Stage de Master 2 Recherche en Informatique

Améliorations algorithmiques d'un moteur de Model Checking et études de cas

Auteur :
Romain Soulat

Maître de stage :
Dr. Laurent Fribourg

Organisme d'accueil :
Laboratoire Spécification et Vérification, ENS Cachan

Secrétariat - tél : 01 69 15 75 18 Fax : 01 69 15 42 72
courrier électronique : Jacques.Laurent@lri.fr

12 mars – 11 septembre 2008

Table des matières

Résumé

Mots clefs

Model Checking, IMITATOR, Arbres de défaillances dynamique, Vérification, Méthode Inverse, Cartographie.

Résumé

Dans ce mémoire, on s'intéresse à la modélisation des Arbres de Défaillances Dynamique, leur modélisation en automate à états fini ainsi que leur étude grâce à des model checkers. On s'intéresse également à des modifications algorithmiques d'un moteur de model checking, IMITATOR, dont la nécessités'est révélée au travers d'études de cas. Ces modifications sont principalement sur l'implémentation de la méthode inverse d'IMITATOR. La partie cartographie des comportements a été étudiée et quelques résultats théoriques sont également présentés.

CHAPITRE 1

Introduction

Mon stage s'est porté sur le Model Checking, [14, 18], et plus particulièrement le model checking sur les automates temporisés paramétriques, au sein du Laboratoire Spécification et Vérification de l'École Normale Supérieure de Cachan sous la direction de Laurent Fribourg. Il s'est articulé en deux temps. Le premier fut pour le projet CRAFT [9], le but était de modéliser un Arbre de Défaillances Dynamique, ADD, [17, 16], en un automate à état fini afin d'en trouver les séquences de coupe. Dans un deuxième temps, je me suis intéressé à l'outil IMITATOR (Model Checker pour les automates temporisés paramétriques dévellopé au sein du LSV), [2], et plus précisément aux algorithmes utilisés dans cet outil, tels que la Méthode Inverse [3] et la cartographie des comportements [4].
Ce premier temps consacré à la modélisation d'ADD m'a permis de me familiariser avec divers model checkers tels que HyTech [12, 13], Uppaal [6], PHAVer [15, 11], notamment au niveau de la syntaxe et les problèmes que l'on peut rencontrés lors de la modélisation, comme la création d'états urgents, les problèmes de synchronisation etc. Dans la seconde partie, j'ai été confronté à des exemples beaucoup plus complexes, tels que le Bounded Retransmission Protocol [10], et les mémoires SPSMALL. J'ai étudié ces exemples grâce à IMITATOR, afin d'observer les performances de ce dernier. Ces exemples ont poussés l'outil dans ses limites et ont forcés à concevoir des modifications algorithmiques afin de pouvoir les traiter.

1.1 Plan du mémoire

Dans une première partie, je présente la modélisation des ADD au travers de l'exemple adopté pour le projet CRAFT, ainsi que les résultats obtenus sur cet exemple. Dans une deuxième partie, je présente l'outil IMITATOR, ses différents algorithmes et les modifications que j'ai suggérées ainsi que quelques résultats thóriques sur la cartographie des comportements.

1.2 Rappels sur les Automates Temporisés Paramétriques

Les automates temporisés paramétriques sont une extension des automates temporisés, [1], permettant l'utilisation de paramètres dans les gardes et les invariants à la place de constantes.
On note $P = \{p_1, ...p_n\}$ un ensemble de paramètres et $X = \{x_1, ..., x_n\}$, un ensemble d'horloges.
Un Automate Temporisé Paramétrique (ATP), $\mathcal{A}$, est un 6-uplet $\mathcal{A} = (\Sigma, \mathcal{Q}, q_0, K, I, \rightarrow)$ où Σ est un ensemble fini d'actions, Q l'ensemble des états de l'automate, $q_0 \in Q$ l'état initial de l'automate, K une contrainte sur les paramètres, I l'invariant qui attribue à chaque état une contrainte I_q sur les horloges et les paramètres et $\rightarrow$ la fonction de transition.
Soit $\pi = (\pi_1, .., \pi_n)$ une valuation des paramètres, on note $\mathcal{A}[\pi]$ l'automate obtenu à partir de $\mathcal{A}$ en remplaçant chaque occurence de p_i par π_i.
Un état de $\mathcal{A}$ est un couple (q, C) avec $q \in \mathcal{Q}$ et C une contrainte sur les paramètres et les horloges.
Une éxécution de $\mathcal{A}$ est une alternance de (q_i, C_i) et d'actions respectant la fonction de transition.
La trace de $\mathcal{A}$ associée à une éxécution correspond à la suite de états et d'actions (on ne considère plus les contraintes).

CHAPITRE 2

Arbres de Défaillances Dynamique (ADD)

Les arbres de défaillances sont une représentation graphique des liens de causes à effets des différentes pannes dans un système. Les feuilles de l'arbre sont les éléments de base du système, ceux susceptibles de tomber en panne, les noeuds de l'arbre sont des éléments plus importants du système dont la panne est une fonction de celles des différents composants élémentaires le constituant, et finalement, la racine de l'arbre représente le système dans sa globalité. On distingue les arbres de défaillances, dits statiques, dans lesquels seules les portes "ou", "et" et "k parmi n" peuvent être utilisées, des arbres de défaillances, dits dynamiques, dans lesquels des portes avec un comportement dynamique peuvent intervenir. C'est l'ajout de ces portes qui rend le problème plus complexe et nécessite donc l'utilisation de nouveaux outils pour mener une étude complète du système.

2.1 Les portes dynamiques

Les différentes portes dynamiques que l'on rencontrera par la suite sont présentées ici de manière informelle.

La porte FDEP

Cette porte est constituée d'une entrée, la "trigger", et d'un ensemble de composants élémentaires dépendants. Quand l'évènement "trigger" survient, la porte FDEP provoque la panne simultanée et instantanée de l'ensemble des composants élémentaires dépendants. Une représentation graphique de la porte est donnée en figure 2.1. Les évènements notés A, B sont les évènements élémentaires dépendants.

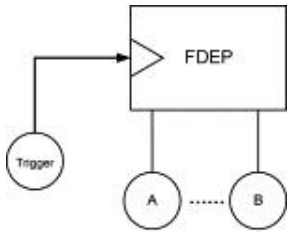


Figure 2.1: Porte FDEP

La porte PAND

Cette porte est une porte à deux entrées, A et B, et une sortie. La porte PAND est équivalente à une porte AND classique avec la condition supplémentaire que la panne de A doit survenir avant celle de B pour que le sous système modélisé par la sortie de la porte PAND passe, elle-aussi, en panne. Une représentation graphique de la porte est donnée en figure 2.2.

Les portes Spare

Les portes Spare (Cold Spare, Warm Spare et Hot Spare) sont utilisées pour modéliser le fait qu'un composant dispose d'un ou plusieurs composants de secours en cas de panne. Une porte

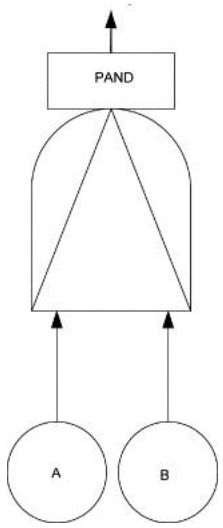


Figure 2.2: Porte PAND

Spare est composée d'une entrée pour le composant principal, de n entrées pour les composants de secours ainsi que d'une sortie. Chaque composant de secours peut être dans 2 états, soit actif, soit dormant. De base, un composant de secours, numéroté i, est dormant ; il ne passe actif que lorsque le composant principal et les composants de secours numérotés de 1 à i-1 sont tombés en panne. Sa probabilité de tomber en panne est alors modifiée suivant la nature de la porte. Elle passe de $\alpha.p$ à p, où p est une constante du composant et $\alpha = 0$, si la porte est une Cold Spare, $\alpha \in]0, 1[$ pour une Warm spare et $\alpha = 1$, pour une Hot Spare.

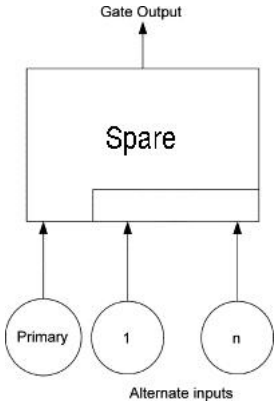


Figure 2.3: Porte Spare

Les séquences de coupes

Une séquence de coupe est une suite de pannes d'évènements de base qui a pour conséquence la panne systémique. On distingue une coupe qui est simplement une liste de pannes d'éléments de base conduisant à la panne systémique, d'une séquence de coupe, impliquant la temporalité des

pannes des évènements de base. Cependant, dans la suite, on utilisera librement coupe ou séquence de coupe pour désigner les deux objets.

2.2 Exemple : L'arbre HCAS

L'arbre HCAS, pour Hypothetical Cardiac Assist System, est présenté dans [7]. Il présente la plupart des difficultés que l'on peut rencontrer dans l'étude des ADD : les 3 types de portes dynamiques, des portes Spares avec un composant de secours commun, mais présente aussi l'avantage de pouvoir se décomposer facilement en sous arbres et permet de se familiariser avec le modèle d'ADD.

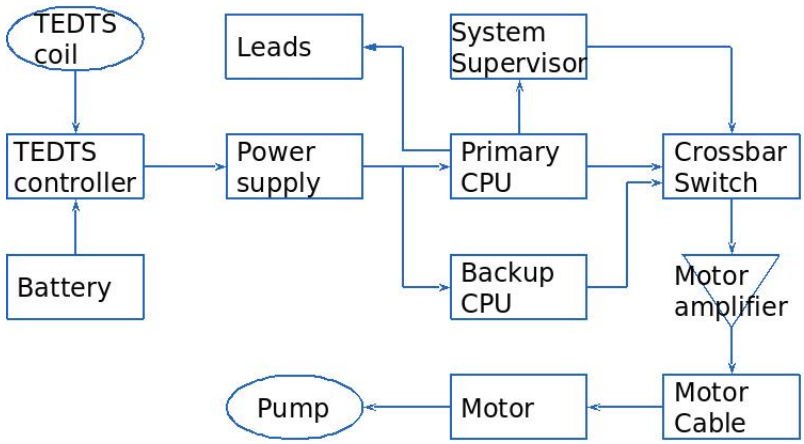


Figure 2.4: Système HCAS, modèle simpliste d'un système d'assistance cardiaque

C'est pour ces raisons qu'il a été choisi comme exemple de départ dans le cadre de ce projet. Une recherche des séquences de coupes avec des outils algébriques, développés par Guillaume Merle, du laboratoire LURPA, donne les séquences de coupes suivantes :

- CS
- SS
- MOTOR.MOTORC
- P.(B<P), B étant dormant lors de sa panne
- B.(P<B), B étant actif lors de sa panne
- PUMP2.(PUMP1<Backup_PUMP).(Backup_PUMP<PUMP2)
- Backup_PUMP.(PUMP2<PUMP1).(PUMP1<Backup_PUMP)

2.3 Modélisation en automates à états finis

La première chose à effectuer est de transformer l'ADD HCAS en un automate à états fini afin d'utiliser un outil de Model Checking, HyTech, afin d'en rechercher les séquences de coupes. La modélisation a été réalisée porte par porte pour permettre la création d'une bibliothèque des différentes portes possibles en vue de l'automatisation de la création de l'automate.
Dans la suite, lorsqu'un "u" est présent dans un état, cela signifie que l'automate ne peux rester dans cet état et doit immédiatement passer à l'un des états suivants, c'est ce qu'on appelle un état urgent. La notation des états est la suivante, un 0 représente l'absence de panne, un 1 signifie que

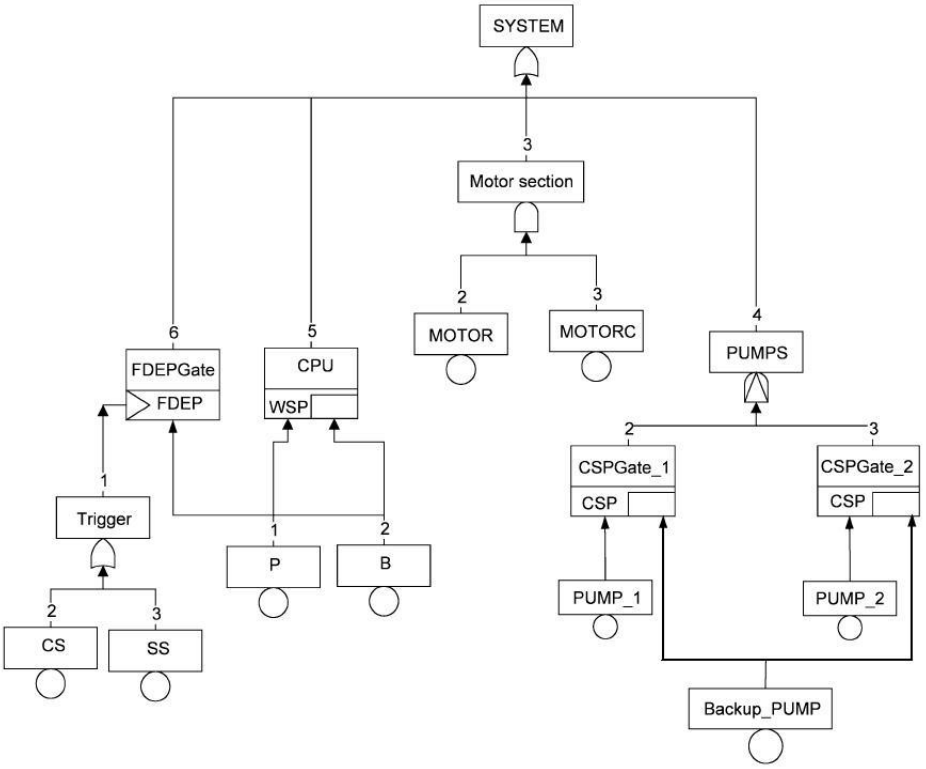


Figure 2.5: Arbre HCAS

le composant correspondant est considéré en panne dans le système. Les états sont notés, de haut en bas, entrée 1, entrée 2, etc. entrée n, sortie 1, sortie 2, etc. Les automates sont présentés sans état initial et sans état final.
Les noms des transitions correspondent au signal de panne émis ou reçu, la notation utilisée dans la suite, sauf indication contraire, est "e"+Numéro, pour une entrée et "s"+Numéro, pour une sortie. Si l'entrée ou la sortie est unique, on ne précisera pas de numéro.

2.4 Les différentes portes

La porte "ou"

La porte présentée en figure 2.6 est la porte "ou" à 2 entrées. De manière similaire on construit les portes "ou" à n entrées. En partant de l'état 000, i.e. les deux entrées et la sortie ne sont pas en panne, deux possibilités peuvent survenir :

- L'entrée 1 passe en panne, dans ce cas, on se rend à l'état 100 urgent, puis immédiatement à l'état 101 en émettant le signal de sortie.
- L'entrée 2 passe en panne, dans ce cas, on se rend à l'état 010 urgent, puis immédiatement à l'état 011 en émettant le signal de sortie.

Dans les deux cas, après l'émission du signal de sortie, on crée les états où la panne complémentaire peut survenir pour des problèmes de synchronisation avec HyTech. Le code Hytech est le suivant :

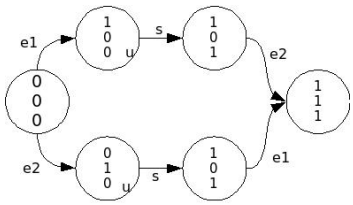


Figure 2.6: Automate pour la porte "ou" à 2 entrées

```
automaton OU
        synclabs: e1, e2, s;
        initially OU_000 & x=0;

loc OU_000: while True wait {}
        when True sync e2 do {x'=0} goto OU_100;
        when True sync s do {x'=0} goto OU_010;

loc OU_100: while x<=0 wait {}
        when x=0 sync e1 goto OU\_101;

loc OU_010: while x<=0 wait {}
        when x=0 sync e1 goto OU\_011;

loc OU_101: while True wait {}
        when True sync s goto OU_111;

loc OU_011: while True wait {}
        when True sync e2 goto OU_111;

        loc OU_111: while True wait {}

end -- OU
```

Dans le code, l'utilisation de l'horloge x permet de simuler le comportement des états urgents. En effet, du fait de la garde $x = 0$, l'automate ne peut rester dans cet état et est forcé à synchroniser l'action, puis à aller dans l'état suivant. La porte AND n'est pas présentée, elle ne présente aucune difficulté supplémentaire par rapport à la porte "ou".

La porte "WSP"

La porte présentée est la porte WSP, c'est-à-dire une porte Spare avec $\alpha \in]0, 1[$. Elle se compose d'une entrée principale et d'une entrée de secours. D'un point de vue purement qualitatif, cette porte n'est rien d'autre qu'un "et" logique. Cependant, une étude quantitative, qui constitue une grande partie de la problématique de ce projet, requiert de savoir si le/les composants de secours sont tombés en panne en étant actif(s) ou dormant(s). C'est pourquoi on sépare les deux branches du "et" logique pour pouvoir garder la notion d'ordre. Une représentation graphique de l'automate pour la porte WSP est présentée en figure 2.7.
La notation $P < B$ signifie que l'entrée principale est tombée en panne avant l'entrée de secours et la notation $B < P$ signifie le contraire.

On peut remarquer que rien n'empêche d'utiliser cet automate pour une porte Hot Spare, HSP, bien que, dans ce cas, l'ordre de panne des différents composants n'influant pas sur les probabilités de pannes, une simple porte "et" suffise.

La porte "CSP"

La porte CSP, porte Spare avec $\alpha = 0$, est la porte dynamique la plus simple à coder en automate. En effet, la panne du composant secondaire ne peut intervenir que si le composant principal est lui même en panne. Le système, ensuite, ne passe en panne que lorsque le composant secondaire passe en panne. L'automate est présenté en figure 2.8.

La porte "FDEP"

La transformation de la porte FDEP en automate nécessite la modification de l'arbre de défaillance en un arbre équivalent. La transformation est présentée en figure 2.9. L'ajout des portes "ou" est nécessaire pour que les évènements P et B restent des évènements élémentaires et que l'automate puisse les traiter comme tels.

La figure 2.10 ne présente que la partie "FDEP" et pas les portes "ou", qui ont déja été décrites auparavant. L'automate est d'une très grande simplicité puisqu'il ne fait que changer le nom d'un signal d'entrée en celui du signal de sortie, c'est la synchronisation ensuite avec les "evt p" et "evt b" qui réalise à proprement parler la fonction de la porte FDEP.

La porte PAND ou Priority AND

Cette porte se comporte de manière proche de celle du "et" classique de par sa définition. En fait il suffit de modifier légèrement le code du "et" pour obtenir la porte PAND. La représentation de l'automate est présentée en figure 2.11. Ici, la condition de priorité est : entrée 1 doit survenir avant entrée 2. L'état "Pas Panne" correspond au fait que si les pannes ne surviennent pas dans le bon ordre, peu importe la suite des évènements, la sortie ne peut passer à l'état "en panne".

Les portes "CSP" avec entrée back up en commun

Dans l'exemple étudié, les portes CSP (CSPGate1 et CSPGate2) possèdent leur entrée de secours en commun. En effet, la "BackupPump" est une pompe de secours prenant le relai de la première des deux pompes à tomber en panne. Cette liaison empêche de traiter les pompes de manière tout à fait indépendante. Il faut tenir compte des signaux qu'émet l'une dans le comportement de l'autre.

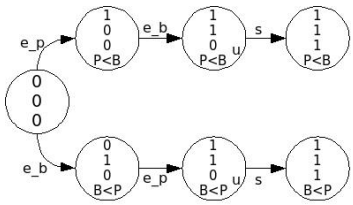


Figure 2.7: Automate pour la porte "WSP" à 1 entrée principale et 1 de secours

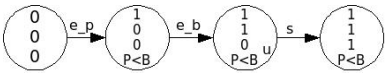


Figure 2.8: Automate pour la porte "CSP" à 1 entrée principale et 1 de secours

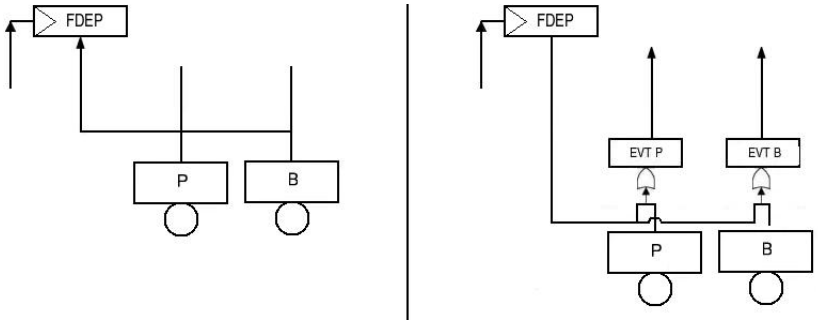


Figure 2.9: Porte originale (gauche) et porte modifiée (droite)

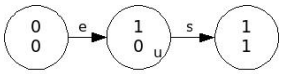


Figure 2.10: Automate pour la porte "FDEP"

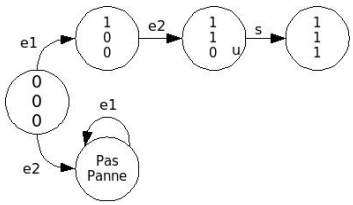


Figure 2.11: Automate pour la porte PAND

Par exemple, après l'initialisation, si jamais l'entrée principale de la seconde porte tombe en panne, elle obtient alors le contrôle de l'entrée de secours commune. La première porte peut se servir de ce signal de panne (entrée principale 2 en panne) pour déclarer localement l'entrée de secours comme en panne. Le comportement de HyTech oblige de déclarer certaines transitions bien que celles-ci n'apportent rien à la porte considérée, c'est pour cela que l'on retrouve plusieurs self-loops dans l'automate.

L'automate est présenté en figure 2.12. Le reste des transitions est assez claire, le modèle est quasiment celui des portes WSP puisque l'entrée de secours peut virtuellement tomber en panne avant le composant principal par la panne du composant principal de la seconde porte.

Les notations utilisées sont les suivantes : entreeP1 désigne l'entrée principale de la porte 1 (celle qui est codée par l'automate présenté), entreeP2 désigne l'entrée principale de la porte qui est reliée à la porte 1, Back up est l'entrée secondaire commune aux deux portes, sortie1 la sortie de la porte 1 et sortie2 la sortie de la porte 2.

Cependant, afin de simuler le fait que les états ne doivent pas être seulement "urgents" mais "prioritaires", impliquant qu'aucune autre synchronisation ne peut avoir lieu, et non pas simplement un simple non écoulement du temps comme le fait la modélisation actuelle, il a fallu ajouter des

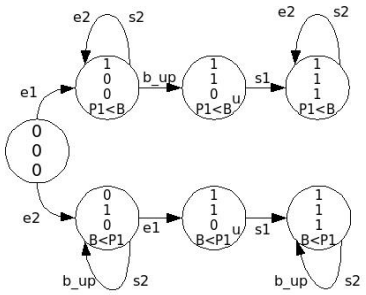


Figure 2.12: Automate pour 2 portes CSP avec entrée de secours en commun

variables discrètes agissant comme des verrous.

2.5 Ajout de variables discrètes

La première variable discrète à être ajoutée pour tout le système est un verrou qui permet de faire une exploration en profondeur des conséquences d'une panne élémentaire. Concrètement, lorsque le signal d'une panne d'un élément de base est émis, la panne est propagée au maximum avant de considérer l'éventualité d'une nouvelle panne.

Il est à noter que la porte FDEP permet la panne instantanée et simultanée des éléments dépendants de cette porte. Cette simultanéité doit être également modéliser. En effet, sous Hytech, les signaux de synchronisation sont émis les uns après les autres. HyTech permet donc de faire émettre un signal de panne d'un des composants dépendant de la porte FDEP, puis d'une panne d'un autre élément indépendant de cette porte, avant de faire émettre la panne des autres éléments dépendants. Ce comportement pose un problème car il permet la création de séquence de coupe non minimales.

Pour régler ce problème, des verrous pour chacun des éléments dépendants de la porte FDEP ont été ajoutés. Cela permet d'émettre tous les signaux de pannes des éléments dépendants de la FDEP puis de propager les pannes grâce au verrou global, puis, si le système n'est pas en panne, de considérer d'autres pannes élémentaires. Ces deux modifications ont permis d'obtenir la liste exacte des séquences de coupes alors qu'une séquence de coupe non minimale était trouvée auparavant.

2.6 Expérimentations et résultats

Pour retrouver l'ensemble des séquences de coupes, on utilise un automate dit d'environnement, dont le code est le suivant :

```
automaton env

        synclabs: PanneSysteme;

        initially paspanne & x=0;

        loc enpanne : while True wait {}
        loc paspanne: while True wait {}
```

```
when True sync PanneSysteme goto enpanne;
```

```
end
```

Le signal "PanneSysteme" étant émis lorsque le système tombe en panne, on va donc chercher les traces qui mènent à l'état "enpanne" pour l'environnement.
Concrètement, on déclare sous HyTech que la région finale que l'on cherche à atteindre est :

```
final_reg :=( loc[env]=enpanne );
```

et que l'état de départ est celui dans lequel tous les composants sont en fonctionnement, c'est-à-dire tous les signaux sont à 0, et tous les verrous sont initialisés à 0.
Un des inconvénients de HyTech est qu'il ne retourne que la trace la plus courte menant à l'état final et non l'ensemble des traces (ce que l'on cherche à obtenir dans notre cas). Pour contourner ce problème, il a fallu, après l'obtention d'une trace, retirer celle-ci de l'état final. Par exemple, la première trace trouvée est MOTOR $\wedge$ MOTORC, on crée alors la région "Obtenues" :

```
Obtenues :=( (loc[motorsection]=motor_111) );
```

et l'on réalise la différence avec la région $final\_reg$

```
final_reg := diff(final_reg, Obtenues);
```

De cette manière, HyTech recherche maintenant une trace ne comportant pas l'état $motor\_111$, i.e. une trace qui ne comporte pas la panne de la partie moteur du système. En continuant de cette manière, jusqu'à ce qu'HyTech ne retourne aucune trace, on obtient les séquences de coupes suivante :

- MOTOR.MOTORC
- P.(B<P), B étant dormant lors de sa panne
- B.(P<B), B étant actif lors de sa panne
- PUMP2.(PUMP1<Backup_PUMP).(Backup_PUMP<PUMP2)
- Backup_PUMP.(PUMP2<PUMP1).(PUMP1<Backup_PUMP)
- SS
- CS

Ce qui correspond exactement aux séquences de coupes de ce système. On a par conséquent réussi à retrouver des séquences de coupes du système. De nombreux problèmes restent à résoudre notamment l'automatisation de la région "Obtenues", la preuve de la correction de la méthode proposée et l'automatisation de la réalisation des automates à partir de la description de l'ADD.

2.7 Conclusion et Travaux Futurs

L'exemple proposé par le projet CRAFT comme exemple d'étude est complètement traîté par la méthode décrite dans ce rapport. Il reste à vérifier que cette méthode peut se généraliser à tous les ADD possibles. Il reste notamment à automatiser la recherche de toutes les séquences de coupes. Actuellement, la région "Obtenues" est actualisée manuellement. Il reste également à automatiser la création de l'automate à partir de l'ADD, notamment la création des différents verrous et la gestion des dépendances entre les différentes portes (comme c'est le cas ici avec les portes partageant une entrée commune). Il reste aussi à fournir une preuve de l'obtention de toutes les séquences de coupe dans le cadre général par cette méthode.

CHAPITRE 3

Les algorithmes de l'outil IMITATOR

IMITATOR est un model checker, au même titre que HyTech, [12], Uppaal [6] etc. Cet outil présente deux aspects. Le premier consiste en l'implémentation de l'algorithme Méthode Inverse[3], le second, la cartographie des comportements [4]. Une partie de mon stage a consisté a me familiariser avec cet outil et les algorithmes utilisés au travers d'exemples. Dans une première partie, je vais présenter les différents algorithmes utilisés dans IMITATOR, puis dans une seconde, je présenterai les modifications algorithmiques pour la méthode inverse et quelques résultats théoriques sur la cartographie des comportements.

3.1 Notations

Les notations et definitions suivantes seront utilisées par la suite. Elles sont directement empruntées de [4].

Définition 1 *Soit π une instanciation des paramètres d'un ATP. Un état (q, C) est dit π-compatible si $\pi \models C$. Sinon il est dit π-incompatible.*

Définition 2 *On note $(\exists X : C)$ la contrainte obtenue de C après élimination des horloges, i.e. $\{ \pi \mid \pi \models C \}$.*

3.2 Méthode Inverse

La méthode inverse pour les ATP consiste à générer une contrainte sur les paramètres telle que l'automate ait un comportement correct. Plus formellement, à partir de π_0, un jeu d'instanciation des paramètres de l'automate, la méthode inverse retourne K_0 une contrainte sur les paramètres telle que $\forall \pi \models K_0$ les traces (alternances d'actions et d'états) de l'ATP sont identiques sous π_0 et sous π. L'algorithme est présenté en algorithme 1.

Algorithm 1: IM$(\mathcal{A}, \pi_0)$

input : Un automate Temporisé Paramétrique $\mathcal{A}$ d'état initial $s_0 = (q_0, K_{init})$
input : Une instanciation π_0 des paramètres
output: Une contrainte K sur les paramètres

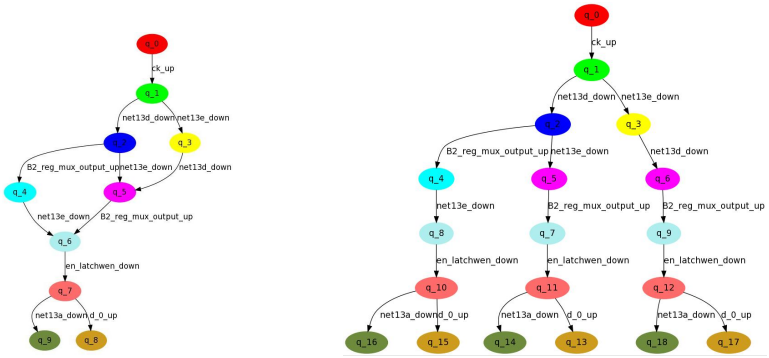
$i \leftarrow 0$; $K \leftarrow true$; $S \leftarrow \{s_0\}$ **while** *true* **do**
  while *Il existe* $(q, C) \in S$ *tel que* $\pi_0 \not\models (\exists X : C)$ **do**
    Sélectionner $(q, C) \in S$ tel que $\pi_0 \not\models (\exists X : C)$;
    Sélectionner $J \in (\exists X : C)$ tel que $\pi_0 \not\models J$;
    $K \leftarrow K \wedge \neg J$;
    foreach $(q, C) \in S$ **do**
      $C \leftarrow C \wedge \neg J$
  if $Post_{(K)}(S) \sqsubseteq S$ **then return** $K \leftarrow \bigcap_{(q,C) \in S} (\exists X : C)$
  $i \leftarrow i + 1$; $S \leftarrow S \cup Post_{(K)}(S)$

De manière informelle, l'algorithme parcourt l'ensemble des états accessibles, puis cherche parmi les nouveaux états ceux qui sont π_0-incompatibles, interdit leur accessibilité en ajoutamt comme contrainte la négation d'une des contraintes qui provoque la π_0-incompatibilité. L'algorithme s'arrête

lorsque plus aucun nouvel état n'est trouvé (i.e. lorsque $\forall s = (q, C) \in Post_{\mathcal{A}(K)}(S),\ \exists s' = (q', C') \in S,\ t.q.\ q = q' \ \wedge\ C = C'$). La correction de l'algorithme est prouvée dans [3].

La mémoire SPSMALL et la nécessité d'une modification algorithmique

La mémoire SPSMALL [8, 5] est un modèle de mémoire dévellopé dans le cadre du projet VALMEM. Cet exemple a permis d'illustrer un comportement théoriquement possible de la méthode inverse mais non observé jusqu'à présent. La méthode inverse garantit que les traces, i.e. l'enchaînement des états et des actions, sont identiques sous π_0 et sous pi (avec $\pi models K_0$ mais ne garantit pas que l'arbre d'accessibilité trouvé sous la méthode inverse soit le même que celui trouvé sous π_0 par accessibilité.
Dans cet exemple, l'arbre est "dupliqué" mais de manières à ce que toutes les traces de cet arbre soit présentes dans l'arbre sous π_0. Une illustration de ce phénomène est présentée en figure 3.1.



(a) Début de l'arbre d'accessibilité en intégralement instancié

(b) Début de l'arbre d'accessibilité en Méthode Inverse

Figure 3.1: Illustration de la duplication des états lors de l'analyse d'accessibilité en Méthode Inverse

Une schématisation de la duplication des états, qui sera utilisée par la suite comme base pour toute explication, est présentée en figure 3.2.

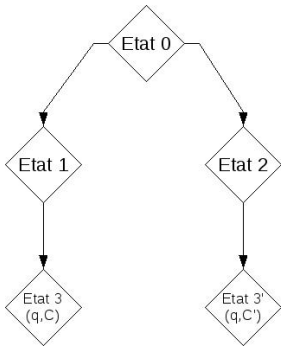


Figure 3.2: Schématisation du phénomène de "duplication"

Intersection à la volée

Dans la figure 3.2, les états 3 et 3' sont les 2 états qui ne sont, dans l'arbre d'accessibilité sous π_0, qu'un seul. L'algorithme Methode Inverse renvoie $K = \bigcap_{(q,C)\in S} \exists X{:}C$ comme contrainte sous laquelle le comportement de l'automate est identique à celui sous π_0. On remarque que, dans cet exemple de la mémoire SPSMALL, on a systématiquement $C \wedge K = C' \wedge K$, où K est la contrainte renvoyée en fin d'éxécution, et même $C \wedge K' = C' \wedge K'$, où K' est $K' = \bigcap_{(q,C)\in\bigcup_{i=1..n}} \exists X{:}C$, avec n le numéro d'itération correspondant à l'apparition des états "3 et 3'". On en conclut donc que tous les états qui succèdent à "3'" ne sont que des copies de ceux qui succède à "3" sous la contrainte K. Le parcours des 2 branches est par conséquent inutile et si l'on avait, à ce point, réalisé le calcul de K', défini comme précédemment, ainsi que la fusion des états devenus identiques par l'ajout de K' à leur contrainte, on aurait pu alors réduire la consommation de temps et de mémoire pour un résultat identique.

Modification de l'algorithme

Pour réaliser la modification proposée, il suffirait de rajouter après la ligne 8, le code présenté en algorithme 2.

Algorithm 2: Modification de l'algorithme de la Méthode Inverse

$K' \leftarrow \bigcup_{(q,C)\in S}(\exists X{:}C)$ **foreach** $(q, C) \in S$ **do**
  $C \leftarrow C \wedge K'$

La correction de cet modification vis-à-vis de l'algorithme original est immédiate puisque si aucun mauvais état ne pouvait être atteint sous l'algorithme original, aucun mauvais état ne peut être atteint maintenant car on considère une restriction des états originaux. De plus, la contrainte renvoyée par les deux algorithmes est identique puisque la seule différence est que le calcul s'opère maintenant à chaque itération plutôt que seulement à la fin.

Résultats sur la mémoire SPSMALL

Pour tester cette modification, on a réalisé l'expérience suivante. A chaque fois que des états semblaient être dupliqués dams cet exemple, on a récupéré les $\exists X{:}C$ de tous les états qui n'ont pas fusionnés, puis on a réalisé l'intersection des $\exists X{:}C$ et on a ajouté les conditions dans l'état initial avant de relancer l'analyse. De cette manière, lorsque l'on arrive à ces mêmes états, les conditions rajoutées provoque l'égalité des états 3 et 3' et donc la fusion.
Le résultat a été concluant puisque cette modification a permis de terminer l'analyse par Méthode Inverse de la mémoire SPSMALL, version sp1, description lsv1, sous l'environnement 1. La version non modifiée de l'algorithme provoqué un abandon de l'algorithme faute de mémoire à l'itération 37, alors qu'ici la recherche se termine à l'itération 97. Les détails des différentes versions et des différents environnements de cette mémoire sont présents dans [5].

Limite de cette modification

Cette intersection à la volée est une amélioration algorithmique qu'il serait intéressant d'implémenter afin de voir si le surcoût de calcul est compensé par la réduction éventuelle du nombre d'états à considérer.
Malheureusement, dans d'autres exemples de cette mémoire (version sp2), cette modification s'est révélée inefficace car les états qui se dupliquent, possèdent le même "$\exists X{:}C$". Par conséquent, la

différence des 2 contraintes se fait sur des morceaux celle-ci portant sur les horloges du système et non sur les paramètres. L'intersection ne provoque alors pas la fusion et la situation reste inchangée.

Travaux futurs

Il reste encore à implémenter cette modification dans le code d'IMITATOR, afin de comparer le temps de calcul à celui de l'ancienne version sur l'ensemble des cas d'étude de cet outil. Il reste également à solutionner le cas où cette modification est insuffisante (les cas d'égalité des "$\exists X$:C" des états dupliqués).

3.3 Cartographie

Présentation de l'algorithme et de la représentation graphique

La cartographie est le second volet de l'outil IMITATOR. La cartographie permet, à partir d'un ATP, $\mathcal{A}$, et d'intervalles bornés sur les valeurs possibles de chacun des paramètres de découper l'espaces des instanciations des paramètres en tuiles. Les tuiles vérifient la propriété suivante, au sein d'une même tuile, le comportement de $\mathcal{A}$ est identique pour toutes les instanciations de cette tuile. Un exemple de la sortie graphique de la cartographie d'IMITATOR est présenté en figure 3.3, les couleurs n'ont aucune signification. La sortie graphique a été dévellopé par Daphné Dussaud. Pour le moment, l'outil cartographie ne fonctionne correctement que pour $V_0 \subset \mathrm{R}^2$.

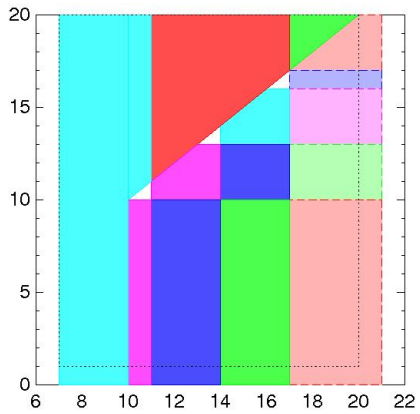


Figure 3.3: Exemple de cartographie et les différentes tuiles

Concrètement, l'algorithme teste tous les points de coordonnées entières dans l'espace des instanciations des paramètres. Si le point appartient à une tuile existante, il passe au suivant, sinon il lance la Méthode Inverse sur cette instanciation et crée la tuile correspondant à la contrainte générée par la Méthode Inverse. L'algorithme termine lorsque tous les points entiers ont été testés. L'algorithme est présenté en algorithme 3.

Limitations de l'algorithme

On remarque immédiatement que l'ensemble des points de V_0 ne sont pas nécessairement cartographiés. En fait, l'algorithme actuel ne garantit que la couverture des points entiers. Une solution pour couvrir plus de zones pourrait être de choisir un pas plus petit mais il est possible d'avoir une infinité de tuiles de taille non nulle et donc l'impossibilité de couvir entièrement V_0,

Algorithm 3: Algorithme de cartographie des comportements de $BC(\mathcal{A}, V_0)$

input : Un ATP $\mathcal{A}$, un rectangle fini $V_0 \subseteq \mathbb{R}^{+M}$
output: *Tuiles* : Liste des tuiles (initiallement vide)

repeat
  Sélectionner un point entier $\pi \in V_0$;
  if π *n'appartient à aucune tuile de Tuiles* **then** Ajouter $IM(\mathcal{A}, \pi)$ à *Tuiles*;
until *Tuiles contiennent tous les points entiers de* V_0 ;

peu importe le pas choisi. Un exemple de ce phénomène de suite de tuiles de taille strictement décroissante est présenté en figure 3.4, tirée de [4]. Il correspond à l'étude du Root Contention Protocol. Les tuiles colorées en rouge correspondent aux tuiles ne vérifiant pas une certaine propriété, celles en bleu, les zones vérifiant cette même propriété.

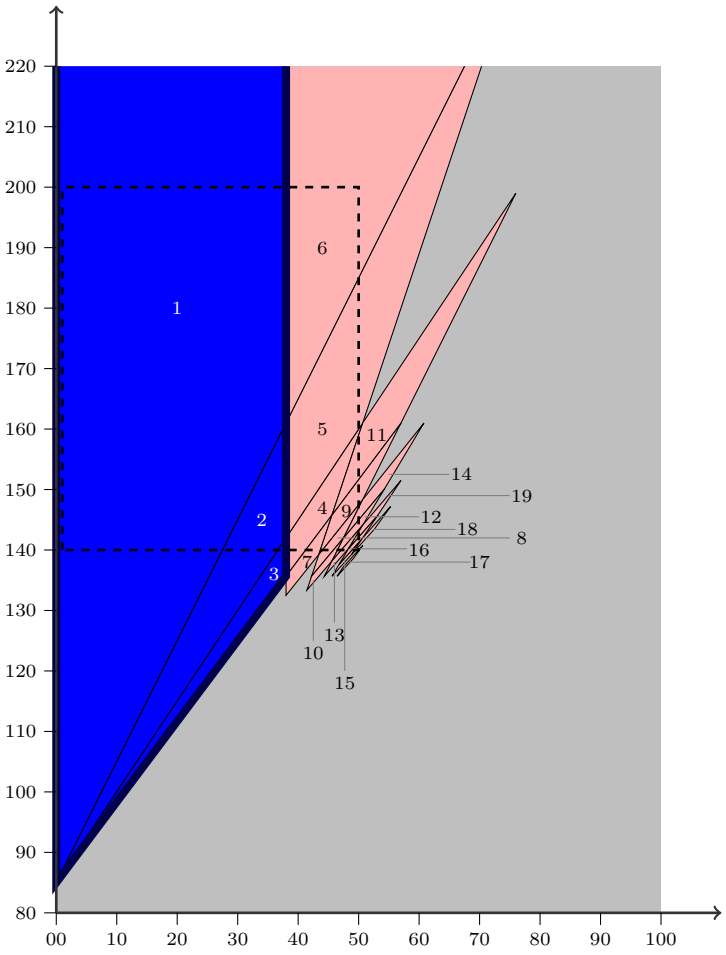


Figure 3.4: Cartographie présentant un nombre infini de tuiles (ici seulement 19 sont représentées)

Durant ce stage, j'ai démontré, pour une certaines classe d'automates, que le nombre de tuiles possibles dans tout rectangle V_0 est fini et donc qu'il est théoriquement possible d'aboutir à la

cartographie complète du rectangle V_0 en un nombre fini d'étapes.

Cartographie finie

Propriété 1 *Les contraintes K générées par la Méthode Inverse sont uniquement des conjonctions de contraintes linéaires sur les paramètres.*

Preuve 1 *La preuve est immédiate en considérant que pour un état $s = (q, C)$, C est une conjonction de garde et d'invariants et sachant que pour un ATP, les gardes et les invariants sont des contraintes linéaires.*
Par conséquent, pour chaque $s = (q, C)$, $\exists X$:C est une conjonction de contraintes linéaires et donc $K = \bigcap_{(q,C)\in S} \exists X$:C est une conjonction de contraintes linéaires.

Une contrainte linéaire est de la forme $\sum_{i\in 1..n}(\alpha_i * p_i)$, avec $\alpha_i \in \mathbb{Z}$ pour tout $i \in I$.

Notation 1 *Soit $(\alpha_1, ...\alpha_n) \in \mathcal{P}^n \subset \mathbb{Z}$, $c \in \mathbb{Z}$. Soit $\prec \in \mathcal{O} = \{<, \leq, >, \geq\}$*
Soit C la contrainte $\sum_{i\in 1..n}(\alpha_i * p_i) \prec c$.
On note $\mathcal{C}_{(\alpha_1,...\alpha_n,c,\prec)} = \{(v_1, ..., v_n) \in \mathbb{R}^n | \sum_{i\in 1..n}(\alpha_i * v_i) \prec c\}$ l'ensemble des points de $\mathbb{R}^n$ qui satisfont C.

Propriété 2 *Soit $V_0 = I_1 \times ... \times I_n$ où I_i est un intervalle borné de $\mathbb{R}$ pour tout $i \in \{1..n\}$.*
Soit $E = \{(\alpha_1, ...\alpha_n, c, \prec) \in \mathcal{P}^n \times \mathbb{Z} \times \mathcal{O} | \mathcal{C}_{(\alpha_1,...\alpha_n,c,\prec)} \cap V_0 \neq \emptyset$ et $\mathcal{C}_{(\alpha_1,...\alpha_n,c,\prec)} \cap V_0 \neq V_0\}$

$$Si\ \mathcal{P}\ est\ borné\ alors\ |E| \leq +\infty$$

E correspond aux contraintes qui définissent une zone non triviale dans V_0.

Preuve 2 *Pour montrer que $|E| < +\infty$ il suffit de montrer que si $(\alpha_1, ...\alpha_n, c, \prec) \in E$ alors c appartient à un intervalle borné de $\mathbb{Z}$. En effet, $\mathcal{P}$ est borné et $|\mathcal{O}| = 4$, donc si c ne peut prendre qu'un nombre fini de valeurs, on aura que E est de cardinal fini.*
Soit $(\alpha_1, ...\alpha_n, c, \prec) \in E$.
$\exists (v_1, ..., v_n) \in V_0 | \sum_{i\in 1..n}(\alpha_i * v_i) \prec c$ car $\mathcal{C}_{(\alpha_1,...\alpha_n,c,\prec)} \cap V_0 \neq \emptyset$
et
$\exists (v'_1, ..., v'_n) \in V_0 | \sum_{i\in 1..n}(\alpha_i * v'_i)(\neg \prec) c$ car $\mathcal{C}_{(\alpha_1,...\alpha_n,c,\prec)} \cap V_0 \neq V_0$
D'où on peut écrire sans perte de généralité que,
$c \leq \sum_{i\in 1..n}(\alpha_i * v_i)$ et $c \geq \sum_{i\in 1..n}(\alpha_i * v'_i)$, quitte à changer les rôles des v_i et des v'_i.
Or, V_0 est borné comme produit cartésien fini de bornés, donc il existe $M \in \mathbb{N}$ tel que $V_0 \subseteq [-M; M]^n$.
Or,

$$c \leq \sum_{i\in 1..n} (\alpha_i * v_i)$$

Ce qu'on peut écrire,

$$c \leq \sum_{i\in 1..n | \alpha_i > 0} (\alpha_i * v_i) + \sum_{i\in 1..n | \alpha_i < 0} (\alpha_i * v_i)$$

$$c \leq \sum_{i\in 1..n | \alpha_i > 0} (\alpha_i * v_i)$$

D'où,

$$c \leq \sum_{i\in 1..n | \alpha_i > 0} (\alpha_i * M)$$

On a également que $\mathcal{P}$ est borné, d'où l'existence de $M' in \mathbb{N}$ tel que $\mathcal{P} \subseteq [-M'; M']$.
D'où,

$$c \leq \sum_{i \in 1..n | \alpha_i > 0} (M' * M)$$

Et on a $|\{i \in 1..n | \alpha_i > 0\}| \leq n$, d'où,

$$c \leq n * M' * M$$

De même, on a que,

$$c \geq -n * M' * M \square$$

On sait qu'une tuile de E est une zone délimitée par un nombre fini de contraintes sur les paramètres. Or il n'y a qu'un nombre fini de contraintes ne définissant pas une zone non trivialle de V_0, d'où le théorème suivant.

Théorème 1 *Si les coefficients des contraintes pouvant être générées par la Méthode Inverse sur l'ATP $\mathcal{A}$ sont bornés alors la cartographie des comportements possède un nomnre fini de tuiles*

Preuve 3 *La preuve est immédiate par la propriété 2.*

Cartographie sans zone d'intérieur vide

Le résultat précédent assure que, dans ce cadre, le nombre de tuiles est fini. Cependant, il peut en exister d'intérieur vide, par exemple un hyperplan, un plan, une droite, voire, un point. Ceci peut être génant si l'on ne posséde pas de fonction qui nous donne un point non encore couvert par les tuiles, ce qui est le cas pour le moment. Dans ce cas, une recherche aléatoire ne trouvera presque-surement pas ce genre de tuiles. Cependant si les tuiles générées par la Méthode Inverse sont d'intérieur non vide et que les coefficients des contraintes linéaires sont bornées alors on a le théorème suivant.

Théorème 2 *Si les coefficients des contraintes pouvant être générées par la Méthode Inverse sur l'ATP $\mathcal{A}$ sont bornés et que les tuiles générées sont d'intérieur non vide alors*

$$\exists \epsilon \in \mathbb{R}^{+*} | \forall\ T \text{ une tuile, } \lambda(T) > \epsilon, \text{ avec } \lambda \text{ la mesure de Lebesgue}$$

Preuve 4 *D'après le théorème 1, on sait qu'il existe un nombre fini de tuiles T.*
Or $\forall T,\ \mathring{T} \neq \emptyset$, i.e. il existe $\pi_T \in T$ tel qu'il existe $\epsilon_T \in \mathbb{R}^{+*}$ tel que $\mathcal{B}(x, \epsilon) \subseteq T$.
Soit $\epsilon' = \min_T(\epsilon_T)$. On a que $\epsilon > 0$ car $\epsilon_T > 0$ pour tout T une tuile et que le nombre de tuiles est fini.
Donc $\mathcal{B}(pi_T, \epsilon') \subseteq T$ pour tout T une tuile.
Par conséquent $\lambda(T) > \epsilon$ avec ϵ la mesure de la boule de rayon ϵ' c

De ce théorème, on peut en déduire l'existence d'un pas tel que, l'algorithme de cartographie appliqué avec ce pas, plutôt que le pas 1 comme dans la version actuelle, terminerait. On pourrait imaginer un algorithme où le pas est calculé à priori grâce à la liste des contraintes, puis la cartographie est réalisé avec ce pas. Cependant, la formule du pas n'est pas encore connu.
Ce résultat d'intérieur non vide pourrait sembler être une condition forte mais il semblerait que la conjecture suivante soit vraie.

Conjecture 1 *Si les invariants des automates sont soit de la forme* $x \leq \sum_{i \in I} u_i$ *et les gardes de la forme* $x \geq \sum_{i \in I} l_i$ *avec* $l_i <= u_i$ *pour tout i, alors les contraintes générées par la Méthode Inverse sont de la forme :*
$\sum_{i \in I}(\alpha_i * l_i) \leq \sum_{j \in J}(\beta_j * u_j) + c$ *ou bien* $\sum_{i \in I}(\alpha_i * u_i) < \sum_{j \in J}(\beta_j * l_j) + c$ *avec les* α_i, β_j *dans* $\mathbb{N}$ *et* $c \in \mathbb{Z}$.

Ce qui semble imposer que les tuiles soient d'intérieur non vide. Si de plus, les constantes α_i, β_j sont bornés alors on aurait un résultat intéressant pour ce type d'automate, qui correspond à tous les automates qui représentent des systèmes dont les temps de transitions sont incertains, ce qui est une large classe des exemples que j'ai rencontrés.

Travaux Futurs

De nombreuses choses restent à faire pour la cartographie des comportements. Le plus intéressant serait d'avoir une fonction qui, à partir de V_0 et des tuiles déjà obtenues, retournerait un point qui n'est pas encore couvert. A défaut, une caractérisation du pas nécessaire dans le cas où la cartographie est finie pourrait permettre de terminer l'analyse dans de nombreux cas.
D'un point de vue plus pratique, il reste à faire une sortie graphique correcte pour plus de 2 paramètres, en faisant par exemple des "tranches" en 2 dimensions suivant certains critères. Il reste également quelques bugs à corriger (notamment la mauvaise gestion des zones lorsque les paramètres peuvent être négatifs), ainsi que ce qui semble être un problème d'arrondi dans le calcul des sommets des tuiles.

Chapitre 4

Conclusion et perspectives

Durant ce stage, j'ai pu me familiariser avec le domaine du model checking au travers d'études de cas. J'ai notamment modéliser un ADD en automate à états finis et j'en ai tiré les séquences de coupes pour le projet CRAFT. J'ai également étudié quelques modélisations au travers d'IMITATOR, notamment la mémoire SPSMALL, qui m'a permis de proposer une modification algorithmique afin de terminer l'analyse en Méthode Inverse dans un plus grand nombre de cas. J'ai également étudié un autre protocole, le Bounded Retransmission Protocol, qui produit des traces infinies. IMITATOR n'est pas conçu pour traîter les automates produisant des traces infinies. Cependant, les traces de cet automates sont cycliques et il est possible de montrer expérimentalement, qu'à partir d'une certaine profondeur, aucun nouvel état π_0-incompatible ne peut être trouvé. L'implémentation d'un algorithme détectant la cyclicité des traces et prouvant la non π_0-incompatibilité des états suivants, pourraient permettre de traiter ce genre de cas, et encore une fois, d'élargir la classe des problèmes pouvant être traités par IMITATOR.
En fin de stage, je me suis intéressé à la partie cartographie d'IMITATOR. Cette partie est la moins étudiée pour le moment et de nombreux résultats restent à être trouvés. Les résultats théoriques permettent de savoir que la cartographie peut aboutir en un nombre fini d'étapes, dans certains cas, mais ils ne proposent pas de méthode pour y arriver. Il reste encore à trouver, éventuellement, un moyen de détecter des zones non cartographiées pour s'ataquer au cas général et pouvoir détecter l'infinité éventuelle d'une cartographie afin d'en arrêter l'étude sous des critères à définir.
Il reste également à améliorer la sortie graphique de la cartographie, notamment en assurant sa correction dans le cas de plus 2 paramètres, corriger certaines erreurs comme la gestion des zones comportant des points de coordonnées négatives et certains artefacts dans la représentation graphique (pentes anormales dues peut-être à des erreurs d'arrondis).
Une amélioration intéressante pourrait être de tester un représentant de chacune des tuiles contre une propriété à vérifier pour démarquer l'espace entre les points qui vérifient cette propriété et ceux qui ne la vérifient pas (par exemple par un appel à Uppaal).

Remerciements

Je tiens à remercier Laurent Fribourg, pour m'avoir encadré pendant ce stage et pour ces conseils avisés. Je souhaite remercier également Étienne André pour avoir patiemment répondu à toutes mes questions sur IMITATOR, la méthode inverse, la cartographie, la syntaxe LaTeX etc. ainsi que toutes les personnes du LSV pour leur accueil.

Bibliographie

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, April 1994.

[2] Étienne André. IMITATOR : A tool for synthesizing constraints on timing bounds of timed automata. In Martin Leucker and Carroll Morgan, editors, *Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing (ICTAC'09)*, volume 5684 of *Lecture Notes in Computer Science*, pages 336–342, Kuala Lumpur, Malaysia, August 2009. Springer.

[3] Étienne André, Thomas Chatain, Emmanuelle Encrenaz, and Laurent Fribourg. An inverse method for parametric timed automata. *International Journal of Foundations of Computer Science*, 20(5) :819–836, October 2009.

[4] Étienne André and Laurent Fribourg. Behavioral cartography of timed automata. In Antonín Kučera and Igor Potapov, editors, *Proceedings of the 4th Workshop on Reachability Problems in Computational Models (RP'10)*, volume 6227 of *Lecture Notes in Computer Science*, Brno, Czech Republic, August 2010. Springer. To appear.

[5] Abdelrezzak Bara and Emmanuelle Encrenaz. Traduction automatique de descriptions vhdl en automates temporisés, projet valmem, 2010.

[6] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems : 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[7] Hichem Boudali and Joanne Bechta Dugan. A bayesian network reliability modeling and analysis framework. phd dissertation. *IEEE Transactions on Reliability*, 55 :86–97, 2005.

[8] Remy Chevallier, Emmanuelle Encrenaz-tiphÃ¨ne, Laurent Fribourg, and Weiwen Xu. Timing analysis of an embedded memory : Spsmall. In *In 10th WSEAS International Conference on Circuits*, 2006.

[9] Projet CRAFT. Craft critical risks analysis by fault trees. 2009.

[10] P. R. D'Argenio, J. P. Katoen, T. C. Ruys, and G. J. Tretmans. The bounded retransmission protocol must be on time ! In H. Brinksma, editor, *Proceedings of the Third Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *Lecture Notes in Computer Science*, pages 416–431. Springer-Verlag, 1997.

[11] Goran Frehse. Phaver : Algorithmic verification of hybrid systems past hytech. In *HSCC* [15], pages 258–273.

[12] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-toi. A user guide to hytech, 1995.

[13] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech : a model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1 :110–122, 1997. 10.1007/s100090050008.

[14] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[15] Manfred Morari and Lothar Thiele. Hybrid systems : Computation and control, 8th international workshop, hscc 2005, zurich, switzerland, march 9-11, 2005, proceedings. In *HSCC*, volume 3414 of *Lecture Notes in Computer Science*. Springer, 2005.

[16] Kevin J. Sullivan and Joanne Bechta Dugan. Formal semantics of models for computational engineering : A case study on dynamic fault trees. In *In Proceedings of the International Symposium on Software Reliability Engineering*, pages 270–282. IEEE, 2000.

[17] N. H. Roberts William E. Vesely. *Fault Tree Handbook*. US Government Printing Office, 1981.

[18] Sergio Yovine. Model checking timed automata. In *In European Educational Forum : School on Embedded Systems*, pages 114–152. Springer-Verlag, 1998.