Tests de politiques d'adaptation pour systèmes cyber-physiques

Jean-Philippe Gros
Institut FEMTO-ST, Univ. Bourgogne Franche-Comté, CNRS
15B avenue des Montboucons, 25030 Besançon, Cedex, France

Abstract

Cet article est dédié à la validation des politiques d'adaptation en utilisant une approche de tests à partir de modèles; à la mise en place d'un verdict basé à la fois sur la vérification à l'exécution et aux propriétés temporelles; à la détection d'inconsistances entre les politiques d'adaptation et les reconfigurations implémentées dans le système. Nous proposons un moyen d'établir un verdict de tests basé sur le respect des politiques d'adaptation ainsi que les mesures de couverture des règles. Ces verdicts nous donnent des informations sur les règles pour détecter d'éventuelles reconfigurations qui n'auraient pas dû avoir lieu, des règles avec une priorité haute qui ne sont jamais déclenchées ou des règles avec une priorité faible qui sont déclenchées trop souvent, des inconsistances dans les règles, ou une mauvaise interprétation des priorités. Les verdicts sont obtenus en analysant les traces d'exécution du système qui est stimulé par un modèle d'usage à transitions probabilistes. Afin d'illustrer notre approche, nous utilisons un système de peloton de voitures autonomes.

1 Introduction

Les systèmes adaptatifs gagnent en importance au fil des années, on les retrouve dans les véhicules intelligents, les infrastructures adaptatives (e.g smartgrids), les robots et de manière générale dans l'industrie. Ces systèmes s'adaptent en fonction des évènements internes et externes qu'ils rencontrent par le biais de reconfigurations dynamiques. Ces reconfigurations dynamiques changent l'architecture de systèmes adaptatifs [4] et sont guidées par des politiques d'adaptation. Une politique d'adaptation a pour rôle de guider le système en indiquant la pertinence de déclencher ses reconfigurations. Une politique d'adaptation est constituée de reconfigurations et d'un ensemble de règles de déclenchement des reconfigurations. Chaque règle étant composée de gardes sur la configuration du système, de priorités à l'activation et de propriétés temporelles. Les priorités, exprimées par de valeurs floues comme dans [5] indiquent la pertinence d'appliquer une reconfiguration.

Lors du développement d'un système adaptatif, il est possible de mal interpréter les choix spécifiés par les politiques d'adaptation. Actuellement, des méthodes de vérification [10] existent afin d'établir qu'un système se comporte correctement du point de vue des propriétés (temporelles) du système. Cependant, il n'existe pas de garantie sur le respect des politiques d'adaptation. Plus précisément, nous souhaitons nous assurer que l'ensemble des politiques d'adaptation ne présente pas d'incohérence. Ensuite, il faut s'assurer que ces politiques d'adaptation soient valides en terme de cohérence et de respect des propriétés fonctionnelles. Dans une autre mesure, il est possible de se convaincre de l'efficacité du système en validant des propriétés non-fonctionnelles. La validation des propriétés non-fonctionnelles consisterait à évaluer les choix des reconfigurations faites en rejouant la même séquence d'évènements. Le fait est que les propriétés et les règles de politiques d'adaptation

ne peuvent évaluer les reconfigurations du système à la volée mais plutôt a posteriori. Afin d'établir de tels verdicts, il est nécessaire d'établir une technique de validation de conformité pour exercer le système à produire des traces pertinentes.

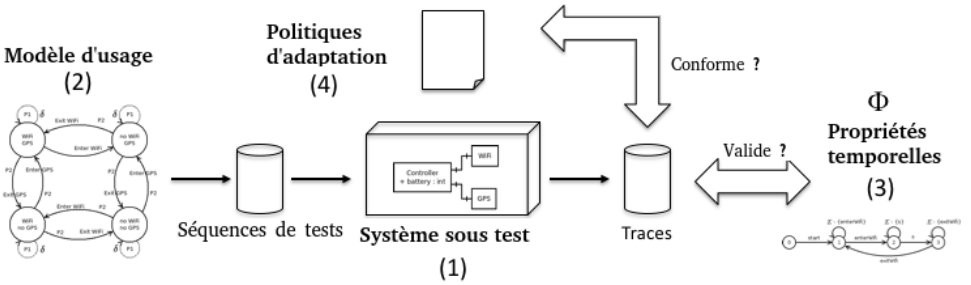

Figure 1: Approche par tests à partir de modèles pour valider les politiques d'adaptation

Afin de satisfaire ces problématiques, nous proposons une approche à partir de modèles dans la Figure. 1, qui se base sur les méthodes proposées dans [1]. Nous proposons de générer des cas de tests tout en établissant une méthode de verdict de test permettant de s'assurer du bon comportement du système sous test.

Les travaux [10] répondent à une première partie de la problématique en vérifiant la validité des propriétés temporelles en analysant rétroactivement les traces produites par le système (3). À partir de ces travaux, nous souhaitons nous assurer que les traces produites par système sous test (1) couvrent suffisamment les comportements possibles du système. Pour cela, nous proposons des critères de couverture qui s'appuient sur les propriétés temporelles (3) et les politiques d'adaptation (4). Les traces utilisées visant à satisfaire les critères de couverture sont obtenues en stimulant le système à l'aide d'un modèle d'usage (2). Ce modèle d'usage génère des séquences d'évènements grâce à un automate probabiliste. Les traces d'exécution sont alors analysées : les propriétés temporelles (3), sont vérifiées pendant l'exécution du système sous test (1). Cette étape s'effectue en utilisant une technique d'analyse du système basée sur les travaux de [8]. Les politiques d'adaptation (4) sont utilisées pour s'assurer que le système effectue les reconfigurations au bon moment. Pour cela, nous vérifions que les reconfigurations souhaitées sont bien déclenchées et qu'aucune reconfiguration non souhaitée n'est déclenchée. Nous nous intéressons également à la cohérence de la politique d'adaptation, pour cela, en se basant sur les critères de couverture, il nous est possible de détecter qu'une règle n'est jamais satisfaite. En outre de ces aspects de cohérence, nous proposons un verdict sur le respect des priorités. Ce verdict permet de détecter lorsqu'une reconfiguration avec une priorité haute n'est pas ou peu déclenchée, ou qu'une reconfiguration avec une faible priorité est trop souvent déclenchée.

Cet article est organisé de la façon suivante : La Section 2 présente le contexte scientifique et définit les notions et notations utilisées dans ce document. Dans la Section 3, nous décrivons notre approche de tests à partir de modèles ainsi que les critères de couverture associés. Nous conclurons et présenterons les futurs axes de recherche de ce travail dans la Section 4.

2 Contexte

Cette section présente le contexte dans lequel nous avons mené nos études. Celles-ci se sont portées sur un système de convoi de véhicules autonomes. Le comportement de ce système est vérifié par des propriétés temporelles et le déclenchement des reconfiguration guidé par des politiques d'adaptation.

2.1 Le convoi de véhicules autonomes

Dans ce système, les véhicules sont organisés en peloton ou individuellement. Dans chaque peloton, on trouve un leader qui se situe à l'avant. Un véhicule seul peut demander à rejoindre un peloton ou créer un nouveau peloton en se groupant avec un autre véhicule seul. Le système est soumis à des évènements internes, en effet, chaque véhicule au sein d'un peloton peut demander à sortir soit parce qu'il a atteint sa destination soit parce qu'il arrive à court d'énergie. Le véhicule désigné leader peut changer soit parce qu'un véhicule a plus d'autonomie que lui soit parce que sa destination est plus éloignée que le véhicule leader actuel. Des évènements externes peuvent avoir lieu, un conducteur peut décider de reprendre la main et quitter le convoi ou un nouveau véhicule peut arriver à portée du peloton pour une éventuelle fusion.

Nous considérons deux séquences différentes comme illustré dans la Figure 2, la séquence d'évènements externes (ou cas de tests) générée par le modèle d'usage et le chemin de reconfiguration (ou traces d'exécution) généré par le système en réponse aux évènements externes. Les séquences et chemins de reconfiguration sont régis par un tic d'horloge assurant que les évènements et les reconfigurations sont échantillonnés selon la même fréquence. Dans le cas d'un convoi de véhicules autonomes, plusieurs évènements peuvent avoir lieu. L'évènement *join* intervient quand des véhicules sont à portée pour se rejoindre, le système peut alors faire rejoindre plusieurs véhicules *acceptJoin* ou ne rien faire *refuseJoin*. Lorsqu'un utilisateur décide volontairement de quitter le peloton, l'évènement *quit* intervient, le système réagit à cet évènement par un (*acceptQuit*). Le système peut également réagir à des évènements internes et choisir de changer de leader avec la reconfiguration (*getRelay*). Si aucune reconfiguration n'a eu lieu entre deux tics d'horloge, on observe une reconfiguration *run* sur le chemin de reconfiguration.

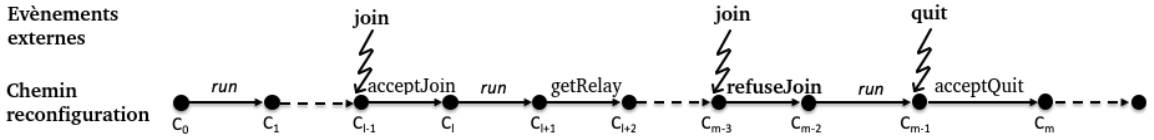

Figure 2: Séquence d'évènements et chemin de reconfiguration

2.2 Propriétés Temporelles

Dans cette section, nous présentons les propriétés temporelles FTPL[1] introduites dans [6]. Le langage FTPL est instancié avec des évènements du système pour exprimer une propriété temporelle. La sémantique de FTPL que nous utilisons se base sur les travaux de [7]. Nous proposons deux exemples liés à notre cas de peloton de véhicules autonomes :

$\varphi 1$: **after** *CreatePlt* **before** *DeletePlt* **always** $PlatoonId.VehicleNb > 2$
Cette propriété s'applique pour chaque convoi de véhicules. Elle vérifie qu'après la création d'un convoi il y a toujours au moins deux véhicules jusqu'à la suppression du peloton.

$\varphi 2$: **after** *Join* **before** *Quit* **always** $VehicleId.Battery > 5$ and $VehicleId.Distance > 2$
Cette propriété s'applique pour chaque *VehicleId*. Elle s'assure qu'après l'entrée du véhicule dans un convoi, les niveaux de batterie et distance restante du véhicule sont supérieurs à respectivement 5% et 2km, jusqu'à la sortie du véhicule.

[1]FTPL vient de la fusion entre TPL (Temporal Pattern Language) et le préfixe 'F' comme 'First order logic'.

Comme nous le montrent ces exemples, une propriété FTPL est constituée entre autres de mots clés temporels (**after, before, always, eventually**), d'évènements (*CreatePl*, *DeletePl*) et de propriétés de configuration (*Distance* > *20*, *Battery* > *5*, *VehicleNb* > *2*). Ces propriétés s'inscrivent dans le contexte global de la route qui inclut les voitures et leurs convois distingués grâce aux identifieurs (*VehicleId* et *PlatoonId*). Ces propriétés assurent à l'exécution que le système se comporte correctement. Toutefois, il est possible que certaines règles soient évaluées à potentiellement vrai (resp. potentiellement faux) dans le cas d'un **always** (resp. **eventually**) qui est évalué à vrai (resp. faux) jusqu'à un instant t mais qui demande d'être évalué à vrai jusqu'à (resp. vrai au moins une fois avant) un instant t'.
Les propriétés sont utilisées pour exprimer des exigences sur le système mais elles seront aussi dans la définition des politiques d'adaptation que nous présentons maintenant.

2.3 Politiques d'adaptation

Des politiques d'adaptation ont été définies dans [2, 5, 7], nous les avons redéfinies afin de les adapter à notre modèle. Dans notre approche, les opérations de reconfiguration sont gardées par des séquences spécifiques d'évènements.
Reprenons notre exemple de convoi de véhicules autonomes, afin de garantir la propriété $\varphi 2$, nous devons réfléchir aux reconfigurations à effectuer. C'est le rôle des politiques d'adaptation et c'est au niveau de leurs règles de reconfiguration que ces reconfigurations sont décidées. Deux exemples de règles de reconfiguration sont données en Figure 3.

```
when (after Join before Quit and VehicleId.battery < 33)
   if (state = leader ) then
        utility of PassRelay is high

when (after Join before Quit and VehicleId.battery > Leader.battery)
   if (state = platooned ) then
        utility of GetRelay is medium
```

Figure 3: Deux règles de reconfiguration du convoi de véhicules autonomes

Ces règles de reconfiguration stipulent, dans un premier temps, la propriété FTPL à valider, elle est décrite après l'utilisation du mot-clé `when` qui délimite l'intervalle d'évènements durant lequel la reconfiguration peut avoir lieu. Dans un second temps, les règles de reconfiguration contiennent une garde sur la configuration du système, qui est décrite après l'utilisation du mot-clé `if` qui définit la configuration courante du système qui peut entraîner une reconfiguration. Pour finir, le nom de la reconfiguration R_N et son utilité F est renseignée sous forme de valeur floue (`high, medium, low`) après le mot clé *utility* afin d'associer une utilité avec un nom d'opération de reconfiguration.
Ces deux exemples s'appliquent aux véhicules et servent à déterminer les moments de passage de relais entre le véhicule leader et un autre véhicule du convoi. Dans le premier cas, la reconfiguration `PassRelay` se déclenche lorsque le leader n'a plus assez de batterie et dans ce cas il doit être rétrogradé. Dans le second cas, la reconfiguration `GetRelay` se déclenche lorsque l'autonomie du véhicule courant est supérieure à celle du leader.
L'implémentation des politiques d'adaptation et plus particulièrement les règles de reconfiguration sont laissées à la discrétion des développeurs qui sont susceptibles de mal interpréter la spécification. Une mauvaise implémentation peut mener le système à se reconfigurer lorsque ce n'est pas nécessaire, ne pas se reconfigurer ou choisir la mauvaise reconfiguration et engendrer une violation de propriété. Le processus de tests décrit dans la prochaine section a pour but de répondre à ce problème en validant que les politiques d'adaptation sont correctement écrites et retranscrites à l'exécution du système.

3 Validation de l'implémentation vis à vis des politiques d'adaptation

Dans cette section, nous présentons les contributions apportées lors de la thèse dont l'ensemble est résumé Fig. 1. Les politiques d'adaptation influencent le système, il est donc nécessaire de s'assurer que le système implémente correctement ces politiques d'adaptation. Cela consiste à vérifier de manière indépendante que le système se reconfigure avec les bonnes reconfigurations au bon moment. Le système doit également toujours répondre aux propriétés fonctionnelles. Nous proposons également un moyen de nous assurer de la conformité des priorités avec la spécification.

Une des difficultés de cette approche est que les vérifications de propriétés sont faites à posteriori et durant l'exécution. En effet, les propriétés temporelles doivent atteindre leur état final pour être évaluées. De la même façon, les reconfigurations déclenchées par le système guidées par les politiques de reconfiguration peuvent mener à une erreur mais après un certain délai. Pour répondre à ces problématiques, nous proposons une analyse à l'exécution des traces produites par le système. Afin de garantir la pertinence de ces traces, nous élaborons des critères de couverture garantissant que le système a bien exploré les propriétés et les politiques d'adaptation. Tout d'abord, nous présentons une méthode de génération de tests à partir d'un modèle d'usage indépendant en simulant l'environnement extérieur au système. Ce modèle a pour but de produire les traces pertinentes pour les critères de couverture définis. Pour finir, nous expliciterons les verdicts de test mis en place et plus précisément les verdicts sur le respect des politiques d'adaptation.

3.1 Critères de couverture

Nos critères de couverture pour les propriétés FTPL s'appuient sur la définition d'un automate associé à ses propriétés et adaptent les précédents travaux de [3].

Definition 1 (Automate de propriétes FTPL)**.** *Soit un automate noté A_φ composé du quadruplet $\langle Q, q_0, Q_f, T\rangle$ avec Q l'ensemble des états, $q_0 \in Q$ est l'état initial, $Q_f \subset Q$ est l'ensemble des états finaux, et $T : Q \times Ev_\varphi \to Q$ est la fonction de transition avec Ev_φ l'ensemble des évènements possibles dans la propriété.*

Nous allons nous appuyer sur les exemples de la Sect. 2.2 et particuliérement la propriété :

after *Join* **before** *Quit* **always** *VehicleId.Battery* > 5 and *VehicleId.Distance* > 2
La création de l'automate visible Fig. 4 est déterminée par les mots-clés temporels et leurs évènements associés. Notre automate est donc constitué des transitions *join* et *quit*. ϵ représente l'ensemble des transitions possibles. L'état final est atteint lorsque le scénario de la propriété a été complété, dans notre exemple lorsqu'un véhicule est entré et sorti du convoi.

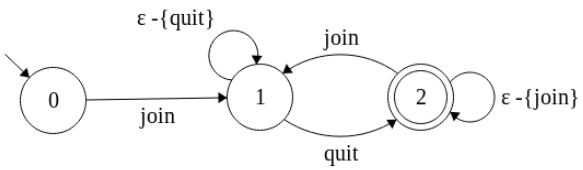

Figure 4: Automate de la propriété φ_1

Definition 2 (Couverture des propriétés FTPL)**.** *Soit une propriété φ et A_φ son automate associé. Un état de l'automate A_φ est dit* couvert *si tous ses couples de transitions entrantes et sortantes ont été*

parcourus par le système. À noter qu'une transition sortante d'un état et entrante dans le même état ne compte pas en tant que transition sortante ou entrante. Ce critère de couverture permet d'assurer une meilleure couverture qu'en se basant uniquement sur le parcours de chaque état ou transition. Une propriété est alors dite couverte *lorsque tous les états de son automate associé ont été couverts.*

Dans notre exemple Fig. 4, pour que A_φ soit *activé*, les trois couples de transitions $0 \xrightarrow{join} 1 \xrightarrow{Quit} 2$, $1 \xrightarrow{Quit} 2 \xrightarrow{Join} 1$ et $2 \xrightarrow{Join} 1 \xrightarrow{Quit} 2$ doivent être sensibilisés.

Definition 3 (Couverture des politiques d'adaptation)**.** *Une règle de politique d'adaptation est dite* activable *si sa garde (partie* `if`*) et sa propriété de déclenchement (partie* `when`*) sont évaluées à vrai et si l'opération de reconfiguration associée est exécutée par le système. Une politique d'adaptation est dite* couverte *lorsque toutes ses règles ont été activées.*

Maintenant que les critères de couverture ont été définis, nous présentons le modèle d'usage permettant de les satisfaire.

3.2 Modèle d'usage et génération de tests

Le modèle d'usage a pour but de produire des séquences d'évènements externes qui exercent le système. Les systèmes adaptatifs réagissent d'une part aux évènements externes et d'autre part aux évolutions de leurs variables internes en accord avec les politiques d'adaptation qui guident à quel moment déclencher les reconfigurations. En d'autres termes, le système se comporte à la manière d'une boîte noire ce qui rend la définition de son modèle impossible. C'est pourquoi nous considérons un modèle d'usage du système sous test. Ce modèle d'usage ne va pas représenter le comportement du système mais simuler l'environnement dans lequel le système évolue. Ce type de modèle a pour but de spécifier les différents évènements qui peuvent avoir lieu dans l'environnement. Pour rappel, nous considérons les systèmes régis par des tics d'horloge, et donc, en plus des évènements externes, nous définissons la notion de *délai*. Cela se traduit par une absence d'évènement externe durant une certaine période, mais de son côté, le système continue d'évoluer et de mettre à jour son état interne. L'automate dans la Figure 5 représentant le modèle d'usage est défini de la même façon qu'un automate de propriété temporelle mais à une différence : la notion de *délai* est présente dans l'automate du modèle alors que l'automate de propriété ne garde que les évènements qui agissent sur sa propriété. L'automate génère pendant l'exécution (en ligne) ce qui permet à l'automate d'être connecté au système. En effet, en plus d'envoyer les séquences d'évènements externes au système, l'automate peut observer les évènements internes du système. Ces concepts sont illustrés Figure 5.

L'exemple des voitures autonomes utilise trois évènements externes marqués par des transitions en

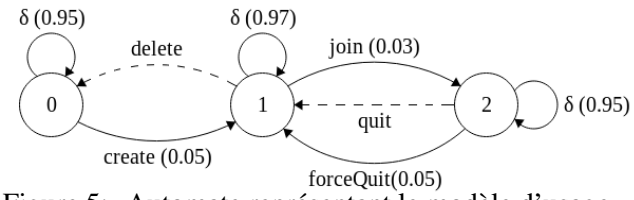

Figure 5: Automate représentant le modèle d'usage

lignes pleines : (*create*) crée un véhicule en l'ajoutant sur la route, (*join*) est déclenché lorsque le véhicule est à portée d'un ou plusieurs véhicules et (*forceQuit*) est déclenché lorsqu'un véhicule veut sortir de son convoi. Les évènements marqués par des transitions en tiret sont des évènements internes observables par le modèle d'usage, ils servent à garantir la cohérence du modèle. En effet,

l'évènement (*delete*) est déclenché lorsqu'un véhicule arrive à destination et ne peut avoir lieu avant la création de ce véhicule. De la même façon, un véhicule peut quitter le convoi avec l'évènement (*quit*) suite à une reconfiguration déclenchée par le système, dans ce cas l'automate d'usage peut à nouveau déclencher l'évènement (*join*).
A noter que l'automate est discret, et donc le *délai* représente une unité de temps. Dans ces conditions, pour générer une séquence d'évènements, nous utilisons un algorithme de type 'Markov random walk' [9] qui consiste à laisser la séquence être générée par parcours aléatoire de l'automate probabiliste. Le processus de génération continue de générer des évènements jusqu'à atteindre les critères de couverture. Avec ces cas de tests, le système est stimulé et peut déclencher des changements internes ou des reconfigurations dans le système. Le système produit une trace de reconfiguration qui est analysée selon le respect des politiques d'adaptation et propriétés FTPL. Cette analyse, les métriques et leurs verdicts sont décrits maintenant.

3.3 Verdicts de tests

L'établissement d'un verdict de test s'appuie sur deux artefacts. La première étape consiste à vérifier les propriétés FTPL. La seconde étape s'occupe de vérifier les conditions d'exécution des reconfigurations. Ce travail se base sur les travaux [8] sur la vérification des propriétés FTPL.

Verdict sur les propriétés FTPL Intuitivement, une propriété FTPL *passe le verdict* pour toute séquence de longueur *i*, si l'état à l'indice *i* est le dernier état final de la propriété FTPL et que la propriété est vérifiée. Le verdict est basé sur le dernier état final de la séquence et donc toute séquence sans état final est dite *inconclusive*. La propriété est dite *violée* s'il existe un état ne respectant pas la propriété peu importe si l'état est final ou non.

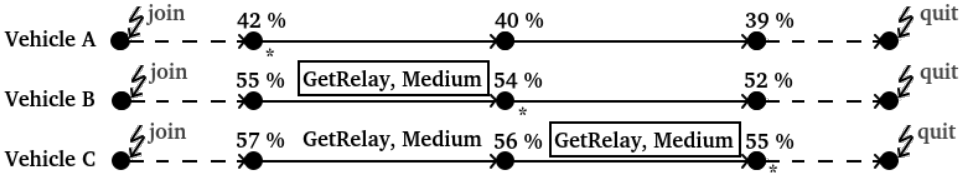

Figure 6: Séquence de reconfigurations observables

Nous définissons à partir des exemples présentés dans la Secion 2.3 notre deuxième partie de verdict basé cette fois sur les politiques d'adaptation en essayant de détecter les reconfigurations non souhaitées.
La Figure 6 montre un exemple de traces produites par trois véhicules au sein d'un même convoi. Les évènement externes *join* et *quit* sont indiqués de la même façon que dans la Figure 2, les reconfigurations internes (*GetRelay*) sont associées à une priorité (*Medium*). À chaque état, le niveau d'énergie restante est indiqué par un pourcentage et le signe $*$ signifie que le véhicule est *leader*. Entre chaque état, les reconfigurations possibles ainsi que leur priorité sont affichées. La reconfiguration choisie par le système est encadrée.
On appelle *eligible* une règle de reconfiguration qui est activable. On appelle *actual* règle de reconfiguration activée. La propriété qui en découle vérifie que la reconfiguration *actual* est présente dans la liste des reconfigurations éligibles. Dans notre exemple, GetRelay est *eligible* pour les véhicules B et C donc, on s'attend à voir une telle reconfiguration être activée, contrairement à PassRelay définie Figure 3 qui déclencherait une erreur si elle était activée dans ce cas de figure.
En complément de ce verdict, nous proposons des métriques sur les règles de reconfiguration : $\#elig$ est donné par le nombre de fois qu'elle a été désignée *eligible* et $\#actual$ est donné par le nombre de

fois qu'une règle est désignée *actual*. Ces métriques nous permettent d'effectuer des ratios permettant de comparer le taux de déclenchement. Si une règle à priorité élevée possède un taux de déclenchement plus faible qu'une règle à priorité basse c'est qu'il existe potentiellement une incohérence dans le déclenchement des reconfigurations ou la définition des politiques d'adaptation.

4 Conclusion et axes de recherche

Ce papier a présenté une approche de tests à partir de modèles qui a pour but de vérifier qu'un système adaptatif implémente fidèlement les politiques d'adaptation spécifiées indépendamment. Cette approche se base sur un modèle d'usage générant des séquences de tests permettant de diriger le système sous test. La validation du système repose sur plusieurs verdicts : le respect des propriétés temporelles ainsi que l'évaluation de la légitimité des reconfigurations. Étant donné que nos mesures et vérifications sont effectuées à l'exécution, la génération des séquences de tests peut être effectuée avant ou pendant l'exécution. Cette approche a été implémentée sur un programme Java modélisant le convoi de véhicules autonomes décrit dans ce papier. Les résultats obtenus valident, pour cet exemple l'approche décrite. Les travaux futurs se dirigent vers une validation des propriétés non fonctionnelles du système en analysant quels paramètres peuvent être modifiés pour mieux répondre à des questions d'efficacité. Nous préparons des expérimentations sur d'autres cas d'étude.

References

[1] B. Beizer. *Black-box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[2] F. Chauvel, O. Barais, N. Plouzeau, I. Borne, and J.-M. Jézéquel. Expression qualitative de politiques d'adaptation pour Fractal. In Y. Aït Ameur, editor, *2ème Conf. sur les Architectures Logicielles (CAL 2008), 3-7 Mars 2008, Montréal, Québec, Canada*, volume RNTI-L-2 of *Revue des Nouvelles Technologies de l'Information*, page 119. Cépaduès-Éditions, 2008.

[3] F. Dadeau, K. Cabrera Castillos, and J. Julliand. Coverage criteria for model-based testing using property patterns. In A.K. Petrenko and H. Schlingloff, editors, *MBT 2014, 9th Workshop on Model-Based Testing, Satellite workshop of ETAPS 2014*, volume 141, pages 29–43, Grenoble, France, apr 2014.

[4] R. De Lemos, H. Giese, H.A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N.M. Villegas, T. Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[5] J. Dormoy and O. Kouchnarenko. Event-based adaptation policies for Fractal components. In *AICCSA 2010, ACS/IEEE Int. Conf. on Computer Systems and Applications*, pages 1–8, Hammamet, Tunisia, may 2010.

[6] J. Dormoy, O. Kouchnarenko, and A. Lanoix. Using temporal logic for dynamic reconfigurations of components. In L. Barbosa and M. Lumpe, editors, *FACS*, volume 6921 of *LNCS*, pages 200–217. Springer Berlin Heidelberg, 2012.

[7] O. Kouchnarenko and J.-F. Weber. Adapting component-based systems at runtime via policies with temporal patterns. In J. L. Fiadeiro, Z. Liu, and J. Xue, editors, *FACS, 10th Int. Symp. on Formal Aspects of Component Software*, volume 8348 of *LNCS*, pages 234–253. Springer, 2014.

[8] O. Kouchnarenko and J.-F. Weber. Decentralised evaluation of temporal patterns over component-based systems at runtime. In I. Lanese and E. Madelaine, editors, *Formal Aspects of Component Software*, volume 8997 of *LNCS*, pages 108 – 126, Bertinoro, Italy, sep 2015. Springer.

[9] A. Sinclair. *Algorithms for Random Generation and Counting: A Markov Chain Approach*. Birkhauser Verlag, Basel, Switzerland, Switzerland, 1993.

[10] Jean-François Weber. *Guider et contrôler les reconfigurations de systèmes à composants: Reconfigurations dynamiques: modélisation formelle et validation automatique. (Guide and control component systems-based system reconfigurations: Dynamic reconfigurations: formal modelling and automatic validation)*. PhD thesis, University of Franche-Comté, Besançon, France, 2017.