche d'un état initial */
2: l ← ⊥; s ← ⊥
3: l_found ← false
4: for loc ∈ locs do
5:    if Init_loc(o_0.val) ≠ ⊥ then
6:       if ¬l_found then
7:          l_found ← true
8:          l ← loc
9:          s ← Init_loc(o_0.val)
10:      else
11:         return KO
12: end for
13: if l = ⊥ then
14:    return KO

15: /* Évaluation itérative de la trace */
16: for in ∈ [1 ... n] do
17:    /* Évolution continue */
18:    s ← Succ_l(s, o_i.val, o_i.l)
19:    if s = ⊥ then
20:       return KO
21:    acts ← o_i.acts
22:    /* Recherche des transitions possibles */
23:    while true do
24:       acts ← acts ∪ {ι}
25:       (acts', s', l_new) ← essayerTransitions(acts, s, l)
26:       /* Non-déterminisme implique non-évaluabilité */
27:       if s' = ⊥ then
28:          return KO
29:       if ((acts', s', l_new) = (acts, s, l)) then
30:          /* S'il ne reste plus de transitions actives,
31:          toutes les actions externes doivent avoir été consommées */
32:          if acts' = {ι} then
33:             break
34:          else
35:             return KO
36:       (acts, s, l) ← (acts', s', l_new)
37:    end while

38:    /* Vérification de l'invariant */
39:    if ¬Inv_l(s) then
40:       return KO
41: end for
```

Algorithm 4.2 Algorithme de vérification dynamique : sous-fonction $essayerTransitions$

```
1: Fonction EssaieTransitions. Entrées : acts_obs, state, location
2: t_found ← false
3: s' ← ⊥
4: acts_new ← acts_obs
5: l_new ← location
6: for all t = (location, acts, l') ∈ D do
7:    /* Pour chaque transition analyser la garde ... */
8:    if ¬(acts ⊂ acts_obs) or ¬Garde_t(s) then
9:       continue
10:   s' ← Reinit_t(state, t)
11:   /* ... et l'invariant après réinitialisation */
12:   if Inv_l'(s') then
13:      if ¬t_found then
14:         t_found ← true
15:         l_new ← l'
16:         acts_new ← acts_obs \ acts
17:      else
18:         /* Si plusieurs transitions sont possibles, la propriété n'est plus évaluable */
19:         return (acts_obs, ⊥, location)
20: end for
21: return (acts_new, s', l_new)
```

théorie de vérifier la conformité d'une trace puisque toutes les transitions discrètes sont précisément renseignées. Toutefois cette définition est purement déclarative, et ne donne aucune indication permettant de savoir si une spécification et un environnement de test donné permettent d'observer des *sampling computations*, à moins de disposer par ailleurs d'informations sur un possible écart minimal entre deux actions émises par le système.

Van Osch [62] adopte une approche différente, en identifiant la relation de conformité qu'il propose à la relation *ioco* proposée par Tretmans [61] pour les systèmes de transitions étiquetées. Il généralise cette relation et l'algorithme de génération de tests associé aux systèmes hybrides en posant que les successeurs d'un état du système ne sont plus des états discrets associés par des transitions discrètes, mais des états hybrides associés par des transitions discrètes ou des évolutions continues (qui laissent le temps progresser). Il est montré que la méthode de génération de tests obtenue est correcte et complète. Toutefois, la portée des hypothèses qui sont faites, telle que sa définition basée sur un modèle de temps continu, la rend inapplicable en pratique sans hypothèses supplémentaires car elle ne traite pas les problèmes de discrétisation du temps que nous avons soulevés. Enfin, elle fait l'hypothèse que tous les signaux du système sont observables, ce qui ne correspond pas au problème que nous traitons ; les questions liés à la non-observabilité des variables et à leur influence sur l'évaluabilité des propriétés n'y sont donc pas considérées.

Enfin, Dang et Nahhal [20] utilisent une relation de conformité discrète proche de celle de van Osch, qui néglige les événements importants (au sens de Manna) qui peuvent se produire entre deux points d'observation. Il suppose également la partition des variables de l'automate en deux ensembles, observables et non-observables et définit une relation de conformité proche de celle que nous décrivons en 4.1.2. Cependant, cette relation n'inclut pas d'aspect d'évaluabilité. En effet, la question de l'évaluation

de cette relation n'est pas posée dans la contribution ; elle est en effet définie dans le cadre de travaux d'animation de modèle, et non pour étudier la conformité d'une implémentation concrète à partir de ses traces. Les traces générées par l'approche proposée sont donc correctes par construction, mais la question de la conformité d'une trace *quelconque* n'est pas posée.

4.4.4 Conclusion

Nous avons dans ce chapitre étudié les problèmes qui se posent pour évaluer la conformité d'une exécution à partir d'informations partielles et échantillonnées. Nous avons proposé une solution partielle à ces problèmes reposant sur la définition d'une spécification approchée. Cette spécification est utilisable par un algorithme de vérification dynamique également proposé et applicable aux cas où les observations sont échantillonnées. Cette approche admet dans certaines limites un non-déterminisme de la spécification originale, la non-observabilité de variables et la discrétisation des observations.

Chapitre 5

Mesure de couverture

Les chapitres précédents ont présenté les propriétés hybrides, en détaillant leur utilité ainsi qu'une méthode permettant de tester la conformité d'une implémentation à celles-ci. Nous avons observé que l'exécution de tests sur une implémentation, en suivant cette méthode, permet de déduire quelles parties de la spécification (autrement dit, des propriétés) ont été exercées ; nous démontrons dans ce chapitre comment cette connaissance peut être exploitée pour évaluer l'adéquation [64] des tests réalisés. L'adéquation caractérise la capacité de ces tests à valider le respect des propriétés par le système.

5.1 Evaluation de jeux de tests dans le contexte hybride

Dans cette première section, nous présentons le concept de critère d'adéquation [64], son rôle dans l'évaluation des besoins de validation sous-jacents au test, et les éléments sur lesquels nous proposons de fonder un tel critère. Nous présentons ensuite la notion de profil opérationnel [48], et montrons qu'elle peut être employée pour formaliser ces éléments.

5.1.1 Critères d'adéquation et besoins sous-jacents

Un *critère d'adéquation* [64] est une mesure qui permet de savoir à quel point un jeu de test donné permet de valider l'artefact (implémentation ou modèle) testé, et donc d'apporter de la confiance. Cette mesure peut être booléenne, auquel cas on parle de critère d'arrêt : elle permet de savoir si un jeu de test donné remplit les conditions minimales pour valider un logiciel. Elle peut aussi être numérique et fournir un résultat dans un espace de valeurs plus important (par exemple un intervalle de $\mathbb{R}$). Ceci peut permettre suivant les caractéristiques du critère de comparer des jeux de test. Une telle comparaison peut être utile pour faire un choix entre deux méthodes de production de tests, par exemple. Il est à noter qu'un critère d'arrêt est un cas particulier de critère quantitatif qui peut prendre deux valeurs, et qu'il est toujours possible de définir un critère d'arrêt à partir d'un critère quantitatif en partitionnant les valeurs qu'il peut délivrer en deux ensembles (ces ensembles caractérisent les jeux de test adéquats et inadéquats). En pratique, si un critère délivre un résultat dans $[0, 1]$, on peut en dériver un critère d'arrêt en considérant comme adéquats les jeux de test qui atteignent un score de 0.95 par exemple.

Les propriétés de sûreté que nous considérons sont formalisées par des automates hybrides. Nous avons vu au chapitre précédent que lors de la vérification dynamique d'une propriété, l'algorithme que nous proposons construit itérativement la séquence des états de la propriété qui sont visités lors d'un test [1]. Il est important de noter que ces états ne peuvent être déduits de la seule spécification et des entrées qui sont soumises au système : les variables internes sont évaluées par l'algorithme à partir des entrées soumises au système sous test, mais à partir aussi des sorties qu'il produit. Pour évaluer quelles parties de la spécification (donc de l'automate) ont été éxercées, il est donc nécessaire de travailler à partir de cette séquence d'états reconstituée à l'issue de l'exécution ; l'adéquation des tests ne peut être estimée *a priori*.

Le critère d'adéquation utilisé spécifie l'usage qui est fait de ces données brutes pour évaluer les tests effectués. Les propriétés sont la représentation de besoins de validation issus du réel, plus ou moins critiques et détaillés. *Pour fournir au testeur un outil d'analyse utile des résultats des tests, nous cherchons à définir un critère d'adéquation qui caractérise dans quelle mesure ces besoins ont été satisfaits, et qui permette d'identifier ceux qui ne le sont pas le cas échant*. Leur définition doit donc nécessairement intervenir dans la définition du critère pour que la validation qui est faite des systèmes soit pertinente. Nous allons à présent présenter les possiblités et limites des approches existantes du point de vue de la représentation des besoins et de leur exploitation. Nous proposons ensuite une formalisation de ces besoins inspirée d'un profil opérationnel, qui repousse certaines de ces limites.

5.1.2 Limites des approches existantes dans la prise en compte des besoins en validation

Dans les travaux de Dang et al. [20], le critère de couverture des tests utilisé caractérise la dispersion des tests. Comme nous l'avons discuté en 3.4, ce type de critère caractrise l'exploration qui est faite de l'espace d'état du point de vue de son étendue et de son uniformité, mais ne mesure que ce type de besoins. Par ailleurs, les auteurs exposent qu'il permet de caractériser l'amélioration de la dispersion apportée par un nouvel état visité à un ensemble existant, mais ne permet pas de comparer deux ensembles d'états quelconques.

Dans les travaux d'Esposito et al. [24] que nous avons présentés en 3.3.2.2, le critère d'adéquation est classiquement paramétré par les dimensions de l'espace d'état (qui est supposé être un produit d'intervalles de $\mathbb{R}$), ainsi que par le pas de la grille qui est superposé à cet espace d'état. La variation du pas de cette grille conduit à l'instantiation d'un critère différent et par suite à l'estimation d'un autre valeur de l'adéquation d'un jeu de test.

Intuitivement, plus le pas est grand, plus favorable sera l'estimation de l'adéquation. Ce critère d'adéquation peut donc traduire des besoins de validation portant sur "la densité d'états à visiter" (un pas plus grand correspondant à des besoins plus faciles à satisfaire). La formalisation par les automates hybrides que nous faisons des propriétés induit l'existence de *modes* qui s'interprètent comme différents modes d'opération du système spécifié. Le critère d'Esposito peut être réutilisé en le définissantau niveau de l'espace d'état de chaque mode, puis en aggrégeant les mesures pour en dériver une adéquation globale. Ceci permet, en utilisant un pas de grille distinct pour chaque mode de trduire des besoins de validation propres à chacun de ces modes.

Sur l'exemple du chauffage écologique que nous avons présenté au chapitre 3, si on dispose d'un

[1] ces états sont calculés et peuvent être exportés à la ligne 36 de l'algorithme 4.1 p.51 et à la ligne 14 de sa sous-fonction présentée par l'algorithme 4.2

ensemble d'états T observés lors des tests, on peut le partitionner en plusieurs ensembles $T_{waitContract}$, T_{steady}, etc ... correspondant à l'espace d'état de chaque mode. Dans chacun de ces modes, on utilise le critère d'Esposito avec un pas de grille spécifique (qui est un vecteur, avec une valeur associée à chacune des variables de l'automate).

Ce critère conserve plusieurs défauts cependant, liés au fait que cette mesure est destinée comme la première à caractériser la dispersion uniforme des états visités plutôt que le remplissage d'un ensemble d'objectifs. Il ne prend pas en compte les besoins en validation dans notre exemple, qui ne sont pas uniformes. Les états tels que $(C - T) > \Delta_{crit}$ par exemple sont interdits ; par conséquent il n'est pas pertinent que le critère les prenne en compte : ils ne pourront jamais être visités par un système conforme.

De plus, supposons que le système ait davantage besoin d'être exercé lorsque C et T sont proches (puisque le comportement décrit par la spécification dans ce cas de figure est relativement complexe), et moins testé lorsque leur écart est important (parce que cela correspond à une situation de grande chaleur où le chauffage n'est pas sensé être actif, et où le testeur sait que le risque de défaillance est faible). Une grille uniforme au niveau de l'espace d'état de chaque mode ne permet pas de faire de telles distinctions.

Enfin dans certains modes des variables peuvent être indéfinies. Ainsi dans l'état $waitContract$, la variable $timeRemaining$ est indéfinie, alors qu'elle est définie dans le mode $followRamp$. Pour l'inclure dans le critère d'un de ces deux modes seulement, il faudrait que la grille ne comporte pas toujours les mêmes variables. Ceci n'est pas prévu dans la proposition originale d'Esposito.

5.1.3 Utilisation d'un profil opérationnel pour caractériser des objectifs

Pour traiter ces questions, nous proposons de définir explicitement les besoins de validation à un niveau plus fin que le mode et d'associer une *importance* explicite à chacun des éléments de la spécification. Nous nous inspirons pour cela de la notion de *profil opérationnel* introduite par Musa [48] (nous emploierons indifféremment les termes de profil opérationnel ou de profil par la suite).

Nous avons présenté le concept de profil opérationnel en 3.2.4 comme une décomposition hiérarchique des cas d'utilisation d'un système, au sein de laquelle chaque élément se voit affecter une fréquence d'utilisation. Ceci permet d'expliciter les connaissances et l'expertise disponibles sur le système, et les utiliser dans le processus de validation. Le type de spécification que nous considérons pour les systèmes hybrides possède pour ses éléments de haut niveau une décomposition hirérarchique en automates puis en modes qui permet de proposer une définition de profil semblable à celle de Musa. En revanche l'espace d'état de chaque mode d'un automate est infini dans le cas général. A ce niveau de décomposition, nous associerons donc une fonction dite d'*importance* ou de *densité* qui associe individuellement à chaque état une importance.

Pour représenter les besoins de validation dans le profil, le testeur définit quantitativement l'importance de chaque élément de la spécification (propriété, mode ou état) en lui affectant une valeur dans $\mathbb{R}_+$. 0 signifie qu'aucune exigence de validation n'est applicable à l'élément considéré. une valeur strictement positive indique que l'élément est plus ou moins important pour le critère en fonction de sa valeur. L'interprétation de ces données numériques sera détaillée dans la prochaine section ; nous nous limitons ici à illustrer la structure d'un profil, définie comme suit (et illustrée en Fig. 5.1) :

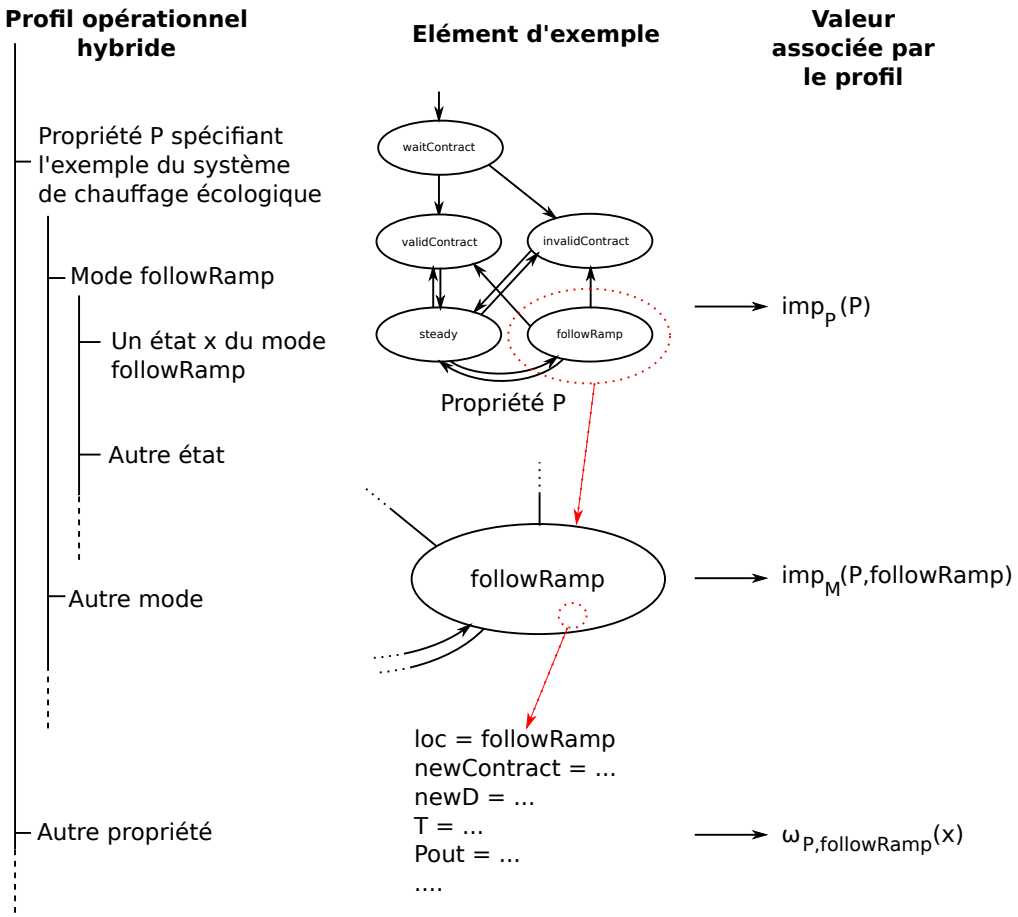


FIG. 5.1 – Composition d'un profil opérationnel hybride

Soit une spécification $\mathcal{S}$ composée d'un ensemble d'IOHA[2] $\{\mathcal{A}_1 \dots \mathcal{A}_n\}$. Un profil opérationnel applicable à $\mathcal{S}$ est un tuple $P = (imp_P, imp_M)$ composé des éléments suivants :

- imp_P est une fonction qui associe à tout automate $\mathcal{A}$ une importance $imp_P(\mathcal{A})$
- imp_M est une fonction qui associe à tout automate $\mathcal{A}$ et un de ses modes m une importance $imp_M(\mathcal{A}, m)$
- imp_E est une fonction qui associe à tout mode m d'un automate $\mathcal{A}$ une fonction de densité $\omega_{\mathcal{A},m}$ définie sur l'espace d'état de m à valeurs dans $\mathbb{R}_+^{|V'_\mathcal{A}|}$, où $V'_\mathcal{A}$ est l'ensembles des variables de $\mathcal{A}$ privé de loc.

On impose en outre que la somme des importances associées aux différents automates, ainsi que les sommes des importances des modes pour chacun de ces automates additionnées valent 1. Lorsque $\mathcal{A}$ et m se déduisent du contexte, on notera ω à la place de $\omega_{\mathcal{A},m}$.

5.2 Une mesure de couverture pour les propriétés hybrides

Nous avons donné une définition de profil opérationnel qui permet d'encoder les besoins en validation en associant un coefficient réel à chaque élément de la spécification, qui spécifie son importance. Nous discutons ici la signification que nous accordons à ces coefficients, et définissons un critère d'adéquation qui évalue la qualité d'une suite de tests sur la base du profil. Les données qui sont évaluées par le critère sont les états visités T de l'automate, inférés par l'algorithme de vérification dynamique que nous avons proposé au chapitre 4.

5.2.1 Interprétation du profil

Les profils opérationnels ont été utilisés pour la génération de test dans les premiers travaux de Musa [48], puis dans d'autres travaux tels que ceux de du Bousquet [22]. Dans ces derniers travaux, le profil est exploité pour calculer une loi de distribution souhaitée pour les données d'entrée. Celle-ci est ensuite utilisée par un processus d'échantillonnage aléatoire de données de test. Des tests statistiques effectués sur les données produites permettent de valider le fait que la génération est bien conforme à la loi de probabilité déduite du profil.

Une première piste pour la définition d'un critère d'adéquation consiterait en s'inspirant de cette approche à analyser la disctribution des éléments de T et de la comparer au profil, que nous interpréterions comme une loi de distribution optimale des tests. Le critère ainsi défini délivrerait le degré de confiance produit par un test statistique sur l'égalité de la loi issue du profil et de la loi observée. Toutefois, un tel critère possèderait une propriété de non-monotonicité : ajouter de nouveaux états à l'ensemble T des états visités inféré par l'oracle pourrait faire décroître la valeur délivrée par le critère. Ceci est incompatible avec le type de critère que nous recherchons, qui doit caractériser la capacité d'un jeu de tests à atteindre un ensemble d'objectifs.

Nous avons donc adapté l'utilisation classique d'un profil opérationnel, qui consiste à l'utiliser pour guider ou évaluer la génération des tests par rapport à une loi de distribution. Nous avons cherché une

[2] Automate hybride à entées/sorties

interprétation qui lui confère la sémantique d'un ensemble d'objectifs à atteindre. Dans notre interprétation, les coefficients associées aux composants de haut niveau (modes et propriétés) de la spécification caractérisent l'importance relative de ces éléments pour le testeur, et les fonctions de densité qui sont associées à chaque état caractérisent *la densité minimale des états visités qui doit être atteinte autour de l'état considéré*.

L'introduction de cette notion de densité locale a été motivée par la constatation suivante : dans un espace d'état infini, les trajectoires discrétisées inférées par l'oracle de test au cours d'une exécution du système ne représentent nécessairement qu'une quantité négligeable de l'espace. Mais s'il n'est pas possible de visiter tous les états, on peut en revanche tenter de se limiter à une exigence de *proximité*. Chaque état de la propriété constitue alors un objectif de test ; s'il possède dans un certain voisinage autour de lui un état visité lors du test on écrira qu'il est *couvert*, et l'objectif correspondant *atteint*.

Le profil intervient pour quantifier ce voisinage : pour un état x appartenant au mode m d'un automate $\mathcal{A}$, la quantité $\omega_{\mathcal{A},m}(x)$ associée à x par le profil caractérise une distance maximale que le plus proche point visité peut avoir avec lui pour qu'il soit considéré *couvert*. Nous allons à présent formaliser ces notions et définir complètement le critère d'adéquation.

5.2.2 Définition du critère

Dans la présente définition de notre critère d'adéquation, nous allons définir tout d'abord la couverture de l'élément de plus bas niveau de la spécification : un état individuel. Nous définirons ensuite la couverture des éléments de plus haut niveau.

5.2.2.1 Couverture d'un état

Nous nous plaçons dans le cas où un automate $\mathcal{A}$ est fixé, ainsi qu'un mode m de cet automate au sein duquel on étudie la couverture des états. Nous avons mentionné précédemment qu'un état appartenant à un automate hybride est couvert s'il a un état visité dans un voisinage quantifié par les données du profil opérationnel. Nous formalisons ici cette première définition intuitive, en nous appuyant sur une notion de *distance* :

Définition 14 (Distance)**.** *Soit $(x, y) \in Val(V) \times Val(V)$ tels que $x \downarrow \{loc\} = y \downarrow \{loc\} = l$ deux états appartenant à un même mode. Soit ω la fonction de densité associée à ce mode par le profil.*

Notons $\omega(x) = (\omega_1(x) \dots \omega_s(x))$, $x = (x_1 \dots x_s, l)$ et $y = (y_1 \dots y_s, l)$, avec $s = |V| - 1$ le nombre de variables de l'automates moins une (loc n'est pas utilisée dans le calcul de la distance). Alors la distance $d_\omega(x, y)$ de y à x est définie ainsi ($\| \|_\infty$ désigne la norme infinie) :

$$d_\omega(x,y) = \begin{cases} 2 \times Sup_{\{i,\ 1 \leq i \leq s \ \wedge\ \omega_i \neq 0\}} |(y_i - x_i) \times \omega_i(x)| & \textit{si } \omega \neq (0, \dots 0) \\ 0 & \textit{sinon} \end{cases}$$

Remarquons que d_ω n'est pas une norme ($d_\omega(x, y) \neq d_\omega(y, x)$ en général). Les deux arguments de la fonction n'ont en effet pas la même sémantique : le premier s'interprète comme les coordonnées d'un état de l'espace à couvrir ; le second représente celles d'un autre état visité lors du test, susceptible de

couvrir le premier. La valeur de la fonction de pondération enfin, qui biaise le calcul de la proximité, est calculée relativement au premier argument (l'état à couvrir). Les expressions $d_\omega(x, y)$ et $d_\omega(y, x)$ n'ont donc pas la même signification et adoptent des valeurs différentes dans le cas général.

Venons-en à la définition de la couverture d'un état. La recherche d'un plus proche voisin est un problème difficile, en particulier lorsque l'on traite des espaces en grande dimension (voir par exemple [10]). Il devient raisonnable si on se limite à un voisinage local de taille réduite. Or cette recherche du plus proche voisin dans notre définition doit être effectuée pour chaque état visité de la spécification, donc elle ne doit pas faire intervenir un algorithme d'une complexité trop importante pour que le critère soit évaluable en un temps raisonnable. De plus, d'après l'interprétation que nous faisons du profil, chaque état possède un voisinage associé, dans lequel un état second visité doit se trouver pour que le premier soit couvert. Il est par conséquent inutile de rechercher des voisins hors de ce voisinage pour estimer une couverture. Notre définition de la couverture d'un état x fait donc référence uniquement au voisinage de x, paramétré par ω :

Définition 15 (Couverture d'un état)**.** *Un état x, dont l'importance est caractérisée par une fonction ω, est couvert par un ensemble d'états visités T ssi $\exists t \in T, d_\omega(x, t) \leq 1$*

Remarquons que cette définition est binaire (un état est couvert ou bien ne l'est pas), contrairement à celle d'Esposito qui donnait un score de couverture réel compris entre 0 et 1 à chaque état. Ceci est conforme à notre interprétation du profil comme un ensemble d'objectifs à remplir. Par ailleurs, le calcul de la distance est définie en utilisant un maximum, et non la norme cartésienne comme le propose Esposito [24]. Ceci également est motivé par l'interprétation que nous faisons du profil : la fonction $\omega(x) = (\omega_1(x) \dots \omega_s(x))$ caractérise la taille d'un voisinage, en spécifiant sa taille dans plusieurs dimensions qui correspondent chacune à une des variables de l'automate (à l'exception de loc). L'utilisation du maximum assure qu'un point visité appartenant au voisinage satisfait indépendamment des conditions de proximité sur *chaque* dimension.

5.2.3 Couverture d'un ensemble d'états et d'un mode

La couverture d'un mode de l'automate est définie par aggrégation des couvertures de chaque état. Elle caractérise ainsi la proportion des objectifs de test qui ont été atteints au sein de ce mode. Cette proportion est biaisée de manière à ce que les différentes zones de l'espace d'état y contribuent à hauteur de l'importance que le profil définit pour elles, via la fonction de densité ω. En particulier, les zones de l'espace où ω s'annule n'interviennent pas dans la mesure. Ceci permet de ne pas compatibiliser les états qui sont ainsi désignés comme inatteignables ou non pertinents pour la validation du système.

Définition 16 (Couverture d'un ensemble d'états)**.** *Soit S un sous-ensemble de l'espace d'état d'un mode m, auquel le profil associe une fonction de densité ω. Pour tout $x \in S$ notons $\omega(x) = (\omega_1(x) \dots \omega_s(x))$, et*

$$\Omega(x) = \begin{cases} \Pi_{i,\omega_i \neq 0}\, \omega_i(x)) & \textit{si } \omega \neq (0, \dots 0) \\ 0 & \textit{sinon} \end{cases}$$

Soit une fonction $pcov$ telle que $pcov(x)$ vaut 1 si un état $x \in S$ est couvert, 0 sinon. Soit T un ensemble d'états de S. On définit $\mu(S)$ l'intégrale (ou la somme) de Ω sur S, et $\mu'(S)$ l'intégrale (ou la somme) de $\Omega.pcov$ sur S. Alors la couverture $scov_\omega(S)$ de S est définie par

$$scov_\omega(S) = \begin{cases} 1 \; si \; \mu(S) = 0 \\ \dfrac{\mu'(S)}{\mu(S)} \; sinon \end{cases}$$

Définition 17 (Couverture d'un mode)**.** *Soit ω la fonction d'importance associée à un mode m d'un automate $\mathcal{A}$ par le profil considéré. Alors la couverture $mcov(\mathcal{A}, m)$ de m par T est égale à $scov_\omega(Val(V))$*

5.2.4 Interprétation de ω comme une densité d'états visités requise au sein d'un mode

Nous avons qualifié les fonctions ω associées par le profil à chacun des modes comme des fonctions d'importance, ou de façon équivalente fonctions de *densité*. Nous pouvons expliquer cette seconde dénomination à présent que nous avons présenté la définition de la couverture d'un mode. Nous allons illustrer cette discussion sur un exemple. Nous n'utiliserons pas tout de suite l'exemple du chauffage pour cela, car sa complexité pourrait encombrer notre discours. Nous allons considérer à la place un exemple-jouet le plus simple possible :

Soit une spécification contenant un unique automate $\mathcal{A}$ avec deux variables d'entrée, x et y. Il possède un seul mode m, au sein duquel les deux variables peuvent évoluer indépendamment dans l'intervalle $[0; 5]$ chacune. Nous définissons pour cet automate un profil opérationnel PO. Puisque la spécification ne contient qu'un seul automate, son importance est nécessairement de 1 : $imp_P(\mathcal{A}) = 1$. Cet automate n'a qu'un mode m, on a donc $imp_M(\mathcal{A}, m) = 1$. Le profil spécifie pour m une fonction de densité ω définie comme suit : $\omega(x, y) = (\omega_1, \omega_2) = (1, 2) \; \forall (x, y) \in [0; 5]^2$.

Imaginons que le premier état visité lors du test soit $t_0 = (2.5, 2.25)$ (Cf Fig. 5.2.a) ; d'après la définition de la distance que nous avons donnée, la distance de t_0 à un état $s = (x, y)$ est de $2 \times Sup(|2.5 - x|, |2.25 - y| \times 2)$. D'après la notion de couverture d'un état, il faut que cette quantité soit inférieure à 1 pour que t couvre x. Il vient que t_0 couvre les points appartenant à la boîte de dimensions 1×0.5 centrée en t_0.

Supposons à présent que d'autres états visités soient ajoutés au calcul de la couverture : les points $(2.5, 2.75)$, $(3.5, 2.25)$ et $(3.5, 2.75)$ sont ainsi ajoutés en sous-figure 5.2.b : de nouveaux états sont alors couverts. Étant donné la façon dont nous avons choisi ces états visités, leurs aires d'influence sont adjacentes : ils couvrent de manière optimale l'ensemble d'états $[2, 4] \times [2, 3]$: on ne peut couvrir cet ensemble d'état avec moins de points. En généralisant à l'ensemble de l'espace d'état, on montre en Fig. 5.2.c comment un ensemble minimal de 50 points visités couvre l'espace d'état, dont le volume est de $5 \times 5 = 25$. La densité d'états visités est donc de 2, ce qui pour tout état x correspond à $\omega_1(x) \times \omega_2(x) = \Omega(x)$. Cette grandeur Ω que nous utilisons dans la définition de la couverture d'un mode pour pondérer le calcul de la propostion de l'espace d'état couvert correspond dans le cas général à la *densité de points minimale* nécessaire pour couvrir l'espace.

On remarque que *sur la dimension associée à la variable x*, chaque état est couvert par un état visité qui se situe à une distance (dans le sens habituel du terme, qui ne fait pas référence à ω) de 0.5. La "densité de voisins visités" sur cette dimension est donc de 1, correspondant à la grandeur ω_1 du profil. De la même façon, il y a 2 états visités par unité de longueur sur la dimension associée à y, correspondant à ω_2.

Les données du profil dans notre interprétation, et plus précisément les fonctions de densité ω, dé-

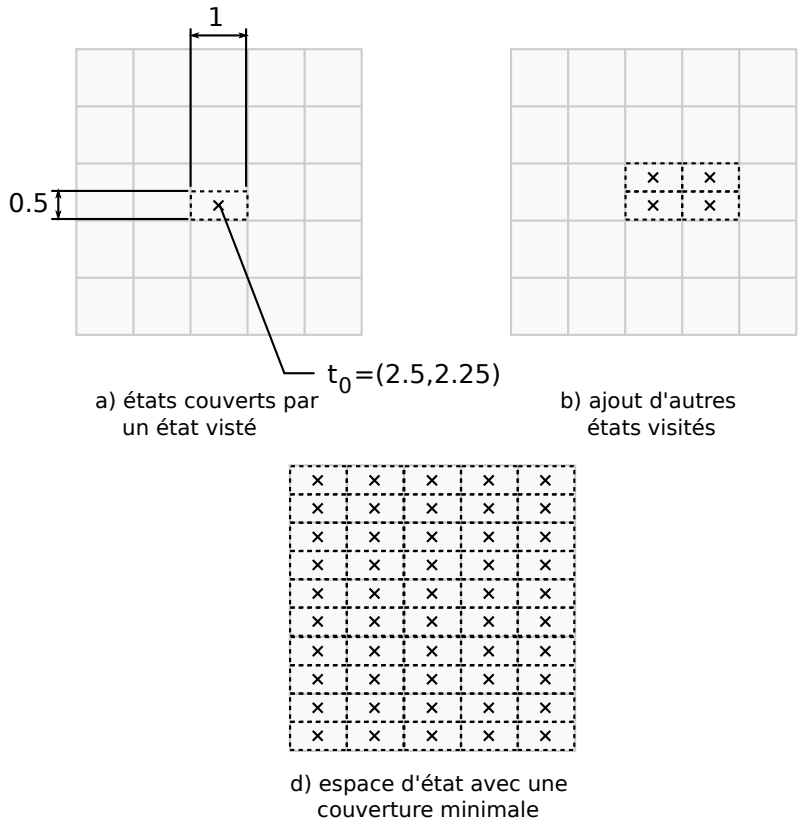


FIG. 5.2 – Illustration de la notion de densité d'états requise

crivent donc une densité d'états requise pour couvrir l'espace d'état, en détaillant cette exigence sur chaque dimension individuellement.

5.2.5 Couverture aux niveaux Propriété et Spécification

La couverture $propcov$ d'une propriété formalisée par un automate $\mathcal{A}$ comportant un ensemble de modes M est définie comme suit :

$$propcov(\mathcal{A}) = \sum_{m \in M} mcov(\mathcal{A}, m) * imp_M(\mathcal{A}, m)$$

La couverture $specov$ d'une spécification S constituée d'un ensemble de propriétés formalisés par un esemble d'automates $\mathcal{A} = \{\mathcal{A}_1, \ldots \mathcal{A}_n\}$ est définie comme suit :

$$specov(S) = \sum_{\mathcal{A}_\gamma \in \mathcal{A}} propcov(\mathcal{A}_i) * imp_P(\mathcal{A}_i)$$

Ainsi, si tous les états des tous les automates d'une spécification sont couverts, la couverture de chaque mode sera égale à 1, par conséquent les couvertures de chaque automate ainsi que celle de la spécification vaudra 1 (de par la contraintes imposée aux coefficients du profil en 5.1.3). A l'inverse, si aucun état n'est couvert, la couverture de la spécification vaut 0.

Cette couverture, qui varie pour chaque élément de la spécification entre 0 et 1 permet d'identifier à chaque niveau le degré de couverture des éléments individuels de la spécification, et donc dans quelle mesure les tests réalisés remplissent les objectifs fixés par le profil pour ces éléments.

5.2.6 Calcul pratique du critère

Nous avons proposé un critère d'adéquation pour les jeux de test hybrides. Il est défini sur un espace d'état quelconque, potentiellement dense. Nous allons voir que dans ce cas, le critère ne peut être évalué par une machine ; toutefois, il est possible d'en calculer une approximation prudente (c'est-à-dire qui ne surévalue pas sa valeur) avec une précision arbitraire en évaluant la couverture d'un nombre fini de points. Cette sous-section décrit cette approximation ainsi que les questions qui se posent pour le choix de ses paramètres.

5.2.6.1 Nécessité d'une approximation calculable du critère

La mesure de couverture proposée pour un ensemble d'états ne peut être calculée dans le cas où l'espace d'état est continu. Lorsque c'est un sous-ensemble S de $\mathbb{R}^n$ par exemple cela implique le calcul d'une infinité de couvertures locales pour évaluer $\mu'(S)$ (Cf. 5.2.3). Nous avons donc cherché une méthode d'approximation de la mesure. Nous allons montrer ici qu'il est possible d'obtenir une estimation prudente de la mesure en évaluant les couvertures locales sur les états d'une grille finie G extraite de S.

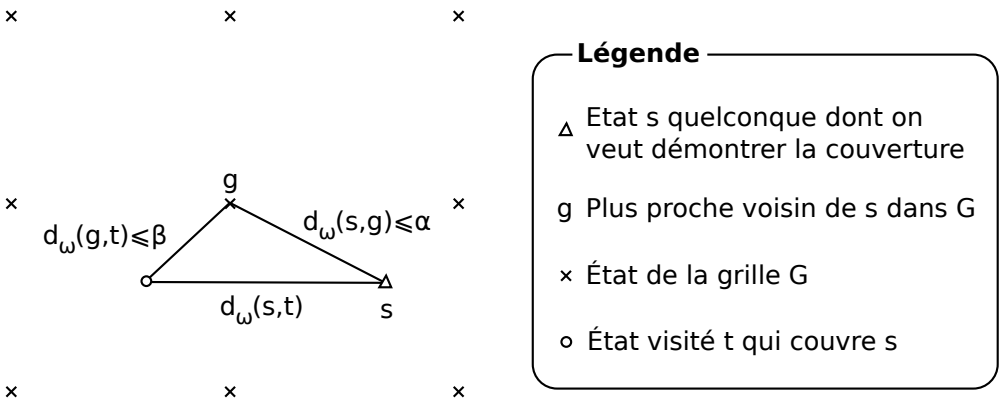


FIG. 5.3 – Intuition fondatrice du théorème d'approximation de la mesure

Dans la suite, nous noterons S un ensemble d'états dont nous souhaitons calculer la couverture approchée, inclus dans l'espace d'état d'un mode m d'automate hybride. On notera ω la fonction de pondération associée à m par le profil opérationnel considéré pour la mesure. On supposera enfin que ω est constante sur S, et on notera $\omega(x) = (\omega_1 \dots \omega_n)\ \forall x \in S$. Enfin, on notera T l'ensemble des états de l'espace d'entrée visités lors des tests.

5.2.6.2 Théorème d'approximation

L'idée fondamentale dans le développement de l'approximation a été de remarquer que lorsque ω est constante, la distance d_ω que nous avons définie en 5.2.3 devient symétrique et acquiert les caractéristiques d'une norme ; en particulier, on obtient la propriété d'inégalité triangulaire $d_\omega(x,y) + d_\omega(y,z) \leq d_\omega(x,z)$. Si on peut montrer qu'une grille G dont on connaît le pas est entièrement couverte, alors on peut alors borner la distance de n'importe quel état de S à un état visité de T. Les différentes distances considérées sont illustrées en figure 5.3 : notre théorème recense des conditions sous lesquelles on dispose d'une borne α sur la distance d'un état quelconque à un point de la grille, et d'une borne β sur la distance de ce point de la grille à un état visité, telles que $\alpha + \beta \leq 1$. Il suit alors que la distance de l'état quelconque à un point visité est inférieure à 1 et donc qu'il est couvert.

Considérons une grille G de pas $d = (d_1 \dots d_n)$ extraite de S et une pondération constante $\omega(x)' = (\omega'_1 \dots \omega'_n) \forall x \in Val(S)$. Notons C la couverture de S par T relativement à la pondération ω, et C' la couverture de G par T relativement à ω'. Nous allons montrer que, si certaines conditions portant sur d et ω' sont vérifiées, alors $C' = 1 \Rightarrow C = 1$.

Par définition, pour tout $s \in S$, la distance (au sens de ω) du plus proche état g de G à s vaut

$$\begin{aligned} d_\omega(s,g) &= \|((g_1 - s_1) * 2\omega_1, \dots, (g_n - s_n) * 2\omega_n)\|_\infty \\ &\leqslant \|\frac{d'_1}{2} * 2\omega_1, \dots, \frac{d'_n}{2} * 2\omega_n)\|_\infty \\ &= \|d'_1\omega_1, \dots, d'_n\omega_n)\|_\infty \\ &= max_{i\in[1;n]}\ d'_i\omega_i \end{aligned}$$

donc $d_\omega(s,g) \leqslant \alpha$ avec $\alpha = max_{i\in[1;n]} * d'_i\omega_i$. Supposons à présent que $C' = 1$, il existe donc un

état visité $t \in T$ "proche" de s :

$$d_{\omega'}(g,t) = \|((t_1 - g_1) * 2w'_1, \ldots, (t_n - g_n) * 2w'_n)\|_\infty \leqslant 1$$

De plus, en notant $\beta = max_{i\in[1;n]} \left\{\frac{\omega}{\omega'}\right\}$ on a cette relation entre d'_ω et d_ω :

$$\begin{aligned} d_\omega(g,t) &= \|((t_1 - g_1) * 2\omega_1, \ldots, (t_n - g_n) * 2\omega_n)\|_\infty \\ &\leqslant \beta.\|((t_1 - g_1) * 2w'_1, \ldots, (t_n - g_n) * 2w'_n)\|_\infty \\ &= \beta.d_{\omega'}(g,t) \leqslant \beta \end{aligned}$$

Finalement, puisque ω est constante d_ω possède les propriétés d'une norme. Par inégalité triangulaire, et sous l'hypothèse précédente que $C' = 1$, on a pour tout état s de S : $d_\omega(s,t) \leqslant \alpha+\beta$. Donc si $\alpha+\beta \leqslant 1$ alors $d_\omega(s,t) \leqslant 1 \forall s \in S$. Il suit que $C = 1$. En résumé, le théorème d'approximation s'exprime ainsi :

Posons, avec les notations précédentes :

$$\alpha = max_{i\in[1;n]} \ * d'_i\omega_i$$
$$\beta = max_{i\in[1;n]} \left\{\frac{\omega}{\omega'}\right\}$$

Si $\alpha + \beta \leqslant 1$ on a :
G **couverte relativement à** $\omega' \Rightarrow S$ **couvert relativement à** ω

5.2.6.3 Arbitrage entre complexité et précision

Les conditions portant sur α et β contraignent le pas de la grille G dont les points seront utilisés pour le calcul de la couverture approchée, et la valeur de la pondération auxiliaire ω' qui paramètre cette msure. Ceci entraîne la possibilité d'un arbitrage entre coût en calcul et précision de l'approximation, résumé en Fig. 5.4 : une forte valeur de α correspond à une grille dotée d'un pas plus important, qui réduit donc la complexité en mémoire et temps de calcul de l'approximation. Au contraire, si α est faible, le pas de la gruille sera réduit, ce qui implique de stocker une plus grande quantité de points de grille en mémoire.

Par ailleurs une forte valeur de β correspond à une pondération auxiliaire ω' modérée, ce qui maximise la valeur de la couverture estimée et améliore la qualité de l'estimation. En particulier, ceci peut permettre de conclure que la couverture de S est complète, alors qu'elle serait sous-estimée si ω' était trop forte ; lorsque $\beta \ll 1$, les coefficients de la pondération ω' sont en effet beaucoup plus forts que ceux de la pondération originale ω. Par conséquent les entrées visitées devront être moins éloignées des états de l'espace d'entrée (au sens de la norme cartésienne) pour les couvrir (par définition de d_ω).

Dans tous les cas, il est toujours souhaitable de respecter la contrainte $\alpha + \beta = 1$. En effet, si $\alpha + \beta < 1$, on pourrait "gratuitement" améliorer le coût ou la précision des calculs de mesure en augmentant l'un des deux termes.

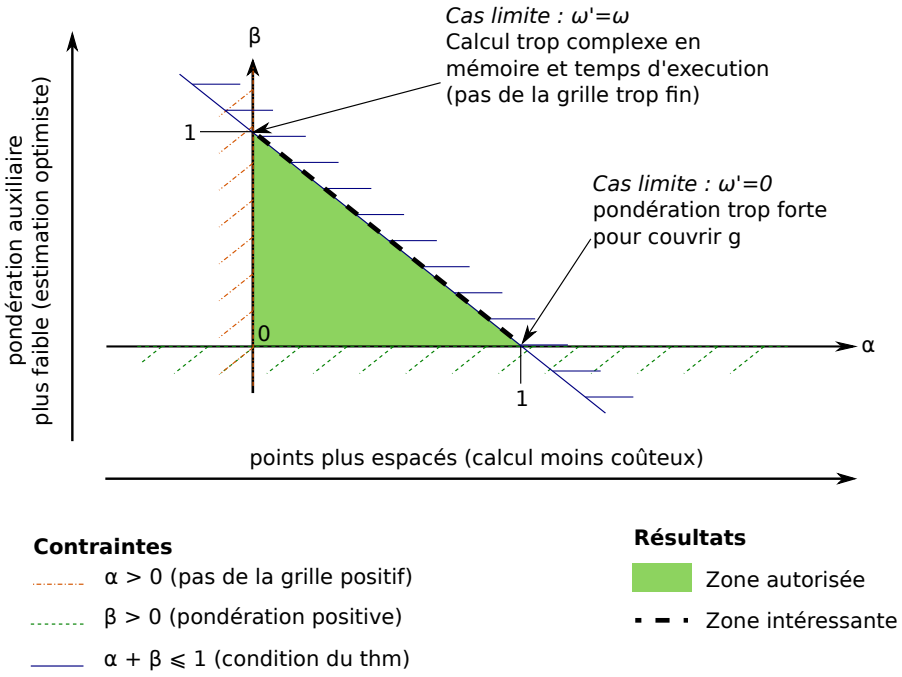


FIG. 5.4 – Influence des paramètres de l'approximation

5.2.6.4 Application à des fonctions d'importance non constantes

Dans le cas où la fonction ω n'est pas constante, mais présente une faible variabilité localement, on peut calculer une approximation prudente en choisissant une partition de l'espace d'entrée en un ensemble d'éléments, au sein desquels la fonction possède des valeurs proches. On approxime ensuite localement la fonction de pondération par une constante égale à la pondération maximale (sur chaque composante du vecteur) de la partie considérée. La Fig. 5.5 donne un exemple d'un tel découpage d'un espace d'état en dimension 2.

Faire cette approximation permet de se rapporter sur chaque élément de la partition au cas précédent de la pondération constante. On peut alors y appliquer le théorème précédent et calculer la couverture. Notons que l'adoption de la pondération "maximale" pour chaque élément est également une approximation pessimiste, nécessaire pour que le théorème précédent reste applicable, et qui peut donc réduire artificiellement le score de couverture obtenu. Ceci pose le problème du choix d'un "bon" découpage, qui ajouté à celui du choix de α et β crée des perspectives intéressantes en matière d'optimisation. Dans le cadre de nos travaux, nous n'avons pas étudié cette question toutefois et n'en avons pas eu besoin car les profils que nous avons utilisés étaient initialement constants par morceaux, ce qui fournissait un découapge immédiat de l'espace d'état.

5.3 Utilisation avancée et discussion du critère

Cette dernière section sur le sujet du critère d'adéquation discute sa sémantique en fonction de particularités que peuvent présenter les profils opérationnels utilisés pour le paramétrer (en particulier lorsque

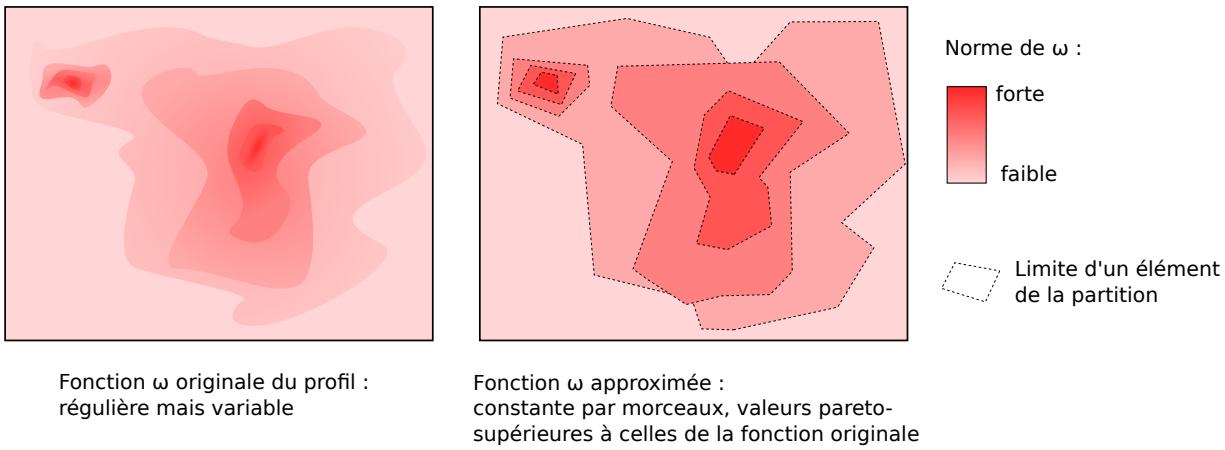


FIG. 5.5 – Approximation du profil pour les calculs de couverture

les fonctions d'importance s'annulent ou comportent des coordonnées nulles sur une partie de l'espace d'état). Elle discute également du rôle de chaque type de variable (interne, d'entrée ou de sortie) et de l'utilisation qui peut en être faite lors de la définition du critère.

5.3.1 Exclusion d'états de la mesure

La définition de la distance d_ω conditionnne l'ensemble des états qui sont couverts par un état visité. Un cas particulier intéressant existe avec l'ensemble des états x tels que $\omega(x) = 0 \ldots 0)$. Par définition de la distance, la distance de n'importe quel autre état à ces états d'importance nulle est également nulle ; n'importe quel autre état les couvre. Par ailleurs, ils n'interviennent pas dans le calcul de la couverture de leur mode, par définition de μ' (voir la définition de la couverture d'un mode en 5.2.3).

Les zones de l'espace d'état qui sont réputées inatteignables ou sans intérêt pour la validation de la propriété peuvent ainsi être exclues du calcul de la couverture.

5.3.2 Exclusion de variables de la mesure

Une propriété similaire existe pour les variables individuelles : si l'une des coordonnées de $\omega(x)$, correspondant à une certaine variable v est nulle pour tout état x de l'espace d'état, alors l'estimation de la couverture de x n'est jamais influencée par la valeur de v. Ceci signifie qu'on peut ainsi construire des profils qui ignorent certaines variables dans des modes particulier.

Ceci permet de ne pas créer d'exigences de couverture artificielles portant sur des variables qui n'ont pas d'intérêt au sein du profil (parce qu'elles ne sont liées à aucun besoin de validation concret). C'est également utile pour exclure les variables qui n'ont pas de signification dans le mode en question et/ou dont la valeur est indéfinie. Dans notre exemple du système de chuffage, nous avons ainsi spécifié que la variable $timeRemaining$ adoptait constamment la valeur $\perp$ dans le mode $waitContract$. Faire porter une exigence de couverture sur cette variable n'aurait pas de sens.

Enfin, d'un point de vue technique cela permet de réduire la complexité en mémoire et en temps de

l'algorithme de calcul de la couverture.

5.3.3 Rôle des variables internes et de sortie dans le profil

Notre profil porte sur toutes les variables des automates qui spécifient des propriétés, qui peuvent être internes, d'entrée ou de sortie. Ceci permet d'exprimer des exigences de plusieurs sortes. Fixer des objectifs portant sur les variables d'entrée permet classiquement de spécifier des données qui doivent être présentées au système. Mais les variables internes et de sortie peuvent également être employées dans le profil.

Considérons pour commencer le rôle des variables internes. Dans notre exemple du système de chauffage, la variable $timeRemaining$ dans le mode $followRamp$ décompte le temps restant avant la réalisation du contrat courant. Elle permet de spécifier la consigne de température à suivre et de contrôler la transition vers le mode $steady$, une fois la nouvelle température de consigne atteinte.

Si le testeur pense que le système risque d'exhiber des défaillances à la fin de ce compte à rebours, il peut construire son profil de manière à ce qu'il requière davantage de test dans les états proches de sa fin. Ceci est possible en faisant intervenir la valeur de la variable $timeRemaining$ dans le calcul de la fonction ω (ω aura des valeurs plus fortes pour les états tels que $timeRemaining$ est proche de zéro). Celà permet de définir des objectifs de test portant non pas sur les valeurs concrètes qui sont soumises au système, mais sur des grandeurs qui sont des artefacts de la spécification (et sont évaluées par l'oracle lors de l'exécution d'un test).

L'utilité d'inclure des variables de sortie dans le profil en revanche est moins évidente. En effet, à moins de disposer d'hypothèses particulières sur l'implémentation du système concret, on ne peut savoir quelles sorties il peut produire. Toutefois, si de telles hypothèses sont disponibles, faire intervenir les variables de sortie dans le profil permet de spécifier que certains scénarios doivent avoir été observées lors du test pour valider le système. Dans notre exemple du système de chauffage, le testeur pourrait savoir que l'implémentation qu'il teste est équipée d'un nouveau modèle de résistance chauffante qui doit être testée à haute température, et il incluerait dans son profil des fonctions de densité dont la composante correpondant à P_{out} ne serait pas nulle pour les grandes valeurs de P_{out}. Le sens intuitif de cette exigence serait : "Il faut observer un certain nombre d'états où la puissance de sortie produite par le système était importante".

Chapitre 6

Étude de cas et validation empirique

Ce chapitre est dédié à l'illustration et la mise en pratique de notre approche sur l'exemple du système de chauffage présenté en 2.1. Cette mise en pratique est réalisée à l'aide de l'outil HyATT (pour Hybrid Automata Testing Tool) que nous avons développé au cours des travaux. Il implémente les algorithmes et la méthode que nous avons proposés pour évaluer la correction de systèmes hybrides vis-à-vis de propriétés de sûreté, et mesurer l'adéquation des tests réalisés. Au fil du texte, nous présentons par étape l'instanciation de l'exemple et la manière dont elle réalisée dans HyATT. Nous discutons enfin les résultats obtenus et les conclusions qu'il est possible d'en tirer sur l'approche.

6.1 Écriture de l'automate formalisant la propriété considérée

Cette section décrit le processus d'écriture de la spécification en temps discret de l'exemple. Cette spécification se compose d'un seul automate, qui est décomposé comme discuté en 4.4.1 en deux parties : une partie discrète qui décrit les modes et les transitions, et une partie numérique qui décrit les aspects quantitatifs de la spécification (trajectoires, gardes, réinitialisations).

6.1.1 Définition de la partie discrète de l'automate

Dans l'outil que nous avons développé, la spécification de l'automate commence par la description de ses signaux (variables et actions) et de leur type (car les actions en particulier doivent être définies pour définir les transitions). Notre exemple comprend 11 variables et 5 actions qui ont été présentées respectivement en 2.2.2 et 2.2.5. Elles peuvent être saisies via une interface illustrée en annexe (Fig. A.1 p.96).

Une fois établie la liste des signaux, nous définissons les modes, et les transitions entre ces modes avec les actions qui leurs sont associées. Ces éléments ont été discutés précédemment en section 2.1 et résumés par la Fig. 2.3 p.20.

Il est à noter que HyATT permet d'associer deux actions à une transition (une d'entrée, facultative, et une de sortie). C'est un artefact syntaxique qui équivaut à spécifier une première transition, qui est

nécessairement suivie d'une deuxième instantanément. Il est introduit pour spécifier des comportements courants tels que la réponse immédiate à une action par une autre. Dans notre exemple, cela correspond entre autres à l'action (*ack* ou *nack*) qui est immédiatement renvoyée par le système lorsqu'il reçoit un nouveau contract (associé à la réception de l'action *newContract*) de l'environnement

Cette partie de la spécification est sauvegardée par HyATT sous la forme d'un fichier XML (reproduit en Fig. A.2 en annexe). La variable interne loc n'y apparaît pas, puisqu'elle est implicitement introduite via la spécification de modes. Comme nous le verrons dans la suite, ce fichier est utilisé ensuite à plusieurs fins : il sert à générer le squelette de la spécification numérique, et à générer du code qui assiste le testeur pour compléter cette spécification et écrire un pilote de test. Il est enfin est utilisé pour instancier l'oracle de test.

6.1.2 Définition de la partie numérique de l'automate

Nous allons à présent présenter les différents éléments qui spécifient les aspects quantitatifs de la spécification d'un automate. Ils sont présentés tour à tour, et accompagnés d'illustrations tirées de l'application de notre approche au système de chauffage. La partie numérique de la spécification du système de chauffage se compose de 810 lignes de C++ (dont 240 écrites par l'utilisateur, le reste étant composé de macros, commentaires, et mise en forme auto-générés). Nous ne l'avons donc pas incluse ici entièrement, mais en présentons des extraits pertinents.

6.1.2.1 Validateurs d'états initiaux

Comme nous l'avons discuté en 4.4.1 la spécification en temps discret d'un automate fait intervenir, en plus de la partie dscrète déjà présentée, une partie numérique. Cette seconde partie est composée de fonctions qui calculent pour chaque mode les invariants, les transitions via leurs les gardes et leurs réinitialisations, et les trajectoires de l'automate. Dans notre implémentation, il s'y ajoute un dernier type de fonction utilisé pour déterminer l'état initial de l'automate lors de l'exécution d'un test ; l'algorithme 4.1 suppose en effet que l'état initial de l'automate est connu initialement. Toutefois, il ne fait pas d'hypothèse sur la manière dont il est déterminé.

Lorsqu'on spécifie un automate avec HyATT, c'est l'écrivain de la spécification qui le fait en fournissant un type de fonction supplémentaire, que nous avons appelé *validateur d'état initial*. Il peut exister au plus un validateur associé à chaque mode, et c'est une fonction dont le paramètre est une valuation des variables externes ; s'il existe des états initiaux de l'automate compatibles avec cette valuation (c'est-à-dire des états initiaux dont la restriction aux variables externes est égale à la valuation passée en paramètre) la fonction en fournit un. Au début de son exécution, l'oracle consulte les valeurs des variables externes et les soumet aux validateurs de chaque mode pour trouver celui qui est actif initialement, et initialiser l'algorithme.

Pour le système de chauffage, nous avons écrit un unique validateur, associé au mode $waitContract$, puisque c'est le seul mode où l'automate peut débuter une exécution ; il est illustré en Fig. 6.1. Ce validateur accepte la valuation qui lui est fournie (en renvoyant la valeur $true$) si la température ambiante est dans sa plage de valeurs admissibles (ceci est vérifié par l'appel à la fonction $tempIsValid$). Cette valuation est alors complétée en affectant des valeurs aux variables internes $oldD$ et $hasNextContract$ (les autres sont laissée indéfinies).

```
bool GreenClimPtyDynamics::initStateValid_location0(ProtectedAccessValuation &newCState) {
    (void) newCState;

    /* This code is the body of a function that MUST return true
     * iff its parameter is a valid initial state in
     * location #0 (name : 'waitContract').
     * See the HyATT manual for more information on how to read and write the state.*/

    double T = READ(T, new);
    WRITE(oldD, T, new);
    WRITE(hasNextContract, -1, new);

    return tempIsValid(T, T);

}
```

FIG. 6.1 – Validateur d'état initial pour le mode $waitContract$

6.1.2.2 Gardes et invariants

Les fonctions spécifiant les gardes et invariants vérifient qu'un état donné permet l'activation de transitions ou qu'il est acceptable pour un mode donné, respectivement. La Fig. 6.2 illustre par exemple la garde associée aux transitions qui quittent le mode *steady* pour le mode *invalidContract*. Cette fonction est évaluée par l'oracle lorsque l'automate, intialement dans le mode *steady*, reçoit un nouveau contrat. Cette évaluation permet de déterminer si une transition vers le mode *invalidContract* est possible, c'est-à-dire si les paramètres du nouveau contrat sont inacceptables.

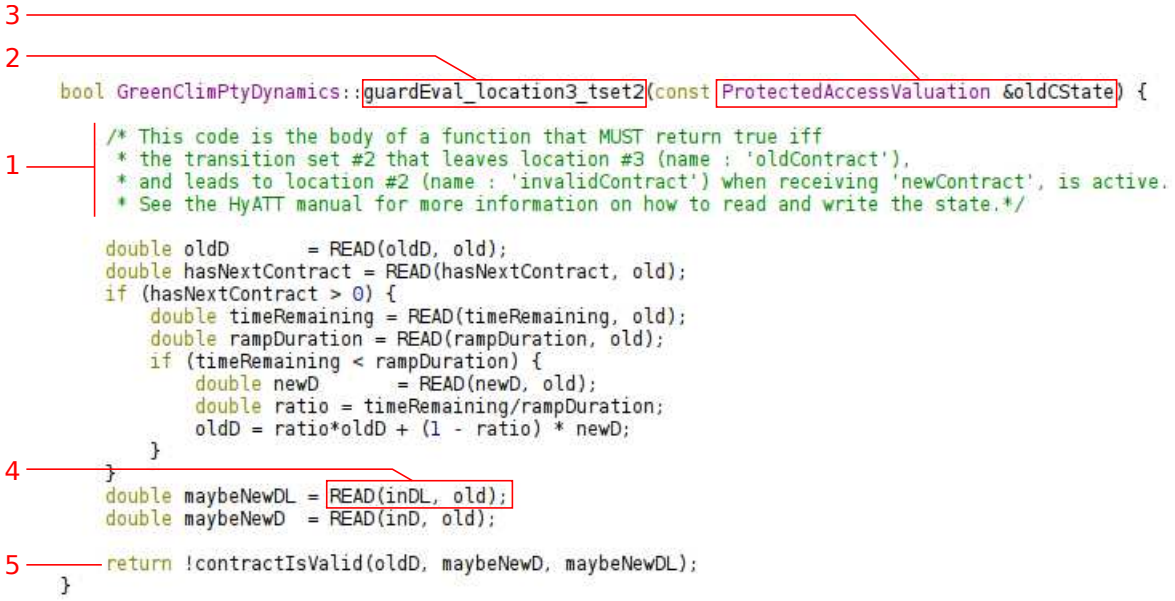
Pour cela, la fonction lit l'état interne de l'automate afin de calculer la consigne courante (représentée dans l'extrait de code de la Fig. 6.2 par la variable *oldD*), en fonction de la présence ou non d'un changement à venir (caractérisé par la valeur de la variable *hasNextContract*) et des paramètres de ce changement mémorisés au sein de l'état. Cette consigne courante est comparée aux paramètres du nouveau contrat reçu (au sein de la fonction contractIsValid, non détaillée ici) pour juger de la faisabilité de celui-ci.

Les fonctions d'évaluation des invariants associées à chaque mode, qui vérifient qu'un état est valide au sein de ce mode, sont spécifiés de la même façon que les gardes.

6.1.2.3 Réinitialisations

Les réinitialisations associées à chaque ensemble de transitions existant entre chaque mode associent un état avant une transition et l'état suivant. Considérons un exemple au sein de la spécification du système de chauffage : lorsque la température de consigne instantanée que suit le système atteint une nouvelle valeur stable après une période d'évolution linéaire, l'exécution correspondante de l'automate qui spécifie ce comportement effectue une transition du mode $followRamp$ (qui spécifie l'évolution linéaire de la température de consigne) au mode $steady$ (qui spécifie une température de consigne stable au cours du temps).

Lors de cette transition, les variables de l'automate changent de valeur : la variable $oldD$ qui représente la consigne initiale du système (la seule, s'il n'y a pas de contrat mémorisé) prend la valeur de la variable $newD$ (la température de consigne cible de l'ex-contrat courant, qui arrive à échéance). La variable $hasNextContract$ qui caractérise l'existence d'un contrat mémorisé en cours d'exécution prend la valeur $false$ (encodée ici par le réel -1). Les variables $newD$, $timeRemaining$ et $rampDuration$ qui



1. Le corps de cette fonction a été généré automatiquement par HyATT à partir de la spécification discrète. Des commentaires sont automatiquement insérés pour renseigner l'utilisateur sur la nature de cette fonction (ici, c'est une garde), ainsi que ses paramètres (modes concernés avec leur nom, action d'entrée éventuelle).
2. Le nom de la fonction est généré automatiquement, ainsi que du code (non présent sur cette capture) qui la connectera au modèle d'automate exploité par l'oracle lors de l'exécution de tests.
3. L'état de l'automate est passé en paramètre de la fonction.
4. Des macros auto-générées (READ, WRITE et KEEP) permettent d'accéder en lecture et/ou en écriture aux variables de l'état.
   – ici, inDL n'est pas une variable locale ou globale définie en C++. C'est le nom d'une variable du modèle tel que défini via l'interface. L'expansion des macros permet d'accéder à sa représentation interne, cachée à l'utilisateur.
   – le second paramètre de READ et WRITE (`old` ou `new`) est utilisé dans les fonctions qui manipulent deux états successifs (comme des fonctions de réinitialisation) pour sélectionner l'état surlequel agit la macro.
5. Du code personnalisé (comme la fonction `contractIsValid()` de cet exemple) peut être ajouté lors de l'assemblage de la classe par HyATT.

FIG. 6.2 – Extrait de la partie numérique de la définition d'un automate, spécifiant une garde

```
bool ClimatisationPtyDynamics::reinit_location4_tset2(const ProtectedAccessValuation &oldCState,
                                                      ProtectedAccessValuation &newCState) {
    /* This code is the body of a function that computes the variables reinitialization when
     * taking the transition set #2 that leaves location #4 (name : 'followRamp'),
     * and leads to location #3 (name : 'steady').
     * See the HyATT manual for more information on how to read and write the state.
     * This function MUST return true iff everything went fine.*/

    // Ces deux variables seront définies à l'itération suivante
    WRITE_NEW_DOUBLE(oldD, READ_OLD_DOUBLE(newD));
    WRITE_NEW_DOUBLE(hasNextContract, -1);

    // Les autres variables ne sont pas réinitialisées : elles auront par la suite
    // une valeur indéfinie

    return true;

}
```

FIG. 6.3 – Réinitialisation d'une partie des variables lors d'une réinitalisation du mode $followRamp$ vers le mode $steady$

caractérisaient les paramètres du contrat mémorisé avant la transition deviennent indéfinies et prennent donc la valeur $\perp$.

La Fig. 6.3 illustre comment ces fonctions sont spécifiées au sein de notre outil. Seules les variables internes peuvent être réinitialisées, les variables externes sont lues à l'exécution depuis les interfaces existant avec le système et l'environnement.

6.1.2.4 Évolution continue de l'état au cours d'une étape de simulation

Les fonctions qui calculent l'évolution continue des variables internes déterminent leurs changements de valeur lorsque le temps progresse d'une durée donnée. Elles acceptent comme paramètres deux états (l'ancien et le nouveau, qui est complété), ainsi qu'un paramètre réel qui représente la durée du pas de simulation (qui peut donc varier d'un appel à l'autre). La Fig. 6.4 illustre ainsi le calcul de l'évolution de l'état qui intervient lorsqu'une durée $timeStep$ s'écoule dans le mode $followRamp$ de l'automate. L'effet de cette fonction est de réduire la valeur de $timeRemaining$ d'une quantité égale au temps qui vient de s'écouler, et de conserver les valeurs des autres variables internes inchangées.

On remarque qu'aucun test n'est fait sur les valeurs des variables pour vérifier si l'évolution linéaire de la température doit s'interrompre car le contrat courant est arrivé à échéance. Lorsque ceci arrive, une transition interne doit pourtant amener l'automate dans le mode $steady$. De façon générale, ces tests sont placés au sein des gardes qui contrôlent l'activation de transition vers d'autres modes. Pour ce mode particulier, nous avons précédemment illustré en Fig. 6.2 la fonction spécifiant la garde des transitions vers le mode $steady$.

6.1.2.5 Protection des accès aux variables

Nous avons illustré comment les fonctions composant la spécification d'un automate en temps discret pouvaient lire et écrire l'état de l'automate. Pour se prémunir de certaines erreurs et oublis dans la spécification, HyATT implémente un mécanisme de contrôle d'accès aux variables que nous détaillons ici.

```
bool GreenClimPtyDynamics::contSuccComp_location4(const ProtectedAccessValuation &oldCState,
            ProtectedAccessValuation &newCState, double timeStep) {

    /* This code is the body of a function that computes a continuous evolution
     * in location #4 (name : 'followRamp').
     * See the HyATT manual for more information on how to read and write the state.
     * This function MUST return true iff everything went fine.*/

    KEEP(hasNextContract);
    WRITE(timeRemaining, READ(timeRemaining, old) - timeStep, new);
    KEEP(rampDuration);
    KEEP(oldD);
    KEEP(newD);
    return true;

}
```

FIG. 6.4 – Évolution continue des variables internes de l'automate

Nous avons vu que le fait de ne pas donner explicitement de valeur à certaines variables internes (par exemple aux variables $newD$, $timeRemaining$ et $rampDuration$ dans l'exemple présenté en 6.1.2.3) signifie du point de vue de l'outil que leur valeur après réinitialisation est indéfinie (c'est-à-dire égale à une valeur spéciale $\perp$ qui n'est pas un élément de $\mathbb{R}$). Cette absence d'affectation est ici volontaire puisque l'objectif est bien de rendre ces variables indéfinies, mais elle pourrait également résulter d'un oubli lors de l'écriture de la spécification. Si HyATT utilisait pour gérer ce genre de cas une stratégie naïve telle que laisser inchangée la valeur d'une variable non affectée, cette valeur inchangée (et incorrecte) serait utilisée dans les calculs suivants. Ceci pourrait provoquer l'apparition de défaillances dont il faudrait rechercher la source. HyATT implémente pour éviter ceci un système de contrôle d'accès aux variables qui interrompt l'évaluation d'un test et renvoie une erreur si une des fonctions écrites par l'utilisateur et exploitée par l'oracle tente d'accéder à une variable indéfinie.

De la même façon, si l'exécution d'une fonction conduit à une tentative d'accès à une variable en lecture ou écriture alors que cela n'a pas de sens au vu de son rôle (par exemple, si une fonction d'évaluation de garde tente d'écrire une partie de l'état), une exception est levée. Nous verrons par la suite que cela est également vrai pour le code d'interface avec l'environnement et le système. Par exemple, si le code écrit pour réaliser l'interface avec le système sous test tente d'écrire une variable interne de l'automate, l'évaluation du test sera interrompue et une erreur sera renvoyée.

6.2 Définition du profil opérationnel

6.2.1 Situation motivant l'application d'un processus de validation

La définition d'un profil opérationnel formalise comme nous l'avons expliqué en 5.1 les exigences qui doivent être remplies par l'exécution d'un jeu de tests, afin de valider un système. Dans notre approche, l'exécution de tests et l'évaluation de la correction des exécutions du système sous test par l'oracle permet de collecter la séquence d'états de l'automate qui ont été visités. Le profil établit quantitativement quelles parties de la spécification doivent être exercées au minimum, ce qui s'interprète comme un ensemble d'exigences portant sur cet ensemble d'états.

Pour motiver les choix de conception du profil dans notre étude de cas, nous imaginons que ce profil est conçu pour vérifier le bon fonctionnement d'un algorithme défectueux qui a été récemment corrigé dans le système. L'algorithme en question est en charge de la détermination de la consigne de température

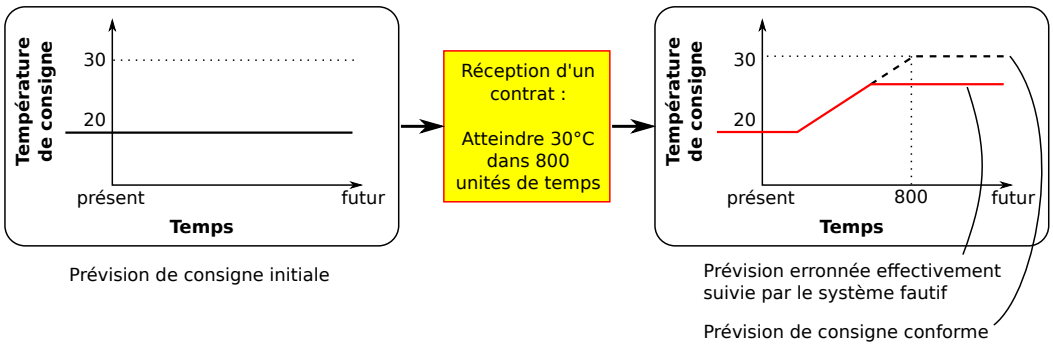


FIG. 6.5 – Type de faute motivant la création du profil

que le système doit suivre. L'ancien algorithme traitait parfois incorrectement les paramètres des contrats reçus et cessait de faire varier la température de consigne avant d'avoir atteint la nouvelle consigne cible.

Nous illustrons en figure 6.5 le type de faute qui était produite par l'algorithme défectueux : imaginons que le système n'ait initialement aucun contrat à venir (il devrait donc sans intervention extérieure tenter de maintenir indéfiniment une température de 20 degrés). A un moment donné il reçoit un contrat consistant à atteindre la température de 30 degrés, 800 unités de temps plus tard (avec la politique de changement de température paresseuse que nous avons détaillée en 2.1, p.11). La courbe en pointillés sur le graphe de droite montre la consigne que devrait tenter de suivre le système. La courbe pleine montre le genre de consigne erronnée que pouvait suivre l'ancien algorithme défectueux.

6.2.2 Besoins traduits par le profil et identification des variables pertinentes

Nous avons montré en 5.3 que le critère d'adéquation que nous proposons ne prend pas en compte les valeurs de certaines variables pour lesquelles le profil opérationnel n'exprime pas d'objectif. Ce cas se produit lorsque les variables en question sont indépendantes des exigences que le profil formalise. Nous allons détailler ici quelles sont ces exigences, et quelles variables de l'automate y sont rattachées.

Pour s'assurer que le problème de prévision de consigne a bien été corrigé, il est raisonnable de tester le système lorsqu'il est en "fin de contrat", dans les situations qui suscitaient précédemment des fautes. Le profil que nous définissons portera donc sur des états du système au sein du mode $followRamp$, lorsque le contrat en cours de traitement est presque atteint ; c'est-à-dire lorsque la variable $timeRemaining$, qui décroit linéairement dans ce mode pour mesurer le temps restant avant la réalisation du contrat, a une valeur proche de zéro. Cette variable interviendra donc dans la définition du profil, et celui-ci n'exprimera d'objectifs que pour les états du mode $followRamp$.

Par ailleurs, on souhaite exercer le système avec différentes températures de consigne cible et de température ambiante afin de vérifier son comportement sur des scénarios courants dans son utilisation normale. Les variables T et $newD$ doivent donc être prises en compte dans le profil.

Enfin, on suppose avoir constaté empiriquement que les fautes se produisaient plus souvent lorsque la puissance renouvelable disponible P_{renew} était importante (c-à-d. supérieure à un seuil estimé à environ $300W$). On souhaite donc valider le comportement du nouvel algorithme sur ce point particulier, et P_{renew} interviendra donc dans le profil.

Reprenons maintenant la liste des variables du système établie en 2.2.2 pour vérifier que nous n'en oublions pas qui soient pertinentes :

Variable	Rôle	Considérée dans le profil
T	Température ambiante	OUI
P_{renew}	Puissance renouvelable disponible	OUI
P_{out}	Puissance consommée	-
inD	Nouvelle consigne	-
$inDL$	Délai accordé	-
$oldD$	Consigne la plus ancienne	-
$hasNextContract$	Présence d'un contrat actif	-
$newD$	Consigne à venir au terme du contrat	OUI
$timeRemaining$	Délai restant pour le contrat courant	OUI
$rampDuration$	Délai de changement	-
loc	Mode de la propriété	OUI

Nous supposons que d'après nos connaissances sur le système, la puissance produite par le système n'est pas corrélée avec l'apparition de fautes ; il est donc inutile de faire intervenir $POut$ dans le profil. Par contre, on ne sait pas si la température ambiante joue un rôle et on souhaite le vérifier. Le profil devra donc spécifier que diverses valeurs de T et $newD$ doivent être exercées.

inD et $inDL$ sont les paramètres des contrats reçus : elles ont bien entendu une influence indirecte sur la consigne que va suivre le système, mais cette influence est capturée par les variables $oldD$, $newD$, etc. qui représentent les paramètre effectivement considérés par le système (puisque tous les contrats ne sont pas acceptés) – en outre, inD et $inDL$ n'ont de sens qu'aux instants où un contrat est envoyé et leur signification n'est pas spécifiée autrement. Nous ne les considérerons donc pas pertinentes pour le profil.

$hasNextContract$ enfin vaut nécessairement $true$ dans le mode $followRamp$ et n'a donc pas d'intérêt pour le profil, puisqu'elle ne varie jamais dans les cas qui nous intéressent.

6.2.3 Identification des domaines des variables à considérer dans le profil

Nous avons présenté les objectifs qui motivent la création du profil, et en avons déduit quelles variables seraient considérées dans celui-ci, pour chaque mode de l'automate (en l'occurrence, les variables T, P_{renew}, $newD$ et $timeRemaining$ sont considérées dans le mode $followRamp$, et aucune variable dans les autres modes). Nous allons à présent quantifier l'importance que le profil associe à chaque état dans le mode $followRamp$ via la définition d'une fonction d'importance ω associée à ce mode.

La figure 6.6 illustre le profil et les données numériques que nous avons choisies pour décrire l'importance de chaque variable, projetées sur les variables P_{renew} et $newD$. La fonction d'importance ω est constante par morceaux, et peut adopter deux valeurs différentes, suivant la valeur de P_{renew}. Lorsque $P_{renew} < 300$, la composante associée à $newD$ est égale à 0.2, ce qui signifie qu'un ensemble d'états visités qui satisfont le profil présente au minimum 0.2 valeurs différentes de $newD$ par degré (autrement dit, on souhaite tester des valeurs de $newD$ réparties au maximum tous les 5 degrés). Cette exigence sur

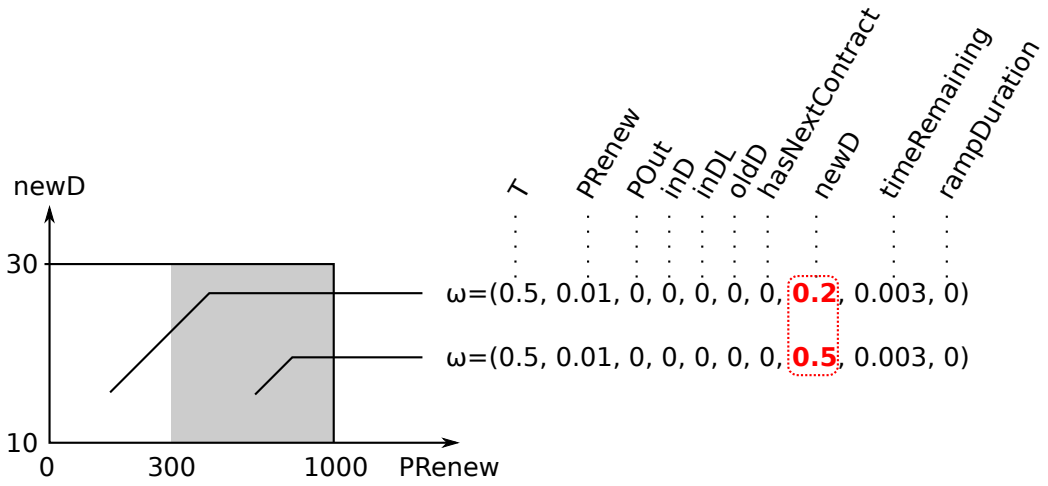


FIG. 6.6 – Profil opérationnel exact utilisé dans l'exemple du système de chauffage

la répartition des valeurs de $newD$ est plus élevée pour les états tels que $P_{renew} \geq 300$ (avec une valeur requise tous les 2 degrés). Ceci traduit l'exigence liée à la validation du système lorsque la quantité d'énergie renouvelable disponible est importante.

Les composantes associées aux variables qui sont exclues du profil, telles que $POut$, on une importance de 0, donc le profil n'exprime aucune exigence dessus et elles n'interviennent pas dans le calcul de la couverture. Les valeurs associées à T, P_{renew} et $timeRemaining$ requierent en revanche une densité d'état positive puisque leurs combinaisons doivent être exercées lors du test, d'après les exigences précédemment exprimées.

Notons que les domaines indiqués pour chaque variable dans le profil sont liés aux objectifs fixés pour la validation de la propriété, ils ne préjugent pas du domaine que sont susceptibles de couvrir effectivement les variables lors d'une exécution. Il est possible d'écrire un profil qui n'exprime pas d'objectifs pour des zones de l'espace d'état qui sont pourtant atteignables et/ou qui spécifie des objectifs que le système dans son environnement ne pourra jamais atteindre, car cette information sur l'(in)atteignabilité de certains états est inconnue lors de l'écriture du profil.

6.2.4 Instantiation du profil modifié pour l'estimation de la couverture

Nous avons développé si-dessus un profil opérationnel applicable à l'exemple du système de chauffage. Il peut être utilisé pour paramétrer le critère d'aédquation que nous proposons en 5.2. Ce critère permet d'estimer la couverture d'une spécification par un ensemble d'états visités, inférés par l'oracle lors de l'exécution d'un test. Nous avons toutefois expliqué en 5.2.6 que cette mesure de l'adéquation de tests vis à vis d'un profil ne pouvait être calculée directement ; elle nécessite en effet une intégration sur un domaine continu et l'exécution d'une infinité de tests en théorie.

La sous-section 5.2.6 montre également qu'on peut sous certaines conditions simples calculer une approximation prudente du critère. L'idée consiste à définir pour chaque mode un partitionnement de l'espace d'état tel que les valeurs de ω varient peu sur chaque élément. Il faut ensuite définir pour chaque

élément une fonction d'importance auxiliaire ω' constante qui est pareto-supérieure à ω en tout point[1]. Sur chaque élément de la partition, on superpose ensuite une grille régulière de points (assimilables à des objectifs de test) sur lesquels on estime la couverture des états visités lors du test, par rapport à ω'. Si cette grille et ω' satisfont aux conditions du théorème exposé en page 66, cette estimation est prudente : la couverture estimée est toujours inférieure à la valeur réelle (c'est-à-dire la couverture de l'espace entier relativement à ω). Elle en est plus ou moins proche en fonction du choix de ω' et du pas des grilles.

Détaillons à présent la façon dont tous ces éléments sont pris en compte par notre outil pour estimer la couverture de tests. HyATT n'effectue pas de pré-traitements sur l'espace d'état et le profil pour déterminer les paramètres de l'approximation ; il prend directement en entrée une partition de l'espace d'état de chaque mode. Il requiert également la fourniture des pas des grilles d'états à couvrir, qui seront superposées à chaque élément des partitions précédentes, pour évaluer la couverture. Les éléments de la partition qu'il est possible de décrire sont des polyèdres convexes, définis comme l'intersection de l'espace d'état du mode considéré avec une collection de demi-espaces, délimités par des hyperplans (l'utilisateur founit les coefficients d'équations linéaires qui caractérisent ces hyperplans).

Nous avons vu ci-dessus qu'au sein du mode $followRamp$, certains états (par exemple ceux pour lesquels la variable $timeRemaining$ possède une valeur élevée, signe qu'ils correspondent à un contrat qui est encore loin d'être terminé) n'ont pas d'intérêt. Le profil associe donc une importance nulle à ces états, et il n'y a pas lieu de définir de grille qui les inclue. Par ailleurs, nous avons vu qu'il convenait de traiter avec davantage d'attention les états tels que P_{renew} possède une valeur importante, supérieure à $300W$. Nous construirons donc deux grilles, l'une qui couvrira raisonnablement les états tels que $timeRemaining$ et P_{renew} sont faibles et une autre qui couvre plus étroitement les états tels que P_{renew} est supérieure à 300 (en testant davantage de valeurs pour $newD$).

Sur la base de ces connaissances, nous coupons les grilles que nous avons établies par deux hyperplans, en supprimant les états qui vérifient l'une des deux équations suivantes :

$$\begin{aligned} T - newD + \delta_{secteur} &< 0 \\ -T + newD + \delta_{nom} &< 0 \end{aligned}$$

Décrivons à présent la formation de la fonction de pondération ω'. Dans notre exemple, la définition de cette fonction auxiliaire est aisée car ω est déjà constante sur chacun des éléments de le partition que nous avons formée (Cf. Fig. 6.6). La constante $1/\beta$ qualifie le ratio qui sera appliqué pour l'estimation de la couverture entre ω et ω' (et conditionne donc le pas des grilles qui seront instanciées, comme discuté en 5.2.6). Nous avons choisi ici $\beta = 2$.

6.2.5 Aspects techniques

Techniquement, les informations relatives au profil sont lues par HyATT depuis un fichier XML ; la représentation du profil complet pour notre exemple est donnée en annexe, figure A.3. Nous décrivons ici le traitement que fait en interne notre outil de ces profils, en le reliant à l'algorithme de calcul de la

[1] C'est-à-dire que pour toute valuation $x = (x_1 \dots x_s)$ des variables de l'automate à l'exception de *loc*, si on note $\omega(x) = (\omega_1(x) \dots \omega_s(x))$ et $\omega'(x) = (\omega'_1(x) \dots \omega'_s(x))$ alors $\omega'_i(x) \geq \omega_i(x)\ \forall i \in [1; s]$

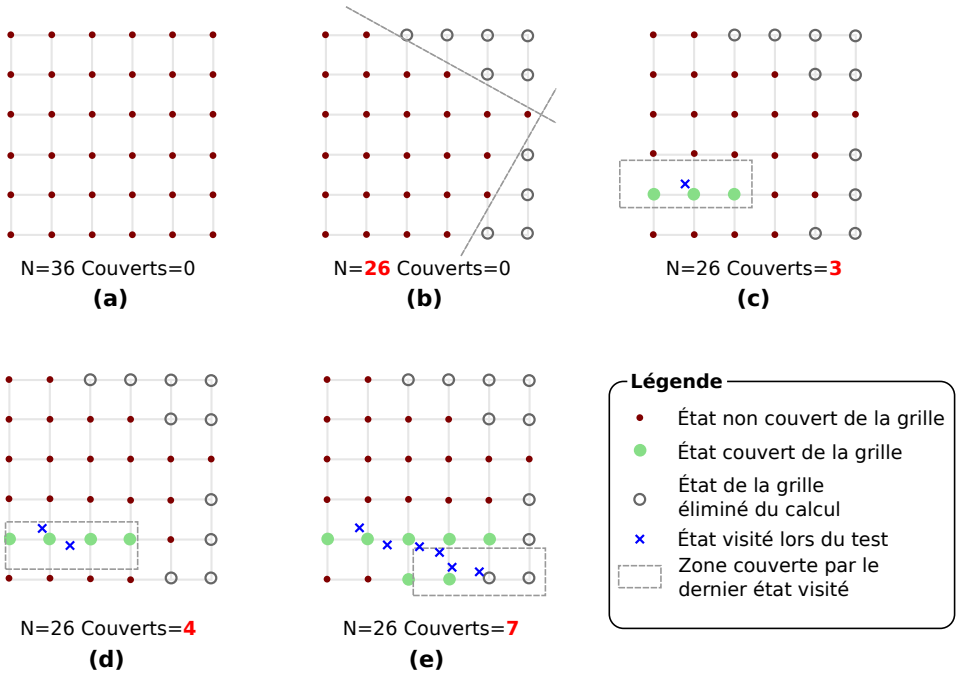


FIG. 6.7 – Comptabilisation des états visités pour l'évaluation de la couverture

couverture approchée que nous avons proposé en 5.2.6. Les étapes en sont décrites ci-dessous, en lien avec l'illustration proposée en Fig. 6.7 :

(a) La grille est instanciée grâce aux paramètres du profil opérationnel lus depuis le fichier XML

(b) Ce fichier comporte également les coefficient d'hyperplans utilisés pour découper les grilles (initialement parallèlépipédiques) et leur donner les formes de polyèdres convexes composant une partition de l'espace d'état (la figure ne représente qu'une seule grille, dans le cas général on en utilisera plusieurs pour paver l'espace d'état).

(c) A chaque itération, l'état atteint de l'automate est reporté sur la grille et couvre certains de ses points. Le pas de la grille et l'étendue du voisinage affecté par un état visité dépendent des paramètres ω, α et β comme expliqué précédemment.

(d) Chaque état visité est susceptible de couvrir de nouveaux états de la grille. Seuls les états couverts de la grille sont mémorisés, la séquence des états visités ne l'est pas (c'est inutile pour le calcul de la couverture). Nous verrons en 6.5 que cette séquence peut toutefois être exportée à l'exécution si elle est utile pour des traitements externes.

(e) Les états éliminés lors de l'application des hyperplans pour le découpage en polyèdres ne peuvent naturellement plus être couverts par les états visités puisqu'ils ne font plus partie de la grille.

6.3 Définition de l'environnement

Nous avons discuté dans la section précédente les détails de la spécification du système de chauffage que nous utilisons comme exemple conducteur, et la façon dont ces éléments sont traités et représentés par HyATT, l'outil que nous avons développé pour la mise en oeuvre de notre méthode. Nous allons à présent décrire l'environnement physique que nous utilisons pour cette application. Cet environnement

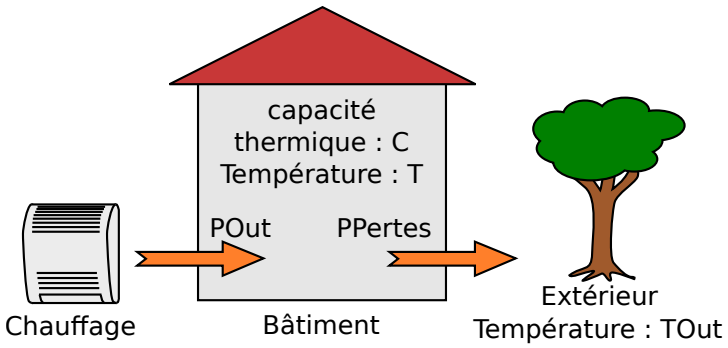


FIG. 6.8 – Modèle thermique d'évolution de la température

est une simulation numérique qui fournit l'évolution des variables externes et l'occurrence des actions externes de l'automate.

6.3.1 Modèle physique

Nous modélisons l'environnement (le bâtiment au sein duquel est installé le système chauffage, habité par des utilisateurs) comme un objet de capacité thermique C constante à la température T, qui reçoit une puissance de chauffe $POut$ du système de chauffage et transmet une puissance $PPertes$ à l'environnement (proportionnelle à l'écart entre T et $TOut$). L'environnement est à la température $TOut$, qui peut varier au fil du temps. Ces éléments sont illustrés en figure 6.8 et liés par l'équation suivante :

$$\frac{dT}{dt} = \frac{POut - \alpha * (TOut - T)}{C}$$

6.3.2 Signaux de commande envoyés au système

La description que nous avons faite du modèle physique de l'environnement contraint partiellement son comportement, mais pas entièrement. Il reste à fixer les règles d'évolution de la température extérieure P_{out} et les dates d'envoi de contrats par les utilisateurs, ainsi que leurs paramètres. Étant donné que l'environnement ici est simulé, ces choix constituent la stratégie de génération de tests qui sera appliquée pour tester le système. Cette stratégie détermine la couverture qui sera faite de la propriété par les tests. Nous présenterons en 6.6 les différentes stratégies que nous avons utilisées pour cela.

6.3.3 Aspects techniques

Nous donnons ici les détails pertinents liés à l'implantation du modèle d'environnement que nous avons utilisé au sein de notre outil de test. Le code de HyATT inclut une classe généraliste dénommée $EnvInterface$ qui représente l'interface du programme avec l'environnement du système sous test. Cette classe, destinée à être dérivée d'une manière propre à chaque contexte applicatif, comporte une fonction essentielle nommée $provideValues$ dont le rôle est d'échantillonner les signaux de l'environnement. Elle peut être implémentée comme une classe d'interface avec des capteurs concrets qui mesurent

des signaux sur un environnement physique, comme un système simulé (ainsi que nous l'avons fait ici), ou comme un mélange des deux. Durant la mise au point du logiciel, nous avons par exemple constaté qu'il était possible de créer simplement un environnement composite où l'évolution des grandeurs physiques était simulée par un code écrit en C++ tandis que l'émission et les paramètres des contrats étaient produits par un modèle écrit en LUTESS [56], un formalisme dédié aux systèmes synchrones.

Dans le détail, la fonction $provideValues$ de notre classe d'interface accepte en paramètre une liste d'actions vide, un état initialement indéfini de l'automate (représentée par le même type de donnée que ceux qui sont passées aux fonctions spécifiant l'automate) et la durée du pas de simulation qui s'achève. Elle écrit les informations relatives aux signaux environnementaux (évolution des variables et occurrence des actions) dans ses paramètres correspondants (passés par référence) ; ces données seront ensuite transmises à l'interface avec le système sous test et enfin à l'oracle.

6.4 Intégration du système

Le dernier composant intervenant dans le processus, en conjonction avec l'environnement et l'oracle (associé à l'algorithme d'évaluation de l'adéquation) est bien sûr le système sous test lui-même. L'objectif que nous poursuivons ici est d'étudier l'évolution du critère que nous avons défini sur un cas d'étude avec plusieurs méthodes de génération, et de discuter l'utilité du profil opérationnel pour guider cette génération. Il est pertinent pour cela d'utiliser un système correct, et nous avons donc réalisé une implémentation simulée du système de chauffage, conforme à la spécification que nous avons décrite au long de ce texte.

D'un point de vue technique, cette implémentation a été intégrée de la même façon que l'environnement au programme de test, en dérivant une classe de HyATT dénommée $SUTInterface$, qui a un rôle similaire à la classe $EnvInterface$ que nous présentions précédemment (elle comprend en particulier une méthode nommée $reactToEnvironment$ appelée à chaque pas de simulation pour ajouter les réactions du système à l'état partiel écrit par l'environnement). On peut remarquer toutefois que l'état partiel qui est passé à la classe pour être complété au cours d'un pas de simulation n'est pas traité de la même façon suivant que le système est simulé ou interfacé via des capteurs : dans le premier cas, le modèle qui simule le système peut exploiter les données fournies par l'environnement pour calculer la réaction du système. Dans le second cas, ces réactions sont mesurées sur le système réel et les données fournies par l'environnement ne sont utilisées que par l'oracle.

6.5 Dérivation du pilote de test

Une fois que l'oracle, ainsi que les connexions avec l'environnement et l'oracle sont définis au sein de l'outil, il reste une dernière étape technique à accomplir pour créer le programme de test : la définition d'un pilote de test. HyATT inclue une classe générique nommée $TestDriver$ qui a pour rôle d'exercer tour à tour l'environnement, le système et l'oracle pour exécuter effectivement l'évaluation de la propriété et de la couverture des tests. Cette classe doit elle aussi être dérivée pour chaque contexte d'application spécifique, afin que le testeur puisse spécifier les derniers détails du processus de test (tels qu'un critère d'arrêt) et ceux liés à l'exportation des résultats.

Pour préciser ces éléments, la classe $TestDriver$ dispose de plusieurs fonctions qui sont appelées à différents points de l'exécution d'un test (notamment à la fin d'une iteration pour exporter des résultats, et une fois par itération pour décider si l'exécution doit continuer). Ces callbacks peuvent être surchargés pour personnaliser l'exécution des tests. Ainsi, pour implanter notre exemple nous avons surchargé une fonction pour exporter l'état de l'automate à intervalles réguliers et une autre pour décider de l'arrêt, l'exécution étant limitée à un nombre fixe de 1.5 millions d'itérations.

6.5.1 Compilation et exécution du programme de test

Pour créer le programme de test complet, il reste à ajouter une fonction principale qui instancie et exécute le test (Telle que celle qui est reproduite en figure A.7), et compiler tous les fichiers précédemment créés en les liant au code d'HyATT. Les figures A.5 et A.6 en annexe résument l'ensemble du processus de spécification et de compilation en présentant les fichiers et opérations mis en jeu à chaque étape.

Pour donner un ordre de grandeur des performances de l'application (peu optimisée pour la performance jusqu'ici), le programme – intrinsèquement séquentiel – exécute 300000 itérations de la boucle de test par seconde, sur une machine à base de Core 2 Duo à 2GHz[2].

6.5.2 Correction des erreurs

Lors des premières exécutions du test, la boucle a été interrompue par diverses erreurs ; certaines étaient des erreurs de l'implémentation (qui ne réinitialisait pas correctement une partie de son état lors de la réception d'un contrat), d'autres provenaient de la spécification. Par exemple, certaines variables qui auraient dû être conservées lors d'une réinitialisation n'étaient pas explicitement affectées (ceci à cause d'un oubli lors de l'écriture de la spécification). L'outil les considérait donc à juste titre comme indéfinies et ceci conduisait à une erreur à l'itération suivante lorsqu'une fonction chargée d'évaluer un invariant tentait d'accéder à leur valeur. De ceci, nous tirons deux enseignements :

- Le système de contrôle d'accès aux variables s'avère utile puisqu'il a effectivement détecté un oubli dans la spécification.
- Des informations détaillées sur l'exécution et les erreurs rencontrées sont nécessaires pour localiser et corriger rapidement ces erreurs. Durant cette étude de cas, nous avons eu besoin de deux éléments :
  - L'historique de l'état de la propriété (et du système sous test, lorsque celui-ci est simulé), exporté par un pilote de test conçu dans ce but, nous a permis de détecter rapidement les incohérences entre le comportement du système et sa spécification en permettant de tracer leur origine. Enfin, il permet de consulter l'état (défini ou non) des variables internes.
  - Des informations sur la nature et l'origine de la faute sont également nécessaires pour localiser l'élément de la spécification concerné, lorsque celle-ci contient une erreur. Actuellement, notre outil fournit le type d'erreur (ex : accès à une variable indéfinie, impossibilité de progresser dans le temps, etc.) et le numéro de l'itération où la faute est apparue. Nous avons remarqué que cela était insuffisant toutefois et qu'il faudrait ajouter d'autres informations aux erreurs

[2]Ce résultat est obtenu en activant les options d'optimisations standard du compilateur (-O2 passé à gcc) et sans exporter de données lors de l'exécution du test.

remontées pour gagner du temps : l'étape en cours de la boucle de test, la variable concernée (le cas échéant) et la fonction de la spécification mise en jeu. En l'occurrence, nous avons dû rechercher certaines de ces informations manuellement à l'aide d'un outil de débogage, ce qui est inutilement long et inadapté à un usage courant.

6.6 Étude empirique de l'influence de la méthode de génération de tests utilisée sur la progression de la couverture

Dans cette section, nous rapportons et discutons les observations que nous avons réalisées en utilisant différentes stratégies de génération de test sur notre exemple conducteur, et en mesurant la progression du critère d'adéquation que nous avons proposé.

6.6.1 Stratégies de génération utilisées

Nous avons appliqué plusieurs méthodes de génération de tests à notre exemple. Pour chacune nous avons mesuré la progression de la couverture au cours du temps, afin de montrer de quelle façon la connaissance du profil opérationnel peut être exploitée pour produire des tests mieux ciblés.

Nous avons employé quatre méthodes, qui exploitent les données du profil de différentes façons. La première méthode $M1$ génère des contrats aléatoires pour le système (tirés uniformément à une distance de 3 degrés au plus de la température ambiante), avec une distribution qui les rend souvent réalisables en leur donnant un délai assez long (la puissance renouvelable disponible effectue quand à elle une marche aléatoire entre les bornes de son domaine).

La seconde méthode $M2$ tente d'explorer systématiquement et uniformément l'ensemble des valeurs de consigne possibles pour les contrats : la valeur de consigne des contrats successifs progresse linéairement d'une borne du domaine autorisé à l'autre, à pas constant ; lorsqu'une valeur de contrat devrait sortir du domaine autorisé, la génération des consignes reprend à partir du voisinage de cette borne, dans l'autre sens.

La troisième méthode $M3$ échantillonne un état appartenant à une des grilles qui n'a pas encore été couvert. Elle génère ensuite une série de contrats très courts (dont la durée est calculée pour être tout juste faisable) afin de parvenir à la consigne de cet état à couvrir. Ceci permet de réaliser plus de contrats et donc plus d'objectifs du profil. En effet, notre profil ne spécifie d'objectifs que pour les états correspondant à une fin de contrat, donc cette méthode de génération doit intuitivement le couvrir plus rapidement.

La dernière méthode $M4$ reprend la stratégie d'exploration systématique des valeurs de consigne possibles de $M2$, en générant des contrats faisables et courts comme la méthode $M3$.

6.6.2 Comparaison des stratégies

6.6.2.1 Production et traitement des résultats

Pour comparer ces méthodes, nous les avons appliquées toutes les quatre sur notre exemple. Pour chacune, nous avons réalisé une simulation de 1.5 million de pas de simulation avec 100 graines aléatoires différentes, pour un total de 600 millions de pas de simulation. Au cours de la simulation (d'une durée de 35 minutes au total sur notre machine à base de Core 2 Duo à 2GHz) nous avons exporté la progression de la couverture pour une centaine de dates régulièrement réparties au sein de la période de temps totale simulée.

Les courbes d'évolution de la couverture que nous présentons sont moyennées sur les 100 exécutions que nous avons réalisées pour chaque méthode. Il est à noter que la progression de la couverture varie peu d'une graine à l'autre ; nous avons choisi pour chaque méthode le pas de simulation p_0 pour lequel les simulations correspondant aux différentes graines avaient atteint en moyenne 50% de couverture et estimé la variance des valeurs de couverture atteintes à p_0. Cette variance est estimée à $9.93 * 10^{-5}$ pour la méthode $M1$, $5.66 * 10^{-5}$ pour la méthode $M2$, $4.37 * 10^{-5}$ pour la méthode $M3$ et $7.04 * 10^{-5}$ pour la méthode $M4$ (la couverture prend ses valeurs entre 0 et 1). Toutes les courbes obtenues pour une même méthode sont donc fortement semblables, ce qui valide empiriquement notre choix de 100 graines aléatoires pour la comparaison.

6.6.2.2 Présentation des résultats et discussion

La figure 6.9 Présente les résultats que nous avons obtenus pour toutes les méthodes. On constate que les méthodes 1 et 2 permettent une progression nettement moins rapide que les autres. Ceci s'explique essentiellement par la façon dont les délais accordés aux contrats sont choisis. En effet, les délais de changement de consigne imposés sont longs, ce qui implique que moins de contrats sont traités par unité de temps. Par ailleurs les contrats peuvent être interrompus en cours d'exécution, remplacés par de nouveaux contrats plus récents. Étant donné que le profil ne progresse que lorsque la propriété est dans un état qui correspond à une fin de contrat mené à terme, il est donc logique que la progression de la couverture soit ralentie par cette caractéristique des méthodes 1 et 2.

Entre les méthodes 1 et 2, on remarque que la méthode 2 progresse plus vite sur le long terme que la méthode 1. Ceci s'explique par la plus grande amplitude des états qu'elle visite : en effet, une marche aléatoire telle que celle qui est mise en oeuvre par la méthode 1 est susceptible de rester dans un voisinage local qui évolue peu. C'est la raison qui a poussé dans le domaine de la robotique à développer des algorithmes d'exploration de l'espace différents, visant à maximiser l'étendue des états visités. L'algorithme RRT dont nous parlions dans les chapitres précédents en est un parfait exemple. L'algorithme 1 est ici pénalisé car le fait de visiter une deuxième fois le même état (ou deux états très proches) de la propriété lors du test ne fait pas progresser l'estimation de la couverture.

La méthode M3 a été conçue pour exploiter les objectifs non encore couverts du profil pour guider la génération. L'algorithme commence par échantillonner un état qui n'a pas encore été couvert (en favorisant les zones les plus importantes du profil), et tente de l'atteindre en générant une séquence de contrats courts qui mènent la propriété à celui-ci. La raison pour laquelle nous avons essayé cette stratégie est basée sur l'intuition que, lorsque la couverture a beaucoup progressé et couvert les objectifs les plus "fa-

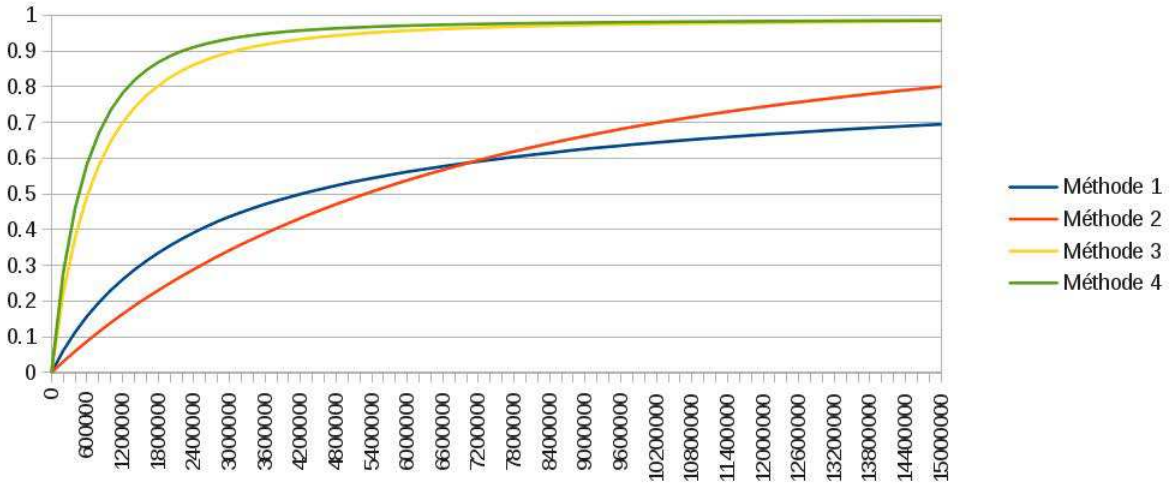


FIG. 6.9 – Évolution de la couverture en fonction du nombre d'itérations pour les quatre méthodes de génération utilisées

ciles" du profil, ceux qui restent nécessitent une stratégie ciblée pour être atteints. Une génération aurait peu de chance de produire une séquence de commande propre à couvrir les états restants rapidement, en particulier si ceux-ci sont "profonds" (c-à-d. nécessitent des conditions environnementales ou une séquence de commande complexe pour être exercés). Nous présentons en figure 6.10 les résultats obtenus par les seules méthodes M3 et M4.

Contrairement à l'intuition initiale, on constate que la méthode M4 se comporte plus efficacement que la méthode M3, et ce tout au long de l'exécution. Nous expliquons ce résultat par deux raisons :

- Le profil ne contient aucun état "profond", ou qui sont plus difficiles à atteindre pour M4 que pour M3 ; par ailleurs, la stratégie d'exploration systématique de M4 consistant à balayer le domaine d'une variable dans les deux sens s'adapte bien au profil et au système considérés. L'intérêt de l'échantillonnage effectué par M3 est donc réduit.
- En échantillonnant aléatoirement ses objectifs parmi ceux qui ne sont pas encore remplis, M3 acquiert des caractéristiques de localité semblables à celles d'une marche aléatoire, dans la mesure où la stratégie peut revisiter une zone de l'espace déjà couverte pour atteindre un objectif non couvert.

Nous en concluons que dans le cas de propriété suffisamment simples portant sur des systèmes contrôlables, il est préférable de maximiser l'étendue des états visités plutôt que de cibler les états non encore visités pour optimiser la vitesse de progression de la couverture.

6.6.2.3 Caractéristiques souhaitables pour les méthodes de génération

En conclusion, c'est la méthode 4 de génération de tests qui produit les meilleurs résultats en termes de couverture du profil, car elle bénéficie de plusieurs avantages sur les autres :

- Elle exploite les caractéristiques du profil pour produire des tests qui le satisferont plus rapidement

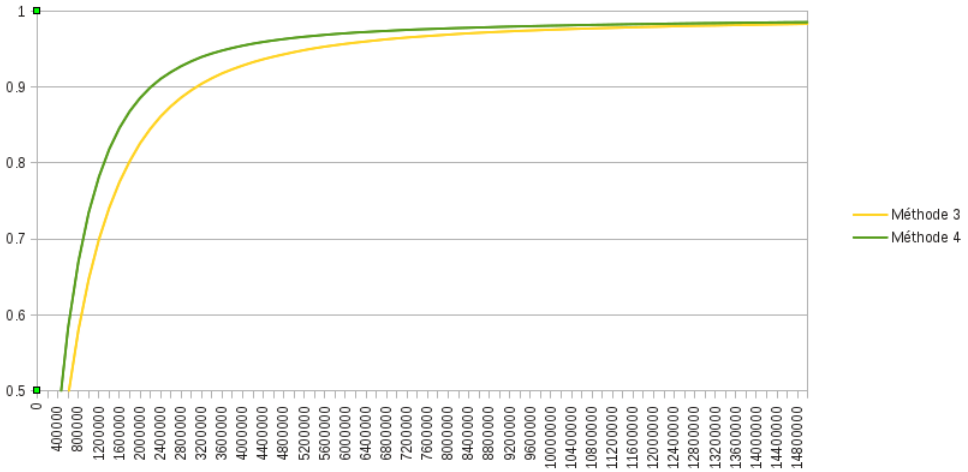


FIG. 6.10 – Évolution de la couverture en fonction du nombre d'itérations pour M3 et M4

(dans notre exemple, en produisant des contrats courts, réalisables et qui ne se chevauchent pas)
– Elle visite des états répartis sur tout le domaine des variables traitées par le profil.

D'autres caractéristiques des méthodes de génération de test pour les propriétés hybrides qui ne sont pas essentielles ici, mais qui semblent constituer des pistes d'étude ultérieure intéressantes sont :

– L'utilisation du profil pour échantillonner des buts à atteindre, notamment lorsque certains correspondent à des états "profonds".
– L'exploitation des connaissances disponibles sur la sémantique de la propriété et le fonctionnement du système sous test pour guider la génération vers ces états profonds.

6.6.2.4 Conclusion sur l'étude de cas

Ces résultats en tant que tels ne présentent pas d'intérêt majeur car les méhodes de génération employées sont rudimentaires ; nous avons mentionné au chapitre 3 l'existence de méthodes plus élaborées dont l'efficacité est solidement établie, une étude complète se devrait de les intégrer à la comparaison. En revanche, nous avons démontré l'applicabilité de notre approche de bout en bout, supportée par le prototype d'outil que nous avons développé, et montré qu'elle conduisait à la production de données pertinentes pour la comparaison de stratégies de génération de test, vis-à-vis d'objectifs et de priorités de validation prédéfinis.

Chapitre 7

Conclusion et perspectives

Dans ce dernier chapitre, nous résumons les contributions et conclusions que nous retirons du travail présenté dans ce manuscrit. Nous replaçons ensuite ce travail dans un contexte plus général et identifions plusieurs perspectives qui en constituent des extension possibles.

7.1 Bilan général

7.1.1 Une approche de test fonctionnel des systèmes hybrides

Dans cette thèse, nous avons abordé la validation des systèmes hybrides sous un angle original. Les contributions proposées dans le domaine des systèmes hybrides s'attachent principalement à l'analyse des modèles hybrides pour établir des propriétés de ces modèles telles que l'atteignabilité de certains états (comme nous l'avons exposé au chapitre 3) ou la synthèse de contrôleurs (c'est-à-dire d'implémentations conformes à ces modèles).

Notre approche traite le problème inverse : soit une implémentation quelconque, nous posons la question de savoir par quel moyen on peut admettre ou réfuter que cette implémentation est conforme à une spécification hybride (composée d'une collection de propriétés hybrides). Si notre approche ne détecte aucun défaut, elle établit que l'implémentation considérée satisfait à certaines exigences de fiabilité spécifiées par un profil opérationnel, ce qui induit de la confiance dans le fait qu'elle est conforme à sa spécification.

7.1.2 Évaluabilité des propriétés en temps discret

Le premier problème que nous avons abordé concerne la testabilité des propriétés hybrides : les propriétés que nous considérons comme spécification peuvent être définies sur un modèle de temps dense, conformément à l'usage dans la modélisation des systèmes hybrides. Elles comportent en outre des signaux (variables et actions) non observables. Or nous faisons l'hypothèse que seules les signaux restants (dits externes, correspondant aux signaux d'entrée et de sortie du système sous test) sont observés, à des

instants qui sont en nombre fini.

Ces contraintes rendent parfois indécidable la correction d'une exécution du système vis-à-vis de sa spécification. Nous analysons les causes possibles de cette indécidabilité, et proposons d'y répondre dans certains cas en utilisant une spécification auxiliaire pour analyser la correction des exécutions. Cette spécification auxiliaire possède une sémantique proche de celle de la proporiété originale, et nous montrons comment elle permet sous certaines conditions de décider de la correction des observations réalisées en temps discret. Ceci fournit un oracle pour le test des propriétés

7.1.3 Critère d'adéquation

Lors de l'évaluation d'une exécution du système sous test, l'algorithme exécuté par l'oracle infère une séquence d'états de la spécification (c'est-à-dire des automates hybrides qui formalisent les propriétés) qui sont visités. L'ensemble de ces états peut être utilisé pour caractériser la qualité des tests, et plus précisément dans notre approche avec quelle efficacité ils exercent la spécification. Cette qualité est formalisée par un critère d'adéquation.

7.1.3.1 Une formalisation d'objectifs de validation pour les systèmes hybrides

Le critère d'adéquation que nous proposons pour caractériser la qualité des tests effectués pour valider un système est paramétré par des exigences de validation. Ces exigences sont formalisées par un profil opérationnel, qui dans notre interprétation spécifie des objectifs de test à atteindre. Ces objectifs peuvent être intuitivement considérés comme une densité minimale d'états de la spécification qui doivent être visités, définie en chaque point de son espace d'état. On peut donc définir des critères d'adéquation qui pour être complètement satisfaits nécessitent que certaines parties de l'espace d'état de la spécification soient exercées avec une plus ou moins forte "intensité".

7.1.3.2 Cas des espaces d'état denses

Le critère d'adéquation que nous proposons en première approximation se heurte lui aussi à problème d'espace dense. Cette fois ce n'est pas la densité du modèle de temps qui pose problème mais celle de l'espace d'état de la spécification. Elle empêche dans le cas général l'évaluation du critère par une machine. Nous montrons comment on peut calculer une approximation prudente de la mesure dans les cas où la mesure originale ne peut être calculée mais où l'espace d'état est borné, via le calcul d'une mesure auxiliaire. Celle-ci est évaluée en un nombre fini de points de l'espace d'état et paramétrique ; il est possible d'effectuer un arbitrage entre qualité de l'approximation et coût de l'algorithme, ce qui permet de prendre en compte les limites des ressources disponibles pour son évaluation.

7.1.3.3 Relation entre adéquation des tests et implémentation

On note que le critère que nous proposons permet parfois de mesurer l'adéquation d'un jeu de test à priori (c'est-à-dire sans exécuter de tests), mais uniquement si celui-ci est paramétré par un profil opérationnel dont les exigences ne font pas intervenir de signaux de sortie du système, ou de signaux

internes dont l'évolution dépend des réactions du système. Notre définition de profil opérationnel permet en effet de décrire des exigences sur ce second type de signaux également pour certaines applications qui sont discutées en 5.3, mais l'évaluation du critère est alors spécifique à une exécution et à un système sous test particuliers : bien qu'elle soit comparable avec une autre mesure effectuée avec le même profil et les mêmes entrées sur un système différent, elle ne permet pas de faire de prédiction sur cette seconde mesure, puisque le second système peut produire des sorties différentes qui influencent la mesure.

7.1.4 Qualités originales du critère

Nous terminons cette partie du bilan liée au critère d'adéquation en résumant plusieurs qualités remarquables que nous avons identifiées sur celui-ci.

7.1.4.1 Monotonicité

Le critère d'adéquation que nous proposons est monotone, c'est-à-dire que l'ajout d'un nouvel état visité ne provoque jamais de décroissance de la mesure. Il caractérise en effet une proportion d'objectifs de validation atteints par le test. Par ailleurs, ce critère qui est évalué entre les valeurs réelles 0 et 1 peut être entièrement satisfait (c'est-à-dire évalué à 1 suite à l'exécution de tests) si la méthode de génération de tests employée ainsi que les caractéristiques de contrôlabilité du système le permettent. Ceci permet d'utiliser ce critère comme mesure de couverture des tests, mais aussi comme critère d'arrêt de façon immédiate : si le critère atteint la valeur 1, tous les objectifs qu'il formalise ont été remplis et il ne permettra plus de mesurer la moindre amélioration de la couverture. Le processus de test peut alors être interrompu, et le système sous test validé (ou testé davantage par rapport une propriété différente).

7.1.4.2 Définition locale du critère

Notre critère est par ailleurs défini localement, c'est-à-dire qu'il associe une valeur de couverture à chaque élément de la spécification (état, mode, automate, propriété et spécification). Au sein d'un mode particulier, il est possible d'évaluer la couverture d'une partie individuelle de ce mode "gratuitement", c'est-à-dire sans augmentation du coût de l'algorithme et tout en évaluant la couverture de la spécification complète normalement. L'algorithme d'évaluation que nous proposons repose en effet sur la partition de l'espace d'état des modes en plusieurs éléments. Il suffit de raffiner cette partition de manière à ce que la zone d'intérêt soit égale à la réunion de certains éléments pour pouvoir établir des statistiques de couverture de cette zone individuellement. Nous conjecturons que ceci peut être utilisé pour guider un processus de génération de test (nous y reviendrons dans les perspectives).

7.1.4.3 Coût prédictible de l'algorithme

Notre algorithme d'évaluation de la couverture estime la couverture d'un nombre d'états fini extraits de l'espace d'état de la propriété que nous nommerons G par un autre ensemble T d'états qui sont les états visités lors du test, inférés par l'oracle. Les états de T peuvent être ajoutés séquentiellement au calcul de couverture au fur et à mesure qu'ils sont inférés par l'oracle (c'est ce qui est fait dans l'outil que nous avons développé pour implémenter la proposition), et ne sont plus utilisés ensuite, les stocker

tous est donc inutile au calcul de la couverture (lorsqu'il est effectué en temps réel). La composition de G étant entièrement définie par le profil et les paramètres de l'algorithme, le coût en mémoire de l'algorithme est donc indépendant de la taille de T.

7.1.4.4 Interprétation physique des données

Considérons à présent un mode particulier d'un automate qui formalise une propriété de la spécification (tel que $followRamp$ dans l'exemple du système de chauffage). Lorsque le profil associe aux états de ce mode une fonction d'importance constante par morceaux, cette fonction peut s'interpréter intuitivement sur les ensembles où elle est constante comme une fonction de densité. En effet, un ensemble de points minimal nécessaire à sa couverture possède une densité égale au produit des coordonnées de cette fonction (Cf. 5.2.4). Bien que non scientifique, nous conjecturons que cette intuition est un guide utile à l'écrivain de la spécification pour construire le profil.

7.1.5 Outil d'application HyATT

Notre proposition a été implémentée au sein d'un outil nommé HyATT. Cet outil permet de spécifier des automates formalisant des propriétés hybrides, et de les tester en s'interfaçant avec un environnement et un système sous test. Il peut aussi estimer la couverture d'un automate si un profil associé à cet automate est également défini.

Nous avons relaté comment l'exemple conducteur du système de chauffage que nous avons utilisé au long du manuscrit avait été instancié avec cet outil et rapportons des conclusions préliminaires sur les facteurs qui peuvent influencer l'efficacité d'une méthode de génération de test pour les automates hybrides.

7.2 Perspectives

Nous refermons ce document en discutant dans cette dernière section les perspectives ouvertes par nos travaux que nous avons identifiées.

7.2.1 Validation de l'approche sur des systèmes à configuration dynamique

Nous conjecturons que de nouvelles applications hybrides seront développées prochainement, sur un modèle ouvert et dynamique. C'est-à-dire qu'un système hybride pourra être composé de plusieurs composants répondant à des normes communes mais issus de fournisseurs différents. Nous identifions comme domaines d'application probables les systèmes domotiques (en particulier, pour le maintien à domicile des personnes âgées et la gestion réactive de l'énergie) et les systèmes de distribution d'énergie distribués (aussi appelés *smart grids*).

L'apparition de tels systèmes si elle se produit fournira un terrain de validation idéal de la pertinence et de l'utilité de notre approche sur deux points. Premièrement, elle permettra de valider le fait que ces

systèmes à architecture dynamique sont adaptés à une spécification hybride par propriété. Deuxièmement, elle permettra d'évaluer le coût de l'établissement des profils opérationnels et la capacité du critère d'adéquation associé à détecter les défaillances sur ces systèmes industriels innovants.

7.2.2 Ajout de restrictions pour assurer des propriétés du formalisme

Le modèle de propriété hybride que nous avons adopté utilise le formalisme d'automates peu contraint de Lynch [40]. La généralité de ce formalisme autorise une très grande expressivité dans l'écriture des spécifications. Nous avons choisi de l'adopter de préférence à des formalismes plus contraints et classiques tels que les automates hybrides linéaires car notre approche ne nécessite pas de propriétés de décidabilité particulières concernant l'atteignabilité de certains états, contrairement aux approches de preuve. Le prix de cette expressivité apparaît cependant au niveau de la testabilité des automates. Nous avons vu qu'il est possible de détecter la non-testabilité d'une propriété sur une trace particulière au vol, lors de l'exécution d'un test. Toutefois nous n'avons pas identifié de moyen permettant de juger à priori de cette testabilité.

Cette absence de garantie n'a pas posé de problème lors de nos première expériences avec HyATT, toutes les propriétés que nous avons écrites étaient aisées à spécifier de façon à être toujours testables. Cependant, sur des propriétés de plus grande taille ou dotées d'une sémantique particulière, cette inspection manuelle pourrait devenir impraticable et laisser passer inaperçus des problèmes de non-testabilité qui compromettent par conséquent l'évaluation du critère d'adéquation associé. Si de telles situations surviennent, il sera pertinent de revenir sur notre première définition du formalisme et de rechercher des contraintes à lui imposer pour limiter ou résoudre ces problèmes.

7.2.3 Génération de tests

Encore une fois pour des raisons d'expressivité du formalisme, nous n'avons pas identifié de méthode de génération de tests qui pourrait être efficace dans le cas général pour compléter notre approche et avons développé celles que nous avons employées de manière empirique. En particulier, l'algorithme RRT que nous avons discuté dans le cadre de certains travaux de référence que nous avons étudiés [24] et discuté en 3.2.4 ne peut pas convenir, du moins pas en présence d'un système physique. Une des raisons en est que le coût de cet algorithme sur un système concret devient prohibitif : lorsqu'il est utilisé explore un espace d'état, l'algorithme RRT ajoute des états à un arbre d'états visités. Chaque ajout d'un nouvel état à l'arbre entraîne dans le cas du test d'un système concret l'exécution de k exécutions (puisque l'état qui est finalement ajouté à chaque itération est choisi parmi un ensemble de k états correspondant à la simulation de différentes entrées). Toute la séquence d'états précédents doit également être rejouée. Si l'arbre final contient N états, ceci correspond à $k.N^2$ exécutions du système sous test alternées avec des réinitialisations. Dans le cas d'un système physique, nous conjecturons que cette complexité est trop importante en pratique et que seules des techniques de monitoring ou de test actif impliquant peu ou pas de réinitialisation sont praticables.

Pour identifier les méthodes de génération adaptées à chaque situation, des travaux ultérieurs pourraient explorer pour commencer les différents facteurs pertinentes pour le choix de l'algorithme. Nous avons employé dans ce manuscrit quatre méthodes de génération de tests. Cela a conduit à l'identification de deux facteurs qui sont la localité des objectifs puirsuivis et l'exploitation de données temporelles de la propriété. Ces facteurs influencent la vitesse de progression de la couverture mesurée par le critère.

En s'appuyant sur la littérature existante, en particulier en matière de test aléatoire et algorithmes de recherche, ainsi qu'en analysant les domaines d'applications pertinents, nous conjecturons que de tels facteurs pourront être identifiés.

Enfin, nous remarquons que certains travaux sur les systèmes hybrides [20] mais aussi dans d'autres domaines comme l'intelligence artificielle [25] exploitent plusieurs algorithmes conjointement pour améliorer les résultats de l'approche globale proposé. De même que la méthode de Dang et al. [20] exploite les données de couverture du modèle pour guider un processus d'exploration de modèle, il est intéressant de poser la question suivante : peut-on utiliser la couverture évaluée par notre critère, qui est définie indépendamment pour différents éléments de la spécification et actualisée en temps réel, pour guider un processus de génération de test ?

7.2.4 Développements de l'outil HyATT

L'outil que nous avons développé pour appliquer notre proposition réalise d'ores et déjà un ensemble fonctions que nous avons détaillées précédemment. Au cours de nos travaux, nous avons identifié un certain nombre de prolongements que nous souhaitons lui ajouter par la suite. A court terme, il s'agit d'améliorer le système de remontée d'erreurs comme discuté en 6.5.2.

A moyen terme, nous souhaitons développer les interactions de l'outil avec des formalismes classiques et fournir un support de documentation afin de l'ouvrir à plus d'applications, et réutiliser des benchmarks existants. L'architecture de HyATT a été conçue pour autoriser l'import d'autres formalismes d'automates hybrides classiques, mais les modules correspondants n'ont pas encore été écrits pour le moment.

Enfin à plus long terme, nous souhaitons définir une surcouche ou une extension du logiciel qui rende l'asemblage de programmes de test plus facile. La méthode actuelle implique de lier différentes unités contenant le code d'interfaçage spécifique à chaque application avec du code généraliste partagé de HyATT (contenant entre autre les algorithmes d'évaluation de la correction et de la couverture). Cet assemblage est aujourd'hui essentiellement manuel et donc complexe.

Annexe A

Mise en pratique de la méthode

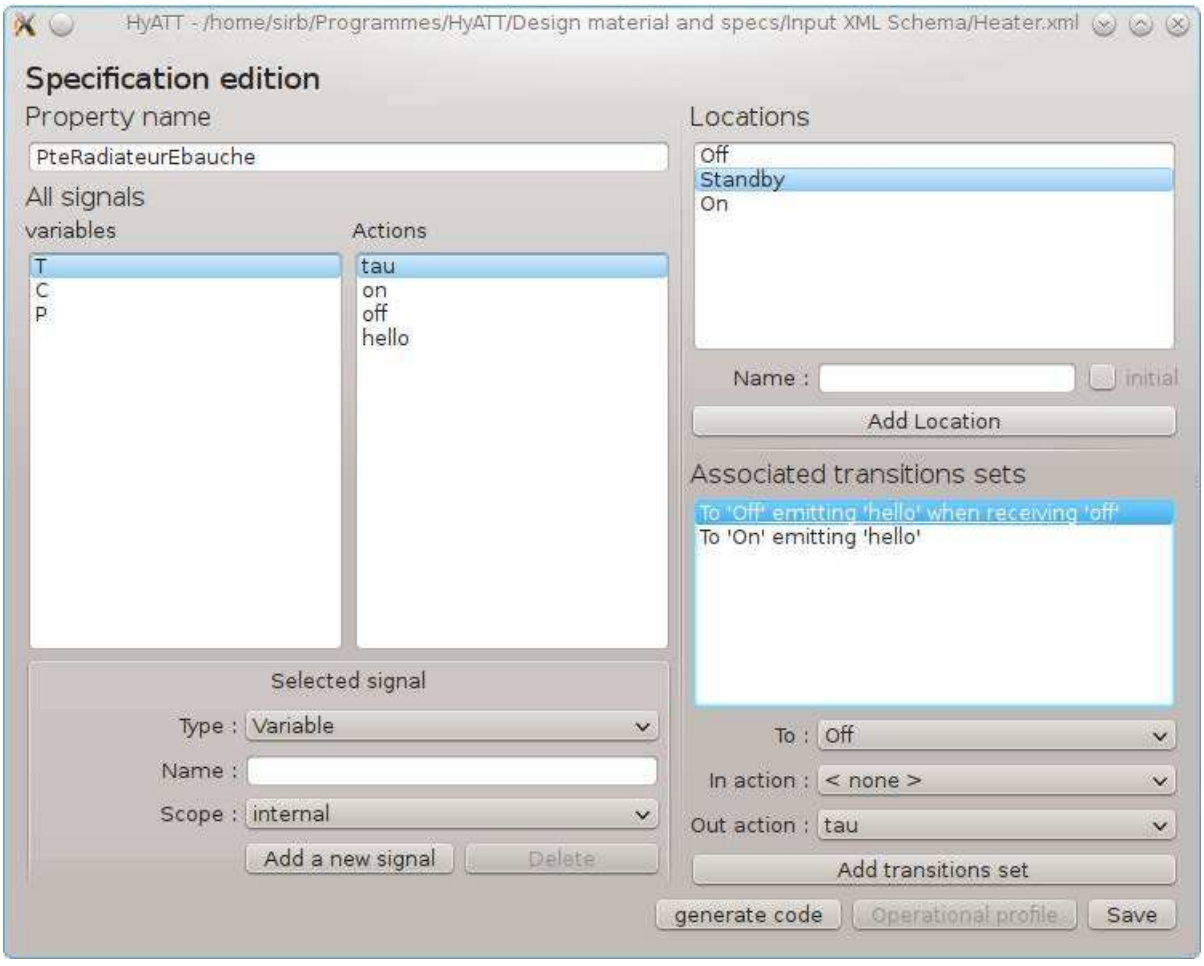


FIG. A.1 – Édition de la partie discrète de la définition d'un automate

```
<?xml version="1.0" encoding="utf-8"?>
<hap:HybridProperties xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:hap="http://www.graslany.org/XML_Namespaces/HybridProperty"
  xsi:schemaLocation="http://www.graslany.org/XML_Namespaces/HybridProperty
                      file:///home/sirb/.local/usr/share/HyATT/HybridProperty.xsd">
 <Property name="Climatisation">
  <Automaton>
   <Variables>
    <Variable name="T" scope="input"/>
    <Variable name="PRenew" scope="input"/>
    <Variable name="POut" scope="output"/>
    <Variable name="inD" scope="input"/>
    <Variable name="inDL" scope="input"/>
    <Variable name="oldD" scope="internal"/>
    <Variable name="hasNextContract" scope="internal"/>
    <Variable name="newD" scope="internal"/>
    <Variable name="timeRemaining" scope="internal"/>
    <Variable name="rampDuration" scope="internal"/>
   </Variables>
   <Actions>
    <Action name="newC" scope="input"/>
    <Action name="ack" scope="output"/>
    <Action name="nack" scope="output"/>
    <Action name="i" scope="internal"/>
   </Actions>
   <Locations>
    <Location name="waitContract"/>
    <Location name="validContract"/>
    <Location name="invalidContract"/>
    <Location name="steady"/>
    <Location name="followRamp"/>
   </Locations>
   <Transitions>
    <TransitionSet inAction="newC" from="waitContract" to="validContract" outAction="ack"/>
    <TransitionSet inAction="newC" from="waitContract" to="invalidContract" outAction="nack"/>
    <TransitionSet from="validContract" to="steady" outAction="i"/>
    <TransitionSet from="invalidContract" to="steady" outAction="i"/>
    <TransitionSet inAction="newC" from="steady" to="validContract" outAction="ack"/>
    <TransitionSet inAction="newC" from="steady" to="invalidContract" outAction="nack"/>
    <TransitionSet from="steady" to="followRamp" outAction="i"/>
    <TransitionSet inAction="newC" from="followRamp" to="validContract" outAction="ack"/>
    <TransitionSet inAction="newC" from="followRamp" to="invalidContract" outAction="nack"/>
    <TransitionSet from="followRamp" to="steady" outAction="i"/>
   </Transitions>
   <InitialLocations>
     <Location name="waitContract"/>
   </InitialLocations>
  </Automaton>
  <!-- Le profil opérationnel se place ici, il n'est pas représenté sur cette capture -->
 </Property>
</hap:HybridProperties>
```

FIG. A.2 – Fichier XML spécifiant la partie discrète de l'automate pour notre exemple conducteur

```
<?xml version="1.0" encoding="utf-8"?>
<hap:HybridProperties ...>
 <Property name="Climatisation">
  <Automaton>
   <!-- La définition de la partie discrète de la propriété intervient ici. -->
  </Automaton>
  <OperationalProfile omegaPrimeOverOmega="2">
   <Location name="followRamp" weight="1">
    <Grid>
     <Dimensions>
      <Dimension variable="T"      min="10" max="30" weight="0.5"/>
      <Dimension variable="PRenew"      min="0" max="300" weight="0.01"/>
      <Dimension variable="newD"      min="10" max="30" weight="0.2"/>
      <Dimension variable="timeRemaining"  min="0" max="400" weight="0.003"/>
     </Dimensions>
     <SplittingHyperplans>
      <Hyperplan>
       <Coefficient variable="T" value="1"/>
       <Coefficient variable="newD" value="-1"/>
       <LastCoefficient value="2.5"/>
      </Hyperplan>
      <Hyperplan>
       <Coefficient variable="T" value="-1"/>
       <Coefficient variable="newD" value="1"/>
       <LastCoefficient value="2"/>
      </Hyperplan>
     </SplittingHyperplans>
    </Grid>
    <Grid>
     <Dimensions>
      <Dimension variable="T"      min="10" max="30" weight="0.5"/>
      <Dimension variable="PRenew"      min="300" max="1000" weight="0.01"/>
      <Dimension variable="newD"      min="10" max="30" weight="0.5"/>
      <Dimension variable="timeRemaining"  min="0" max="400" weight="0.003"/>
     </Dimensions>
     <SplittingHyperplans>
      <Hyperplan>
       <Coefficient variable="T" value="1"/>
       <Coefficient variable="newD" value="-1"/>
       <LastCoefficient value="2.5"/>
      </Hyperplan>
      <Hyperplan>
       <Coefficient variable="T" value="-1"/>
       <Coefficient variable="newD" value="1"/>
       <LastCoefficient value="2"/>
      </Hyperplan>
     </SplittingHyperplans>
    </Grid>
   </Location>
   <!-- On pourrait avoir ici un autre élément "Location" si le profil exprimait des objectifs
         pour d'autres modes ... -->
  </OperationalProfile>
 </Property>
</hap:HybridProperties>
```

FIG. A.3 – Fichier XML spécifiant le profil opérationnel pour notre exemple conducteur

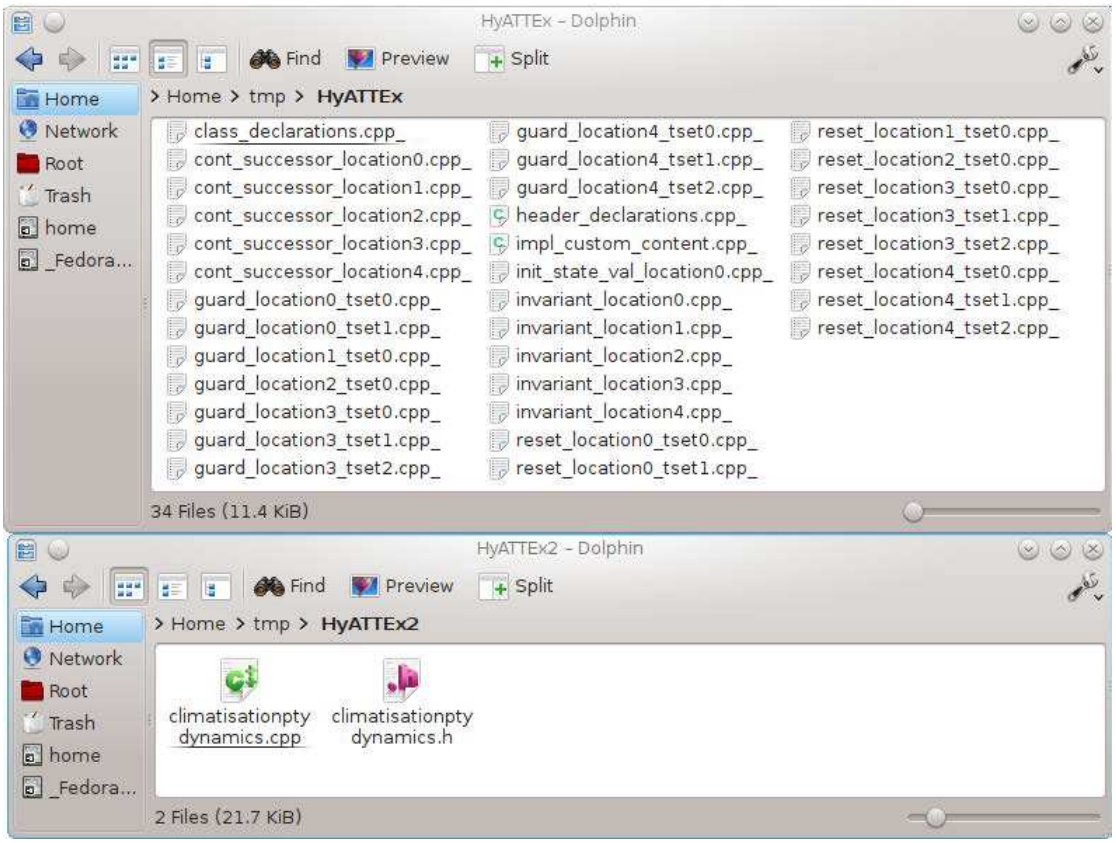


FIG. A.4 – Fichiers intermédiaires et finals générés pour la partie continue de la spécification

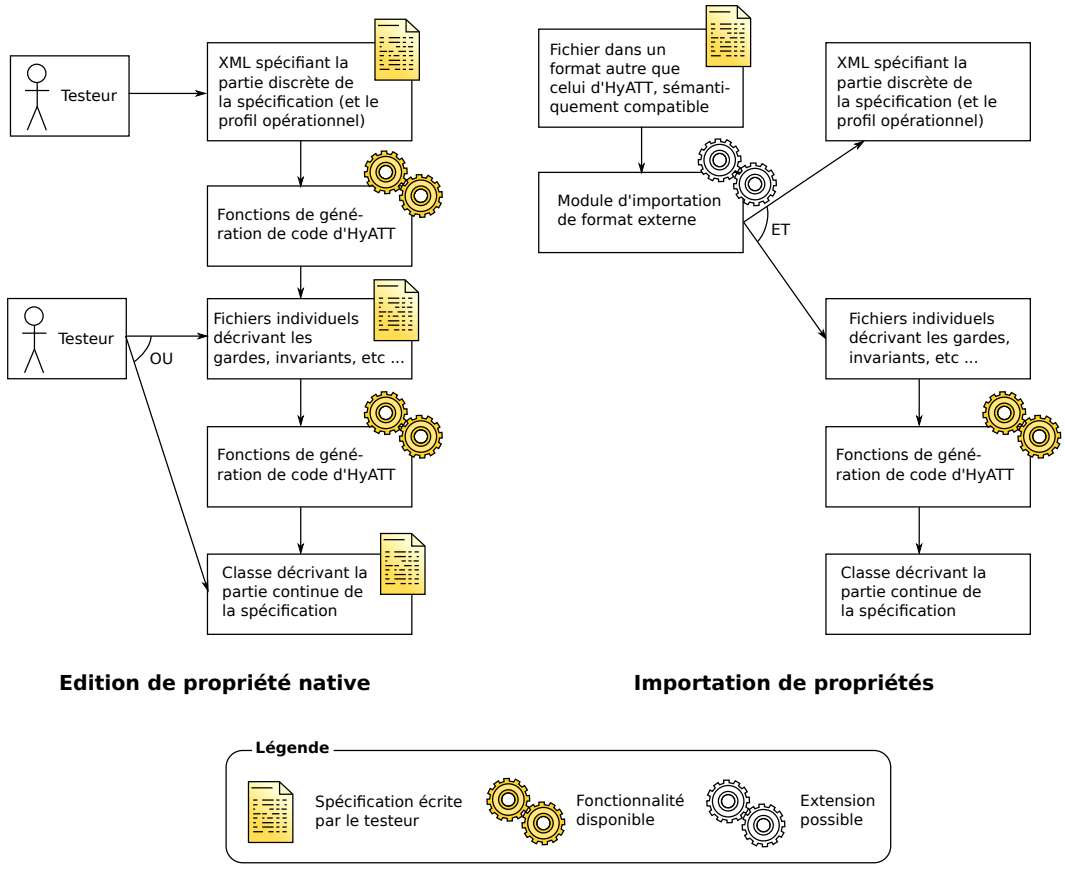


FIG. A.5 – Création ou importation de propriétés dans HyATT

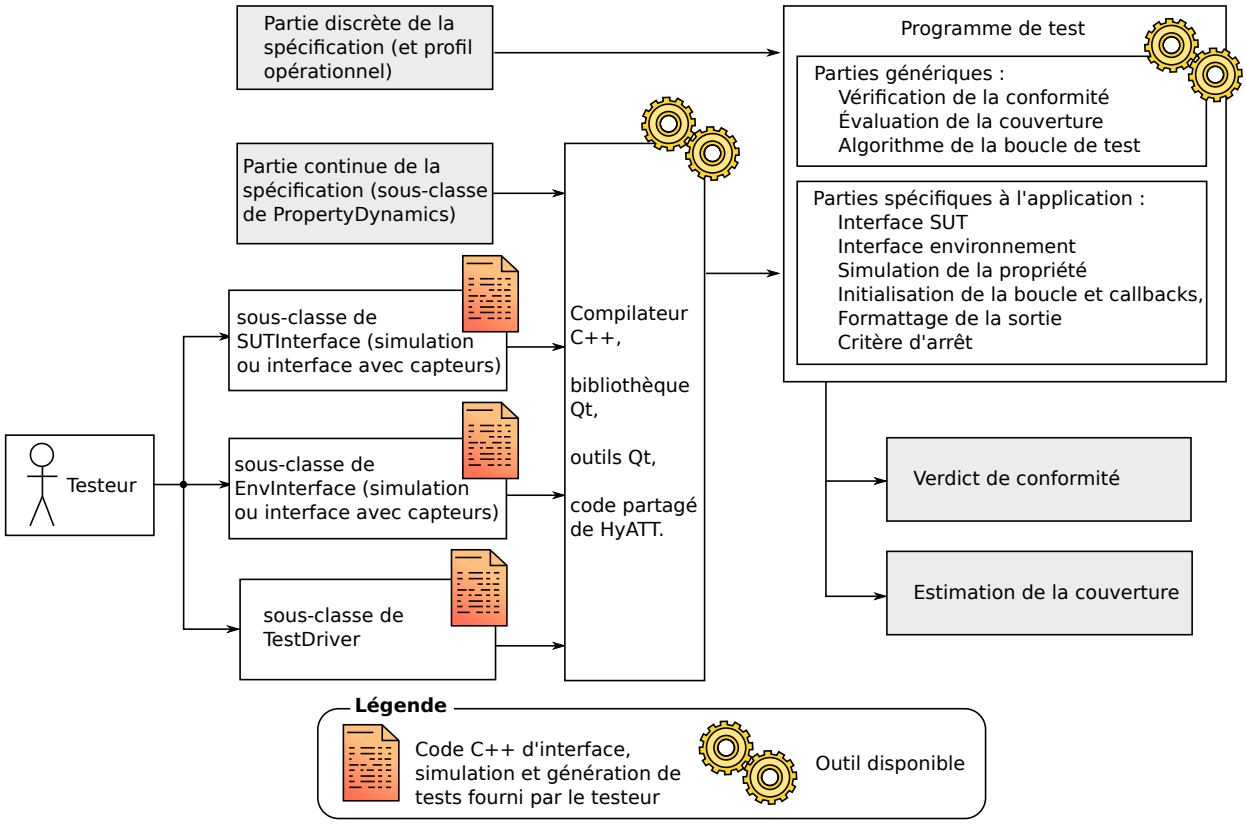


FIG. A.6 – Composition et assemblage du programme de test

```
#include "PropertyIO/nativepropertyreader.h"
#include "Properties/coveragecomputingautomaton.h"
#include "Control/testdriver.h"
#include "Examples/ClimThese/climatisationptydynamics.h"
#include "Examples/ClimThese/climtheseenv.h"
#include "Examples/ClimThese/climthesesut.h"
#include "Examples/ClimThese/climthesetestdriver.h"

#include <QDebug>
#include <QApplication>
#include <QVector>

int main(int argc, char *argv[]) {
    QApplication a(argc, argv); (void) a;
    const QString propertyFile("/home/sirb/Programmes/HyATT/Source/HyATT/Examples"
                               "/ClimThese/ClimThese.xml");
    const QUrl schemaFile("file:///home/sirb/.local/usr/share/HyATT/HybridProperty.xsd");

    Automaton *automaton = 0;
    ClimTheseEnv *env = 0;
    TestDriver *testDriver = 0;
    SUTInterface *mySUT = 0;

    // Création d'une instance de propriété hybride à partir du fichier XML créé
    // via l'interface d'HyATT (ou par tout autre moyen approprié).
    automaton = NativePropertyReader (propertyFile, schemaFile).extractAutomaton();
    if (automaton==0) {
        qDebug() << "main() error : Unable to instantiate the property.";
        return 1;
    }

    // La classe ClimatisationPtyDynamics a été dans ce cas précis générée par HyATT
    // et complétée par le testeur pour y programmer les invariants, gardes, etc ...
    // Elle est ici affectée à l'automate pour être utilisée par la boucle de test.
    ClimatisationPtyDynamics continuousDynamics;
    continuousDynamics.assignToAutomaton(*automaton);

    // Instantiation du système de test (simulé) et de l'environnement qui
    // génère les tests. Ici, ces deux composants ont été programmés par le testeur.
    // On pourrait également avoir dans ces classes du code qui lit et remonte
    // des flux depuis des capteurs.
    mySUT = new ClimTheseSUT(*automaton);
    env = new ClimTheseEnv(*automaton);

    // Le driver de test, surchargé pour exporter la couverture de la propriété
    // à chaque itération dans un fichier, et cesser le test au bout d'une certaine durée.
    testDriver = new ClimTheseTestDriver(automaton, env, mySUT);

    // Exécution des tests
    testDriver->launchRun();

    // Affichage d'un message à la fin de l'exécution donnant la couverture finale.
    CoverageComputingAutomaton *ca = dynamic_cast <CoverageComputingAutomaton *> (automaton);
    if (ca && ca->getCoverageInstance()) {
        qDebug() << "main() information : The achived coverage was : "
                 << ca->getCoverageInstance()->computeCoverage();
    } else {
        qDebug() << "main() warning : The property does not compute coverage oO ?";
    }

    // Nettoyage
    if (automaton!=0)
        delete automaton;
    delete env;
    delete testDriver;
    delete mySUT;
    return 0;
}
```

FIG. A.7 – Rédaction d'une fonction principale chargée d'instancier et d'exécuter le test

Table des figures

Bibliographie

[1] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, page 1, 1990.

[2] Parosh Abdulla, Bengt Jonsson, Ahmed Rezine, and Mayank Saksena. Proving liveness by backwards reachability. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 – Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 95–109. Springer Berlin / Heidelberg, 2006.

[3] Shadi Abras, Sylvie Pesty, Stéphane Ploix, and Mireille Jacomino. Advantages of mas for the resolution of a power management problem in smart homes. In Yves Demazeau, Frank Dignum, Juan M. Corchado, and Javier Bajo, editors, *PAAMS*, volume 70 of *Advances in Soft Computing*, pages 269–278. Springer, 2010.

[4] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2 :117–126, 1987. 10.1007/BF01782772.

[5] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1) :3–34, 1995.

[6] Rajeev Alur, Thao Dang, and Franjo Ivančić. Counterexample-guided predicate abstraction of hybrid systems. *Theoretical Computer Science*, 354(2) :250 – 271, 2006.

[7] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2) :183–235, 1994.

[8] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[9] Yashwanth Annpureddy, Che Liu, Georgios Fainekos, and Sriram Sankaranarayanan. S-taliro : A tool for temporal logic falsification for hybrid systems. In Parosh Abdulla and K. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 254–257. Springer Berlin / Heidelberg, 2011.

[10] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6) :891–923, 1998.

[11] Eugene Asarin, Thao Dang, Goran Frehse, Antoine Girard, Colas Le Guernic, and Oded Maler. Recent progress in continuous and hybrid reachability analysis. In *IEEE Symposium on Computer-Aided Control Systems Design, Proceedings*, pages 1582–1587. IEEE, 2006.

[12] Eugene Asarin, Thao Dang, and Antoine Girard. Hybridization methods for the analysis of nonlinear systems. *Acta Informatica*, 43 :451–476, 2007. 10.1007/s00236-006-0035-7.

[13] Eugene Asarin, Thao Dang, and Oded Maler. The d/dt tool for verification of hybrid systems. In Ed Brinksma and Kim Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 746–770. Springer Berlin / Heidelberg, 2002.

[14] André Baresel, Hartmut Pohlheim, and Sadegh Sadeghipour. Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In Erick Cantú-Paz, James A. Foster, Kalyanmoy Deb, Lawrence Davis, Rajkumar Roy, Una-May O'Reilly, Hans-Georg Beyer, Russell K. Standish, Graham Kendall, Stewart W. Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, Mitchell A. Potter, Alan C. Schultz, Kathryn A. Dowsland, Natasa Jonoska, and Julian F. Miller, editors, *GECCO*, volume 2724 of *Lecture Notes in Computer Science*, pages 2428–2441. Springer, 2003.

[15] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[16] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9) :1270 –1282, sep 1991.

[17] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing : The art of test case diversity. *Journal of Systems and Software*, 83(1) :60–66, 2010.

[18] Edmund Clarke. Model checking. In S. Ramesh and G Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *Lecture Notes in Computer Science*, pages 54–56. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0058022.

[19] Edmund Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at ltl model checking. In David Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 415–427. Springer Berlin / Heidelberg, 1994.

[20] Thao Dang and Tarik Nahhal. Using disparity to enhance test generation for hybrid systems. In *TestCom/FATES*, volume 5047 of *Lecture Notes in Computer Science*, pages 54–69. Springer, 2008.

[21] Pieter-Tjerk de Boer, Dirk P. Kroese, Shie Mannor, and Reuven Y. Rubinstein. A tutorial on the cross-entropy method. *Annals OR*, 134(1) :19–67, 2005.

[22] Lydie du Bousquet, Farid Ouabdesselam, and Jean-Luc Richier. Expressing and implementing operational profiles for reactive software validation. In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, pages 222 –230, nov 1998.

[23] Clément Escoffier, Jonathan Bardin, Johann Bourcier, and Philippe Lalanda. Developing user-centric applications with h-omega. In Cristian Hesselman and Carlo Giannelli, editors, *MOBILWARE Workshops*, volume 12 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 118–123. Springer, 2009.

[24] Joel M. Esposito, Jongwoo Kim, and Vijay Kumar. Adaptive rrts for validating hybrid robotic control systems. In *Algorithmic Foundations of Robotics VI*, volume 17 of *Springer Tracts in Advanced Robotics*, pages 107–121. 2005.

[25] David A. Ferrucci. Build watson : an overview of deepqa for the jeopardy ! challenge. In Valentina Salapura, Michael Gschwind, and Jens Knoop, editors, *PACT*, pages 1–2. ACM, 2010.

[26] Goran Frehse. Phaver : Algorithmic verification of hybrid systems past hytech. In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems : Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer Berlin / Heidelberg, 2005.

[27] Antoine Girard and Colas Le Guernic. Zonotope/hyperplane intersection for hybrid systems reachability analysis. In Magnus Egerstedt and Bud Mishra, editors, *HSCC*, volume 4981 of *Lecture Notes in Computer Science*, pages 215–228. Springer, 2008.

[28] Antoine Girard and George J. Pappas. Approximation metrics for discrete and continuous systems. *Automatic Control, IEEE Transactions on*, 52(5) :782 –798, may 2007.

[29] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with pvs. In Orna Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer Berlin / Heidelberg, 1997.

[30] Robert L. Grossman, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors. *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*. Springer, 1993.

[31] Joachim Hänsel, Daniela Rose, Paula Herber, and Sabine Glesner. An evolutionary algorithm for the generation of timed test traces for embedded real-time systems. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 170 –179, march 2011.

[32] Hadi Hemmati, Andrea Arcuri, and Lionel C. Briand. Empirical investigation of the effects of test suite properties on similarity-based test case selection. In *ICST*, pages 327–336. IEEE Computer Society, 2011.

[33] Thomas A. Henzinger. The theory of hybrid automata. In *LICS*, pages 278–292. IEEE Computer Society, 1996.

[34] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech : A model checker for hybrid systems. In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 460–463. Springer, 1997.

[35] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata ? *J. Comput. Syst. Sci.*, 57(1) :94–124, 1998.

[36] Muhammad Iqbal, Andrea Arcuri, and Lionel Briand. Environment modeling with uml/marte to support black-box system testing for real-time embedded systems : Methodology and industrial case studies. In Dorina Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 286–300. Springer Berlin / Heidelberg, 2010.

[37] Agung Julius, Georgios E. Fainekos, Madhukar Anand, Insup Lee, and George J. Pappas. Robust test generation and coverage for hybrid systems. In *Hybrid Systems : Computation and Control*, volume 4416 of *Lecture Notes in Computer Science*, pages 329–342. Springer, 2007.

[38] Steven M. LaValle and James J. Kuffner Jr. Randomized kinodynamic planning. *I. J. Robotic Res.*, 20(5) :378–400, 2001.

[39] Michael D. Lemmon, James A. Stiver, and Panos J. Antsaklis. Event identification and intelligent hybrid control. In Grossman et al. [30], pages 268–296.

[40] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid i/o automata. *Information and Computation*, 185(1) :105 – 157, 2003.

[41] The modelica association website. https ://modelica.org/.

[42] Yashwant K. Malaiya. Antirandom testing : getting the most out of black-box testing. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 86 –95, oct 1995.

[43] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties (invited paper, 1989). In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, pages 377–410, New York, NY, USA, 1990. ACM.

[44] Zohar Manna and Amir Pnueli. Verifying hybrid systems. In Grossman et al. [30], pages 4–35.

[45] Aditya P. Mathur. *Foundations of Software Testing*. Addison-Wesley Professional, 1st edition, 2008.

[46] Phil McMinn. Search-based software test data generation : a survey. *Software Testing, Verification and Reliability*, 14(2) :105–156, 2004.

[47] The matlab commercial website. http ://www.mathworks.fr/products/matlab/.

[48] John D. Musa. Operational profiles in software-reliability engineering. *Software, IEEE*, 10(2) :14 –32, mar 1993.

[49] Glenford J. Myers. *The Art of Software Testing*. 1979. ISBN : 9780471043287.

[50] Masoud Najafi and Ramine Nikoukhah. Implementation of hybrid automata in scicos. In *Control Applications, 2007. CCA 2007. IEEE International Conference on*, pages 819 –824, oct. 2007.

[51] Masahide Nakamura, Yusuke Fukuoka, Hiroshi Igaki, and Ken ichi Matsumoto. Implementing multi-vendor home network system with vendor-neutral services and dynamic service binding. In *IEEE SCC (2)*, pages 275–282. IEEE Computer Society, 2008.

[52] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1) :25–53, 2003.

[53] Erion Plaku, Lydia Kavraki, and Moshe Vardi. Hybrid systems : from verification to falsification by combining motion planning and discrete search. *Formal Methods in System Design*, 34 :157–182, 2009. 10.1007/s10703-008-0058-5.

[54] Sriram Sankaranarayanan and Georgios Fainekos. Falsification of temporal properties of hybrid systems using the cross-entropy method. In *Proceedings of the 15th ACM international conference on Hybrid Systems : Computation and Control*, HSCC '12, pages 125–134, New York, NY, USA, 2012. ACM.

[55] Peter Schrammel and Bertrand Jeannet. From hybrid data-flow languages to hybrid automata : a complete translation. In Thao Dang and Ian M. Mitchell, editors, *HSCC*, pages 167–176. ACM, 2012.

[56] Besnik Seljimi and Ioannis Parissis. Automatic generation of test data generators for synchronous programs : Lutess v2. In Alan Hartman, Mika Katara, and Amit M. Paradkar, editors, *DOSTA*, pages 8–12. ACM, 2007.

[57] Izaias B. Silva and Bruce H. Krogh. Formal verification of hybrid systems using checkmate : a case study. In *American Control Conference, 2000. Proceedings of the 2000*, volume 3, pages 1679 –1683 vol.3, 2000.

[58] Li Tan, Jesung Kim, Oleg Sokolsky, and Insup Lee. Model-based testing and monitoring for hybrid embedded systems. In *Information Reuse and Integration, 2004. IRI 2004. Proceedings of the 2004 IEEE International Conference on*, pages 487 – 492, nov. 2004.

[59] Eric Thiémard. An algorithm to compute bounds for the star discrepancy. *Journal of Complexity*, 17(4) :850 – 880, 2001.

[60] Claire J. Tomlin, Ian Mitchell, Alexandre M. Bayen, and Meeko Oishi. Computational techniques for the verification of hybrid systems. *Proceedings of the IEEE*, 91(7) :986 – 1001, july 2003.

[61] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[62] Michiel van Osch. Hybrid input-output conformance and test generation. In Klaus Havelund, Manuel Núñez, Grigore Rosu, and Burkhart Wolff, editors, *FATES/RV*, volume 4262 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2006.

[63] Elaine J Weyuker. Axiomatizing software test data adequacy. *IEEE Trans. Softw. Eng.*, 12(12) :1128–1138, December 1986.

[64] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4) :366–427, December 1997.