

# Model-Checking aléatoire: une approche entre test et vérification

Marc BOYER  
ENSEEIH T – TésA / IRIT (IRT)  
2, rue Camichel  
31071 TOULOUSE Cedex 7  
Marc.Boyer@enseeiht.fr

Jean-Christophe PINCE  
M3-Systems  
1, rue des oiseaux  
31410 Lavernose  
pince@m3systems.net

5 mars 2004

## Résumé

Dans ce papier, nous rapportons l'expérience de la modélisation d'un système industriel embarqué réel, en montrant comment une PME a pu, dans le cadre d'un DRT (Diplôme de Recherche Technologique) étudier un bus satellitaire. De surcroîts, cette étude a donné lieu à une nouvelle façon d'utiliser en milieu industriel les outils de model-checking, nommée "model-checking aléatoire", une approche entre le test et la vérification, qui permet de faire mieux que le test, sans avoir l'exhaustivité du model-checking, mais qui permet surtout de limiter le problème de taille et donc d'utiliser en vérification un modèle qui semble plus naturel dans la culture d'entreprise.

## 1 Introduction

La robustesse du code est une exigence des industriels travaillant sur les systèmes embarqués ou critiques. Elle s'appuie principalement sur des critères de qualité du code, de tests (voire de redondance), mais peu encore sur la vérification formelle, pourtant développée dans ce but. A cela, on peut trouver trois raisons majeures :

1. la nécessité de développer un prototype spécifiquement destiné à la vérification, puisque la vérification de code reste marginale,
2. une image de trop grande complexité des langages de vérification, qui les réserveraient à des chercheurs,
3. l'inadéquation flagrante entre la taille des problèmes industriels et ceux que les outils de vérification arrivent à traiter, qui demande une expertise certaine pour écrire un modèle qui ressemble assez au système réel pour que les preuves apportées soient significantes, mais assez petit pour être traitable par les outils.

Dans ce papier, nous rapportons l'expérience de la modélisation d'un système industriel embarqué réel, en montrant comment une PME a pu, dans le cadre d'un DRT (Diplôme de Recherche Technologique) étudier un bus satellitaire. De surcroîts, cette étude a donné lieu à une nouvelle façon d'utiliser en milieu industriel les outils de model-checking, nommée "model-checking aléatoire", une approche entre le test et la vérification, qui permet de faire mieux que le test, sans avoir l'exhaustivité du model-checking, mais qui permet surtout de limiter le problème de taille et donc d'utiliser en vérification un modèle qui semble plus naturel dans la culture d'entreprise.

Nous allons tout d'abord présenter le principe du model-checking aléatoire (partie 2), son principe et les bénéfices attendus, puis, après une présentation du cas d'étude (SOIF sur bus MIL-STD-1553B, dans la partie 3), nous présenterons dans la partie 4, les vérifications menées sur le système, avec le model-checking classique et notre approche de model-checking aléatoire. La présentation se termine dans la partie 5 par une présentation d'autres usages possibles du model-checking aléatoire (que nous n'avons pas menés dans cette étude), puis se conclue sur les travaux en cours (partie 6).

## 2 Le model-checking aléatoire

L'approche présentée dans cet article est nommée "model-checking aléatoire". En peu de mots, on peut la définir comme le fait de tester toutes les réponses possibles à une entrée aléatoire, en utilisant des outils de model-checking.

Présentons la plus en détail.

### 2.1 Entre test et vérification : s'immiscer dans le processus industriel

On peut souvent définir un système comme une boîte possédant une séquence en entrée (stimulus) et produisant une séquence de sortie (réponse). La démarche de test peut alors se voir comme le fait d'envoyer dans la boîte un ensemble de séquences d'entrée, et de vérifier si, pour chacune, la séquence de sortie associée est correcte, *i.e.* vérifie certaines propriétés. Pour augmenter la confiance, la démarche de test considère souvent le système comme une boîte blanche, pour s'assurer d'avoir couvert un maximum de cas significatifs (mais chaque test individuel voit le système comme une boîte noire). Bien sur, le test n'apporte qu'une confiance partielle sur les propriétés du systèmes, puisque on n'observe qu'un sous-ensemble des comportements possibles.

L'approche par model-checking s'intéresse à l'ensemble des comportements possibles d'un système, représenté par un graphe dans lequel les états possibles du système sont les nœuds et les arcs sont les actions qui font évoluer le système d'un état à un autre. Dans cette approche, seul le nom d'une action permet de distinguer si elle est une entrée, une sortie ou une action interne. Le model-checking permet d'avoir une confiance totale sur les propriétés du systèmes, puisqu'il prend en compte tous les comportements possibles.

Pour relier les deux approches, on peut considérer une démarche de tests comme un ensemble de trajectoires à travers l'ensemble des comportements possibles.

De ce point de vue, le model-checking paraît supérieur au test. Cette supériorité théorique est limitée par deux aspects pratiques : le langage de description du système et la taille du système :

- Le test permet souvent de tester le système réel, celui qui sera déployé, écrit dans un langage de programmation. Le model-checking nécessite généralement qu'on écrive (et décrive) le système dans un langage spécifique (Nous nommerons cette description du système "modèle"). Cela nécessite donc un effort supplémentaire, et on est pas vraiment sûr que le modèle décrive correctement le système. Comme il arrive fréquemment que le langage de modélisation soit moins riche que le langage de programmation, le modèle n'est souvent qu'une abstraction du système, et se pose que la question alors de savoir ce que les propriétés vérifiées sur le modèle prouvent sur le système réel.
- Comme le model-checking nécessite de stocker *tous* les états (alors que le test ne s'intéresse qu'à une trajectoire à la fois), il ne permet de travailler que sur des systèmes plus petits que ne le permet le test. Souvent, un système industriel réel est trop gros pour le model-checking, et on doit construire un modèle réduit, une abstraction. En général, construire une abstraction qui soit de taille suffisante pour les outils et qui ne soit pas une simplification abusive de la réalité demande une expertise qu'on ne trouve que dans le milieu académique. Le model-checking n'existe pas encore de manière autonome dans l'industrie.

Il faut néanmoins préciser que ces deux approches se situent<sup>1</sup> généralement à des étapes très différentes du cycle de développement. Le test suppose que le système existe. Il arrive donc très tard dans la démarche de développement. Le model-checking peut être fait sur un modèle lors de la phase de conception, bien en amont du développement lui même.

Cette affirmation doit bien sur être nuancée par le fait que la conception peut passer par le développement d'un prototype, ou d'une spécification avec une sémantique opérationnelle (comme SDL [SDL99]) et qu'on peut donc tester et/ou vérifier le prototype ou la spécification. Ceci n'invalide en rien notre approche : pour des systèmes réels, même un prototype un peu réaliste peut s'avérer trop gros pour les outils de model-checking et demander un effort important pour être réécrit dans un langage formel.

Pour limiter les effets de la spécificité du langage de modélisation, l'équipe "Distributed and Complex System" du VERIMAG a décidé de définir un langage de modélisation, IF [BFG<sup>+</sup>99, BFG<sup>+</sup>00], très proche de SDL, un langage de description de systèmes temps réels asynchrones, utilisés dans l'industrie pour la spécification, voire la génération automatique de code. IF est si proche de SDL qu'il existe un traducteur automatique de SDL vers IF. Ce traducteur automatique, comme beaucoup, ne donne pas un code IF très agréable à

---

<sup>1</sup>ou devraient se situer...

lire. Néanmoins, les deux sémantiques sont tellement proches que, comme l’a confirmé l’expérience relatée dans cette étude, un industriel connaissant SDL passe sans difficulté au langage IF.

Un industriel peut donc, de façon relativement autonome, produire un modèle qui soit assez proche de la spécification de son système réel. Reste le problème de la taille : le modèle étant proche du système réel, il est généralement trop gros pour être traité par les outils de model-checking. C’est là que se situe notre contribution.

Souvent, l’ensemble des états accessibles suite à *une* stimulation est de taille assez faible. L’idée est d’utiliser une possibilité laissée par l’outil de model-checking `generator` pour explorer l’ensemble de états accessibles suites à des stimulations aléatoires.

## 2.2 Générer un trafic aléatoire en pratique

Dérouler une stimulation consiste, grossièrement, à choisir à chaque étape une valeur dans un ensemble, l’ensemble pouvant être représenté de façon explicite ou implicite (les valeurs possibles étant alors souvent les solutions d’un ensemble de contraintes, seules les contraintes étant explicites). Dans la modélisation, une machine à état offre, depuis chaque état, l’ensemble des valeurs possibles, d’une manière ou d’une autre. Le moteur de model-checking, voyant ses différentes possibilités, les explore toutes.

L’idée de l’approche aléatoire est de masquer ses différents choix dans une unique transition, que nous appelons *in* qui choisit au hasard une des possibilités. Cette possibilité est offerte dans le langage IF qui permet de mettre du code C++ dans une transition. L’hypothèse faite par l’explorateur d’espace d’état `generator` est que cette transition est déterministe. Depuis chaque état où cette transition est franchissable, il ne l’utilise qu’une seule fois, n’explorant qu’une entrée parmi toutes celles possibles.

Comment cela s’arrête-il ? Comme toujours en model-checking : quand on retombe sur un état déjà visité. On applique *in* une seule fois à chaque état. La stimulation se termine quand on retrouve un état déjà connu. C’est donc cette notion d’état qu’il faut éclaircir. Si le système suit l’hypothèse d’un système réactif dans lequel la réaction est instantanée, c’est à dire que la boîte noire traite le stimulus complètement avant qu’une nouvelle entrée ne se présente, on risque d’explorer bien peu d’états. Si en plus le système n’a que peu de mémoire de son passé, la séquence risque d’être très courte. Dans un tel cas, il y a de fortes chances pour que le système puisse se traiter avec des outils de model-checking classiques. Si par contre le système garde mémoire des entrées, par exemple parce qu’il n’a pas fini de les traiter, la séquence devient plus significative.

Illustrons cette approche par un exemple simple, symbolisé dans la figure 1. Imaginons un système producteur–consommateur relié par une FIFO de taille bornée à 2, sans écrasement possible de données. Le producteur peut envoyer 0 ou 1. Pour différencier entrée et sortie, le consommateur produit *a* quand il lit un 0 et *b* quand il lit un 1.

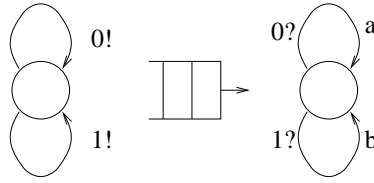


FIG. 1 – Système producteur–consommateur avec une FIFO de taille 2

Le graphe d'état étiqueté du système est présenté dans la figure 2. Il contient 7 états et 14 transitions.

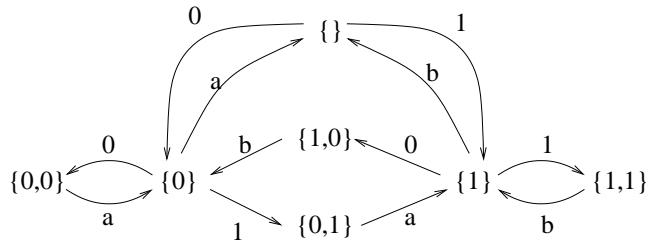


FIG. 2 – Graphe d'état étiqueté du système producteur–consommateur

Quel pourrait être le résultat d'une exploration aléatoire ?

Commençons par transformer le modèle de manière à ce que le choix de la valeur 0 ou 1 émise soit reporté dans une fonction  $f()$ . Cette fonction fait un choix aléatoire entre 0 et 1, mais le moteur de model-checking la croit déterministe, comme illustré sur la figure 3.

Une exploration par model-checking aléatoire dépend bien sûr de la séquence produite par le générateur, ici  $f()$ . Imaginons que cette séquence soit 1, 0, 1. Alors, le graphe d'état étiqueté du système aléatoire est celui de la figure 4, avec 5 états et 6 transitions (soulignons que l'action de consommation est systématiquement explorée, puisque le model-checker explore toutes les réponses à une stimulation donnée).

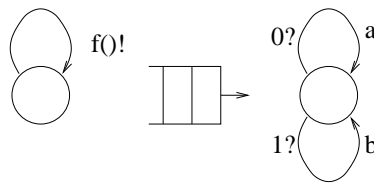


FIG. 3 – Système producteur–consommateur avec une FIFO de taille 2 modifié

Quel pourrait être le résultat d'un test ? Dans une telle démarche, il faut aussi choisir quand a lieu la consommation. Prenons la même séquence de produc-



model-checking aléatoire est tout de même supérieur au test pour les systèmes parallèles puisqu'il permet de vérifier toutes les réponses à un stimulus et non une seule.

Le model-checking aléatoire a donc les avantages et inconvénients des deux techniques. Là où il semble novateur, c'est qu'il peut s'intégrer dans les démarches industrielles en bouleversant moins les habitudes que la vérification classique. De plus, sans trop d'effort, et en réduisant les paramètres du système, l'utilisateur peut s'essayer à la vérification.

## 2.4 Lien avec le model-checking à la volée

L'idée qu'il n'est pas forcément nécessaire de parcourir tout le graphe d'état pour trouver un contre exemple à une propriété n'est pas nouvelle. Elle a été appliquée dans le model-checking à la volée, dans lequel le moteur prend en entrée un système *et* une propriété à vérifier, et s'arrête dès qu'il peut conclure de la validité de la propriété (le plus souvent en trouvant un contre exemple). La limitation des moteurs de model-checking à la volée, c'est qu'ils ne permettent pas de guider finement l'exploration, offrant en général uniquement deux alternatives : privilégier la profondeur ou la largeur, sans distinction entre les entrées et les réponses. Parfois, le contre exemple est "trop loin" et échappe à ces deux stratégies. L'approche présentée ici pourrait être vue comme un système "en profondeur aléatoire" pour les entrées, et exhaustif pour les réponses.

## 3 Le cas d'étude : SOIF sur bus MIL-STD-1553B

Afin de permettre une réutilisation de composants dans le domaine des satellites, l'agence spatiale européenne (ESA) a établi en décembre 1999 une "spacecraft onboard interface" (SOIF) qui est un modèle de référence en couches pour les communications embarquées et un ensemble de services pour accéder à ces communications.

La société M3-Systems avait eut en charge, dans le cadre du projet DABUTUS, l'étude et le prototypage (logiciel et matériel) de la partie du modèle relative aux communications à l'intérieur d'un sous-réseau simple au dessus d'un bus MIL-STD-1553. Lorsque la société a choisi d'évaluer, dans le cadre du DRT de J.C. PINCE la possibilité de valider par vérification formelle une étude industrielle typique de son activité, c'est donc ce cas qu'elle a choisi.

Dans le modèle SOIF, on distingue deux types d'hôtes : des unités inintelligentes (souvent des capteurs ou des périphériques de sortie) qui transmettent un flot de donnée temps réel (LODI) et des unités intelligentes (souvent des actionneurs), qui transmettent un flot de donnée non temps réel, IP dans notre étude.

Le flot temps réel se décompose en un flot synchrone périodique (échanges de valeurs, programmé lors de la définition de la mission du satellite), et un flot asynchrone (comme des alarmes).

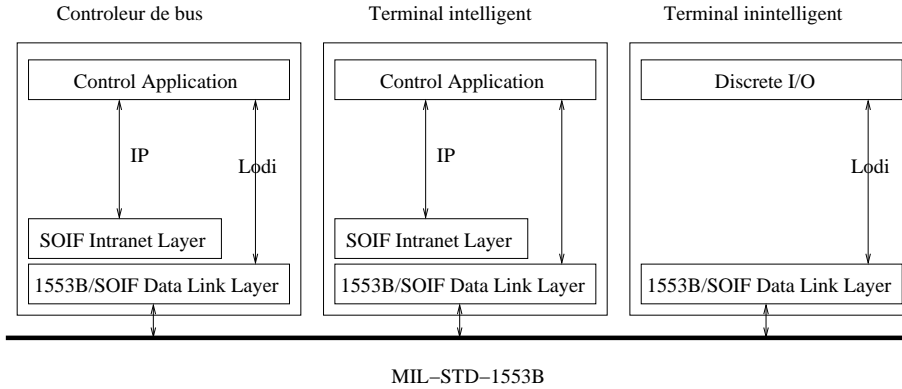


FIG. 6 – Architecture SOIF sur bus MIL-STD-1553B

Le bus MIL-STD-1553 est un bus maître-esclave, dans lequel le maître attribue les tours et temps de “parole”. L’implémentation de SOIF sur MIL-STD-1553 est organisée autour d’une table d’allocation, divisée en  $n$  tranches (*slots*) de durées égales. Dans chaque tranche de temps, le bus commence par donner la parole aux LODI synchrones prévus dans cette tranche, puis il interroge tous les terminaux, intelligents et inintelligents, pour savoir s’ils ont des messages à émettre (*polling*). Ensuite, il donne la parole prioritairement aux LODI, puis en fonction du temps qu’il reste, il laisse passer les messages IP, en considérant prioritairement ceux qui sont émis par le contrôleur de bus, et ensuite seulement ceux des terminaux. Après avoir effectué la  $n^{\text{ème}}$  tranche, il recommence à la première.

- Les principales propriétés qui intéressaient la société dans ce cas étaient :
- l’absence de blocage et de boucle infinie (*deadlock* et *livelock*),
  - la non saturation des buffers, et
  - un temps de latence borné pour les LODI asynchrones

## 4 Vérifications du modèle : model-checking classique et aléatoire

La démarche classique de vérification par model-checking consiste donc à modéliser le système, ce qui fut fait en utilisant le langage IF de manière assez rapide grâce à l’expertise déjà présente sur le sujet et sur le langage SDL. Il y eut ensuite bien sur une phase d’abstraction, classique, en coopération entre la société M3-Systems et l’équipe IRT de l’IRIT (fusion des différents terminaux en un seul processus pour réduire les entrelacements, abstraction des valeurs des données transférées sur le bus pour ne prendre en compte que la taille, etc.).

## 4.1 Paramètres des configurations testées

Une fois faite ces simplifications, des valeurs représentatives d'une mission typique ont été choisies, présentées dans les tableaux 1 et 2. Le chercheur habitué au model-checking voit tout de suite qu'un problème évident apparaît : la disparité d'ordre de grandeur entre les différentes valeurs numériques, toujours pénalisantes pour les outils de vérification, mais incontournable car reflet des valeurs réelles du système.

Temps inter-messages <sup>1</sup>	8 $\mu$ s
Temps de réponse <sup>1</sup>	8 $\mu$ s
Temps transfert d'un octet	8 $\mu$ s
Temps de <i>polling</i>	32 $\mu$ s
Durée d'un slot	10 ms
Période de <i>polling</i>	1 s
Nb de slot simulés	150
Nb de terminaux intelligents	2
Nb de buffers IP dans les terminaux	3
Nb de buffers IP dans le contrôleur de bus	6
Période de génération IP	1s
Taille min paquet IP	1000 octets
Taille min Lodi asynchrones	4 octets

<sup>1</sup>Durée imposée par le standard MIL-STD-1553B

TAB. 1 – Constantes numériques du système

On ajoute aussi au systèmes deux flux de LODI synchrones : un flux de commande de 7 octets toutes les 50ms et une flux d'acquisition de 10 octets toutes les 10ms.

	s0	s1	s2	s3	s4	s5	s6	s7
Nb Lodi asynchrones par slot	2	3	4	3	2	3	4	3
Gigue génération IP	0s	0s	0s	0s	.5s	.5s	.5s	.5s
Taille max paquet IP	1001	1001	1001	1051	1001	1001	1001	1051
Taille max Lodi asynchrones	5	5	5	16	5	5	5	16

les tailles sont exprimées en octets

TAB. 2 – Valeurs associées aux différentes configurations

Les configurations s0, s1 et s2 vont en complexité croissante par augmentation du nombre de flux de Lodi. La configuration s3 reprend la configuration s1, mais avec des tailles de messages qui peuvent être plus importantes.

Les configurations s4 à s7 sont respectivement les mêmes que les s0 à s3, en augmentant la gigue pour le flux IP.

## 4.2 Vérifications des propriétés

Comme présenté dans le paragraphe 3, les propriétés recherchées dans cette étude sont l'absence de blocage et de boucle infinie, la non saturation des buffers, et le respect du délais sur les Lodis.

Pour vérifier le respect des délais, il a fallu ajouter un observateur au système. Les études ont donc porté sur deux modèles différents : un modèle système, et un modèle système plus observateur.

Le tableau 3 montre les résultats de l'étude faite sur le système. On voit très nettement la différence de taille de l'espace d'états visités entre le model-checking classique et l'approche aléatoire. On voit que dès que le système devient moins déterministe (taille de message pouvant être plus importante comme dans s3, gigue sur les arrivés de message IP), les systèmes sont trop gros pour être traité par les outils.

En ce qui concerne les propriété vérifiées, on a débordement de buffer dans la configuration s2, détecté par l'approche classique et que ne voit pas l'approche aléatoire. L'approche aléatoire ne voit écrasement de buffer que dans s7, qui est une version avec gigue de s3, qui lui même est identique à s1, hormis des tailles de paquets plus grandes.

Le tableau 4 montre les résultats de l'étude faite sur le système avec un observateur qui regarde si le temps de latence des Lodis dépasse la borne désirée.

Là encore, la différence de taille entre le model-checking classique et sa version aléatoire est impressionnante. Mais cette fois, la version aléatoire détecte une violation de propriété sur des modèles que la version classique ne peut pas traiter.

De point de vue du système étudié, ces résultats montrent la complémentarité des deux approches : la configuration s3 est une version simplifiée de la réalité dans laquelle les messages non temps réels arrivent à période fixe... Détecter un débordement de buffer dans une situation aussi favorable signifie un mauvais dimensionnement évident du système.

Dans le cas, plus réaliste, où les flux IP arrivent à des dates inconnues, la mise en évidence d'un dépassement du délais toléré doit être suivie d'une analyse de la cause de l'erreur. L'outil `evaluator` [CAD] utilisé pour cette étude le permet puisqu'il donne un contre exemple quand une propriété est violée.

## 5 Autres usages du model-checking aléatoires

### 5.1 Génération guidée : se rapprocher du test

Dans les expériences présentées, le flux d'entrée est généré aléatoirement dans un certain ensemble de contraintes (ici, une variable aléatoire dans un intervalle de valeurs), mais, avec la possibilité de générer la séquence grâce à une code C++. On peut bien sur aller plus loin et diriger complètement la séquence : il suffit de stocker la séquence (dans un fichier par exemple), et d'ajouter une nouvelle variable dans le modèle, qui est le numéro de pas dans la séquence de

Tab. 3 – Vérifications faites sur le modèle du système

	Model-checking					Model-checking aléatoire				
	Nb états	Nb Trans.	Blocage	Boucle	Débordement	Nb états	Nb Trans.	Blocage	Boucle	Débordement
s0	35.080	35.379	Non	Non	Non	2.762	2.791	Non	Non	Non
s1	49.525	49.827	Non	Non	Non	3.966	3.989	Non	Non	Non
s2	53.152	53.451	Non	Non	Oui	4.212	4.235	Non	Non	Non
s3	out of memory					538.046	551.509	Non	Non	Non
s4	out of memory					3.586	3.608	Non	Non	Non
s5	out of memory					5.316	5.339	Non	Non	Non
s6	out of memory					8.316	8.408	Non	Non	Non
s7	out of memory					2.507.214	2.574.529	Non	Non	Oui

	Model-checking			Model-checking aléatoire		
	Nb états	Nb Trans.	Latence Lodi	Nb états	Nb Trans.	Latence Lodi
s0	35.380	36.399	OK	2.787	2.870	OK
s1	49.828	50.847	OK	3.991	4.074	OK
s2	53.452	54.471	OK	4.237	4.320	OK
s3	out of memory			1.515.471	1.562.594	Trop importante
s4	out of memory			10.896	11.080	OK
s5	out of memory			17.784	17.899	OK
s6	out of memory			28.132	28.316	OK
s7	out of memory			8.318.500	9.301.299	Trop importante

TAB. 4 – Vérifications faites sur le modèle du système plus un observateur

stimulation. Incrémenter ce numéro de pas permet de s’assurer que le moteur d’exploration ne retombera pas sur un état déjà connu avant d’avoir traitée toute la séquence.

Cette possibilité permet, suivant l’expertise du testeur, de guider l’exploration vers les zones qui lui semblent le plus intéressantes. Là encore, face à un test classique, on gagne l’exhaustivité de la réponse (face à une stimulation donnée).

Il s’agit ensuite de faire le lien avec toute l’expertise du domaine du test pour générer des flux d’entrée plus pertinents qu’un simple aléa.

## 5.2 Ajouter de la mémoire pour agrandir l’exploration

On peut remarquer sur l’étude que, pour certaines configurations (s4, s5 et s6), la partie de l’espace d’état explorée est relativement petite (comparée à la puissance de calcul de la machine) mais que l’exploration classique ne tient pas en mémoire.

On pourrait souhaiter pouvoir explorer plus d’états, pour avoir une plus grande confiance en le résultat.

Pour ce faire, on peut ajouter au modèle une variable inutile au système. Chaque fois que le système génère une nouvelle donnée en entrée, on met à jour cette variable inutile, avec une valeur aléatoire ou avec la précédente valeur d’entrée. On diminue ainsi les chances de retomber sur un état déjà exploré, et donc on augmente la taille de la recherche.

## 6 Conclusion

Le model-checking aléatoire permet de venir occuper un créneau entre le test et la vérification, dans une démarche qu’un industriel peut mener de manière autonome. Il ne donne une confiance que partielle, mais supérieure à celle du test, et sur des systèmes plus proches de la réalité. Il peut permettre aussi de familiariser les industriels avec les outils de model-checking et donc de faciliter son utilisation.

C'est une démarche prometteuse, mais qui demande à être encore étudiée sur plusieurs points.

Du point de vue théorique, nous sommes en train d'étudier le rapport moyen entre une telle exploration aléatoire et la taille globale du système. La partie la plus intéressante est bien sûr dans la répétition des expérimentations : si une exploration aléatoire couvre en moyenne  $n\%$  du graphe total, combien couvrent  $m$  explorations, comment diminue le nombre d'états non visités ?

Cette répétition d'expérience est aussi à étudier du point de vue expérimental : si l'exploration exhaustive prend un temps  $t$ , combien de vérification aléatoire peut-on faire pendant la même durée ?

Il faut aussi bien sûr intégrer toutes l'expertise du test, pour ne pas simplement générer une entrée aléatoire, mais des entrées pertinentes, et passer du model-checking aléatoire au model-checking partiel. Néanmoins, pour aller plus loin dans ce sens et dépasser le stade de l'expérimentation, il faudra sûrement entrer dans un moteur même d'un outil.

## Références

- [BFG<sup>+</sup>99] M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, L. Mounier, and J. Sifakis. IF : An intermediate representation for SDL and its applications. In *Proceedings of SDL-FORUM'99*, Montreal, Canada, June 1999.
- [BFG<sup>+</sup>00] M. Bozga, J.Cl. Fernandez, S. Ghirvu, L. and. Graf, J.P. Krimm, and L. Mounier. IF : A validation environment for timed asynchronous systems. In *Proceedings of CAV'2000*, Chicago, USA, July 2000.
- [CAD] Cadp (caesar/aldebaran development package) home page. <http://www.inrialpes.fr/vasy/cadp/>.
- [SDL99] Specification and description language (SDL). Recommendation Z.100, International Telecommunication Union, 1999.