Analyse d’une preuve informelle pour guider le test

Guillaume Lussier, Hélène Waeselynck
LAAS-CNRS
7 Avenue du Colonel Roche
31077 Toulouse Cedex 4 - France
Phone: +33/ 5 61 33 78 13
Fax: +33/ 5 61 33 64 11
Email: {glussier,waeselyn}@laas.fr

Résumé

Notre objectif est la vérification d’algorithmes génériques de tolérance aux fautes. Nous souhaitons améliorer le processus de test en utilisant des informations extraites des preuves, formelles ou informelles, de l’algorithme. À titre d’exemple, nous étudions l’algorithme d’ordonnancement FT-RMS (Fault-Tolerant Rate Monotonic Scheduling), prouvé par une démonstration informelle, et révélé incorrect par la suite. Nous nous attachons à l’analyse de sa preuve informelle, que nous restructurons sous la forme d’un arbre de preuve semi-formel, dont sont extraits plusieurs cas fonctionnels utilisés pour tester un prototype de l’algorithme. Les résultats expérimentaux montrent qu’une preuve informelle fausse peut ne pas apporter d’informations pertinentes pour le test. Il nous reste à étudier si une preuve formelle partielle nous permettrait d’établir une meilleure connexion avec les fautes.

1. Introduction

Cet article étudie l’utilisation de preuves pour guider la conception de données de test. Le champ d’application ciblé est celui de la vérification d’algorithmes de tolérance aux fautes.

Un grand nombre de techniques de traitement d’erreur ou de faute ont été proposées dans la littérature. Dans certains cas, les algorithmes sous-jacents ont été prouvés à l’aide d’outils (c.f. [14]). Mais la preuve peut aussi être donnée sous la forme d’une démonstration informelle, auquel cas il existe toujours un risque de découvrir tardivement que la démonstration est fausse et que l’algorithme ne fournit pas les propriétés souhaitées. Des exemples de telles “preuves” d’algorithmes incorrects existent dans la littérature, et l’étude de cas présentée dans cet article en est un.

Les mécanismes de tolérance aux fautes étant des composants critiques d’une architecture sûre de fonctionnement, il est souhaitable que leur correction ait été établie de façon rigoureuse, mais tel n’est pas toujours le cas. De plus, les tentatives de démonstration formelle peuvent échouer, dans le sens où elles peuvent rester partielles à cause de lemmes non prouvés et laissés en suspens.

Le test peut être vu comme une approche pragmatique pour obtenir plus d’assurance dans le cas d’une preuve informelle, ou d’une preuve formelle mais partielle. Il est à noter que le test restera toujours nécessaire pour valider l’implémentation finale au sein d’une architecture tolérante aux fautes. Mais ce n’est pas le point discuté dans cet article: notre intérêt se porte sur la vérification de propriétés fondamentales des algorithmes proposés. L’objet testé peut être un prototype de l’algorithme, ou une spécification exécutable. L’étude de cas que nous avons choisie comme exemple est un algorithme générique d’ordonnancement de tâches incorporant un mécanisme de tolérance aux fautes. Avant d’implémenter cet algorithme au sein d’un système d’exploitation temps-réel, il est fortement souhaitable de vérifier qu’il traite correctement le problème pour lequel il a été conçu (sous certaines hypothèses, aucune tâche ne manque son échéance).

Tester des algorithmes de tolérance aux fautes nécessite d’échantillonner sur de larges espaces d’entrées. En effet, il faut considérer à la fois des entrées liées à l’activité fonctionnelle, et des entrées spécifiques liées aux fautes devant être tolérées. Le choix d’un échantillon peut être guidé par un critère de couverture basé sur un modèle fonctionnel ou structurel de l’algorithme. Mais si le test est destiné à compléter une preuve existante, il devrait idéalement se concentrer sur les faiblesses de cette preuve. De cette façon, les efforts investis dans le développement de la preuve ne sont pas complètement perdus, et le test est dirigé pour révéler des fautes dans l'algorithme qui n'auront pas été trouvées par la preuve. Il est évident que l'effort de test peut être réduit si une preuve formelle a montré que la propriété recherchée était satisfaite pour une partie de l'espace d'entrée. Même dans le cas d'une démonstration informelle, il peut être possible d'extraire des informations pertinentes pour le test. Par exemple, une preuve par cas peut suggérer des cas de test potentiellement significatifs vis-à-vis de la correction de l'algorithme. Identifier les parties les plus complexes, ou les moins convaincantes, de la preuve informelle peut suggérer des sous-espaces devant être échantillonnés plus finement que les autres. Toutefois, il reste à déterminer comment mettre ces idées en pratique et dans quelle mesure elles permettent d'améliorer l'efficacité des tests. Cet article présente des premiers résultats dans cette direction.

La partie 2 traite de travaux sur la collaboration entre différentes techniques de vérification, et plus spécifiquement le test et la preuve. La partie 3 introduit l'algorithme FT-RMS (Fault-Tolerant Rate Monotonic Scheduling) qui sera utilisé comme cas d'étude. La partie 4 se consacre à des résultats antérieurs établissant la présence de (au moins) deux fautes dans l'algorithme. La partie 5 contient une discussion détaillée de la démonstration informelle de l'algorithme, que nous avons restructurée sous la forme d'un arbre de preuve semi-formel en déduction naturelle. Dans la partie 6, des expériences de test sont présentées. Elles ont pour but de 1) évaluer la difficulté de révéler les deux fautes par un test en aveugle, 2) évaluer le gain en efficacité lorsque l'on prend en compte la couverture de cas fonctionnels issus de la démonstration, ainsi que la couverture structurelle de l'algorithme. La partie 7 conclut cet article et propose des directions pour continuer ces recherches.

2. Autres travaux dans la littérature

Des collaborations fructueuses entre diverses techniques de vérification ont déjà été développées, notamment pour la validation de protocoles. Un exemple est l'intégration dans l'assistant de preuve PVS de techniques de model-checking. Le principe de cette approche [16, 17] est d'utiliser une interaction étroite entre la preuve de théorèmes et la génération d'abstractions pour le model-checking. Chaque étape consiste à appliquer alternativement l'une des deux techniques en exploitant les informations obtenues lors des étapes précédentes. Ce processus itératif, lorsqu'il termine, donne un contre-exemple indiquant comment la propriété a été violée, ou une preuve que cette propriété est satisfaite. Un autre exemple est le bénéfice que le test de protocoles a pu tirer des avancées du model-checking. Des méthodes et des outils ont été développés pour automatiser la génération de jeux de test, soit en adaptant les algorithmes de graphes utilisés par les model-checkers [5], soit en utilisant la capacité des model-checkers à générer des contre-exemples [6].

Dans le cas du test et de la preuve, l'analyse de la littérature montre qu'il existe cependant peu de contributions proposant des solutions opératoires pour combiner ces techniques. L'idée a pourtant été exprimée dès les années 70: *"A judicious combination of direct program proving and empirical judgment can reduce size of a complete test."* [10].

Dans la communauté des méthodes formelles, une contribution originale est celle de Hayashi [11]. Il propose d'utiliser *l'isomorphisme de Curry-Howard* pour déverminer des preuves formelles, qu'elles soient partielles ou non. Cet isomorphisme établit une correspondance entre preuves et programmes, dans une logique constructive. Dans une telle logique, les preuves et les programmes sont équivalents selon les règles de cet isomorphisme: on peut donc extraire des programmes directement des preuves développées. La méthode proposée par Hayashi, appelée Animation de Preuve, revient à "exécuter" les preuves comme des programmes fonctionnels en utilisant des cas de test. Cette méthode a pour but de faciliter le développement de preuves formelles et un outil, ProofWorks, aide à trouver les bogues dans les preuves en cours de développement (preuves fausses) ou même dans les preuves terminées (erreurs de formalisation). Mais l'utilisation de l'isomorphisme de Curry-Howard restreint l'utilisation de cette méthode aux preuves en logiques constructives qui sont peu familières à la plupart des utilisateurs.

En ce qui concerne les travaux émanant de la communauté du test, Geller a considéré le problème dans [7], pour la vérification d'assertions de programmes. Il utilise une approche inductive, testant des valeurs spécifiques du domaine d'entrées et généralisant par la suite ces résultats à des *classes d'équivalence* en utilisant des preuves mathématiques. (Une classe d'équivalence est un ensemble d'entrées équivalentes dans leur capacité à stimuler le système testé.) Mais, dans toute approche inductive, le problème le plus délicat concerne la généralisation et non pas la vérification des cas de base: le test n'apporte donc pas d'aide significative à la preuve dans cette approche. Dans un esprit différent, Richardson et Clarke proposent dans [15] une méthode combinant un partionnement fonctionnel et structurel en classes d'équivalence. Le but est à la fois de guider la sélection de données de test et de vérifier formellement la cohérence des deux partitions obtenues. Plus proches de notre objectif, les travaux de Suri et Sinha [18, 19] concernent une approche pour guider une validation basée sur l'injection de fautes. Ils ont construit un modèle de l'algorithme FT-RMS dans le langage de spécification de PVS, et interrogé ce modèle pour étudier la validité de la preuve informelle du FT-RMS. Les résultats ont mis en évidence des problèmes dans certains des lemmes utilisés dans cette preuve informelle. Les interrogations du modèle ont alors été ciblées en considérant des ensembles de tâches spécifiques, permettant l'identification de deux classes de scénarios de test provoquant une défaillance du FT-RMS. Dans la partie 4, nous présenterons ces résultats de façon plus détaillée, et ce même cas d'étude servira de premier exemple pour nos travaux.

3. L'algorithme FT-RMS

Le FT-RMS est un algorithme d'ordonnancement de tâches ayant pour but de tolérer des fautes transitoires par réexécution des instances de tâches fautives. Il a été implémenté dans deux systèmes d'exploitation temps-réel [1, 4]: RT-Mach, un système académique, et DEOS (Digital Engineering Operating System), un système commercial pour les applications avioniques. La première définition du FT-RMS dans [8] contenait (au moins) deux fautes, l'une d'entre elles ayant été corrigée dans une version révisée de l'algorithme [9]. Comme nous l'avons mentionné précédemment, cet algorithme a déjà été étudié par d'autres travaux visant une collaboration entre techniques de vérification, travaux qui ont amené la seconde faute à être révélée. Notons cependant que les implémentations de l'algorithme dans RT-Mach et DEOS correspondent à d'autres variantes de l'algorithme et peuvent tout à fait être correctes. Dans cet article, nous nous focaliserons sur les deux versions incorrectes connues de l'algorithme, présentées dans [8, 9].

Dans le but de distinguer les fautes que l'algorithme est censé tolérer, et les fautes de conception affectant l'algorithme lui-même, nous utiliserons le terme de "défaut" pour ces dernières dans la suite de cet article.

Avant d'expliquer le fonctionnement du FT-RMS, nous introduisons brièvement les principes du RMS (Rate Monotonic Scheduling) sur lesquels il est basé.

3.1. L'algorithme RMS

Le RMS est un algorithme statique d'ordonnancement de tâches indépendantes à priorités fixées. Chaque tâche τ_i est caractérisée par son temps d'exécution C_i et sa période T_i. La $k^{ième}$ instance de la tâche τ_i, activée à l'instant $t_{0i} + (k-1)T_i$, doit être terminée avant que la requête suivante pour τ_i ne soit activée à l'instant $t_{0i} + kT_i$. L'assignation de priorités RM (Rate Monotonic) est liée au taux d'activation des tâches, c'est-à-dire que les tâches ayant des périodes plus courtes ont des priorités plus hautes. La tâche *ready* (dont on a reçu la requête) de plus haute priorité est toujours choisie pour s'exécuter, et l'exécution d'une tâche est préemptée lorsqu'une tâche de plus haute priorité devient *ready*. Les priorités étant attribuées de façon définitive, l'ordonnancement complet des tâches peut donc être déterminé hors-ligne avant leur exécution (ordonnancement statique).

Les schémas d'ordonnancement statique à priorités fixes ont été étudiés de façon approfondie. Nous rappelons ci-dessous un certain nombre de résultats théoriques.

Une condition suffisante pour qu'un ensemble de n tâches soit ordonnançable selon la politique RMS a été établie par Liu et Layland [13] :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \quad \leq \quad U_{LL} = n(2^{1/n} - 1) \tag{1}$$

U est le *taux d'utilisation*, c'est-à-dire la fraction du temps processeur passée à exécuter l'ensemble de tâches. La limite sur U est obtenue par une analyse de pire cas. Liu et Layland introduisent la notion d'*instant critique*, quand toutes les tâches deviennent *ready* en même temps: les tâches ont alors le temps de réponse le plus long. Ils considèrent ensuite les ensembles de tâches remplissant la condition d'*utilisation maximale* du processeur: l'ordonnancement est faisable pour ces ensembles de tâches mais toute augmentation dans le temps de calcul de l'une d'entre elles le rendrait infaisable. Ces ensembles de tâches sont de la forme:

$$\begin{cases} C_i = T_{i+1} - T_i & \forall\, i.\ 1 \leq i \leq n-1 \\ C_n = 2T_1 - T_n \end{cases} \tag{2}$$

les n tâches étant ordonnées selon leur priorité ($T_1 \leq T_2 \leq \ldots \leq T_n$). Il a été démontré que le minimum des taux d'utilisation sur tous les ensembles de tâches en *utilisation maximale* est: $n(2^{1/n} - 1)$. Ainsi tout ensemble de tâches ayant un taux d'utilisation inférieur à cette valeur est ordonnançable en RMS.

La condition (1) correspond à une borne pessimiste, une analyse exacte a été proposée dans des travaux postérieurs [12]. Elle consiste à calculer le pire temps de réponse R_i de chaque tâche τ_i, et de vérifier que $R_i \leq T_i$. Soit $\{\tau_1, \ldots, \tau_{i-1}\}$ l'ensemble des tâches ayant une priorité plus forte que τ_i. Alors R_i est la plus petite solution de l'équation:

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j \tag{3}$$

Une façon pratique de calculer ces R_i est de réécrire l'équation 3 comme une relation de récurrence et d'itérer le calcul jusqu'à soit obtenir un point fixe, soit dépasser T_i (c.f. [12]).

3.2. Deux versions de l'algorithme FT-RMS

L'algorithme FT-RMS étend le RMS pour incorporer de la tolérance aux fautes. Les fautes sont supposées être transitoires: une seule instance de tâche est affectée par chaque faute et le recouvrement est effectué par réexécution. La détection d'erreur a lieu à la fin de l'exécution de chaque tâche et son coût est considéré comme étant inclus dans le temps d'exécution C_i de chaque tâche. Des hypothèses supplémentaires concernent l'intervalle de temps entre deux fautes, mais nous les ignorerons car le comportement erroné de l'algorithme peut être révélé en se limitant à une unique faute transitoire.

Le principe de l'algorithme est d'utiliser le RMS lorsqu'il n'y a pas de faute, et, lorsqu'une faute est détectée, de réexécuter la tâche concernée selon un certain *schéma de recouvrement* RS (Recovery Scheme). Il doit y avoir assez de redondance temporelle pour que ni la tâche réexécutée, ni aucune autre tâche, ne manquent leurs échéances. Pour cela du *temps mort* (ou *slack*) est réservé pour la réexécution, et le schéma de recouvrement décrit comment ce slack doit être utilisé pour la réexécution dynamique. La réservation de slack est identique dans les deux versions de l'algorithme [8, 9], mais le schéma de recouvrement n'est pas le même.

La réservation de slack impose une condition sur l'ensemble de tâches considéré. Il doit être ordonnançable en considérant une politique RMS avec slack inséré: le schéma IBRMS (Inserted Backup RMS). Selon le IBRMS, la quantité de slack insérée sur un intervalle de temps donné doit être proportionnelle à la longueur de cet intervalle. Cette proportion constante U_B est égale au maximum des taux d'utilisation des tâches, c'est à dire: $U_B = max(C_i/T_i)$. Les auteurs affirment que cela peut être interprété comme l'ajout d'une tâche de recouvrement (backup) τ_B ayant pour période $T_B = gcd(T_i)$ et pour temps de calcul $C_B = U_B \times T_B$, mais ils utilisent, en fait, une politique d'insertion de slack différente[1]. Le principe de l'IBRMS consiste à insérer un slack entre deux débuts de périodes consécutifs. Ce slack inséré est proportionnel à l'espace temporel entre ces deux débuts de périodes: si ces deux instants sont distants de L unités de temps, sa longueur est $U_B \times L$. Un exemple de IBRMS est donné Figure 1 pour un ensemble de trois tâches. Nous rappelons que la priorité des tâches est assignée inversement à leur période, ainsi τ_1 a la plus haute priorité et τ_3 la plus basse. Dans la construction d'un ordonnancement IBRMS, l'instant $t = 0$ est généralement un *instant critique*.

1. La différence est significative lorsque l'on calcule les pires temps de réponse R_i des différentes tâches dans les deux cas.

τ_i	C_i	T_i	$U_i = C_i/T_i$
τ_1	16	60	0.276
τ_2	23	73	0.315
τ_3	2	116	0.017
			$U_B = 0.315$

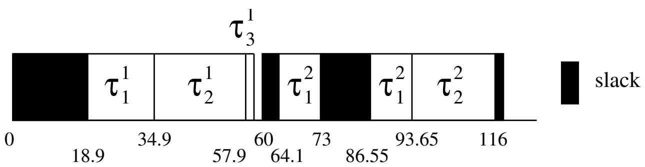

Figure 1. Ordonnancent IBRMS

Considérons maintenant qu'un ensemble de tâches satisfait la condition de faisabilité de l'IBRMS. Durant l'exécution effective des tâches, une politique RMS est suivie en absence de faute. Le slack peut donc être considéré comme repoussé dans le temps par l'exécution des tâches, ce phénomène correspond à la permutation temporelle entre le slack et l'exécution. Lorsqu'une erreur est détectée, ce slack est récupéré en suivant le schéma de recouvrement.

Le premier schéma de recouvrement [8] est le plus simple, la tâche en faute est réexécutée à la même priorité. Mais, comme cela est montré par la Figure 2.a, ce principe est imparfait: des tâches de priorité supérieure peuvent préempter la réexécution, empêchant la tâche fautive d'obtenir le slack réservé avant son échéance. Cela a conduit au nouveau schéma de recouvrement proposé dans [9].

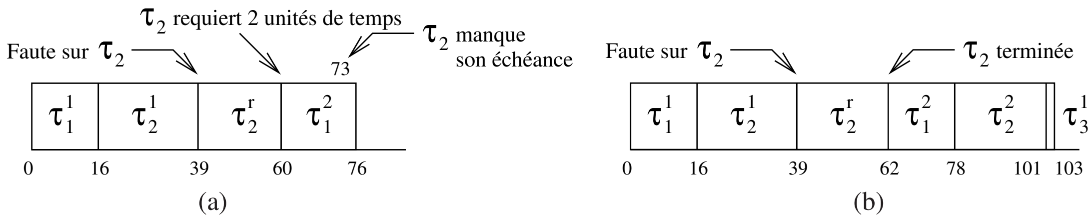

Figure 2. Ordonnancement FTRMS, version 1 et 2

Avec ce deuxième schéma de recouvrement, la réexécution de la tâche fautive τ_r ne peut plus être préemptée par des tâches ayant une échéance plus tardive que celle de τ_r, même si leur priorité est supérieure (c.f. Figure 2.b). Mais ce schéma s'est aussi avéré incorrect, comme nous le verrons dans la partie suivante.

Quel que soit le schéma de recouvrement, la preuve informelle de l'algorithme est basée sur le fait que l'ensemble de tâches respecte le critère de faisabilité de l'IBRMS. Les auteurs étudient par ailleurs une autre condition liée à une analyse de pire cas avec un ensemble de tâches en utilisation maximale pour déterminer une borne sur le taux d'utilisation comparable à celle du RMS. La borne résultante sur ce taux d'utilisation est:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq U_{FT-RMS} \qquad U_{FT-RMS} = U_{LL}(1 - U_B) = n(2^{1/n} - 1)(1 - U_B) \tag{4}$$

Cela correspond à une variante de l'analyse de Liu et Layland (c.f. Partie 3.1, équations 1 et 2), l'ensemble de tâches ayant été modifié pour tenir compte de la réservation de *slack*:

$$\begin{cases} C_i = (T_{i+1} - T_i)(1 - U_B) & \forall\, i.\ 1 \leq i \leq n-1 \\ C_n = (2T_1 - T_n)(1 - U_B) \end{cases} \tag{5}$$

Ce seuil U_{FTRMS} a été pris en compte par Suri et Sinha qui ont mis en évidence un défaut dans le deuxième *schéma de recouvrement*, c'est-à-dire dans la seconde version de l'algorithme.

4. Vérification du FT-RMS, résultats antérieurs

Comme exposé dans [2], les techniques actuelles de model-checking ne permettent pas une vérification complète des ordonnanceurs temps-réel. La preuve de théorème est une alternative valide, mais requiert des efforts importants et peut aussi échouer à fournir une preuve complète. En ce qui concerne le test, la taille du domaine d'entrées soulève le problème du degré de couverture qui peut être atteint en pratique. Le FT-RMS a donc été identifié par Suri et Sinha comme un exemple pertinent pour aborder l'utilisation de méthodes formelles pour guider le test. La première version de l'algorithme étant connue comme fausse, ils désiraient également étudier la capacité de leur approche à révéler le défaut de conception correspondant.

L'approche proposée [18] analyse le comportement de l'algorithme en utilisant des exécutions symboliques et des vérifications de propriétés, basées sur l'outil PVS. Le modèle PVS de la première version du FT-RMS [19] n'a pas pour but de fournir une description opérationnelle de l'algorithme lui-même, par exemple, la construction de l'échéancier n'est pas décrite dans le modèle PVS. Les auteurs proposent une interprétation de l'algorithme en termes de calcul de temps de réponse, en utilisant des formules telles que l'équation (3) de la partie 3.1. Les auteurs introduisent également un certain nombre de lemmes sur le modèle PVS, pour tester la valeur de vérité de certaines propriétés de l'algorithme. Ces conditions de vérité peuvent être directement dérivées de lemmes de la preuve informelle, ou des résultats d'interrogations antérieures du modèle PVS.

Une première analyse du comportement du modèle a amené Suri et Sinha à établir que la réservation de slack pour le FT-RMS était erronée. Il existe une différence entre la quantité de slack qui est censée être disponible selon l'IBRMS, et le slack réellement accessible pour la réexécution dynamique. À partir de ces résultats, les auteurs ont affiné leurs recherches en choisissant un cas spécifique ciblant le minimum de slack naturel. Ils ont choisi la condition d'utilisation maximale (c.f. Paragraphes 3.1 et 3.2). On peut toutefois noter qu'il n'a pas été prouvé que ce critère, démontré dans le cadre statique du RMS, soit réellement un pire cas pour le FT-RMS. Cela est fortement discutable pour la deuxième version du FT-RMS qui incorpore une règle d'ajustement dynamique des priorités. Néanmoins, ce critère s'est avéré efficace pour leurs expériences.

Les ensembles de tâches considérés sont définis comme ci-dessous, où Δ est un nombre positif aussi petit que désiré:

$$\begin{aligned} C_i &= T_{i+1} - T_i, \quad \forall\, i.\, 1 \leq i \leq n-1, \\ C_n &= (2T_1 - T_n)/2 + \Delta, \\ & \qquad avec \sum_i U_i \leq U_{FT-RMS} \end{aligned} \tag{6}$$

Dans cette définition, la réservation de slack est prise en compte par la limitation sur le taux d'utilisation global car elle est incluse dans le calcul de U_{FT-RMS} (c.f. équation 4). Si on la compare à la définition originale de Liu et Layland (équation 2), le temps de calcul de la dernière tâche a été divisé par deux. La valeur Δ ajoutée à C_n empêchera la réexécution de cette tâche dans la première version de l'algorithme [8], et cela bien que la condition de faisabilité de l'IBRMS soit respectée.

Suri et Sinha déterminent un ensemble de quatre tâches satisfaisant les équations ci-dessus, et l'incorporent au modèle PVS de l'algorithme. Cela leur permet de vérifier que la première version de l'algorithme échoue si la quatrième tâche doit être réexécutée. Mais ils observent aussi un échec de l'algorithme en cas de réexécution de la troisième tâche (c.f. Figure 3). Ce défaut, contrairement au premier, n'est pas corrigé dans la deuxième version du FT-RMS [9].

Cette approche a donc permis d'obtenir deux classes de scénarios de test, qui satisfont toutes deux les contraintes précédentes (équation 6). La première correspond à une faute sur la tâche de plus faible priorité τ_n et la deuxième sur τ_{n-1}. Ces classes de scénarios sont présentées comme des classes d'équivalence, c'est-à-dire que tous les membres d'une de ces classes ont la même capacité à exercer le comportement de l'algorithme. Elles peuvent provoquer, respectivement, la défaillance des versions 1 et 2.

τ_i	C_i	T_i	$U_i = C_i/T_i$
τ_1	0.5	4.0	0.125
τ_2	0.5	4.5	0.111
τ_3	1.0	5.0	0.2
τ_4	1.01	6.0	0.1683
			$\Sigma_U = 0.604$

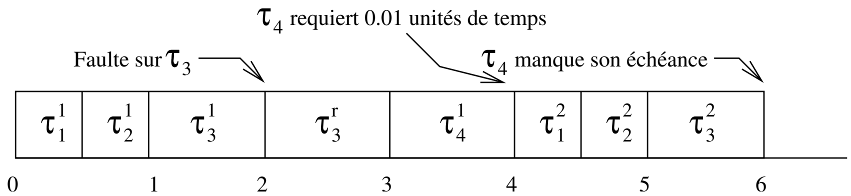

Figure 3. Échec de la version 2 du FT-RMS

Nous avons malheureusement découvert que la deuxième classe d'équivalence n'est pas homogène dans sa capacité à révéler le défaut de la version 2. En effet, un ensemble de tâches vérifiant les équations 6 sera ordonnançable sous les deux versions de l'algorithme avec réexécution de τ_{n-1} si $C_{n-1} \leq C_n - 2\Delta$, comme cela est illustré par la Figure 4. Dans cet exemple, $U_B = max(U_i) = 0.23$, et le seuil FT-RMS est $U_{FT-RMS} = 0.58$. L'ensemble de tâches vérifie toutes les contraintes de la deuxième classe de scénarios (réexécution de τ_{n-1}) mais aucune défaillance n'est constatée. Les résultats de Suri et Sinha se sont avérés biaisés par le choix de valeurs spécifiques pour instancier leurs ensembles de tâches.

τ_i	C_i	T_i	$U_i = C_i/T_i$
τ_1	0.5	4.0	0.125
τ_2	0.5	4.5	0.111
τ_3	0.5	5.0	0.1
τ_4	1.26	5.5	0.23
			$\Sigma_U = 0.566$

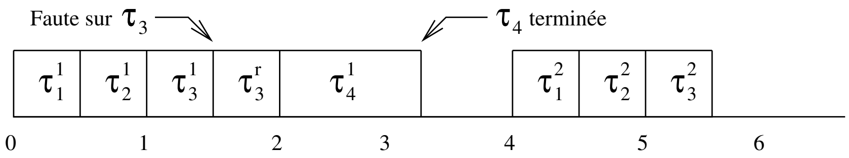

Figure 4. Ensemble de tâches en *utilisation maximale* sans échec

5. Analyse de la preuve informelle

Notre proposition est d'utiliser la preuve de l'algorithme pour obtenir plus d'informations destinées à la conception du test. Le modèle PVS développé dans [19] n'est pas, dans sa forme actuelle, utilisable pour étudier la preuve du FT-RMS. Nous aurions besoin d'un modèle opérationnel (par exemple une machine à états), de lemmes établissant une relation explicite entre ce modèle et le calcul de temps de réponse des tâches, et de théories mathématiques pour raisonner sur ces calculs (comme le calcul de points fixes). Ces points sont par exemple couverts par la spécification et la preuve en PVS du PCP (Priority Ceiling Protocol) présentées dans [2]. Il est dans notre intention d'étudier si le développement du PCP peut être adapté au FT-RMS. Mais nous avons, dans un premier temps, étudié la preuve informelle, correspondant à la seconde version de l'algorithme, exposée dans [9].

Nous utilisons la *déduction naturelle* de Gentzen sous forme de séquent pour restructurer la preuve informelle du FT-RMS, et analyser l'arbre de preuve semi-formel résultant. Nous utiliserons un certain nombre de notations spécifiques dans nos preuves. Pour signifier la preuve de branchement due à une décomposition en sous-cas, par exemple, lors de l'application d'une règle $\vee_e$, nous utiliserons la notation complétude?. L'axiome HYP sera utilisé lorsqu'un lemme pourra être identifié aux hypothèses. Les branches non prouvées des arbres de preuve seront aussi indiquées par des notations particulières: $trivial$ sera utilisé pour des buts pouvant aisément être prouvés (des preuve évidentes de complétude par exemple), $assume$ dénotera une partie des preuves qui a été soit oubliée, soit considérée valide par les auteurs sans la démontrer, mais que nous considérons comme manquante.

Nous allons maintenant présenter une analyse détaillée de la preuve informelle du FT-RMS. Notre but est à la fois d'identifier les faiblesses de cette preuve et d'extraire des cas fonctionnels pertinents pour le test. La preuve informelle de cet algorithme est basée sur trois lemmes intermédiaires:

$[S_1]$: Pour chaque tâche τ_i un slack d'au moins C_i doit être présent entre les instants kT_i et $(k+1)T_i$.

$[S_2]$: Si une faute est détectée durant l'exécution de τ_r, alors le schéma de recouvrement doit permettre à τ_r de se réexécuter pendant une durée de C_r avant son échéance.

$[S_3]$: Lorsqu'une tâche se réexécute, elle ne doit amener aucune autre tâche à manquer son échéance.

La Figure 5 montre comment ces lemmes sont utilisés dans la structure de la preuve.

$$\dfrac{\dfrac{\dfrac{\text{cf. Fig 6}}{\Gamma \vdash [S_1]} \quad \dfrac{\dfrac{\dfrac{\text{cf. Fig 7}}{\Gamma, [S_1] \vdash [S_2]} \quad \dfrac{\text{cf. Fig 8}}{\Gamma, [S_1] \vdash [S_3]}}{\Gamma, [S_1] \vdash [S_2] \wedge [S_3]}}{\Gamma \vdash [S_1] \Rightarrow [S_2] \wedge [S_3]}}{\Gamma \vdash [S_2] \wedge [S_3]} \qquad \dfrac{assume}{\Gamma \vdash [S_2] \wedge [S_3] \Rightarrow \text{ordonnancement valide}}}{\Gamma \vdash \text{ordonnancement valide}} \Rightarrow_e$$

Figure 5. Arbre de la preuve informelle du FT-RMS

La liste Γ des hypothèses intègre plusieurs notions: le classement des tâches selon leur priorité (($T_1 \leq T_2 \leq \ldots \leq T_n$), la condition de faisabilité de l'IBRMS, l'occurrence d'une faute unique, et l'algorithme lui-même (permutation du slack avec les exécutions et schéma de recouvrement). Sous ces hypothèses, aucune tâche ne doit manquer son échéance, ce qui revient à dire que l'ordonnancement déterminé est faisable. Dans la preuve, la validité de la première règle $\Rightarrow_e$, introduisant les sous-buts $[S_2]$ et $[S_3]$, est considérée comme acquise par les auteurs. Mais l'implication n'est pas si évidente à démontrer. $[S_2]$ exprime une propriété sur l'instance de la tâche τ_r devant être réexécutée, mais rien n'est dit en ce qui concerne ses autres instances. Comme la réexécution peut (dans la deuxième version) repousser dans le temps des tâches de plus haute priorité, et comme ces tâches peuvent elles-même préempter les instances suivantes de τ_r, cela mériterait plus d'attention. Plus généralement, le fait que les conditions $[S_2]$ et $[S_3]$ soient exprimées en termes de tâches et non d'instances de tâche peut être considéré comme un manque de précision. Les preuves de $[S_1]$, $[S_2]$ et $[S_3]$ sont présentées Figures 6 à 8. $[S_2]$ et $[S_3]$ utilisent une preuve par cas, ceux-ci sont détaillés dans la Table 1.

La preuve de $[S_1]$ (Fig. 6) est basée sur le critère de faisabilité de l'IBRMS. En IBRMS, une quantité de slack $U_B\, T_i$ est insérée sur toute période T_i, et comme $U_B \geq C_i \,/\, T_i$, alors la quantité de slack sur T_i est au moins C_i. Cette preuve est évidente si l'on considère que $[S_1]$ est une propriété liée à l'IBRMS. Mais les commentaires accompagnant la démonstration et l'utilisation future de $[S_1]$ nous amènent à penser que les auteurs en ont une interprétation plus large: $[S_1]$ étant amenée à caractériser la quantité de slack présente dans un échéancier FT-RMS. En fait, le slack est considéré comme étant repoussé dans le temps par l'exécution des tâches. Mais la relation entre le slack inséré dans l'IBRMS et la distribution de temps morts en FT-RMS n'est jamais clairement posée. Nous considérons donc que l'utilisation des lemmes (1) et (2) pour prouver $[S_1]$ n'est pas suffisante pour prouver sa validité dans le cadre du FT-RMS. Il est à noter que cette confusion entre le *slack* en IBRMS et en FT-RMS avait aussi été mise en évidence par Suri et Sinha.

$$\dfrac{\dfrac{\dfrac{\text{HYP}}{\Gamma \vdash \forall i.\ slack(T_i) = U_B T_i} \quad \dfrac{\text{HYP}}{\Gamma \vdash\ \forall i.\ U_B \geq C_i \,/\, T_i}}{\Gamma \vdash (1)\, \forall i.\ slack(T_i) = U_B T_i\ \wedge (2)\, \forall i.\ U_B \geq C_i \,/\, T_i} \quad \dfrac{assume}{\Gamma \vdash (1) \wedge (2) \Rightarrow [S_1]}}{\Gamma \vdash [S_1]} \Rightarrow_e$$

Figure 6. Arbre de la preuve informelle de la condition $[S_1]$

La preuve de S_2 (Fig. 7) est, elle, construite en introduisant une nouvelle condition suffisante pour S_2, notée *slack dans* D_r, et qui signifie que le slack réservé ne peut pas être repoussé au-delà de la deadline D_r de l'instance fautive de τ_r. Justifier que cette condition est suffisante nécessite d'utiliser la définition de S_1 comme applicable au FT-RMS, ce que nous avons critiqué ci-dessus. La preuve de la condition elle-même est décomposée en trois cas. Chacun d'entre eux considère une tâche τ_h de priorité supérieure à τ_r, et pouvant potentiellement la préempter (c.f. Table 1). Dans le premier cas, τ_h a une deadline D_h inférieure ou égale à D_r, et préemptera donc l'exécution ou la réexécution de τ_r. Les deux autres cas correspondent à une tâche τ_h ayant une deadline strictement postérieure à celle de τ_r, et dont la requête est reçue respectivement après et avant E_r.

Nous avons arrêté notre arbre de preuve au niveau des différents cas dégagés par les auteurs. Cela est du aux limites de l'approche informelle. Nous avons extrait, ci-dessous, le Cas 1 (τ_r est préemptée et $D_h \leq D_r$) de la preuve de $[S_2]$ comme exemple:

> Selon le schéma de recouvrement (RS), τ_h va préempter τ_r et le slack réservé sera permuté avec l'exécution de τ_h. Ainsi le slack repoussé se trouve dans D_h. Et puisque $D_h \leq D_r$ on peut conclure que tout le slack repoussé par τ_h se trouve dans D_r.

Une analyse fine de cette démonstration nécessiterait des définitions formelles claires de notions comme le *schéma de recouvrement*, le *slack réservé*, le *slack repoussé* et *"se trouve dans"*. Certaines remarques peuvent toutefois être faites sur la décomposition choisie. Chacun des trois cas n'implique que deux tâches: τ_r et τ_h, ce qui limite fortement la généralité de telles démonstrations. Par exemple, des préemptions multiples par plusieurs tâches de priorités supérieures ne sont pas prises en compte.

$$\dfrac{\dfrac{Cas1 \quad Cas2 \quad Cas3 \quad \dfrac{assume}{\text{complétude?}}}{\Gamma, [S_1] \vdash \text{slack dans} D_r} \qquad \dfrac{assume}{\Gamma, [S_1] \vdash \text{slack dans} D_r \Rightarrow [S_2]}}{\Gamma, [S_1] \vdash [S_2]} \Rightarrow_e$$

Figure 7. Arbre de la preuve informelle de la condition $[S_2]$

La condition $[S_3]$ est probablement la plus difficile à prouver (Fig. 8), et nous savons qu'elle est fausse car Suri et Sinha ont pu mettre en évidence un contre-exemple. $[S_3]$ peut se reformuler de la façon suivante: $\forall\ i.\ (\ i \neq r \Rightarrow$ τ_i *se termine avant sa deadline*$)$. Après avoir passé $i \neq r$ en hypothèse, la preuve est divisée en deux sous-buts selon les priorités comparées de τ_i et τ_r. La première preuve se consacre aux tâches de plus faible priorité. Elle est conduite en décomposant l'espace d'état en trois cas (4 à 6), selon le placement de la période de τ_i par rapport à celle de τ_r. La deuxième partie de la preuve traite des tâches de priorité supérieure. Les auteurs indiquent que le seul cas pertinent est celui où une tâche de priorité supérieure est repoussée par la réexécution de τ_r (Cas 7). Mais cette fois encore, l'analyse de chaque cas n'est faite que pour deux tâches. Elle est centrée sur les interactions directes entre τ_r et une autre tâche, en termes d'utilisation du slack et de repoussement de tâche. En ignorant des cas plus complexes, les auteurs ne considèrent pas des chaînes d'interactions indirectes. Par exemple, une tâche de priorité inférieure repoussée par la réexécution peut ensuite être préemptée par une tâche de priorité supérieure, elle sera donc encore plus retardée. C'est précisément le cas mis en évidence par Suri et Sinha.

$$\dfrac{\dfrac{\dfrac{Cas4 \quad Cas5 \quad Cas6 \quad \dfrac{assume}{\text{complétude?}}}{\Gamma, [S_1], i > r \vdash \tau_i \text{ se termine avant sa deadline}} \quad \dfrac{\dfrac{Cas7}{\tau_i \text{ retardée}} \quad \dfrac{trivial}{\neg\tau_i \text{ retardée}} \quad \dfrac{trivial}{\text{complétude?}}}{\Gamma, [S_1], i < r \vdash \tau_i \text{ se termine avant sa deadline}} \quad \dfrac{trivial}{\text{complétude?}}}{\Gamma, [S_1], i \neq r \vdash \tau_i \text{ se termine avant sa deadline}} \vee_e}{\Gamma, [S_1] \vdash [S_3]}$$

Figure 8. Arbre de la preuve informelle de la condition $[S_3]$

Table 1. Cas extraits de la démonstration informelle

τ_r est la tâche en réexécution, τ_h est une tâche de *priorité supérieure*, τ_l est une tâche de *priorité inférieure*

R_i et D_i sont respectivement l'instant d'arrivée de la requête (*ready time*) et la deadline de τ_i

E_r est l'instant correspondant à la fin de l'exécution de τ_r

Cas de la preuve	**Conditions Sous-Buts**	**Priorité τ_i Vs τ_r**	**Description des Cas**	
1	$[S_2]$	supérieure	τ_h préempte τ_r	$D_h \leq D_r$ & $R_r \leq R_h <$ fin de réexécution
2	$[S_2]$	supérieure	τ_h est retardée par la réexécution de τ_r	$D_h > D_r$ & $E_r \leq R_h <$ fin de réexécution
3	$[S_2]$	supérieure	τ_h préempte l'exécution de τ_r	$D_h > D_r$ & $R_r \leq R_h < E_r$
4	$[S_3]$	inférieure	les périodes de τ_r et τ_l se chevauchent et τ_r est ready avant τ_l	$R_r < R_l$ & $D_r < D_l$
5	$[S_3]$	inférieure	la période de τ_r est entièrement contenue dans celle de τ_l	$R_r \geq R_l$ & $D_r \leq D_l$
6	$[S_3]$	inférieure	les périodes de τ_r et τ_l se chevauchent et τ_r est ready après τ_l	$R_r > R_l$ & $D_r > D_l$
7	$[S_3]$	supérieure	la réexécution de τ_r préempte τ_h	$D_h > D_r$ & $E_r \leq R_h <$ fin de réexécution

En conclusion, l'ensemble des feuilles de l'arbre de preuve partiel (cas de la preuve) ne donne pas de forte assurance quant à la validité de l'algorithme. Mais nous n'avons pas pu mettre en évidence des problèmes triviaux de logique, tout comme nous n'avons pas pu déterminer de partie spécifiquement faible dans la preuve. Les problèmes que nous avons relevés sont plus liés à un niveau "méta" de raisonnement qui affecte l'ensemble de la structure de la preuve. L'un de ces problèmes est la difficulté à établir une distinction claire entre l'IBRMS (de nature entièrement statique) et le FT-RMS (ayant une nature dynamique liée à l'utilisation du *slack*). L'ensemble des cas utilisés dans les preuves se limitent également à étudier deux tâches avec une seule instance de chacune d'entre elles. Ce dernier point a amené les auteurs à ne pas considérer les schémas d'interactions multiples qui sont pourtant la principale difficulté de l'analyse d'ordonnançabilité.

L'identification de ces faiblesses ne nous fournit toutefois aucune information constructive utilisable pour aider à la conception de jeux de tests sur le FT-RMS. Il nous est cependant possible d'extraire des informations de la preuve informelle, sous la forme des différents cas fonctionnels (Table 1) considérés par les auteurs comme représentatifs du comportement de l'algorithme. En se basant sur l'analyse précédente, on peut s'attendre à ce qu'un critère de test basé sur ces cas fonctionnels soit imparfait. Nous pouvons cependant étudier dans quelle mesure les cas fonctionnels peuvent être reliés au comportement incorrect de l'algorithme, et si leur imperfection peut être compensée en activant chacun de ces cas plusieurs fois avec des entrées de test différents [20].

6. Expériences de test

Dans le but de déterminer la difficulté à révéler les défauts de conception du FT-RMS, nous utiliserons, dans un premier temps, une approche de test en aveugle avec des ensembles de tâches générés aléatoirement. Nous étudierons aussi si la couverture structurelle de l'algorithme peut être utile pour améliorer les résultats du test (Partie 6.1). Par la suite, nous étudierons les différents cas fonctionnels extraits de la preuve informelle étudiée précédemment et leur connexion éventuelle avec les entrées de test qui activent le défaut de l'algorithme (Partie 6.2).

6.1. Approche de test en aveugle

Pour pouvoir tester le FT-RMS, nous avons développé un prototype de cet algorithme en langage C pour chacune de ses deux versions [8, 9] décrites dans la partie 3.

Chaque entrée de test est un ensemble de tâches incluant:

- le nombre n de tâches
- la période T_i et le temps d'exécution C_i de chaque tâche
- la description de la faute (tâche fautive τ_r et numéro de l'instance devant être réexécutée)

Nous utilisons un temps discret, de sorte que tous nos paramètres sont des valeurs entières. L'instant temporel $t = 0$ est considéré comme étant un *instant critique*, c'est-à-dire que les premières requêtes de toutes les tâches arrivent à cet instant (toutes les tâches sont *ready* en 0). En réponse à une entrée, les deux prototypes du FT-RMS calculent l'échéancier et reportent tout problème dû à un dépassement d'échéance.

La condition de faisabilité de l'IBRMS est incorporée dans la génération aléatoire des ensembles de tâches, afin de respecter les hypothèses de la preuve informelle. Chaque tâche τ_i est telle que $5 \leq T_i \leq 130$, $1 \leq C_i < T_i$ et $C_i/T_i < 0.4$. La construction d'un ensemble de tâches est réalisée en ajoutant de nouvelles tâches tant que l'ensemble reste ordonnançable en IBRMS. Chaque ensemble doit contenir au moins deux tâches. Comme le PPCM des $T_i s$ peut être très grand, nous avons dû considérer une fenêtre temporelle limitée: le prototype du FT-RMS calcule un ordonnancement partiel de $t = 0$ à $t = D_r + T_{n-1} + T_n$, la deadline D_r de l'instance à réexécuter devant être inférieure à $t = 2^{20}$.

Nous avons fait des expériences avec trois variantes de la distribution aléatoire, la différence se situant au niveau du placement de la faute. La distribution `i0` choisit toujours la première instance de la tâche de C_i maximum. La distribution `i1` conserve la faute sur la première instance mais choisit la tâche aléatoirement. Et finalement la distribution `i2` choisit la tâche aussi bien que l'instance de façon aléatoire. On pourra noter que les distributions `i0` et `i1` imposent que la faute soit proche de l'instant critique, alors que `i2` n'impose aucune limitation quant à la configuration des différentes tâches au moment de la faute. La Table 2 présente les résultats obtenus avec les ensembles de tâches générés selon ces trois distributions.

Table 2. Résultats des trois ensembles de tâches aléatoires

Distribution des fautes	Nombre d'entrées	Temps machine	Défaillances Version 1	Défaillances Version 2	Taux de défaillance Version 2
`i0`	10^2	1s	46	5	5%
`i1`	10^3	5s	141	35	3.5%
`i2`	10^4	1h30	93	4	0.04%

Après analyse, les différentes défaillances observées n'ont pas révélé de nouveau défaut dans l'algorithme par rapport à ceux déjà découverts par les travaux précédents. Comme nous pouvons le constater, la défaillance de la deuxième version de l'algorithme est bien plus difficile à provoquer que celle de la première version. La distribution `i2`, la plus générale, s'avère particulièrement inefficace, puisque le taux de défaillance observées de la version 2 sous ce profil est de 0.04%. On peut améliorer ce résultat en se concentrant sur l'instant critique, et en forçant la tâche fautive à être celle ayant le temps d'exécution le plus long. Mais nous n'avons aucune garantie que les distributions de fautes `i0` et `i1` suffisent pour révéler tout défaut potentiel de l'algorithme, car elles excluent de nombreuses configurations de fautes.

C'est un fait connu que le test aléatoire en aveugle n'est généralement pas efficace pour activer toutes les fonctionnalités d'un programme, ni l'ensemble de sa structure. Nous avons voulu savoir si les mauvais résultats de `i2` pouvaient être expliqués par une faible couverture structurelle de l'algorithme. Nous nous sommes basés sur l'utilitaire Unix `tcov` pour évaluer la couverture des branches du prototype an langage C, réalisé pour la deuxième version de l'algorithme. Analyser la couverture des chemins était impossible dû à leur nombre très important lié à la nature temporelle du FT-RMS. Les résultats de cette étude sont présentés Table 3. Les trois jeux de test activent 100% des branches de la structure de l'algorithme. Dans le cas de `i2`, la branche la moins couverte est activée 2242 fois. La couverture structurelle n'est donc pas un facteur pouvant expliquer la faible efficacité observée.

Table 3. Couverture structurelle du prototype C de la version 2 du FT-RMS

Ensemble de test	nombre d'activations de la branche la moins couverte
`i0`	100
`i1`	347
`i2`	2242

Pour conclure, ces premiers résultats expérimentaux confirment que le défaut de conception dans la deuxième version du FT-RMS est difficile à révéler en utilisant une approche de test aveugle. Nous avons aussi montré que l'utilisation d'une approche classique de test structurel, le test des branches, était également inefficace, même s'il est requis que chaque branche soit activée un grand nombre de fois. Le test doit, de façon évidente, incorporer plus d'informations fonctionnelles sur le FT-RMS. Concentrer notre étude sur l'instant critique (ensembles de tâches `i0` et `i1`) était une tentative dans ce sens, mais nous n'avons pas de justification formelle que l'algorithme ne peut pas être mis en défaillance sous d'autres configurations. Nous allons maintenant étudier si les cas extraits de la preuve informelle peuvent améliorer ces résultats.

6.2. Couverture des cas fonctionnels extraits de la preuve informelle

Pour pouvoir obtenir des mesures de couverture des cas fonctionnels, nous avons du instrumenter le prototype en langage C. Notre étude ayant porté sur la preuve informelle de la deuxième version de l'algorithme, nous n'avons instrumenté que cette version (qui est celle où le défaut de conception est la plus difficile à révéler).

Les différents cas déterminés sont regroupés dans la Table 4. Il y a de légères différences avec les cas fonctionnels originaux extraits de la preuve (Table 1, fin du paragraphe 5). Le Cas 1 a été séparé en deux sous-cas pour étudier de façon séparée la préemption de l'exécution (a) ou de la réexécution (b) de τ_r par une tâche de priorité supérieure. Certains cas se sont également avérés équivalents, ils ont donc été regroupés comme un cas unique durant les expériences. Les Cas 2 et 7, ainsi que 4 et 6 ont ainsi été regroupés, sous les appellations 2(7) et 4(6). Les Cas 4 et 6 avaient été traités séparément dans la preuve informelle car celle-ci ne considère qu'une seule instance pour chaque tâche, alors qu'ils représentent en fait une même tâche dont on considère deux instances successives. Nous avons finalement ajouté deux cas supplémentaires: *inconnue supérieure* et *inconnue inférieure*. Ils nous permettent de détecter des tâches qui n'appartiennent à aucun des cas de la démonstration informelle et de les classer selon leur priorité vis-à-vis de la tâche en réexécution.

Table 4. Cas expérimentaux

Cas expérimentaux	Cas de la preuve
1a	1
1b	1
2(7)	2 et 7
3	3
inconnue supérieure	ni 1(a ou b), ni 2(7), ni 3
4(6)	4 et 6
5	5
inconnue inférieure	ni 4(6), ni 5

Les résultats de couverture fonctionnelle pour les trois jeux de test précédents sont présentés Table 5. Les mesures correspondent au nombre d'entrées de chaque jeu de test activant la configuration. On pourra noter que la distribution `i2` couvre deux cas qui ne sont jamais activés par `i0` ou `i1`: *inconnue supérieure* et le Cas 4(6). Ces deux cas correspondent à des configurations qui ne peuvent pas être rencontrées avec ces deux distributions car elles sont limitées dans le placement de la faute. Le cas *inconnue supérieure* correspond à une tâche de priorité supérieure n'interagissant pas avec τ_r, or, à l'instant critique, toutes les tâches de priorité supérieure préemptent l'exécution de τ_r. Le Cas 4(6) requiert un chevauchement des périodes, ce qui est impossible à l'instant critique car toutes les tâches de priorité supérieure sont entièrement contenues dans T_r et sont donc dans le Cas 5.

Table 5. Nombre d'entrées de test activant les cas fonctionnels

Cas	Jeu de test `i0`	Jeu de test `i1`	Jeu de test `i2`
1a	98	762	5284
1b	38	209	2563
2 (7)	79	251	1825
3	5	40	863
4 (6)	0	0	6089
5	38	761	4631
inconnue supérieure	0	0	3540
inconnue inférieure	0	0	0

Ces résultats n'offrent pas plus d'explication que ceux de la couverture structurelle quant aux faibles résultats du jeu de test `i2`: 0.04% d'entrées révélant le défaut. Pour ce jeu de test, nous avons une couverture correcte de tous les cas, le moins activé (le Cas3) étant couvert par 863 entrées. Dans le but d'étudier plus finement l'éventuelle corrélation entre ces cas fonctionnels et le comportement erroné de l'algorithme, nous avons poursuivi une analyse plus poussée des entrées provoquant une défaillance de l'algorithme. La Table 6 présente les résultats de cette analyse pour les distributions `i0` et `i2` (nous n'avons pas présenté les résultats de `i1` pour des raisons de place). Cette fois encore, il n'est pas possible d'établir une corrélation claire entre le déclenchement d'une défaillance et un quelconque cas (ou combinaison de cas). La combinaison des cas 1a et 5 est très souvent observée (elle est systématiquement provoquée par une faute lors de l'instant critique), mais elle n'est ni suffisante ni nécessaire (l'entrée 1733 du jeu de test `i2` le démontre).

Table 6. Couverture des cas pour les entrées révélant le défaut

Jeu de test `i0`	Cas couverts
entrée 20	1a, 1b, 5
entrée 60	1a, 1b, 5
entrée 71	1a, 2(7), 3, 5
entrée 73	1a, 1b, 5
entrée 96	1a, 1b, 2(7), 5

(a)

Jeu de test `i2`	Cas couverts
entrée 1083	1a, 4(6)
entrée 1733	4(6), *inconnue supérieure*
entrée 3762	1a, 5
entrée 9095	1a, 5

(b)

La preuve informelle du FT-RMS ne nous a donc apporté aucune information pertinente pour améliorer le test de l'algorithme. Le (ou les) défaut qui n'a pas été trouvé par la preuve informelle peut aussi échapper au test, et ce même si un grand nombre d'activations est demandé pour chaque cas fonctionnel issu de la preuve. Cela est notamment dû au fait que les schémas d'interactions indirects ont été complètement ignorés dans la preuve, et les défaillances constatées impliquent de tels schémas. Pour illustrer cela, nous présentons ci-dessous (Figure 9) un exemple de défaillance constatée lors de nos expériences. L'instance 88 de τ_3 manque son échéance, et provoque une défaillance de l'algorithme, car elle se trouve prémptée par l'instance 93 de τ_1. Ce n'est donc pas une interaction directe avec τ_2, ou sa réexécution, qui a causé la défaillance dans ce cas, mais la préemption par une instance d'une tâche de priorité supérieure qui n'avait en rien été affectée par la réexécution.

τ_i	C_i	T_i	$U_i = C_i/T_i$
τ_1	21	107	0.196
τ_2	29	112	0.259
τ_3	29	112	0.259
			U_B = 0.259

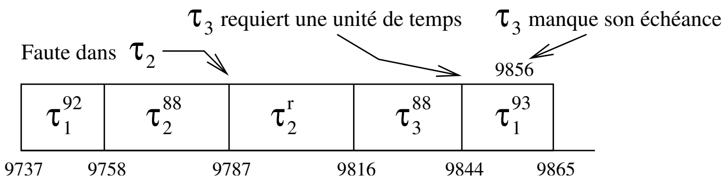

Figure 9. Jeu de test `i2`, entrée 9095, exemple de défaillance

7. Conclusion et perspectives

Nous avons utilisé la déduction naturelle de Gentzen sous forme de séquents pour restructurer la preuve informelle du FT-RMS. L'arbre de preuve semi-formel ainsi obtenu donne une représentation compacte de la structure de la preuve qui nous a permis de l'analyser pas à pas. Nous avons pu trouver plusieurs problèmes dans cette preuve. Certains sont liés à la sémantique imprécise de notions telles que le slack, dont la validité prouvée en IBRMS est supposée en FT-RMS. D'autres sont liés à l'absence de plusieurs preuves de validité des décompositions en cas, et des branchements, utilisés dans la preuve informelle. Le fait qu'elle se limite, par exemple, à étudier deux instances dans les cas de fonctionnement n'a pas été démontré comme suffisant, et nous avons pu voir que les défaillances de l'algorithme impliquent des schémas plus complexes. Ces défauts sont plus liés à un niveau "méta" de la preuve qu'à de simples mauvaises applications de règles logiques, même si certaines preuves de validité des branchements auraient pu les révéler. L'exemple du FT-RMS confirme une fois de plus la difficulté de révéler des défauts subtils par une analyse manuelle.

Nous avons également conduit des expériences sur des prototypes de l'algorithme FT-RMS réalisés en langage C. Les résultats obtenus tendent à démontrer qu'une preuve informelle trop imparfaite peut s'avérer inefficace pour guider le test, car les cas fonctionnels extraits seront trop éloignés des fautes de conception. Mais un tel résultat ne peut être généralisé à partir d'un seul exemple, le test structurel s'étant lui aussi avéré inefficace dans le cas du FT-RMS.

Les différents problèmes rencontrés lors de l'étude de la preuve informelle vont dans le sens d'une formalisation de celle-ci. Différentes notions telles que le slack pourront être clairement définies, ce qui préviendra tout glissement dans leur sémantique au cours de la preuve. Mais un problème prévisible, qui ne s'était pas posé pour la preuve informelle analysée, consiste en la difficulté d'établir un lien entre un lemme formel non prouvé et le comportement de l'algorithme en termes de configuration des tâches. De plus, les branchements dans l'arbre de preuve formel peuvent être aussi inadéquats que dans le cas de la preuve informelle. Si leur preuve ne termine pas, rien ne nous assure de leur validité, ni de leur pertinence pour le test. Nous souhaitons néanmoins poursuivre cette approche dans le cadre du FT-RMS en construisant une preuve formelle de cet algorithme, et nous nous baserons pour cela sur les travaux de B. Dutertre sur l'algorithme PCP [3]. Nous élargirons également notre étude à d'autres algorithmes de tolérance aux fautes, en partant cette fois d'algorithmes prouvés corrects, et en analysant l'impact sur les arbres de preuve, de fautes de conception insérées dans ces algorithmes.

Bibliographie

[1] Libin Dong, Rami Melhem, Daniel Mossé, Sunondo Ghosh, Walt Heimerdinger, and Aaron Larson. Implementation of a transient-fault-tolerance scheme on DEOS. In *Real-Time Technology and Application Symposium (RTAS)*, 1999.

[2] Bruno Dutertre. Formal analysis of the priority ceiling protocol. In *IEEE Real-Time Systems Symposium (RTSS'00)*, pages 151–160, Orlando, FL, November 2000.

[3] Bruno Dutertre and Victoria Stavridou. Formal analysis for realtime scheduling. In *19th AIAA/IEEE Digital Avionics Systems Conference*, Philadelphia, PA, October 2000.

[4] E. Egan, D. Kutz, D. Mikulin, R. Melhem, and D. Mossé. Fault-tolerant RT-Mach (FTRT -Mach) and an application to real-time train control, 1999.

[5] J. C. Fernandez, C. Jard, T. Jéron, L. Nedelka, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *8th Int. Conf. on Computer-Aided Verification (CAV)*, number 1102 in LNCS, pages 348–359. Springer-Verlag, August 1996.

[6] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Joint 7th Eur. Software Engineering Conf. and 7th ACM SIGSOFT Intern. Symp. on Foundations of Software Eng. (ESEC/FSE99)*, number 1687 in LNCS, pages 146–162. Springer-Verlag, September 1998.

[7] Matthew Geller. Test data as an aid in proving program correctness. *Communications of the ACM*, 21(5):368–375, 1978.

[8] Sunondo Ghosh, Rami Melhem, and Daniel Mossé. Fault-tolerant rate monotonic scheduling. In *Sixth IFIP Conference on Dependable Computing for Critical Applications (DCCA-97)*, March 1997.

[9] Sunondo Ghosh, Rami Melhem, Daniel Mossé, and Joydeep Sen Sarma. Fault-tolerant rate monotonic scheduling. *Real-Time Systems*, 15(2):149–181, 1998.

[10] John B. Goodenough and Susan L. Gerhart. Towards a theory of test data selection. *IEEE Transaction on Software Engineering*, SE-1(2):156–173, 1975.

[11] Susumu Hayashi. Towards animation of proofs - testing proofs by exemples. *special issue of TCS for TTP Tokyo meeting '97*, 1997.

[12] M. Joseph. *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall, 1996. London.

[13] C. L. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, jan 1973.

[14] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[15] D. J. Richardson and L. A. Clarke. Partition analysis: a method combining testing and verification. *IEEE Transactions on Software Engineering*, 11(12):1477–1490, 1985.

[16] John Rushby. Integrated formal verification: Using model checking with automated abstraction, invariant generation and theorem proving. In *Theoretical and Practical Aspects of SPIN Model Checking : 5th and 6th International SPIN Workshops*, pages 1–11. Springer-Verlag, 1999. LNCS 1680.

[17] Vlad Rusu and Eli Singerman. On proving safety properties by integrating static analysis, theorem proving and abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, pages 178–192. Springer-Verlag, 1999. LNCS 1579.

[18] Purnendu Sinha and Neeraj Suri. Identification of test cases using a formal approach. In *29th Symposium on Fault-Tolerant Computing (FTCS)*, pages 314–321, 1999.

[19] Purnendu Sinha and Neeraj Suri. On the use of formal techniques for analysing dependable real-time protocols. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 126–135, 1999.

[20] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. *Software statistical testing*, pages 253–272. Springer Verlag, 1995. B. Randell, J-C. Laprie, H.Kopetz and B. Littlewood (Eds).