Diagnostic à la demande sur un système à événements discrets

Pierre-Jean Meyer

14 mars - 9 septembre 2011

Résumé

Le travail réalisé par Sampath sur le diagnostic des systèmes modélisés par des machines à états finis, bien que très performant, est beaucoup trop volumineux pour être utilisé dans une application embarquée. Pour remédier à ces problèmes de mémoire, il est nécessaire de partir d'un modèle moins développé, mais plus lent. C'est pour cette raison que, à l'aide de la plateforme DIADES, nous avons approché ce problème grâce à un algorithme de diagnostic dit « à la demande », qui doit avoir la capacité d'être interrompu à tout moment pour fournir immédiatement un résultat intermédiaire correct. A l'aide de recherches d'arbre simples, nous avons développé plusieurs versions de cet algorithme de diagnostic, dont la plus complète est une recherche bidirectionnelle associée à une fonction de raffinage en arrière plan. Nous avons ensuite mis en place des critères de performance pour trouver la version la plus efficace et connaître la plage de fonctionnement correct de l'algorithme. La recherche d'algorithmes anytimes pour le diagnostic basé sur ce type de modélisation étant assez récente, il reste bien sûr de nombreuses possibilités pour améliorer cet algorithme ou trouver des stratégies de recherche plus performantes.

Table des matières

Remerciements **2**

Introduction **3**

1 Présentation du sujet **4**
1.1 Diagnostic 4
1.2 Modélisation du diagnostiqueur 5
1.2.1 Diagnostiqueur de Sampath 5
1.2.2 Retour à la réalité 7
1.3 Algorithme à la demande 8
1.3.1 Justification du besoin 8
1.3.2 Qualité et approximations 8
1.3.3 Objectifs 10

2 Contributions **11**
2.1 Mise en place 11
2.1.1 Algorithme de base 11
2.1.2 Rendre anytime l'algorithme 12
2.2 Algorithmes 14
2.2.1 Recherche d'arbre 14
2.2.2 Algorithme complet 15
2.3 Critères de qualité 20
2.3.1 Les critères 20
2.3.2 Les mesures 21

3 Résultats **22**
3.1 Plan des expérimentations 22
3.2 Observations générales 22
3.3 Etape par étape 23
3.4 Synthèse des résultats 25

Conclusion **27**

Annexe : Résultats des expérimentations **28**

Bibliographie **36**

Remerciements

Je tiens à remercier tout particulièrement mes superviseurs Yannick Pencolé et Elodie Chanthery pour avoir accepté ma candidature pour ce stage, mais surtout pour avoir su me guider avec justesse tout au long de ce projet de six mois.

Je souhaiterais également remercier l'ensemble du personnel du groupe de recherche DISCO pour leur accueil chaleureux, et pour avoir eu la patience de répondre à mes interrogations concernant leur travail.

Introduction

Dans le cadre de la formation d'ingénieur à l'ENSEEIHT de Toulouse, il nous est demandé tous les ans de réaliser un stage en entreprise ou en laboratoire de recherche. Ce rapport relate du travail réalisé au cours de mon Projet de Fin d'Etudes marquant l'aboutissement de cette formation. Ce stage s'est étendu du 14 mars au 9 septembre 2011, et s'est déroulé dans le laboratoire de recherche du LAAS-CNRS. En effet, pour cet unique stage long, je souhaitais découvrir plus en détail le travail et l'organisation dans un laboratoire de recherche, dans l'éventualité de réaliser ensuite une thèse.

Parmi les enseignements suivis à l'ENSEEIHT, le domaine m'intéressant le plus est celui des systèmes à événements discrets. C'est pour cette raison que j'ai cherché auprès de Yannick Pencolé et Elodie Chanthery du groupe de recherche DISCO (DIagnostic, Supervision et COnduite), pour trouver le stage intitulé *Diagnostic à la demande sur un système à événements discrets*. L'objectif principal de ce projet est de développer un algorithme de diagnostic qui puisse être interrompu à tout moment et tout de même renvoyer un résultat « correct », pour compenser le fait que les algorithmes de diagnostic embarqués sont souvent trop lents.

Dans ce rapport, je vais donc commencer par présenter ce sujet, en détaillant les concepts de diagnostic, diagnostiqueur et algorithme à la demande. Par la suite, je décrirai les programmes réalisés au cours de ce stage pour mettre en place plusieurs versions d'un algorithme de diagnostic à la demande, et pour mesurer la qualité de leurs résultats. Le dernier chapitre présentera les séries d'expérimentations réalisées pour connaître la plage de fonctionnement correct de notre algorithme. Ces expérimentations aboutiront sur une comparaison des différentes versions de l'algorithme pour trouver lesquelles sont les plus intéressantes.

Chapitre 1

Présentation du sujet

1.1 Diagnostic

Ce projet de fin d'études a été réalisé au sein du LAAS (Laboratoire d'Analyse et d'Architecture des Systèmes), un laboratoire de recherche à Toulouse, rattaché au CNRS (Centre National de la Recherche Scientifique). Faisant partie du pôle MOCOSY (Modélisation, Optimisation et COnduite des SYstèmes) qui s'intéresse aux systèmes dynamiques incertains, le groupe de recherche DISCO (DIagnostic, Supervision et COnduite), dans lequel j'ai réalisé mon projet, oriente ses travaux principalement sur deux axes de recherche. Le premier axe traite de la surveillance et de la supervision par des méthodes d'apprentissage. Le second concerne le diagnostic et les décisions qui en découlent. C'est dans ce second axe que s'est déroulé mon stage.

De nos jours, les systèmes complexes tels que les satellites, les robots ou les drones ont des besoins d'autonomie de plus en plus importants. Pour leur donner cette autonomie, il va falloir embarquer dans ces systèmes des moyens de raisonnement pour leur permettre d'achever leur mission quelles qu'en soient les conditions. Plus particulièrement, on s'intéresse ici au cas où une défaillance venait à apparaître. Ainsi, il va falloir donner au système en question des moyens de s'auto-diagnostiquer, c'est à dire de détecter, d'isoler et d'identifier les fautes. Une fois que ce diagnostic a été réalisé et que les fautes sont connues, viendra l'étape de réaction dans laquelle le planificateur adaptera ses actions à la situation décrite par le diagnostic. La figure 1.1 représente le fonctionnement global et les interactions entre les modules de contrôle et de diagnostic du système.

Le diagnostiqueur peut prendre diverses formes puisque son fonctionnement dépend du type de modélisation utilisé pour le système. Il existe trois principales catégories de modélisation selon si l'on considère le comportement du système comme continu, discret ou encore hybride. Toutefois, même parmi chacune de ces catégories, il existe plusieurs types de modélisations qui donneront des diagnostiqueurs de formes complètement différentes. Pour ma part, mon stage s'est concentré sur le diagnostic d'un système à événements discrets, dont les composants sont modélisés par des automates. C'est pourquoi mon travail a été encadré par Elodie Chanthery, maître de conférences à l'INSA (Institut National des Sciences Appliquées) de Toulouse, et Yannick Pencolé, chargé de recherche, dont les travaux de recherches sont relatifs aux systèmes à événements

discrets, respectivement dans les domaines de la planification et du diagnostic.

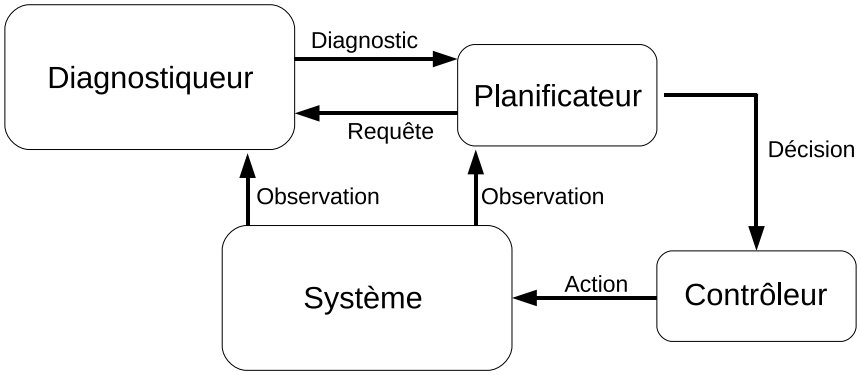

FIG. 1.1 – Architecture de contrôle et de diagnostic d'un système

1.2 Modélisation du diagnostiqueur

1.2.1 Diagnostiqueur de Sampath

Au cours de ce projet, nous avons utilisé une modélisation du système et du diagnostiqueur très proche de celle décrite par Meera Sampath dans son article [Sampath96] dont nous allons résumer ici les éléments importants. Dans cet article, chaque composant du système est représenté par un automate. On considère qu'il est parfaitement modélisé, c'est à dire qu'il va comprendre à la fois le fonctionnement normal du composant, mais aussi son fonctionnement induit par l'occurrence de fautes. La synchronisation de l'ensemble de ces automates va générer une machine à états finis correspondant au modèle du système global, définie comme suit.

Définition 1 (Modèle global) *Le modèle global, G, d'un système à événements discrets, est une machine à états finis $G = (X, \Sigma, T, x_0)$ avec :*
- *X, l'ensemble des états issus de la synchronisation des automates des composants ;*
- *Σ, l'alphabet des événements, constitué de deux ensembles distincts Σ_o (les événements observables) et Σ_{io} (les événements inobservables) ;*
- *$T \subseteq X \times \Sigma \times X$, l'ensemble des transitions ;*
- *$x_0 \in X$, l'état initial du modèle global.*

A partir de ce modèle du système global, Sampath va créer une nouvelle machine à états finis, correspondant au diagnostiqueur.

Définition 2 (Diagnostiqueur de Sampath) *Le diagnostiqueur G_d d'un modèle global G, est une machine à états finis $G_d = (X_d, \Sigma_d, T_d, q_0)$ avec :*
- *X_d, l'ensemble des états de croyance (voir définition 3) ;*
- *$\Sigma_d = \Sigma_o$, l'alphabet est réduit aux événements observables ;*
- *$T_d \subseteq X_d \times \Sigma_d \times X_d$, l'ensemble des transitions ;*
- *$q_0 = (x_0, \{\}) \in X_d$, l'état de croyance initial du diagnostiqueur.*

Définition 3 (Etat de croyance) *Un état de croyance $q_d \in X_d$ est un état du diagnostiqueur tel que $q_d = \{(x_1, l_1), \ldots, (x_n, l_n)\}$, avec :*

- *$x_i \in X$, un état dans lequel le modèle global pourrait se trouver ;*
- *$\Sigma_f \subseteq \Sigma_{io}$, l'ensemble des événements associés à une faute ;*
- *$l_i \subseteq \Sigma_f$, une liste (potentiellement vide) des fautes rencontrées pour arriver à l'état x_i.*

Ainsi, lorsque l'on a généré le diagnostiqueur correspondant au modèle d'un système global, l'occurrence d'un événement observable déclenchera une transition du diagnostiqueur entre deux de ses états de croyance. Et le nouvel état de croyance dans lequel il se trouvera correspondra à l'ensemble des états — chacun associé à une liste de fautes rencontrées sur son parcours — dans lesquels le modèle pourrait se trouver après l'occurrence de l'événement observable. En d'autres termes, l'analyse de l'état de croyance courant correspond au diagnostic de la séquence des observations réalisées à partir de l'état initial.

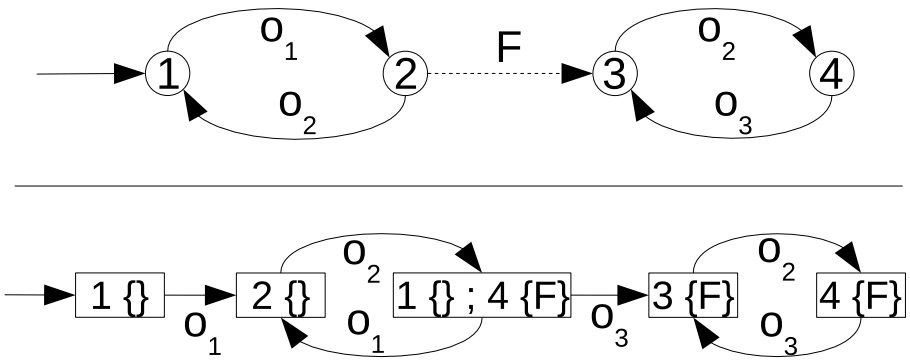

FIG. 1.2 – L'automate du modèle global (en haut), et celui du diagnostiqueur (en bas) pour un exemple basique

Pour illustrer ce fonctionnement, la figure 1.2 représente un exemple simple. Dans cet exemple, on a pris un modèle global contenant 4 états, 4 événements (3 observables et un de faute) et 5 transitions. Aussi, on considère que l'état initial de cet automate est l'état 1. On a donc maintenant tous les éléments pour construire la machine à états finis du diagnostiqueur (dans la partie basse de la figure 1.2), dont l'état initial sera l'état initial du modèle global associé à un ensemble de fautes vide. On peut voir que chaque état du diagnostiqueur est un état de croyance correspondant à une liste d'états du modèle global, chacun associé à une liste d'événements de fautes. Aussi, on voit que seuls les événements observables sont encore associés à des transitions, alors que les événements de fautes sont passés dans les états de croyance — et les événements inobservables non fautifs, s'il y en avait, n'apparaîtraient plus.

Prenons maintenant une possible séquence d'observations adaptée à cet exemple : $[o_1, o_2, o_3]$. En partant de l'état initial du diagnostiqueur, et en exécutant successivement ces trois observations, on peut suivre l'évolution du diagnostic comme le montrent les résultats présentés dans le tableau 1.1.

L'analyse de ces résultats nous montre qu'après l'observation o_1, l'état de croyance du diagnostiqueur ne contient qu'un élément qui n'est associé à aucune

Observation	Etat de croyance	Diagnostic
Etat initial	$1\{\}$	Sain
o_1	$2\{\}$	Sain
o_2	$1\{\}; 4\{F\}$	Ambigu
o_3	$3\{F\}$	Fautif

TAB. 1.1 – Résultats de la séquence $[o_1, o_2, o_3]$ pour l'exemple de la figure 1.2

faute. On peut donc en conclure sans ambiguïté que le système est dans un état de fonctionnement normal. Lorsque la seconde observation apparaît, on voit qu'il y a deux chemins possibles dans le modèle, dont l'un où l'événement de faute a été déclenché. Le diagnostic est donc ambigu, puisqu'à ce stade rien ne nous permet de savoir dans laquelle des deux possibilités le modèle global se trouve vraiment. Il faut donc attendre l'arrivée de l'observation suivante, o_3, pour lever cette ambiguïté en voyant que cette observation ne peut pas être exécutée à partir de l'état 1. Ainsi, cette séquence d'observations nous a permis de réaliser que l'événement inobservable de faute F est apparu au cours de cette séquence (en analysant l'automate de cet exemple très simple, on peut même déterminer que la faute est apparue entre les observations o_1 et o_2).

1.2.2 Retour à la réalité

Une telle modélisation complète du diagnostiqueur est belle et très efficace puisqu'à chaque nouvelle observation, il suffit de suivre la transition correspondante sur l'automate du diagnostiqueur. Ainsi, les calculs pour obtenir un nouveau diagnostic seront toujours presque instantanés. Toutefois, cette modélisation est également utopique puisque l'espace mémoire requis pour l'utilisation directe de ce diagnostiqueur est trop important. En effet, le nombre d'états du diagnostiqueur est au pire exponentiel en le nombre d'états du modèle global (déterminisation du modèle global), alors que le nombre d'états du modèle global est au pire exponentiel en le nombre de composants (synchronisation des automates). La taille du diagnostiqueur est donc doublement exponentielle en le nombre de composants du système. Or, comme cela a été dit précédemment, le diagnostic est principalement utilisé pour améliorer l'autonomie d'un système, c'est à dire dans des systèmes embarqués qui ont généralement de très fortes contraintes en terme d'espace mémoire.

Il existe tout un spectre d'algorithmes de diagnostic allant de l'algorithme sans pré-compilation à base de composants au diagnostiqueur de Sampath [Schumann07]. Afin de pouvoir embarquer un algorithme de diagnostic, il va donc falloir travailler à partir d'un modèle moins développé que celui du diagnostiqueur de Sampath. Bien sûr, les temps de calcul du diagnostic seront de plus en plus long si l'on diminue la quantité de pré-compilation réalisée.

1.3 Algorithme à la demande [1]

1.3.1 Justification du besoin

Le choix précédent, consistant à n'utiliser l'algorithme de diagnostic qu'à partir d'un modèle pas ou peu pré-compilé, nous permet donc d'abaisser les besoins en espace mémoire à une valeur plus convenable pour une application embarquée. Toutefois, comme nous le soulignions dans la partie précédente, cette réduction de la pré-compilation va augmenter les temps de calcul, qui auront donc de fortes chances de dépasser les temps entre deux observations consécutives. Il nous faudrait donc trouver une solution pour pouvoir diagnostiquer une observation, sans dépasser le temps disponible et sans avoir recours directement à la machine à états finis du diagnostiqueur complet.

Une autre difficulté de la mise en place d'un diagnostiqueur, vient de la nécessité de donner l'état de santé du système dès que le planificateur en fait la demande. Il faut donc trouver une solution pour permettre au diagnostiqueur de renvoyer un résultat au planificateur même lorsqu'il n'a pas eu le temps de terminer les calculs menant au résultat final. Bien sûr, il faudra aussi que le résultat renvoyé soit toujours « correct » aux vues des exigences qui seront décrites plus tard.

Pour ces deux raisons, nous avons décidé de nous orienter vers l'utilisation d'un algorithme de type anytime [1]. Les algorithmes anytimes nous donnent en effet la possibilité de faire des concessions sur la qualité du résultat dans le but de diminuer le temps de calcul [Zilberstein96]. Bien sûr, ces algorithmes sont aussi toujours construits pour que, plus il y aura de temps de calcul disponible, meilleure sera la qualité du résultat. Dans certains cas, ce type d'algorithme peut être dit « à contrat » lorsque le temps alloué à l'algorithme est planifié à l'avance. Pour pouvoir répondre aux requêtes du planificateur à tout moment, nous choisirons plutôt un algorithme anytime de type « interruptible » pour justement être capable de l'interrompre sans savoir à l'avance quand en viendra la demande.

Un exemple extrêmement basique d'un algorithme anytime pour du diagnostic serait de réutiliser un algorithme classique non-interruptible, et à chaque fois que le planificateur demande l'état de santé du système : si l'algorithme est terminé, on renvoie son résultat ; sinon, on répond simplement que tout est possible (pour être sûr de ne pas éliminer des fautes qui sont vraiment arrivées). Evidemment, nous n'allons pas réellement utiliser un algorithme aussi naïf. Notre but dans la suite sera de développer des algorithmes qui puissent donner des résultats intermédiaires le plus tôt possible, pour avoir un diagnostic de qualité la meilleure possible avant que l'algorithme ne soit interrompu.

1.3.2 Qualité et approximations

Dans son article, Zilberstein introduit aussi la notion étendue de « qualité » évoquée précédemment. En effet, le principe binaire de correction d'un résultat traditionnel (correct ou incorrect) n'est plus suffisant pour caractériser

[1] Pour plus de clarté, dans la suite, le caractère « à la demande » des algorithmes sera plutôt qualifié d'« anytime ».

la qualité d'un résultat d'algorithme anytime. La possibilité d'interrompre l'algorithme pour n'obtenir qu'un résultat partiel ouvre un nouveau spectre de caractérisations de la qualité. Le choix du type de mesure de qualité liée à un algorithme anytime dépend à la fois de la manière dont le résultat exact est approché par l'algorithme, mais aussi de l'utilisation que l'on fera ensuite de ce résultat. Parmi les exemples les plus utilisés, on peut mesurer la qualité du résultat selon :

- la *certitude*, en calculant par exemple la probabilité que le résultat soit correct ;
- la *précision* du résultat par rapport à la solution optimale, qui est généralement utilisée dans les algorithmes de planification [Likhachev05] ;
- la *spécificité*, qui mesure le niveau de détail du résultat.

Dans le cadre d'un algorithme de diagnostic, c'est ce dernier type de qualité que nous allons choisir. En effet, le plus important lorsque l'on retourne un résultat au planificateur, c'est que ce résultat soit correct quelque soit la situation : on préfère être dans le flou que de générer un diagnostic erroné. Ainsi, dans les algorithmes, il faudra s'arranger pour que le résultat englobe toujours la solution finale, et alors l'augmentation de la qualité consistera à confirmer ou infirmer les possibilités présentes dans le résultat courant.

Ces notions de qualité du diagnostic peuvent être étudiées plus en profondeur en s'intéressant à l'utilisation d'approximations pour faciliter les décisions et l'accès aux données utiles contenues dans le résultat [VanHarmelen95]. En effet, même si le résultat final contient les informations que l'on recherche, il n'est pas toujours nécessaire de calculer ce résultat complètement pour pouvoir prendre la bonne décision. L'un des exemples donnés dans cet article concerne un centre téléphonique d'appel d'urgence. Il est vital que l'opérateur ne cherche pas à déterminer le diagnostic complet de son interlocuteur (ce qui prendrait beaucoup de temps au téléphone), au contraire il n'a besoin de trouver qu'un seul symptôme nécessitant l'envoi d'une ambulance avant de prendre sa décision. Le cas contraire peut se présenter lorsqu'un médecin a la possibilité d'administrer un traitement dangereux comme solution à un diagnostic possible : la prise de décision doit alors d'abord considérer les autres diagnostics possibles dont le traitement serait moins dangereux (et si aucun ne résout le problème, on reviendra au traitement dangereux).

Il existe beaucoup d'autres types d'approximations possibles pour faciliter l'obtention et l'utilisation des résultats d'un algorithme de diagnostic, qu'il soit classique (comme dans le dernier exemple) ou anytime. Si par exemple on souhaite réaliser un diagnostic sur un système dont le modèle est sous la forme d'une architecture de composants et sous-composants, il est possible de partir d'un diagnostic très flou concernant une faute sur le composant principal, puis d'améliorer la qualité du diagnostic en étudiant ses sous-composants [Cobb99]. Il est aussi possible de travailler dans l'autre sens en cherchant tous les composants fautifs, puis remonter vers les sur-composants jusqu'à ce que toutes les fautes soient contenues dans un même composant, qui sera celui que l'on choisira de remplacer. Une autre possibilité serait de chercher le diagnostic impliquant le moins de composants possibles [Verberne00]. Ainsi, sur un même type de modélisation, on a déjà pu trouver trois types d'approximations pour extraire un diagnostic, ce qui montre bien que la manière de réaliser les recherches est

grandement influencée par l'objectif du diagnostic : isolation, réparation rapide, réparation économique, ...

Un dernier exemple d'approximation pour le diagnostic peut être utilisé lorsque le modèle du système est sous la forme d'une liste de règles de comportement [Mouaddib95]. On pourrait alors obtenir un diagnostic avec une abstraction maximale lorsque l'on décide d'omettre les règles les moins importantes. Si il reste du temps disponible après cela, on pourra alors relancer l'algorithme avec davantage de règles pour raffiner le diagnostic.

On voit donc que l'utilisation d'approximations dans la détermination du diagnostic peut nous être imposée, par la nécessité de réduire les temps de calcul ou pour nous permettre de rendre anytime l'algorithme. Mais indépendamment de cela, certaines approximations peuvent aussi être souhaitables selon l'objectif de l'algorithme.

1.3.3 Objectifs

Ce projet traite donc du diagnostic appliqué à un système à événements discrets, dont les composants seraient modélisés par des machines à états finis. L'objectif principal est d'améliorer la performance générale des algorithmes de diagnostic utilisant peu de pré-compilation, grâce à des techniques algorithmiques dites à la demande (anytime). Le caractère anytime de l'algorithme devra se traduire par sa capacité à être interrompu à tout instant pour délivrer un résultat correct.

Comme il n'existe à l'heure actuelle aucun algorithme de diagnostic anytime pour ce type de modélisation dans l'état de l'art, nous avons décidé de limiter cette première étude à la mise en place d'un algorithme anytime sur un modèle global (issu de la synchronisation des composants). Cette limitation nous permet de nous focaliser sur les aspects anytime uniquement et non sur les aspects liés notamment à la décentralisation des algorithmes [Pencolé05] nécessaire à la mise en place d'un algorithme sur les modèles de composants. Il est à noter néanmoins que les techniques développées au cours de ce stage sont exploitables sur des algorithmes travaillant directement sur les modèles des composants, car au final ces algorithmes reconstruisent à la volée des sous-parties du modèle global. Enfin, toujours dans le même esprit de simplification, les algorithmes développés n'utilisent pas de techniques symboliques (utilisation de diagramme de décision binaire par exemple) [Schumann07], qui améliorent l'efficacité générale des recherches dans les graphes mais n'apporte rien de plus conceptuellement dans le développement de techniques anytime.

Par la suite, nous tenterons d'améliorer cet algorithme afin de le rendre plus efficace aux vues des besoins. Le but de cet algorithme anytime n'est pas d'être plus rapide que l'algorithme de base, mais plutôt d'obtenir des résultats intermédiaires le plus tôt possible pour les avoir avant l'interruption.

Enfin, afin de mesurer l'efficacité de nos algorithmes, il serait bon de développer un critère de qualité, d'une part pour pouvoir comparer les différentes versions de l'algorithme entre elles, mais aussi pour vérifier que notre algorithme anytime est effectivement plus performant que l'algorithme de base.

Chapitre 2

Contributions

2.1 Mise en place

2.1.1 Algorithme de base

L'intégralité de ce projet a été réalisée sur la plateforme DIADES (DIAgnosis of Discrete Event Systems) en majeure partie développée par Yannick Pencolé. DIADES est un ensemble de bibliothèques C++ relatives au diagnostic de systèmes à événements discrets, et de programmes utilisant ces bibliothèques. Parmi ces bibliothèques, on peut trouver *Graph* qui crée une structure de graphe utilisée dans la plupart des autres bibliothèques comme *Petri* pour les réseaux de Petri, ou encore *Automata* pour les machines à états finis. C'est cette dernière bibliothèque que l'on a utilisée le plus au cours de ce projet, puisqu'elle permet par exemple d'implémenter les automates des composants du système, mais aussi d'en réaliser la synchronisation.

Au sein de DIADES, Yannick Pencolé a également créé un programme qui réalise un diagnostic classique (non-anytime) à partir de l'automate représentant le système global, en utilisant une stratégie de recherche en largeur d'abord [Russel]. Aussi, il utilise des états de croyance comme on les a décrits dans la définition 3, pour que les résultats soient sous la même forme que ceux du diagnostiqueur de Sampath.

Nous allons expliquer le fonctionnement de cet algorithme à l'aide de l'exemple de la figure 2.1. Si l'on considère que le précédent diagnostic [1] est simplement composé de 1 {} et que l'observation en cours de diagnostic est o_1, l'algorithme se comportera de la manière décrite dans la suite et dans le tableau 2.1. Pour chaque élément du diagnostic précédent (ici on n'en a qu'un, 1 {}), on va ajouter à la fin d'une file l'ensemble des successeurs de cet élément. Ces successeurs seront associés à un ensemble de fautes correspondant à l'union des fautes de l'état d'origine, avec l'ensemble des fautes rencontrée pour atteindre l'état courant. Cette file que l'on utilise ici correspond à l'ensemble des états à visiter : on visite le premier élément (que l'on retire de la file), et on ajoute les nouveaux états à visiter (les successeurs de cet élément) à la fin de la file. Ce n'est pas illustré dans l'exemple de la figure 2.1, mais lorsqu'un état a déjà été visité (il faut que l'état et l'ensemble des fautes soient les mêmes),

[1] Ici et souvent dans la suite, le terme « diagnostic » désignera en réalité un état de croyance.

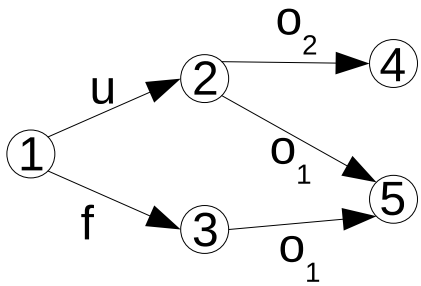

FIG. 2.1 – Automate partiel pour illustrer l'algorithme de base (o_i est observable, u est inobservable, f est une faute)

Etat visité	File	Etat de croyance
1 {}	2 {} ; 3 {f}	
2 {}	3 {f} ; 4 {} ; 5 {}	
3 {f}	4 {} ; 5 {} ; 5 {f}	
4 {}	5 {} ; 5 {f}	
5 {}	5 {f}	5 {}
5 {f}		5 {} ; 5 {f}

TAB. 2.1 – Résultats de l'algorithme pour la figure 2.1 (diagnostic précédent : 1 {} ; observation : o_1)

alors on ne l'ajoutera pas un seconde fois dans la file.

Lorsque l'état que l'on visite a été atteint avec l'événement observable en cours de diagnostic (ici o_1), alors on peut ajouter cet état au nouveau diagnostic (en lui associant son ensemble de fautes) et on arrête la recherche pour cette branche. Au contraire, si un état est atteint avec un événement observable autre que celui diagnostiqué (c'est le cas de l'état 4), on supprime cet état de la file et on passe au prochain. En résumé, les solutions du diagnostic d'une observation sont tous les états accessibles depuis le précédent diagnostic avec cette observation précédée d'une séquence (potentiellement vide) d'événements inobservables (fautes incluses).

La dernière colonne du tableau 2.1 nous montre clairement que cet algorithme n'est pas anytime. En effet, si on l'interrompait en cours de réalisation, le résultat relevé serait un sous-ensemble du diagnostic final (5 {} ; 5 {f}), alors que pour un algorithme anytime, on cherche justement à se mettre dans le cas contraire : avoir un résultat intermédiaire qui contient le diagnostic final.

2.1.2 Rendre anytime l'algorithme

Comme on le disait dans la partie 1.3.2 et à nouveau dans le paragraphe précédent, pour que l'on puisse considérer un résultat de diagnostic comme « correct », il est indispensable que le diagnostic final soit contenu dans ce résultat. Ainsi, à l'inverse de l'algorithme précédent, pour que notre algorithme soit anytime, il faut qu'il commence par remplir son état de croyance courant de telle sorte que le diagnostic final y soit contenu. Par la suite, au cours de

l'algorithme, il faudra désormais chercher à retirer des éléments de cet état de croyance, plutôt que d'en ajouter comme dans l'algorithme précédent.

Pour obtenir un état de croyance initial contenant le résultat final, on pourrait d'abord penser à utiliser l'ensemble des états de l'automate du modèle, chacun associé à toutes les combinaisons de fautes possibles. Ainsi, cet état de croyance initial recouvrerait l'intégralité des éléments de solution possibles pour ce modèle. Il y a toutefois des informations supplémentaires que l'on pourrait utiliser pour diminuer la taille de ce diagnostic initial. En effet, on sait que l'algorithme de diagnostic est à chaque fois utilisé pour une observation donnée, c'est à dire que quand on cherche à diagnostiquer cette observation à partir d'un précédent diagnostic, on veut trouver toutes les paires état-liste de fautes que l'on peut atteindre depuis le précédent diagnostic avec un séquence d'événements inobservables qui se termine par l'observation en question. Toutes les solutions d'une étape de diagnostic ont donc un point commun : la dernière transition utilisée pour atteindre chacune des solutions est associée à l'événement observable que l'on est en train de diagnostiquer. On peut donc réduire le diagnostic initial à l'ensemble des états cibles d'une transition associée à l'observation. On continue d'associer toutes les combinaisons de fautes possibles à chacun de ces états, puisqu'au début de l'algorithme on n'a pas d'information utile permettant de réduire le nombre de ces possibilités.

En pratique, on n'utilise pas la liste des combinaisons de fautes pour représenter l'ensemble des possibilités initiales, pour ne pas risquer de confondre ces possibilités initiales avec les vraies solutions trouvées plus tard. Pour cette raison, on va plutôt associer uniquement l'événement fictif *non-raffiné* à chacun des états de l'état de croyance initial. Alors, on obtient deux types d'actions pouvant améliorer la qualité du diagnostic :

- **La confirmation d'une solution**, quand on trouve un chemin entre un état du diagnostic précédent et un des états de l'état de croyance de départ pour cette étape (états cibles d'une transition associée à l'observation diagnostiquée) ;
- **Le raffinage d'un état**, lorsque l'on sait que pour cet état, aucune nouvelle solution ne peut être confirmée. Cela peut se réaliser en deux occasions. Si toutes les combinaisons de fautes (il en existe $2^{nbFautes}$) sont présentes dans les solutions confirmées liées à cet état, aucune autre solution ne pourra être trouvée pour cet état. La seconde possibilité, utilisable seulement dans une recherche en arrière (cf. partie 2.2.2), est lorsque tous les chemins menant à cet état ont été explorés : toutes les solutions liées à cet état qui doivent être confirmées le sont déjà. Lorsqu'un état est raffiné, la paire contenant cet état et l'événement fictif *non-raffiné* est retirée de l'état de croyance.

Si l'on reprend l'exemple de la figure 2.1 en utilisant cette fois un algorithme anytime, on obtiendra les résultats présentés dans le tableau 2.2. On considère que dans cet exemple, on réutilise une stratégie de recherche en largeur d'abord. La colonne des états visités et celle de la file ne changent donc pas par rapport à l'exemple non-anytime. Ainsi, seul l'état de croyance est modifié comme expliqué dans les paragraphes précédents : l'observation diagnostiquée est o_1 et l'état 5 est le seul état cible de transitions associées à o_1, on commence donc par mettre la paire $(5, \{non - raffiné\})$ dans l'état de croyance.

Aucune solution n'est trouvée lors de la visite des quatre premiers états, donc rien n'est changé dans l'état de croyance. A l'avant dernière étape, la solu-

Etat visité	File	Etat de croyance
1 $\{\}$	2 $\{\}$; 3 $\{f\}$	5 $\{non - raffiné\}$
2 $\{\}$	3 $\{f\}$; 4 $\{\}$; 5 $\{\}$	5 $\{non - raffiné\}$
3 $\{f\}$	4 $\{\}$; 5 $\{\}$; 5 $\{f\}$	5 $\{non - raffiné\}$
4 $\{\}$	5 $\{\}$; 5 $\{f\}$	5 $\{non - raffiné\}$
5 $\{\}$	5 $\{f\}$	5 $\{non - raffiné\}$; 5 $\{\}$
5 $\{f\}$		5 $\{\}$; 5 $\{f\}$

TAB. 2.2 – Résultats d'un algorithme anytime pour la figure 2.1 (diagnostic précédent : 1 $\{\}$; observation : o_1)

tion $(5, \{\})$ est **confirmée**, on l'ajoute donc à l'état de croyance. A la dernière étape, la solution $(5, \{f\})$ est également **confirmée**, on l'ajoute donc aussi. Après cette seconde confirmation, on peut remarquer sur l'automate de la figure 2.1 que tous les chemins allant à l'état 5 (en partant de l'état initial) ont été explorés. L'état 5 a donc été entièrement **raffiné**, et on peut donc retirer la paire $(5, \{non - raffiné\})$ de l'état de croyance.

On se rend compte qu'avec cette mise en place, si l'algorithme est interrompu en cours de calcul, tous les états qui n'ont pas fini d'être raffinés vont conserver leur événement *non-raffiné* lorsqu'ils seront transmis dans la variable représentant le diagnostic précédant dans l'étape de diagnostic suivante. Les abstractions induites par cet événement fictif vont donc certainement s'accumuler.

2.2 Algorithmes

2.2.1 Recherche d'arbre

Le principe algorithmique utilisé dans les exemples de la partie 2.1 sur l'automate de la figure 2.1, peut être associé à une recherche d'arbre [Russel]. En effet, un automate peut être considéré comme un arbre si au lieu de reboucler sur les états existants, on crée un nouveau nœud. La méthode utilisée précédemment est une recherche en largeur d'abord [2]. Dans les algorithmes, cette stratégie de recherche se traduit par une variable des états à visiter de type First In-First Out (file d'attente).

Une autre stratégie de recherche que l'on associe généralement à la précédente, est la recherche en profondeur d'abord [3] que l'on utilise avec une variable des états à visiter de type Last In-First Out (pile) dans les algorithmes. La différence entre ces deux recherches (toutes deux utilisées dans les algorithmes) est illustrée dans la figure 2.2.

Bien sûr, il existe de nombreuses autres stratégies de recherche d'arbre plus complexes ou plus efficaces selon les situations. Toutefois, notre objectif étant avant tout de réaliser un algorithme anytime, nous avons préféré commencer par travailler avec des stratégies de recherche assez simples. C'est pour cette raison que dans la suite, nous n'avons utilisé que les recherches en largeur d'abord et en profondeur d'abord.

[2] Breadth-First Search (BFS) en anglais.
[3] Depth-First Search (DFS) en anglais.

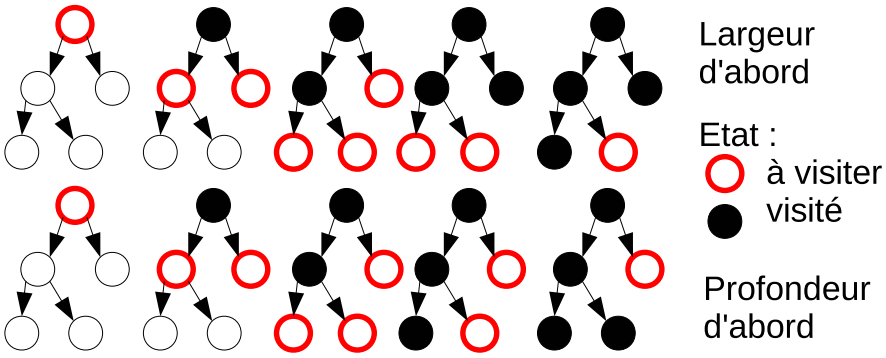

FIG. 2.2 – Comparaison entre les recherches en largeur et profondeur d'abord.

2.2.2 Algorithme complet

Fonction principale L'algorithme 1 décrit la fonction principale du programme. Dans une situation réelle, les observations seraient fournies à cette fonction au fur et à mesure de leur occurrence. Puisque dans notre cas, le système est simulé, nous avons décidé d'utiliser une séquence des observations qui vont se produire, chacune étant associée à son instant d'occurrence.

Aussi, cette fonction doit être capable d'interrompre la fonction de diagnostic pour des raisons externes à cette fonction de diagnostic (occurrence d'une nouvelle observation, requête du planificateur). Il est donc nécessaire que cette fonction puisse agir même lorsque la fonction de diagnostic est en cours. Nous avons donc choisi d'appeler la fonction de diagnostic dans un processus séparé pour qu'elle puisse travailler en parallèle de la fonction principale. Pour mettre cela en place, nous avons décidé d'utiliser la bibliothèque *boost : :thread* pour profiter de la présence de Boost dans DIADES. La réalisation des interruptions de la fonction de diagnostic est décrite dans l'algorithme 5.

Algorithme 1 Fonction principale : pour chaque observation, appel et interruption de la fonction de diagnostic.

Entrée : Modèle global $G = (X, \Sigma, T, x_0)$
Entrée : Observations et instants d'occurrence $\langle (o_1, t_1), \ldots, (o_n, t_n) \rangle$
procRaffinage $\leftarrow$ NouveauProcessus(*fctRaffinage*)
Afficher(*diagInitial*)
Attendre(t_1)
pour $i = 1$ à n **faire**
  procDiagnostic $\leftarrow$ NouveauProcessus(*fctDiagnostic*(o_i))
  Attendre(t_{i+1})
  si *procDiagnostic* pas fini **alors**
    stopProcessus(*procDiagnostic*)
  fin si
  Afficher(*diagCourant*)
fin pour

Fonction de diagnostic L'algorithme 2 décrit la fonction de diagnostic. Le dernier diagnostic obtenu devient le diagnostic précédent pour cette étape. Et comme on l'a expliqué dans la partie 2.1.2, on prend les états cibles de transitions associées à l'observation diagnostiquée, et on les met dans le nouveau diagnostic courant en les associant à l'événement fictif *non-raffiné*. On obtient ainsi un diagnostic courant qui contient le diagnostic final de l'étape.

Dans la suite, nous utilisons deux stratégies de recherche combinées pour parcourir l'automate du modèle à la recherche de solutions à confirmer. La première est une recherche en largeur d'abord, que l'on appelle « en avant » car elle part des états de $diagPrécédent$ et essaye ensuite de trouver ceux de $diagCourant$. La seconde recherche est en profondeur d'abord et « en arrière », puisqu'au contraire de la précédente elle va commencer sa recherche à partir des états de $diagCourant$ pour retrouver ceux du diagnostic précédent. Les caractéristiques propres à ces deux recherches seront développées dans les paragraphes suivants.

Pour revenir sur le fonctionnement de la fonction de diagnostic, on remarque que l'on va appeler la recherche en avant dans un nouveau processus, alors que juste après la recherche en arrière est appelée dans le même processus que celui de la fonction de diagnostic. Cela vient du fait que l'on veuille que ces recherches fonctionnent en parallèle pour associer leurs avantages dans une recherche bidirectionnelle (il n'est pas nécessaire d'appeler la fonction en arrière dans un nouveau processus, puisqu'elle se trouve déjà dans un processus différent de celui de la fonction principale). Enfin, la commande d'attente de la fin du processus « en avant » permet de s'assurer que l'on ne sortira de la fonction de diagnostic qu'une fois que les deux fonctions de recherche seront finies (que ce soit un arrêt naturel ou forcé).

Algorithme 2 Fonction de diagnostic (*fctDiagnostic*) : préparation des variables et lancement des recherches.

Entrée : Modèle global $G = (X, \Sigma, T, x_0)$
Entrée : Observation diagnostiquée Obs
Entrée : Diagnostic courant $diagCourant$
$diagPrécédent \leftarrow diagCourant$
$diagCourant \leftarrow \{(état, non-raffiné) \mid (x, Obs, état) \in T, x \in X\}$
$procAvant \leftarrow$ NouveauProcessus($fctAvant(Obs)$)
$fctArrière(Obs)$
Attendre($procAvant$)
Incrémenter($obsSem$) {sémaphore pour la fonction de raffinage}

Fonctions de recherche L'utilisation de ces deux fonctions de recherches en parallèle nous permet de combiner leurs avantages en une recherche bidirectionnelle. L'objectif de la recherche en arrière (algorithme 3) est de raffiner les états : on part d'un état non-raffiné et on explore tous les chemins qui mènent à cet état. L'inconvénient de cette recherche est sa lenteur puisque, contrairement à une recherche en avant, elle ne peut pas interrompre l'exploration d'une branche après y avoir confirmé une solution. C'est pour compenser cela que nous avons introduit la recherche en avant (algorithme 4), nettement plus rapide pour l'exploration et la confirmation de solutions, mais qui n'est pas aussi efficace pour raffiner des états (cf. définition Raffinage, partie 2.1.2).

La stratégie en profondeur d'abord de la recherche en arrière permet de ne passer au prochain état que lorsque le premier est complètement raffiné. Ainsi, la liste des états à visiter par cette recherche est sous la forme d'une pile. Pour cette pile, au lieu de partir des états *non-raffinés* que l'on a ajouté dans le diagnostic courant dans la fonction de diagnostic, on va plutôt utiliser leurs prédécesseurs : les états sources d'une transition associée à l'observation diagnostiquée et dont la cible est un des états *non-raffinés*. De cette manière, nous n'avons plus besoin de nous occuper des observations dans la suite des recherches en arrière.

Algorithme 3 Recherche en arrière (*fctArrière*) : en profondeur d'abord.

Entrée : Modèle global $G = (X, \Sigma, T, x_0)$
Entrée : Fautes possibles *FautesPossibles*
Entrée : Observation diagnostiquée *Obs*
Donnée partagée : Diagnostics *diagCourant*, *diagPrécédent*
Donnée partagée : Etats visités *étatsVisitésAvant*, *étatsVisitésArrière*
pour tout *étatNonRaffiné* $\in$ *diagCourant* **faire**
  pour tout $(source, Obs, étatNonRaffiné) \in T$ **faire**
    pour tout $(source, fautes) \in étatsVisitésAvant$ **faire**
      $diagCourant \ll (étatNonRaffiné, fautes)$ {Confirmation}
    fin pour
    $étatsAVisiter \ll (source, \emptyset)$
  fin pour
  tant que $étatsAVisiter \neq \emptyset$ **faire**
    $(état, fautes) \leftarrow$ Début($étatsAVisiter$)
    Supprimer Début($étatsAVisiter$)
    pour tout $(source, e \in \Sigma_{io}, état) \in T$ **faire**
      si $e \in FautesPossibles$ **alors**
        $fautes \ll e$
      fin si
      pour tout $(source, fautes2) \in étatsVisitésAvant$ **faire**
        $diagCourant \ll (étatNonRaffiné, fautes \cup fautes2)$ {Confirmation}
      fin pour
      $étatsVisitésArrière \ll (source, étatNonRaffiné, fautes)$
      $étatsAVisiter \ll (source, fautes)$ {au début de la pile (LIFO)}
    fin pour
  fin tant que
fin pour{Raffinage de l'état *étatNonRaffiné*}

Le choix d'une stratégie en largeur d'abord pour la recherche en avant, vient du souhait de visiter des états répartis sur tout l'automate pour aider plus efficacement la recherche en arrière grâce à la variable *étatsVisitésAvant*. La liste des états à visiter pour cette recherche est donc sous la forme d'une file d'attente initialisée à partir des états du diagnostic de l'étape précédente.

Dans les deux fonctions, pour chaque état à visiter, on regarde les transitions inobservables (entrantes ou sortantes selon le sens de recherche) pour voir si l'état à l'autre extrémité a été visité par l'autre recherche. En effet, la recherche bidirectionnelle tire ses avantages du partage des résultats intermédiaires des deux recherches dans les variables globales *étatsVisitésAvant* et *étatsVisitésArrière*. Ainsi, puisque les recherches se font en parallèle, il suffit

à l'une de tomber sur un état visité par l'autre pour confirmer une solution. La recherche en avant peut aussi confirmer des solutions en franchissant une transition associée à l'observation diagnostiquée.

Algorithme 4 Recherche en avant (*fctAvant*) : en largeur d'abord.

Entrée : Modèle global $G = (X, \Sigma, T, x_0)$
Entrée : Fautes possibles *FautesPossibles*
Entrée : Observation diagnostiquée *Obs*
Donnée partagée : Diagnostics $diagCourant$, $diagPrécédent$
Donnée partagée : Etats visités $étatsVisitésAvant$, $étatsVisitésArrière$
$étatsVisitésAvant \leftarrow diagPrécédent$
$étatsAVisiter \leftarrow diagPrécédent$
tant que $étatsAVisiter \neq \emptyset$ **faire**
  $(état, fautes) \leftarrow$ Début($étatsAVisiter$)
  pour tout $(état, e, cible) \in T$ **faire**
    si $e = Obs$ **alors**
      $diagCourant \ll (cible, fautes)$ {Confirmation}
    sinon si $e \in \Sigma_{io}$ **alors**
      si $e \in FautesPossibles$ **alors**
        $fautes \ll e$
      fin si
      pour tout $(cible, étatFinal, fautes2) \in étatsVisitésArrière$ **faire**
        $diagCourant \ll (étatFinal, fautes \cup fautes2)$ {Confirmation}
      fin pour
      $étatsVisitésAvant \ll (cible, fautes)$
      $étatsAVisiter \ll (cible, fautes)$ {à la fin de la file (FIFO)}
    fin si
  fin pour
  Supprimer Début($étatsAVisiter$)
fin tant que

Fonction d'interruption Pour l'interruption d'une étape de diagnostic, nous utilisons la fonction *stopProcessus* de l'algorithme 5. Pour ne pas risquer une interruption au milieu d'une action importante (écriture dans l'état de croyance par exemple), nous n'avons pas utilisé la fonction fournie dans la bibliothèque des processus. A la place, nous mettons à *vrai* les booléens *stopProcAvant* et *stopProcArrière*. Dans les fonctions de recherche, au début de chaque boucle, nous avons ajouté des tests (non représentés dans les algorithmes 4 et 3) qui vont sortir de la fonction si le booléen correspondant est *vrai*. Cela nous permet aussi d'interrompre les processus dans l'ordre inverse de leur création.

Fonction de raffinage La fonction de raffinage, lancée au début de l'algorithme 1, est détaillée dans l'algorithme 6. Lorsque l'on arrête une étape de diagnostic avant la fin, les états qui n'ont pas été raffinés vont conserver leur événement fictif *non-raffiné*. Cet événement sera donc transmis dans les étapes suivantes comme s'il était un événement de faute classique, et ne pourra donc plus être raffiné. C'est pour remédier à ce problème que nous avons introduit cette fonction de raffinage.

Algorithme 5 Fonction d'interruption (*stopProcessus*)

Entrée : Processus du diagnostic *procDiagnostic*
Donnée partagée : Processus de la recherche en avant *procAvant*
Donnée partagée : Booléens *stopProcAvant*, *stopProcArrière*
stopProcAvant $\leftarrow$ *vrai*
Attendre(*procAvant*)
stopProcArrière $\leftarrow$ *vrai*
Attendre(*procDiagnostic*)

L'idée est de reprendre les étapes de diagnostic dans l'ordre jusqu'à ce qu'elles soient complètement raffinées, puis de passer à la suivante si elle est disponible. Pour savoir si la prochaine étape est prête à être raffiner, nous avons utilisé un sémaphore sur le nombre d'étapes disponibles. Ce sémaphore est incrémenté dans l'algorithme 2 lorsque les deux fonctions de recherche sont terminées. Comme pour les processus, nous avons utilisé les sémaphores fournis dans les bibliothèques de Boost.

Contrairement aux deux fonctions de recherche, le processus de raffinage est construit pour fonctionner en continu, et n'est donc pas interruptible. Cela n'est toutefois pas un problème puisque cette fonction ne travaille que sur les étapes de diagnostic qui ont déjà été arrêtées.

Algorithme 6 Fonction de raffinage (*fctRaffinage*) des anciennes étapes.

Entrée : Diagnostic initial *diagInitial*
Entrée : Diagnostics $anciensDiag = [(obs, diagnostic, étatsNonRaffinés)]$
Entrée : Sémaphore sur le nombre d'observations à raffiner *obsSem*
diagPrécédent $\leftarrow$ *diagInitial*
étatsRaffinés $\leftarrow \emptyset$ {états raffinés par la recherche en arrière}
Attendre(*obsSem*) {attend que le premier diagnostic soit prêt}
anciensDiagIt $\leftarrow$ Début(*anciensDiag*) {itérateur sur *anciensDiag*}
tant que *anciensDiagIt* $\neq$ Fin(*anciensDiag*) **faire**
  tant que *étatsRaffinés* $\neq \emptyset$ **faire**
    Recherche en avant en largeur d'abord
    anciensDiagIt.diagnostic $\ll$ solutions
  fin tant que
  tant que *anciensDiagIt.étatsNonRaffinés* $\neq \emptyset$ **faire**
    Recherche en arrière en profondeur d'abord
    anciensDiagIt.diagnostic $\ll$ solutions
    étatsRaffinés $\leftarrow$ solutions
  fin tant que
  diagPrécédent $\leftarrow$ *anciensDiagIt.diagnostic*
  Attendre(*obsSem*) {attend le prochain diagnostic}
  $++$ *anciensDiagIt*
fin tant que

Pour chaque étape, cette fonction part du diagnostic incomplet et de la liste des états qui n'y ont pas été raffinés. On commence le raffinage par une recherche en avant en largeur d'abord à partir des états qui ont été raffinés par la précédente étape de la fonction de raffinage (*étatsRaffinés*). Son but est

de supprimer les événements *non-raffinés* transmis par l'étape de diagnostic précédente, en trouvant les solutions qu'ils représentaient. Ensuite, on passe à une recherche en arrière en profondeur d'abord qui va finir de raffiner les états *non-raffinés* créés dans cette étape par la fonction de diagnostic.

Versions Au final, notre algorithme complet de plus de 2500 lignes de code C++ est composé principalement de trois fonctions travaillant en parallèle : deux fonctions de recherche qui s'associent en une recherche bidirectionnelle, et une fonction de raffinage des anciennes étapes. Les programmes sont construits de telle sorte que l'on peut désactiver n'importe laquelle de ces trois fonctions, il suffit pour cela de mettre en commentaire la création du processus de la fonction correspondante. La seule contrainte est de garder active au moins l'une des deux fonctions de recherche. On obtient ainsi 6 versions possibles pour notre programme : 3 configurations de recherche (Avant, Arrière, Bidirectionnelle), chacune pouvant être avec ou sans la fonction de Raffinage.

2.3 Critères de qualité

2.3.1 Les critères

Après avoir réalisé un algorithme de diagnostic anytime, il est maintenant nécessaire de vérifier qu'il est effectivement plus avantageux que l'algorithme original dans les situations pour lesquelles il a été créé. Aussi, comme on vient de voir que notre programme regroupe plusieurs versions de cet algorithme, il pourrait être intéressant de les comparer entre elles pour trouver les versions les plus efficaces. C'est donc pour mesurer les performances des algorithmes que nous avons développé plusieurs critères de qualité, afin de couvrir tous les aspects importants pour un algorithme de diagnostic anytime.

D'un point de vue algorithmique, l'amélioration de l'état de croyance peut se faire de deux manières : la confirmation d'une solution, et le raffinage d'un état. Nous avons donc créé un critère pour mesurer les performances de chacun de ces deux aspects (définitions 4 et 5). Du point de vue du diagnostic, on s'intéresse plutôt aux fautes. Ainsi, le critère de la définition 6 compare l'état de confirmation de chaque faute à celui du résultat final.

Définition 4 (Critère de raffinage) *Le critère C_r donne une mesure (entre 0 et 1) de l'avancement du raffinage.*

- $C_r = \dfrac{|X| * 2^{|\Sigma_f|} - |\acute{e}tatsCouverts|}{|X| * 2^{|\Sigma_f|} - |diagnosticFinal|}$ *avec :*
- *$\acute{e}tatsCouverts$ est l'ensemble des solutions potentielles que couvre le diagnostic courant, en remplaçant les événements* non-raffiné *par toutes les combinaisons de fautes possibles ;*
- *$diagnosticFinal$ est l'état de croyance final, obtenu avec l'algorithme original non-anytime ;*
- *$|X| * 2^{|\Sigma_f|}$ est le nombre total d'éléments qu'un diagnostic peut couvrir :*
  - *X est le nombre d'états du modèle global ;*
  - *Σ_f est l'ensemble des événements de faute.*

Définition 5 (Critère de confirmation) *Le critère C_c donne la progression des confirmations de solutions :*

- $C_c = \frac{|solutionsConfirmées|}{|diagnosticFinal|}$ *avec :*
- *solutionsConfirmées est l'ensemble des solutions confirmées dans le diagnostic courant, dans lesquelles l'événement* non-raffiné *n'est pas présent ;*

Définition 6 (Critère de faute) *Le critère C_f compare l'état de chaque faute dans le diagnostic courant, avec celui du résultat final :*

- $C_f = \frac{|\Sigma_f| - distanceFautes}{|\Sigma_f|}$ *avec :*
- *distanceFautes est la distance de Hamming entre les statutFautes du diagnostic courant et du résultat final ;*
- *statutFautes est une liste associant à chaque faute de Σ_f, un état parmi : Sain ; Sûr ; Ambigu ; SainOuAmbigu (seulement des solutions sans cette faute ont été confirmées jusque là) ; SûrOuAmbigu ; Inconnu.*

2.3.2 Les mesures

Les mesures de ces critères doivent se faire par le biais d'un autre programme car le calcul de ces critères nécessite de connaître les résultats du diagnostic final. C'est pourquoi ce nouveau programme va commencer par lancer le diagnostic non anytime, puis va sauvegarder pour chaque étape les données suivantes : le temps de calcul ; le nombre d'éléments dans l'état de croyance final ; le statut de confirmation des fautes. Ensuite, ce programme pourra être utilisé de deux manières selon l'objectif de la mesure.

La première version permet de mesurer la qualité obtenue par l'algorithme anytime, lorsqu'on le fait fonctionner pendant la durée utilisée pour que l'algorithme non anytime se termine. Ainsi, pour chaque observation, on lance le diagnostic anytime que l'on interrompt après le temps relevé dans l'étape correspondante du diagnostic non anytime, puis on relève la valeur des critères et on lance la prochaine étape. Lorsque la dernière étape est terminée, on va également interrompre le processus de raffinage s'il était en marche. Pour ce processus, il suffit de relever les critères pour l'étape qu'il était en train de traiter, puisque toutes les étapes précédentes ont atteint la valeur maximale des critères, et les étapes suivantes n'ont pas changé.

La seconde version est utilisée pour comparer les différentes versions de l'algorithme entre elles, plutôt que de les comparer à l'algorithme non anytime. En effet, cette méthode permet de mesurer le temps nécessaire à chaque étape de l'algorithme pour que les critères atteignent une qualité minimale. Dans cette version, la valeur des critères est réévaluée à chaque modification de l'état de croyance, et l'étape est interrompue lorsque tous les critères ont dépassé la valeur seuil (il est possible de modifier cela si l'on souhaite que seulement la moyenne des critères dépasse le seuil).

Pour faciliter la visualisation et l'exploitation des résultats, nous les avons écrits dans un fichier au format CSV [4] qu'il est possible d'ouvrir dans une feuille de calcul. Les données sauvegardées dans ce fichier sont les suivantes : le numéro de l'étape, le temps de calcul de l'étape de l'algorithme anytime, la valeur de chacun des critères, et le type de comparaison (avec ou sans seuil).

[4] Comma-Separated Values

Chapitre 3

Résultats

3.1 Plan des expérimentations

Tout au long de ce projet, les exemples que nous avons utilisés pour tester les algorithmes n'étaient pas extraits de systèmes réels. En effet, la plateforme DIADES possède également un générateur aléatoire d'automates pouvant être directement utilisés avec nos applications. Ce sont donc des modèles extraits de ce générateur (dans lequel il suffit de préciser un intervalle pour le nombre d'états et d'événements) que nous utilisons dans la suite pour mesurer les performances de nos algorithmes.

Dans nos expérimentations, nous allons générer une série de modèles en fixant la valeur de tous les paramètres à l'exception de celui que nous voulons tester. En comparant les résultats — obtenus à partir des deux algorithmes de mesure décrits dans la partie 2.3.2 —pour les différentes valeurs de ce paramètre, nous pouvons en extraire un intervalle sur lequel l'algorithme anytime est utile et fonctionne correctement. Nous fixons donc ce paramètre à une valeur intermédiaire dans cet intervalle, et nous recommençons pour un autre paramètre.

Les principaux facteurs que nous pouvons faire varier sont la taille du modèle (nombre d'états, de transitions, d'événements), la répartition des événements (proportion de fautes, d'observations), et la version utilisée par l'algorithme (parmi les six possibilités présentées dans la partie 2.2.2.) Puisque l'objectif principale est de comparer ces six versions entre elles et avec l'algorithme non-anytime de base, nous avons décidé que ce serait le dernier paramètre testé, et la version utilisée pour les autres tests sera la recherche Bidirectionnelle avec fonction de Raffinage. Les expérimentations se feront toutes sur une séquence de cinq observations, en faisant varier les paramètres dans l'ordre suivant : le nombre d'états ; le nombre maximal de transitions sortantes pour chaque état ; la proportion d'événements observables ; la proportion d'événements de faute.

3.2 Observations générales

Avant d'aller dans le détail des résultats de chaque expérimentation, il serait bon de relever certaines observations concernant tous les tests. Premièrement, on remarque que le temps de calcul de l'algorithme de base pour la première

observation d'une séquence est toujours beaucoup plus court que les autres. Cela s'explique par le fait que le diagnostic initial ne contient toujours qu'un seul élément, et que cet algorithme utilise une recherche en avant. Ainsi, la valeur des critères de qualité pour l'algorithme anytime sera toujours assez faible pour la première étape, puisque très peu de temps sera disponible quand on utilise la première méthode de mesure de la partie 2.3.2.

La seconde observation importante est que dans la suite, seul le critère de confirmation est véritablement utilisé pour mesurer la qualité des résultats, car les deux autres sont peu exploitables. Comme nous utilisons des modèles de taille relativement importante, et particulièrement avec beaucoup de transitions, le statut de confirmation des fautes est presque toujours *ambigu* : au moins une solution avec cette faute, et une autre sans. Aussi, le critère de raffinage est calculé par rapport au nombre total de possibilités, alors que dès le début de l'algorithme on réduit la liste des états couverts seulement aux états cibles d'une transition associée à l'observation diagnostiquée. Cette première étape de raffinage explique donc pourquoi les valeurs de ce critère sont toujours très hautes (généralement entre $0,9$ et 1). Ainsi, seul le critère de confirmation peut vraiment être exploité pour connaître la progression des algorithmes.

3.3 Etape par étape

Les paragraphes qui suivent détaillent la mise en place des expérimentations, analysent leurs résultats, et concluent sur un choix pour la série de tests suivante. Pour voir le détail des résultats (tableaux et graphiques), reportez vous en annexe page 28.

Nombre d'états Dans cette série de 5 tests, le nombre d'états varie entre 20 et 100. Les autres paramètres ne sont pas exactement fixes, nous les avons fait varier de manière proportionnelle au nombre d'états :

- $nbEvénements = nbEtats/2$;
- $nbObservations = nbEtats/4$;
- $nbFautes = nbEtats/20$;
- $transitionsSortantesMax = nbEtats/10$.

Les deux premiers modèles, possédant respectivement 20 et 40 états, contiennent trop peu d'éléments pour que leurs résultats puissent réellement être analysables. En effet, le premier modèle par exemple ne contient qu'une seule faute, ce qui rend binaire le critère de faute, et chaque état à raffiner ne peut être associé qu'à deux combinaisons de fautes différentes. Or, les temps de calcul de l'algorithme de base étant très courts (inférieurs à une milliseconde), la chance va grandement influencer les résultats de l'algorithme anytime qui est naturellement un peu plus lent que l'algorithme de base, car plus complexe.

Le temps nécessaire aux calculs de chaque étape augmente de façon exponentielle en fonction du nombre d'états du modèle. Ainsi, lorsque l'on utilise la méthode de comparaison où les interruptions sont fixées par les temps de calcul de l'algorithme de base, l'algorithme anytime a assez de temps disponible pour que l'on fasse confiance au résultat. Dans notre cas, l'efficacité de l'algorithme anytime bidirectionnel varie entre 60 et 80% de l'algorithme de base pour un nombre d'états entre 60 et 100. Nous avons donc décidé de fixer le nombre d'états à 80 pour ne pas avoir un modèle trop petit où l'algorithme anytime

est inutile, ni un modèle trop grand pour que les temps de simulations restent raisonnables.

Transitions sortantes L'étape précédente a donc fixé le nombre d'états à 80, et le nombre d'événements à 40 (dont 20 observations et 4 fautes). Le paramètre que l'on va faire varier ici est le nombre maximal de transitions sortantes par état. Pour chaque état, le nombre de transitions sortantes assignées par le générateur aléatoire est déterminé à partir d'une loi de probabilité uniforme entre 1 et ce nombre maximal. En moyenne, chaque état aura donc un nombre de transitions sortantes égal à la moitié de cette limite. Bien que cette limite puisse aller jusqu'au nombre d'événements du modèle, on constate que dans les applications réelles, les systèmes n'ont jamais autant de transitions. C'est pourquoi nous n'allons faire varier ce paramètre qu'entre 1 et 16.

Comme on le dit dans le paragraphe précédent, on sait que l'algorithme anytime est plus lent que l'algorithme de base, puisqu'il est plus complexe. On remarque ainsi à nouveau sur cette série de tests que l'algorithme anytime est particulièrement peu efficace sur les petits modèles. Pour les modèles plus conséquents (à partir d'une limite des transitions sortantes à 9), la qualité des résultats de l'algorithme anytime se stabilise à 80% de ceux de l'algorithme de base pour une même durée de calcul. Si l'on compare les temps de calcul de l'algorithme de base à ceux nécessaires pour que l'algorithme anytime atteigne un seuil de qualité de 80%, pour notre paramètre variant entre 9 et 16, les temps de calcul augmentent $2,5$ fois moins vite pour l'algorithme anytime. Nous allons donc fixer notre paramètre à 13 : une valeur intermédiaire de l'intervalle $[9; 16]$, pour éviter les mauvaises performances constatées sur des modèles plus petits.

Observations Dans cette étape, nous allons faire varier l'observabilité de la liste d'événements. Sans modifier le nombre total d'événements (40), nous allons donc réaliser 7 tests où le nombre d'événements observables variera entre 5 et 35.

De la même manière que précédemment, lorsque le modèle est trop facilement diagnosticable (ici, lorsque plus des trois quarts des événements sont observables) l'algorithme de base est très rapide et l'algorithme anytime n'arrive pas à conserver une efficacité correcte. Au contraire, lorsque le modèle est peu observable, l'efficacité de l'algorithme anytime se stabilise à 80% de celle de l'algorithme de base. Aussi, lorsque l'on utilise un seuil de qualité de $0,8$ pour l'algorithme anytime, les temps de calcul peuvent être jusqu'à 4 fois plus faibles que pour l'algorithme de base pour un modèle très peu observable. L'algorithme anytime est donc plus utile et plus efficace sur un modèle assez inobservable. Pour ne pas non plus se placer dans le cas le plus extrême d'inobservabilité, nous allons choisir d'utiliser 10 observations (un quart des événements).

Fautes Cette fois, nous allons faire varier la proportion d'événements fautifs (de 1 à 8 fautes) parmi les événements inobservables (30). En traçant le logarithme du temps de calcul des étapes en fonction du nombre de fautes, on observe que le temps de calcul est exponentiel en le nombre de fautes. Cela s'explique facilement par le fait que le nombre de combinaisons de fautes que l'on peut associer à chaque état est au plus $2^{nbFautes}$. On voit donc bien à quel point l'algorithme non-anytime de base est limité, puisqu'un modèle à 9 fautes

nécessiterait environ 7 heures de calcul par étape. Le temps de calcul nécessaire à l'algorithme anytime varie de la même manière que pour l'algorithme de base. Ainsi, quelque soit le nombre de fautes testé, la qualité de l'algorithme anytime oscille autour de 80% des résultats complets.

C'est donc davantage du point de vue des temps de calcul par étape qu'il faut se placer pour connaître l'intervalle sur lequel l'algorithme anytime est intéressant. L'algorithme anytime a besoin en moyenne de 6 (respectivement 28) secondes par étape pour atteindre le seuil de qualité $0,8$, pour un modèle contenant 5 (resp. 6) événements de faute. Puisqu'un quart des événements sont des observations, cela revient à avoir l'occurrence d'un événement toutes les $1,5$ (resp. 7) secondes. Cela dépendra du système modélisé, mais dans beaucoup de cas, notre algorithme risque de ne pas être capable d'assurer une qualité suffisante si le nombre de fautes est supérieur à 5 ou 6. D'un autre côté, il n'est pas très intéressant d'utiliser un algorithme de diagnostic sur un système avec très peu de fautes. C'est pourquoi dans la suite, nous prendrons le nombre de fautes égal à 5.

Algorithmes Nous nous sommes donc placés dans une configuration (80 états ; 40 événements dont 10 observations et 5 fautes ; au plus 13 transitions sortantes par état) où l'algorithme anytime est à la fois utile et efficace, et c'est désormais le type d'algorithme utilisé que nous allons faire varier. Pour cette expérimentation, nous avons toujours utilisé le même modèle, mais avec trois séquences d'observations différentes. Les résultats étant similaires dans les trois cas, les commentaires qui suivent concernent leur moyenne.

Quelque soit le type de recherche utilisée, on remarque que les fonctions de recherche sont très peu ralenties par la présence de la fonction de Raffinage. Il semble donc bon de garder cette fonction qui peut s'avérer très utile.

Lorsque l'on assigne la même durée de calcul aux algorithmes anytime et non-anytime, on observe que la recherche en avant est la plus efficace (obtient 91% du résultat final), suivie par la bidirectionnelle (78%) et la recherche en arrière (40%). En utilisant l'autre méthode de comparaison, on remarque aussi que seule la recherche en arrière nécessite plus de temps que l'algorithme de base pour atteindre un seuil de qualité de 80%. On observe aussi que, malgré le fait que la recherche en avant ne peut pas raffiner des états aussi facilement que les autres, sa grande rapidité lui permet tout de même d'être la plus efficace.

3.4 Synthèse des résultats

Les premiers tests réalisés sur notre algorithme nous ont permis de voir que quelque soit la version utilisée, si on ne l'interrompt pas avant la fin, l'algorithme anytime obtiendra le même résultat que l'algorithme non-anytime dont on se sert comme référence. Bien sûr, les algorithmes anytimes développés ne sont pas plus rapides que l'algorithme de base : ce n'était pas l'objectif, et c'était d'ailleurs prévisible qu'ils soient un peu plus lents puisqu'ils sont plus complexes. Par contre, ils remplissent bien leurs objectifs : être interruptible à tout instant, et être capable de donner un résultat « correct » (c'est à dire dans notre cas, qui contient le résultat final) lorsqu'il est interrompu.

Suite aux expérimentations de la partie précédente, nous avons pu observer deux types de limitations à cet algorithme anytime. La première concerne les

modèles très rapides à diagnostiquer (peu d'états, d'événements, de transitions ; peu de fautes ; grande observabilité). Dans ce cas de figure, l'algorithme anytime est très peu utile, puisqu'il est moins performant que l'algorithme de base, qui lui-même est assez rapide pour se passer de l'algorithme anytime.

La seconde limitation concerne cette fois les modèles de trop grande taille. Le type de recherche utilisée par les algorithmes anytime et non-anytime étant similaire, les temps de calcul requis par les deux algorithmes évolueront de la même manière en fonction de la taille du système diagnostiqué. Alors, l'algorithme anytime aura beau être interruptible, ses résultats seront toujours de faible qualité si les temps de calcul requis par le modèle sont bien supérieurs au délai avant la prochaine interruption.

En ce qui concerne les différentes versions de l'algorithme, la première remarque est que la fonction de raffinage ne ralentit que très peu les fonctions de recherches. Il sera donc bon de garder cette fonction, puisque d'une part elle permet de raffiner toutes les étapes avec un certain retard (lorsque l'on souhaite recevoir des résultats exacts sans que cela ne soit pressé), d'autre part s'il y a une pause suffisamment longue entre deux observations, elle va permettre à la fonction de diagnostic de repartir d'un état de croyance complètement raffiné.

Lors de la comparaison des trois méthodes de recherche, nous avons remarqué que la recherche en arrière est beaucoup trop lente par rapport aux deux autres et à l'algorithme de base. Bien que la recherche en avant soit plus rapide que celle bidirectionnelle, elle n'a pas la possibilité d'orienter ses recherches vers un état final particulier. Ainsi, si l'objectif de l'algorithme de diagnostic anytime est d'avancer le plus possible pendant le temps disponible, alors le meilleur choix est celui de la recherche en avant. Mais si dans la suite l'on souhaitait prendre en compte des requêtes du planificateur pour orienter les recherches sur un état particulier, alors il sera plus intéressant d'utiliser la recherche bidirectionnelle.

Conclusion

Au cours de ce stage, nous avons développé un algorithme de diagnostic « à la demande », c'est à dire interruptible à tout instant tout en restant capable de fournir un résultat correct. L'architecture de cet algorithme est composée de plusieurs processus fonctionnant en parallèle : la fonction principale qui va lancer et interrompre le diagnostiqueur ; deux fonctions de recherche (en arrière pour raffiner les états, et en avant pour confirmer rapidement de nouvelles solutions) alliant leurs avantages en une recherche bidirectionnelle ; une fonction pour raffiner les abstractions dans les anciennes étapes interrompues. L'utilisation de processus nous a également donné la possibilité de n'utiliser qu'une partie de ces fonctions, en créant ainsi 6 versions différentes pour cet algorithme. Afin de pouvoir mesurer les performances de notre algorithme, nous avons ensuite mis en place un ensemble de critères de qualité, ainsi que deux programmes permettant de comparer les performances de notre algorithme à celle d'un algorithme classique (pas « à la demande »). Nous avons fini par réaliser une série de tests pour démontrer l'utilité de notre algorithme, trouver ses limitations, et déterminer quelles versions sont les plus intéressantes.

Ce projet a donc permis de réaliser un nouvel algorithme dans le domaine encore très peu développé du diagnostic anytime. Plus précisément, nous n'avons rien trouvé dans la littérature en ce qui concerne un tel algorithme pour la modélisation à base d'automates que nous avons utilisée. Le travail réalisé semble donc assez novateur, et il reste par conséquent de nombreuses voies à explorer pour continuer d'améliorer cet algorithme. On pourrait par exemple essayer de mettre en place des stratégies de recherche plus complexes, ou encore utiliser une version modifiée des critères de qualité pour passer d'une recherche à une autre selon le moment où elles sont le plus efficace. Il pourrait aussi être intéressant de voir comment prendre en compte des requêtes du planificateur plus complexes qu'une interruption : concentrer les recherches sur une faute, un état, ...

J'ai beaucoup apprécié ce stage pour deux principales raisons. D'une part, j'ai été très intéressé par son sujet puisque, comme je le recherchais, il portait sur les systèmes à événements discrets, mais aussi parce qu'il m'a permis de découvrir le domaine du diagnostic que je ne connaissais pas du tout. D'autre part, cette immersion dans le monde de la recherche m'a beaucoup apporté pour comprendre l'organisation du travail dans un laboratoire de recherche, à la fois par le travail que j'ai eu à réaliser, mais aussi en m'entretenant avec les chercheurs et doctorants autour de moi. Cela m'a permis de me faire une meilleure idée des possibilités qui se présentaient à moi pour la suite, et je pense tenter de me diriger vers la réalisation d'une thèse dans un domaine qui me plaît beaucoup, celui des systèmes à événements discrets.

Annexe : Résultats des expérimentations

Cette annexe présente les résultats importants obtenus au cours des tests décrits dans le chapitre 3. Toutes les simulations ont été réalisées sur une séquence de 5 observations, et les données présentées correspondent à la moyenne des résultats sur ces 5 observations. A l'exception de la dernière série de tests, toutes les simulations ont été réalisées en utilisant l'algorithme bidirectionnel associé à la fonction de raffinage. Dans la suite, les résultats présentés sont les suivants :

- les valeurs du critère de confirmation (entre 0 et 1) lorsque les interruptions sont fixées par les temps de calcul de l'algorithme de base (les deux autres critères se sont avérés peu utiles) ;
- le temps de calcul moyen pour l'algorithme de base non-anytime ;
- le temps de calcul moyen pour que l'algorithme anytime dépasse un seuil de $0,8$ sur tous ses critères de qualité.

Nombre d'états

Etats	$20 \rightarrow 100$
Evénements	$nbEtats/2$
Observations	$nbEtats/4$
Fautes	$nbEtats/20$
Max transitions sortantes	$nbEtats/10$

Dans cette première série de tests, les paramètres sont fixés comme dans le tableau ci-dessus, et on fait varier le nombre d'états du modèle. Les résultats obtenus sont présentés dans le tableau 1.

Etats	**Confirmation** (C_c)	**Temps moyen par étape (ms)** (non-anytime)	(anytime : seuil 0.8)
20	1	0	0.4
40	1	1	4.6
60	0.69	273	269
80	0.78	2 579	1 510
100	0.61	8 491	5 944

TAB. 1 – Résultats pour différentes tailles (nombre d'états) du modèle.

La figure 1 donne l'histogramme du critère de confirmation, et la figure 2 met en avant la dépendance exponentielle entre les temps de calcul et le nombre d'états.

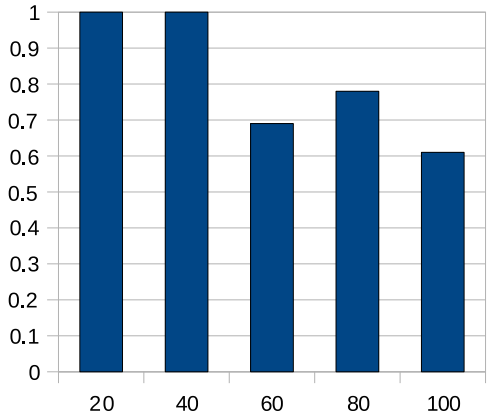

Fig. 1 – Critère de confirmation en fonction du nombre d'états

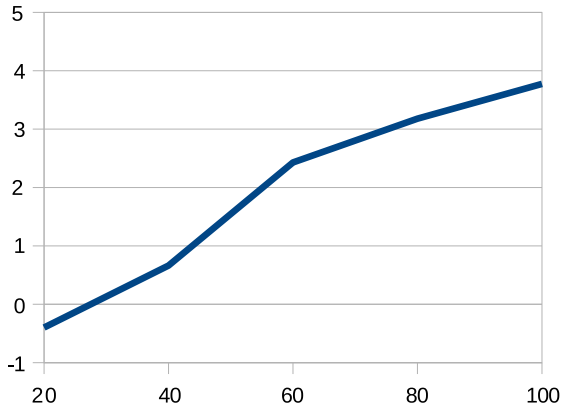

Fig. 2 – Logarithme du temps par étape, en fonction du nombre d'états

Transitions sortantes

Etats	80
Evénements	40
Observations	20
Fautes	4
Max transitions sortantes	$1 \rightarrow 16$

Dans cette série de tests, les paramètres sont fixés comme dans le tableau ci-dessus, et on fait varier le nombre maximal de transitions sortantes que chaque

état peut avoir. Les résultats obtenus sont présentés dans le tableau 2. La figure 3 donne l'histogramme du critère de confirmation, et la figure 4 donne la comparaison des temps de calcul nécessaires aux algorithmes anytime et non-anytime.

Max transitions sortantes	**Confirmation** (C_c)	**Temps moyen par étape (ms)** (non-anytime)	(anytime : seuil 0.8)
1	0.8	0	1.2
2	0.2	0.2	3.2
3	0.2	0.4	4.2
4	0.43	14	67
5	0.4	50	203
6	0.55	51	354
7	0.29	867	3 807
8	0.34	1 040	2 369
9	0.78	2 764	1 079
10	0.84	3 275	1 179
11	0.8	2 812	2 267
12	0.82	3 348	1 066
13	0.84	6 151	1 839
14	0.83	5 541	2 166
15	0.81	8 475	1 793
16	0.82	7 216	2 853

Tab. 2 – Résultats pour différentes tailles (nombre de transitions) du modèle.

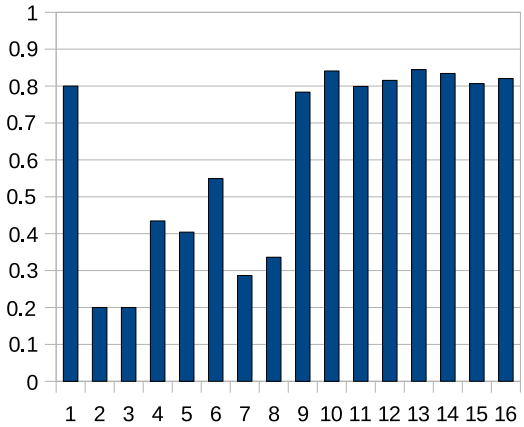

Fig. 3 – Critère de confirmation en fonction de la limite des transitions sortantes

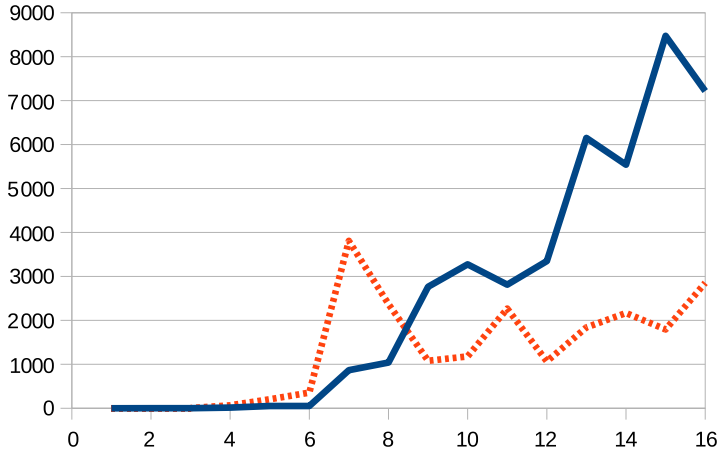

FIG. 4 – Temps par étape (ms) en fonction de la limite des transitions sortantes (lignes pleines : non-anytime ; pointillés : anytime avec seuil 0.8)

Nombre d'observations

Etats	80
Evénements	40
Observations	5 → 35
Fautes	4
Max transitions sortantes	13

Dans cette série de tests, les paramètres sont fixés comme dans le tableau ci-dessus, et on fait varier le nombre d'observations (en gardant le nombre total d'événements fixe). Les résultats obtenus sont présentés dans le tableau 3. La figure 5 donne l'histogramme du critère de confirmation, et la figure 6 donne la comparaison des temps de calcul nécessaires aux algorithmes anytime et non-anytime.

Observations	**Confirmation**	**Temps moyen par étape (ms)**	
	(C_c)	(non-anytime)	(anytime : seuil 0.8)
5	0.78	9 905	2 557
10	0.79	4 940	2 463
15	0.84	4 962	2 238
20	0.67	3 731	3 525
25	0.84	2 502	1 644
30	0.53	1 218	2 023
35	0.23	6	49

TAB. 3 – Résultats pour différentes proportions d'événements observables.

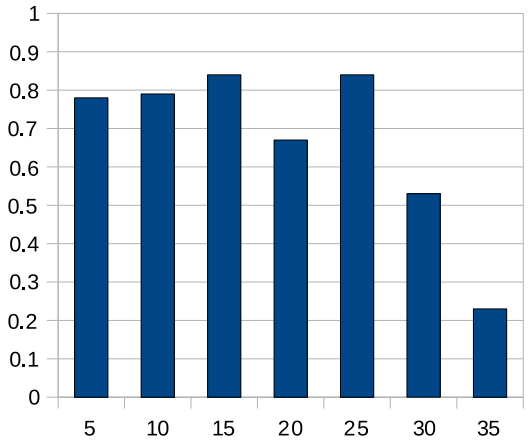

FIG. 5 – Critère de confirmation en fonction du nombre d'observations

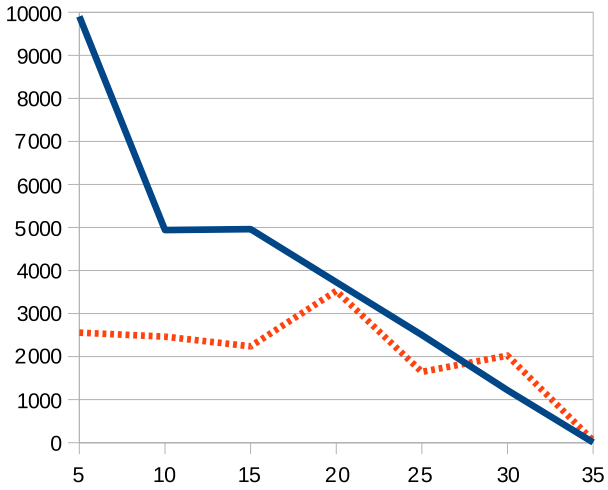

FIG. 6 – Temps par étape (ms) en fonction du nombre d'observations (lignes pleines : non-anytime ; pointillés : anytime avec seuil 0.8)

Nombre de fautes

Etats	80
Evénements	40
Observations	10
Fautes	$1 \rightarrow 8$
Max transitions sortantes	13

Dans cette série de tests, les paramètres sont fixés comme dans le tableau ci-dessus, et on fait varier le nombre de fautes (en gardant le nombre total d'événements fixe). Les résultats obtenus sont présentés dans le tableau 4. La figure 7 donne l'histogramme du critère de confirmation, et la figure 8 met en avant la dépendance exponentielle entre les temps de calcul et le nombre de fautes.

Fautes	**Confirmation** (C_c)	**Temps moyen par étape (ms)** (non-anytime)	(anytime : seuil 0.8)
1	1	326	81
2	0.87	717	162
3	0.85	1 588	710
4	0.76	3 803	2 012
5	0.75	12 905	6 611
6	0.76	64 057	28 858
7	0.82	397 396	107 819
8	0.82	2 923 849	1 038 220

Tab. 4 – Résultats pour différentes proportions d'événements de faute.

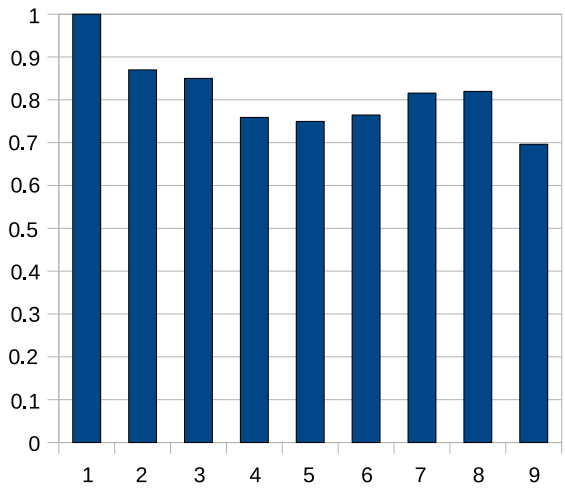

Fig. 7 – Critère de confirmation en fonction du nombre de fautes

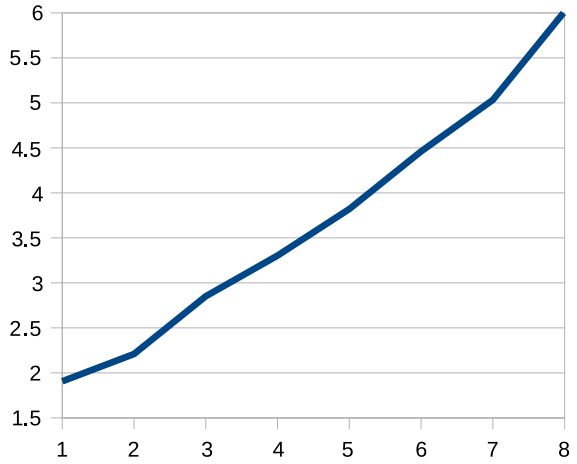

FIG. 8 – Logarithme du temps par étape, en fonction du nombre de fautes

Algorithmes

Etats	80
Evénements	40
Observations	10
Fautes	5
Max transitions sortantes	13

Dans cette dernière série de tests, les paramètres sont fixés comme dans le tableau ci-dessus, et on compare les 6 versions de l'algorithme sur un même modèle. Les résultats obtenus sont présentés dans le tableau 5. La figure 9 donne l'histogramme du critère de confirmation en fonction de la version de l'algorithme utilisée, et la figure 10 donne la comparaison des temps de calcul nécessaires aux algorithmes anytime et non-anytime.

Algorithme	**Confirmation** (C_c)	**Temps moyen par étape (ms)** (non-anytime)	(anytime : seuil 0.8)
Avant	0.92	17 654	1 478
Av. + Raf.	0.91	18 583	2 280
Bidirectionnel	0.79	17 702	10 624
Bi. + Raf.	0.78	17 044	6 524
Arrière	0.46	19 240	27 463
Ar. + Raf.	0.35	17 191	22 581

TAB. 5 – Résultats pour comparer les 6 versions de l'algorithme.

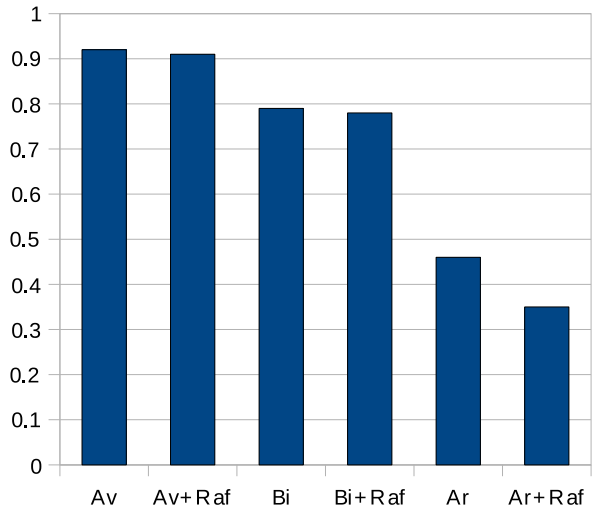

FIG. 9 – Critère de confirmation en fonction de la version de l'algorithme

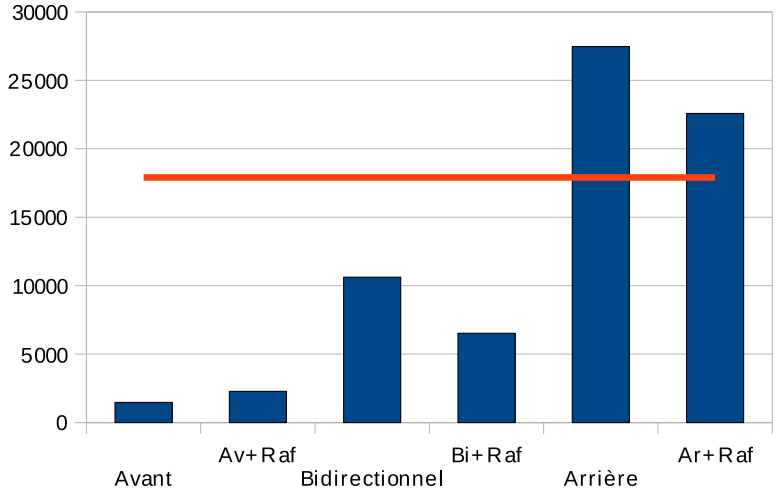

FIG. 10 – Temps par étape (ms) en fonction de la version de l'algorithme (droite : non-anytime ; histogramme : anytime avec seuil 0.8)

Bibliographie

[Cobb99] Paul Cobb, Eric S. Yager et Charles Jacobus, *Anytime Diagnosis Using Model-Based Methods for Satellite Diagnostics*, Proceedings of the Twelfth International FLAIRS Conference, 1999.

[Likhachev05] Maxim Likhachev, Dave Ferguson, Geoff Gordon, Anthony Stentz, et Sebastian Thrun, *Anytime Dynamic A* : An Anytime, Replanning Algorithm*, 2005.

[Mouaddib95] Abdel-Illah Mouaddib et Shlomo Zilberstein, *Knowledge-based anytime computation*, 14th International Joint Conference on Artificial Intelligence, août 1995.

[Pencolé05] Yannick Pencolé et Marie-Odile Cordier, *A formal framework for the decentralised diagnosis of large scale discrete event systems and its application to telecommunication networks*, Artificial Intelligence Vol. 164 No. 1-2 pages 121-170, 2005.

[Russel] Stuart Russell et Peter Norvig, *Artificial Intelligence, A Modern Approach*, chap. 3-4, Prentice Hall.

[Sampath96] Meera Sampath, Raja Sengupta, Stéphane Lafortune, Kasim Sinnamohideen, et Demosthenis C. Teneketzis, *Failure Diagnosis Using Discrete-Event Models*, IEEE transactions on control systems technology Vol. 4 No. 2 pages 105-124, mars 1996.

[Schumann07] Anika Schumann, Yannick Pencolé et Sylvie Thiébaux, *A Spectrum of Symbolic On-line Diagnosis Approaches*, Twenty-Second Conference on Artificial Intelligence (AAAI-07), juillet 2007.

[VanHarmelen95] Frank van Harmelen et Annette ten Teije, *Approximations in diagnosis : motivations and techniques*, Proceedings of the Symposium on Abstraction, Reformulation and Approximation, août 1995.

[Verberne00] Alan Verberne, Frank van Harmelen et Annette ten Teije, *Anytime Diagnostic Reasoning using Approximate Boolean Constraint Propagation*, Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning, avril 2000.

[Zilberstein96] Shlomo Zilberstein, *Using Anytime Algorithms in Intelligent Systems*, AI Magazine Vol. 17 No. 3 pages 73-83, automne 1996.