

Rapport d'IRL: Implémentation de l'algorithme MAS avec la méthode B

JONATHAN JULOU

Encadrants: Akram Idani, Roland Groz

Equipe VASCO, LIG

25 mai 2020

TABLE DES MATIÈRES

I	Introduction	2
II	Inférence de modèles	3
III	Algorithme L*	3
i	rappels utiles de théorie des langages	4
ii	Table d'observations	4
iii	Création de l'automate à partir de la table d'observations	5
iv	Membership query	5
v	Equivalence query	5
vi	Algorithme complet	6
IV	Algorithme MAS	7
i	Formalisation des traces d'exécution	7
ii	Particularités de MAS par rapport à L*	7
iii	construction du diagramme état-transition	8
iii.1	Définition des états	9
iii.2	Définition des transitions	9
iii.3	Etat initial et états finals	9
iv	particularités pour une application domotique	9
V	Implémentation en B	10
i	outils	10
i.1	Eclipse Modeling Tools	10
i.2	Meeduse	10
i.3	AtelierB	10
i.4	ProB	11
ii	quelques bases de syntaxe B	11
iii	Implémentation de L*	12
iv	Implémentation de MAS	13
VI	Conclusion	14
i	Résumé	14
ii	Bilan et perspectives	14
ii.1	performances	14
ii.2	preuve de l'algorithme implémenté	14
ii.3	autres algorithmes possibles	14
VII	Bibliographie	15

I. INTRODUCTION

Le titre original de cette IRL était *Mix DSL et Méthode B pour la preuve et l'exécution d'un algorithme d'inférence de modèles*. L'objectif était donc double. D'une part il s'agissait de s'initier à la méthode B et aux techniques de modélisation par un langage dédié pour implémenter un algorithme d'inférence de modèles. D'une autre, on voulait avoir des garanties sur l'exécution.

L'inférence de modèles est une branche du machine learning cherchant pour un système à en obtenir un modèle approché en observant son comportement. Cela prend tout son sens par exemple dans le cas où on cherche à analyser un système qui comporte des "boîtes noires", des composants dont on ignore la nature, mais pour lesquels on peut obtenir des traces d'entrées-sorties. Dans les faits, ces boîtes noires peuvent être des composants au fonctionnement volontairement flou, mal documentés, ou encore des composants extérieurs qui peuvent interagir avec le système. Cet exemple est un des principaux axes d'étude de l'équipe VASCO.

La méthode B est un processus de développement formel centré autour du langage de programmation B. L'objectif est de pouvoir concevoir une spécification du logiciel que l'on veut développer, et en tirer un programme informatique qui découle directement des spécifications. Le principe général est d'écrire un programme B de très haut niveau, appelé "machine abstraite", qui spécifie notamment toutes les contraintes que doit respecter le logiciel. Il est ensuite raffiné, c'est à dire que des machines B de plus en plus concrètes sont codées de sorte à respecter les contraintes de la machine abstraite. A la fin, si on a suffisamment raffiné, il est possible d'automatiquement traduire le code B en code compilable par une transformation prouvée correcte, et ainsi obtenir un logiciel utilisable. Le but est de pouvoir relier chaque raffinement à la machine abstraite par une preuve de consistance. Cela se fait au moyen d'un invariant : un ensemble de propriétés qui doivent être vérifiées tout au long du programme.

L'environnement logiciel est principalement centré sur l'outil Meeduse développé par Akram Idani. C'est un plugin de l'Eclipse Modelling Framework qui permet de faire des preuves formelles sur des DSLs (langages dédiés), tout en ayant une fonctionnalité d'animation pas à pas du DSL sur un exemple. Pour cela, le DSL est traduit en machine B. Ensuite Meeduse peut se connecter à ProB pour animer le modèle. On peut aussi éditer le code B, ou le raffiner, et faire des preuves sur le code B dans

AtelierB [8].

La majeure partie de l'IRL aura consisté à essayer d'adapter un algorithme d'inférence de modèles en B, pour être utilisable dans Meeduse et avoir des garanties simples sur l'exécution, dans une lointaine optique de prospection pour une potentielle application domotique. L'intérêt d'une implémentation en B est de chercher à se convaincre que l'implémentation correspond bien à l'algorithme.

Dans une première partie, un algorithme d'inférence de modèles, MAS, est présenté. Nous verrons d'abord un algorithme plus simple sur lequel il est basé, puis comment MAS est construit par dessus. Dans une seconde partie, la méthode B est introduite, suivie d'une discussion sur l'implémentation de MAS en B. Il y a enfin un bref résumé de l'IRL et une ouverture qui porte un regard critique sur les résultats obtenus.

II. INFÉRENCE DE MODÈLES

L'algorithme d'inférence de modèles auquel on s'intéresse s'appelle *Minimally Adequate Synthesizer* (MAS). Il a été décrit par Mäkinen et Systä [1]. Il s'applique dans le cas où on veut modéliser un système pour lequel on peut facilement obtenir des diagrammes de séquence, afin d'en tirer un automate décrivant le système. Un diagramme de séquence est un ensemble de lignes de vie, une par composant du système, entre lesquelles on peut placer des messages que s'échangent les composants.

L'algorithme suppose que l'on se place du point de vue d'un seul composant, qui sera donc le composant modélisé par l'automate. On suppose que les autres composants ont un comportement déterministe par rapport aux actions du composant principal, de sorte à avoir une certaine causalité entre messages envoyés et messages reçus. Par exemple, sur la **Figure 1**, on modéliserait l'humain.

L'exemple du réveil décrit dans le papier original [1] est très bien, mais pour simplifier encore plus, décrire facilement chaque étape, et se rapprocher de la domotique, on va prendre un autre exemple. Ici on peut imaginer que l'on se place juste après que le réveil ait sonné, et on décrit quelques actions que l'humain fait avant de sortir de la chambre. L'objectif implicite sera d'obtenir un automate qui éteint forcément la lumière avant de sortir.

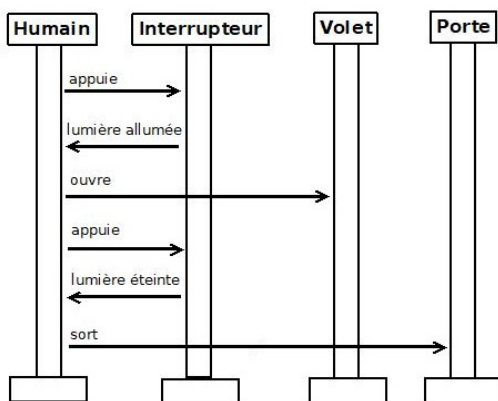


Figure 1 – diagramme de séquence pour l'exemple

En sortie, on obtient un diagramme état-transition de Harel [7], que l'on appellera aussi par anglicisme *statechart diagram*. Il s'agit d'une extension par rapport aux automates état-transition classiques (machines de Moore ou de Mealy notamment) ajoutant la notion de hiérarchie, de communication, et d'exécution concurrente. Cette représentation est donc très bien adaptée à la description de systèmes complexes, et a été intégrée dans le Unified Modeling Language (UML).

Cette représentation permet donc des choses très avancées, comme découper un système en sous-systèmes ou l'exécution parallèle d'automates. Du côté de MAS, on a vu que l'on se restreignait à un seul composant. La caractéristique de ces diagrammes qui nous intéresse vraiment ici est de pouvoir définir des transitions par rapport à des messages reçus, et des états qui effectuent une action, ou envoient un message, comme sur la **Figure 2**. Un autre intérêt est que le diagramme obtenu peut être intégré à un diagramme plus grand, pour y remplacer une boîte noire par exemple.

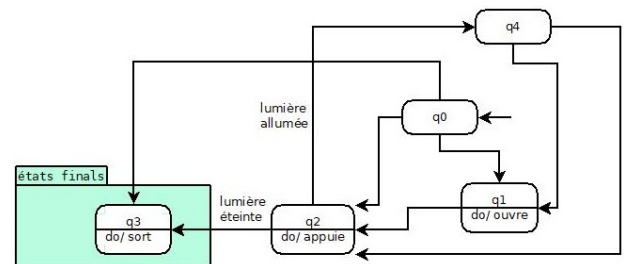


Figure 2 – diagramme état-transition de Harel final obtenu pour l'exemple

III. ALGORITHME L*

MAS se base sur l'algorithme L* présenté par D. Angluin [4]. À première vue L* n'a pas grand chose en commun avec ce qui a été évoqué au dessus, mais c'est le cœur de MAS. Il convient donc d'expliquer d'abord le fonctionnement de L*.

L* permet d'obtenir un automate fini déterministe accepteur pour un langage régulier inconnu U. Le principe de L* est de réaliser un apprentissage interactif du langage, c'est à dire en permettant à l'algorithme (l'élève) d'interagir avec un "enseignant". L'enseignant connaît un certain nombre de choses sur le langage U, de sorte que l'élève peut lui demander si un mot appartient à U, c'est une "membership query". Quand l'élève en sait assez, il propose un automate, et l'enseignant peut alors soit l'accepter, soit le réfuter en donnant un contre-exemple, c'est une "equivalence query".

cette approche, appelée inférence active, s'oppose à l'inférence passive, dans laquelle on va plutôt analyser une base de données de mots du langage. Un avantage de l'inférence active en terme de garanties est que comme on consulte directement le système pendant le processus de synthèse, on limite les généralisations indésirables.

i. rappels utiles de théorie des langages

Definition 1. Alphabet

Ensemble quelconque

Ses éléments sont appelés les **lettres**

Definition 2. Mot

Séquence finie de lettres

On note le mot vide λ

On peut concaténer deux mots. On note cette opération "."

On note A^* l'ensemble de tous les mots possibles sur un alphabet A

("*" est l'**étoile de Kleene**, telle que pour un ensemble E , E^* désigne l'ensemble de tous les mots obtenus par un nombre quelconque de concaténations de mots de E)

Definition 3. Préfixe

On appelle préfixe s d'un mot w tout mot tel qu'il existe un mot r tel que $w = s.r$

Definition 4. Suffixe

On appelle suffixe r d'un mot w tout mot tel qu'il existe un mot s tel que $w = s.r$

Definition 5. Langage

On appelle langage sur un alphabet A tout sous-ensemble de A^*

Definition 6. Langage régulier

Il y a 3 définitions équivalentes de ce que sont les langages réguliers sur un alphabet A :

- Langages décrits par les expressions rationnelles
- Langages construits à partir de A et \emptyset en utilisant seulement l'union ensembliste, la concaténation, et l'étoile de Kleene
- Langages reconnus par les automates finis

C'est la dernière définition qui nous intéresse dans le cadre de l'algorithme L^* .

Definition 7. Automate fini déterministe

Un automate fini est un quintuplet (Q, V, q_0, δ, F) tel que :

- Q est un ensemble fini d'états
- V est le vocabulaire d'entrée
- q_0 est l'état initial
- δ est la fonction de transition $\delta : Q \times V \mapsto Q$
- F est l'ensemble des états finals

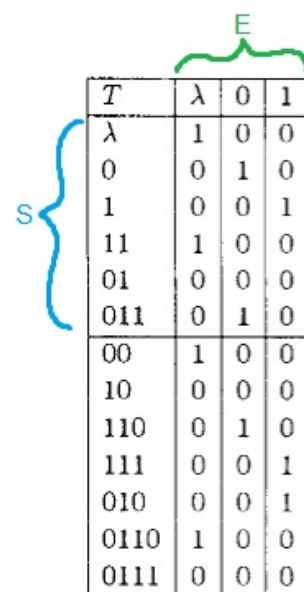
ii. Table d'observations

En reprenant la métaphore de l'élève qui apprend, il faut qu'il se souvienne de ce qu'il sait. C'est ce rôle que remplit la table d'observation, que l'on notera T .

Pour les notations, on cherche à inférer le langage U inconnu sur l'alphabet A . On notera S un ensemble de mots fermé par le préfixe (tout préfixe d'un mot de S est dans S) et E un ensemble de mots fermé par le suffixe. On va stocker dans la table l'appartenance ou non des mots de $S \cup (S.A)$ au langage U .

Intuitivement, la table d'observations est une structure de donnée construite pour être à mi-chemin entre un ensemble de mots acceptés et refusés, et un automate. On peut imaginer les mots de S comme une suite de lettres qui nous amène dans un certain état, et on se demande dans quel état (vu comme un ensemble de continuations possibles du mot ici, d'où les suffixes dans E) on arriverait si on leur ajoutait une lettre, ce qui se rapproche d'une transition dans un automate. C'est $S.A$.

Habituellement, la table d'observation est disposée comme ci-dessous, avec les lignes indexées par les éléments de S d'abord (ceux qui nous intéressent le plus), puis ceux de $(S.A)$ privé de S dans un deuxième temps. Les colonnes sont indexées par les éléments de E . Ainsi, le booléen situé à l'intersection de la ligne s et de la colonne r donne l'appartenance de $s.e$ à U . On peut alors noter $T(s.e) = 1$, et 0 sinon.



		E		
	T	λ	0	1
S {	λ	1	0	0
	0	0	1	0
	1	0	0	1
	11	1	0	0
	01	0	0	0
	011	0	1	0
	00	1	0	0
	10	0	0	0
	110	0	1	0
	111	0	0	1
	010	0	0	1
	0110	1	0	0
	0111	0	0	0

Figure 3 – exemple de table d'observation extraite de [4], dans un exemple sur l'alphabet $\{0,1\}$

Definition 8. Row

pour toute ligne s de T , on note

$$row(s) = \{e \in E \mid T(s.e) = 1\}$$

Cette quantité peut être vue de manière équivalente comme la suite de booléens composant la ligne.

On a besoin de deux propriétés sur la table d'observation pour pouvoir en tirer un automate fini déterministe :

Definition 9. Table fermée

$$\forall t \in S.A, row(t) \neq \emptyset, \exists s \in S, row(s) = row(t)$$

Cette propriété sera utile comme les états de l'automate généré seront les rows des éléments de S . Ainsi sans cette propriété, on pourrait avec certaines lettres partir d'un état de l'automate vers une row qui n'est pas un état, et donc quitter l'automate en quelque sorte.

Definition 10. Table consistante

$$\forall s_1, s_2 \in S, row(s_1) = row(s_2), \forall a \in A,$$

$$row(s_1.a) = row(s_2.a)$$

Cette propriété sera utile comme les états de l'automate généré seront les rows des éléments de S . Ainsi sans cette propriété, une même lettre pourrait apparaître sur deux transitions différentes partant du même état.

Tout l'algorithme consistera ensuite à itérer sur la table d'observation en ajoutant des éléments pour corriger ces deux propriétés, jusqu'à ce que les deux soient vraies. Quand c'est le cas, la table correspond à un automate raisonnable. On peut alors passer à la création de l'automate.

On corrige la fermeture en ajoutant un des t qui pose problème dans S , ainsi on a bien pour tous les autres de même row l'égalité de row avec un élément de S .

On corrige la consistance en cherchant la colonne e où ça pose problème (on n'a pas l'égalité de $T(s_1.a.e)$ et $T(s_2.a.e)$). On ajoute alors $a.e$ dans E . La colonne est donc corrigée, et on a bien égalité des row .

iii. Création de l'automate à partir de la table d'observations

Comme évoqué précédemment, les états de l'automate seront les différentes row possibles.

Pour construire l'automate (Q, V, q_0, δ, F) , on suit ces règles :

- $Q = \{row(s) \mid s \in S\}$
- $V = A$
- $q_0 = row(\lambda)$
- δ est construit tel que
 $\forall s \in S, \forall a \in A, \delta(row(s), a) = row(s.a)$
- $F = \{row(s) \mid s \in S \text{ et } T(s) = 1\}$
 (le $T(s) = 1$ équivaut à $\lambda \in row(s)$)

Dans [4], D.Angluin démontre qu'en suivant ces règles on obtient bien un automate qui va accepter (resp. refuser) les mots acceptés (resp. refusés) par la fonction T qui donne l'appartenance d'un mot de la table d'observations au langage inconnu U .

iv. Membership query

On a vu la structure de la table d'observation T , les mécanismes qui permettent de l'étendre (correction de fermeture et consistance), et comment on en tirait un automate. Cependant, il reste à construire la fonction T , qui dit si un mot appartient ou pas à la table.

C'est ici qu'interviennent les membership queries. Le principe est simple. On a une case de la table correspondant un préfixe s et un suffixe e . On va simplement demander à l'enseignant si $s.e$ appartient ou non au langage inconnu U .

Le nombre de membership queries posées par L^* pour remplir T peut être assez conséquent. Un mécanisme qui permet de le réduire est que si $s.e$ apparaît déjà dans la table sous une autre combinaison préfixe-suffixe, on connaît déjà $T(s.e)$.

v. Equivalence query

Une fois qu'un automate a été construit, il faut le tester. Pour cela on soumet une equivalence query à l'enseignant. Il doit accepter ou refuser l'automate.

Si il accepte, on s'arrête là.

Si il refuse, il doit donner un contre-exemple qui soit appartient à U mais n'est pas reconnu par l'automate, soit n'appartient pas à U et est reconnu par l'automate. Ce contre-exemple ainsi que tous ses préfixes sont ajoutés à S et le processus de recherche de fermeture et consistance de la table reprend.

vi. Algorithme complet

Algorithm 1: L^*

Result: automate acceptant U

initialiser S et E à $\{\lambda\}$;

while *Enseignant non satisfait* **do**

while *T n'est pas fermée et consistante* **do**

 instructions;

if *T non consistante* **then**

 trouver s_1 et s_2 dans S , a dans A et

r dans R tels que

$row(s_1) = row(s_2)$ et

$T(s_1.a.e) \neq T(s_2.a.e)$

 ajouter $a.e$ à E

 étendre T à $(S \cup S.A).E$ en utilisant
 les membership queries

end

if *T non fermée* **then**

 trouver s_1 dans S , a dans A tels que

$\forall s \in S, row(s_1.a) \neq row(s)$

 ajouter $s_1.a$ à S

 étendre T à $(S \cup S.A).E$ en utilisant
 les membership queries

end

end

 faire une conjecture d'accepteur M

if *l'enseignant refuse avec un contreexemple t*

then

 ajouter t et tous ses préfixes dans S

 étendre T à $(S \cup S.A).E$ en utilisant les
 membership queries

else

 On sort de la boucle principale,
 l'enseignant est satisfait

end

end

IV. ALGORITHME MAS

i. Formalisation des traces d'exécution

L^* est très abstrait, et il faut donc pouvoir modéliser une exécution du système par un mot d'un langage pour l'utiliser dans le cadre de MAS. La transformation proposée par E. Mäkinen et T. Systä est la suivante :

On sépare les messages reçus des messages envoyés, puis on les regroupe par paires. Ainsi, à message envoyé ei , on associe un message reçu ej . On représente alors cet échange par le couple (ei, ej) . Or pour chaque message envoyé, on a soit un message reçu, soit rien. De même pour un message reçu, soit il répond à un message envoyé, soit il ne répond à rien. On représente ce "rien" par un message artificiel NULL.

On représente la fin de l'exécution par la réception d'un message artificiel VOID. Le papier définit aussi DESTROYED, qui signifie qu'on a arrêté explicitement le composant, mais il est équivalent à VOID en pratique.

On définit alors un langage dont les lettres sont ces couples de messages. L'exécution est alors entièrement modélisée par une suite de ces lettres. Par exemple, dans le cas de la **Figure 1**, si on note on l'allumage de la lampe et off son extinction, la trace d'exécution serait le mot $(appuie, on)(ouvre, NULL)(appuie, off)(sort, VOID)$

ii. Particularités de MAS par rapport à L^*

Avec cette représentation des traces d'exécution, on pourrait directement utiliser L^* pour obtenir un automate. Cependant, L^* est réputé assez inefficace car il requiert un nombre important de membership queries, ce qui va énormément ralentir le procédé, que ce soit un humain qui réponde ou que l'on questionne directement le système.

La structure de MAS reste très proche de celle de L^* . L'algorithme ci-dessus est d'ailleurs aussi celui de MAS. Mais dans le cadre de traces d'exécution, un certain nombre d'optimisations sont possibles. Les auteurs de MAS les regroupent sous le nom de "*state determinism*" [2]. En effet, elles découlent du fait que l'on analyse un système réel et dont les interactions sont supposées déterministes. Ainsi on a ces trois règles :

- Si pour un préfixe d'exécution $(e_0, e_1) \dots (e_{i-1}, e_i)$, il est suivi par un suffixe $(e_{i+1}, e_{i+2}) \dots (e_{n-1}, e_n)$, alors tout autre suffixe d'exécution doit commencer par le même e_{i+1} , sinon la réponse a un message du composant principal ne serait pas déterministe.
- Si la lettre $(e_i, NULL)$ apparaît dans un mot du langage, on ne peut pas avoir de mot contenant (e_i, e_j) avec $e_j \neq NULL$, car le NULL indique que e_i ne provoque pas de réponse immédiate.

La deuxième condition se généralise en :
Si $(e_i, e_f)(e'_i, e'_f)$ et $(e_i, e_f)(e'_i, e'_f)$ apparaissent dans des mots du langage, alors $e'_f = e''_f$

On requiert aussi une condition implicite qui est que l'on ne peut avoir plus d'une fois le message VOID dans un mot du langage, car cela signifierait que l'on termine plusieurs fois, donc que l'on continue après avoir terminé.

On va rajouter ces vérifications dans les membership queries, de sorte que si une condition de state determinism est violée, on ne propose pas la query. En pratique, cela réduit énormément le nombre de membership queries et l'approche devient totalement abordable, même avec des systèmes assez grands.

Une autre différence assez importante est que l'on commence MAS avec une trace donnée en entrée, avant la moindre membership query. Plutôt que d'initialiser l'ensemble S des entrées fermé par le préfixe à λ , on va mettre tous les préfixes de cette trace initiale. Cela permettra de déjà avoir une bonne base pour modéliser le système si cette trace est suffisamment exhaustive, et aussi de pouvoir profiter dès le début des optimisations ci-dessus. L'algorithme MAS réalise donc une inférence mi-active, mi-passive, même si le cœur de l'algorithme est interactif.

Enfin, par les contre-exemples, il est possible de rajouter de nouvelles lettres inconnues dans l'alphabet. Ce cas de figure correspond à un échange de messages non-prévu par le diagramme de séquence initial mais possible dans le système. Cela ne pose pas de problème, on ajoute juste dans la table d'observations tous les préfixes de mots de S suivis de cette nouvelle lettre.

- Si un mot ne se termine pas par une lettre dont la sortie est VOID, l'exécution ne termine pas, donc le mot ne peut appartenir au langage.

La méthode proposée pour transformer l'automate accepteur que l'on a en diagramme état-transition est la suivante :

iii.1 Définition des états

Les états du diagramme état-transition seront a priori soit des actions effectuées par le composant étudié, soit des états sans actions, par exemple pour une attente.

Pour chaque état q de notre automate, on regarde les couples constituant toutes les transitions sortantes.

- Si ils commencent tous par le même message e_i , on crée un état qui effectue l'action do/e_i .
- Si ils commencent tous par un message NULL, on crée un état sans action.
- Si ils commencent par n messages différents, on crée un état sans action, ainsi que n états effectuant les actions correspondant aux messages, puis on crée des ϵ -transitions permettant de passer de notre état vide aux n états d'actions.

iii.2 Définition des transitions

Les transitions seront les messages reçus par le composant étudié. Cela correspond donc au second message de chaque couple sur les transition dans notre automate.

Pour chaque transition (e_i, e_j) de q à q' dans notre automate, on crée une transition entre chaque état "feuille" de q et chaque état "racine" de q' pour le message e_j .

Par état "feuille" on entend les n états d'action dans le 3^e cas ci dessus.

Par état "racine" on entend l'état sans action menant aux autres par ϵ -transition dans le 3^e cas ci dessus.

Dans les autres cas que le 3^e, les deux désignent juste le nouvel état créé.

iii.3 Etat initial et états finals

L'état initial du diagramme état-transition sera l'état racine généré par l'état initial de l'automate.

Les états finals du diagramme état-transition seront les états feuilles générés par les états finals de l'automate.

En appliquant cette méthode à l'automate de l'exemple, on obtient ce diagramme état-transition :

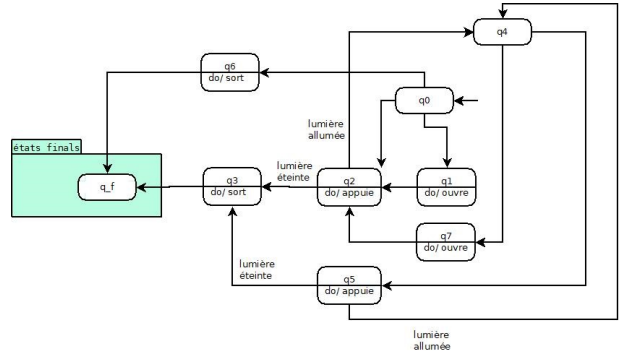


Figure 7 – diagramme état-transition final obtenu pour l'exemple

La création de l'état q_0 dans le diagramme état-transition à partir de l'état initial $\{[(sort, VOID)], [(appuie, eteint)(sort, VOID)], [(ouvre, NULL)(appuie, eteint)(sort, VOID)]\}$ est un bon exemple du 3^e cas de figure quand on définit les états.

Comme on avait trois premiers messages différents dans les transitions sortantes, *appuie*, *sort* et *ouvre*, on a dû créer 3 autres états, respectivement q_2 , q_6 et q_1 , atteignables depuis q_0 par ϵ -transition.

On constate que certains états ne servent à rien, comme q_6 qui sert de passage entre deux ϵ -transition, et que des états sont équivalents, comme q_2 et q_5 , ainsi que q_1 et q_7 . On supprime ou fusionne tous ces états pour rendre le diagramme plus lisible. On obtient bien :

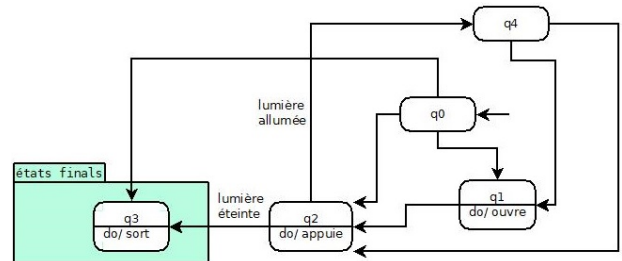


Figure 8 – diagramme état-transition final obtenu pour l'exemple

iv. particularités pour une application domotique

On remarque sur le diagramme état-transition final qu'il y a dans certains cas plusieurs ϵ -transition sortantes possibles. L'automate sous-jacent n'est donc pas déterministe.

C'est probablement dû à un bug de l'implémentation décrit plus loin qui fait qu'elle ne respecte pas exactement le state determinism.

Dans les applications industrielles visées par MAS, cela ne devrait pas être possible, donc si malgré une bonne implémentation du state determinism on avait un tel résultat, il faudrait refuser le diagramme et donner un contre-exemple qui force une transition plutôt que l'autre, ce qui changerait l'automate et apporterait peut-être d'autres cas comme celui-ci, et il faudrait recommencer jusqu'à avoir un diagramme déterministe.

En revanche, dans le cas où on modélise un humain, il semble plus intéressant de garder le non-déterminisme. En effet, cela peut signifier que le composant modélisé a "le choix", ce qui est tout à fait raisonnable pour un humain. L'objectif si on veut encore raffiner le diagramme serait plutôt de ne garder que les choix intéressants. On peut alors se demander si la condition de state-determinism utilisée par MAS est pertinente dans le domaine d'application de la domotique.

V. IMPLÉMENTATION EN B

La méthode B se base sur la théorie des ensembles et la logiques des prédicats pour décrire un logiciel. Un code B ressemble donc syntaxiquement énormément à une définition mathématique, et finalement assez peu à un programme informatique usuel.

Comme indiqué rapidement en introduction, le processus de développement conseillé par la méthode B se découpe en plusieurs étapes. D'abord on spécifie une ou plusieurs machines abstraites. Par "machine" on entend les données du programme ainsi que les opérations possibles sur ces données, mais aussi les contraintes qu'ils doivent respecter. Cette machine abstraite est une description de haut niveau du logiciel. Ensuite, elle est améliorée itérativement en ajoutant des raffinements, qui reprennent les propriétés de la machine abstraite, mais rentrent plus dans les détails de ce que sont les données et ce que font les opérations aux données détaillées.

Les contraintes sont principalement précisées sous forme d'un invariant, un ensemble de propriétés qui doit être respecté durant toute l'exécution du programme. On peut facilement vérifier s'il a été violé ou non durant une exécution, mais la vraie plus-value de la méthode B est de pouvoir prouver statiquement et automatiquement que l'invariant sera toujours respecté.

On peut prouver les machines B avec le logiciel AtelierB, mais on peut aussi les animer, avant d'avoir un raffinement suffisant pour compiler, en utilisant ProB.

Durant l'IRL, l'accent a été mis sur la partie animation, afin de chercher des pistes pour implémenter MAS de façon fonctionnelle. Les aspects preuves auraient dû venir en second lieu mais ont finalement été peu abordés.

i. outils

Comme indiqué rapidement dans l'introduction, les outils utilisés sont Eclipse Modeling Tools, Meeduse, AtelierB et ProB.

i.1 Eclipse Modeling Tools

C'est un ensemble d'outils interfacés à l'IDE Eclipse et centré sur un framework permettant de décrire formellement des structures d'informations. Il contient notamment Ecore, un méta-modèle, qui permet donc de décrire d'autres modèles. Autours de ce framework, une grande quantité de plugins peut être ajoutée pour décrire différents types de systèmes.

Pour l'IRL, on a utilisé le plugin Xtext pour spécifier un DSL décrivant l'entrée de l'algorithme. Xtext génère ensuite un modèle Ecore.

i.2 Meeduse

Meeduse peut utiliser un modèle Ecore pour générer une machine B décrivant la sémantique du modèle, et permettant d'interagir avec. On lui donne donc le modèle généré par Xtext et on obtient du code B décrivant l'entrée de l'algorithme.

Meeduse permet de facilement faire interagir la machine avec un exemple écrit dans le DSL spécifié, en faisant appel à ProB.

i.3 AtelierB

Ce logiciel permet principalement de prouver du code B. Il peut notamment vérifier que les types sont constants tout au long du programme. Il peut aussi vérifier statiquement que l'invariant est toujours vérifié. Il vérifie aussi des conditions de "well-definedness", qui sont liées au type de certaines variables. La plupart des preuves sont effectuées automatiquement. L'outil propose plusieurs forces

de preuve, qui vont tester avec de plus en plus d'hypothèses (ce qui augmente le temps de preuve). Il est aussi possible de rédiger une preuve à la main.

i.4 ProB

ProB permet principalement d'animer la machine B. Il propose aussi des fonctionnalités de model-checking et de visualisation. L'animation consiste à proposer les opérations possibles. Ensuite l'utilisateur en choisit une que l'interpréteur exécute et vérifie l'invariant, puis ProB propose les opérations correspondant au nouvel état. On peut aussi lancer une animation aléatoire qui ne va pas attendre d'entrée utilisateur et plutôt choisir une opération possible au hasard.

ii. quelques bases de syntaxe B

On va juste voir la structure d'un code B pour comprendre les choix de conceptions. Pour plus de détails sur les opérations possibles, se référer à [9].

Le code commence par un header où on déclare s'il s'agit d'une machine abstraite ou d'un raffinement, et où on peut importer une autre machine B. Ensuite, la structure globale du programme B est :

- déclarer des ensembles abstraits
- déclarer des variables. On peut avoir des ensembles, des fonctions, des relations, etc...
- définir un invariant
- initialiser les variables
- déclarer des opérations

Certaines sections peuvent être omises si elles sont inutiles dans le code.

L'invariant est une longue propriété composée d'une conjonction de plusieurs propriétés plus petites, qui donnent le type des variables mais peuvent aussi assurer d'autres contraintes qui doivent être vraies tout au long de l'exécution de la machine. C'est ici que l'on spécifierait les propriétés du programme que l'on veut prouver.

Pour déclarer une opération, il faut spécifier une **précondition**, qui dit dans quel cadre on peut utiliser l'opération, puis on peut effectuer des modifications sur les variables. Ces modifications sont des substitutions. Il y en a plusieurs types, et il y a une théorie des substitutions qui permet de faire des preuves. On utilise souvent la substitution simple (notée " := ") car elle ressemble à l'affectation de variable en programmation classique

Pour l'IRL on a utilisé deux types de précondition :

le mot-clef **PRE**, qui va vérifier une propriété globale sur les variables.

le mot-clef **ANY ... WHERE**, qui va chercher des valeurs pour un certains nombre de variables locales listées après le ANY qui vérifient certaines conditions précisées après le WHERE. Ces variables locales sont utilisables dans la partie qui va effectuer les modifications.

Voici le début du code de la machine B générée par Meeduse pour notre DSL codant l'entrée de l'algorithme MAS (voir section outils) :

```

1- MACHINE
2-   myDsl
3-
4- SETS
5-   SEQUENCEDSL;
6-   MESSAGE;
7-   WORD;
8-   MARKER
9-
10- ABSTRACT_VARIABLES
11-   SequenceDSL,
12-   Message,
13-   Word,
14-   Marker,
15-   A_messages_sequenceDSL,
16-   A_input_word,
17-   A_word_output,
18-   A_word_next,
19-   A_sequence_sequenceDSL,
20-   A_accepted_words_sequenceDSL,
21-   A_refused_words_sequenceDSL,
22-   A_nullmessage_marker,
23-   A_marker_voidmessage,
24-   A_markers_sequenceDSL
25-
26- INVARIANT
27-   SequenceDSL : FIN(SEQUENCEDSL) &
28-   Message : FIN(MESSAGE) &
29-   Word : FIN(WORD) &
30-   Marker : FIN(MARKER) &
31-   A_messages_sequenceDSL : Message +-> SequenceDSL &
32-   A_input_word : Word +-> Message &
33-   A_word_output : Word +-> Message &
34-   A_word_next : Word +-> Word &
35-   A_sequence_sequenceDSL : Word +-> SequenceDSL &
36-   A_accepted_words_sequenceDSL : Word +-> SequenceDSL &
37-   A_refused_words_sequenceDSL : Word +-> SequenceDSL &
38-   A_nullmessage_marker : Marker +-> Message &
39-   A_marker_voidmessage : Marker +-> Message &
40-   A_markers_sequenceDSL : Marker +-> SequenceDSL
41-
42- INITIALISATION
43-   SequenceDSL := {} ||
44-   Message := {} ||
45-   Word := {} ||
46-   Marker := {} ||
47-   A_messages_sequenceDSL := {} ||
48-   A_input_word := {} ||
49-   A_word_output := {} ||
50-   A_word_next := {} ||
51-   A_sequence_sequenceDSL := {} ||
52-   A_accepted_words_sequenceDSL := {} ||
53-   A_refused_words_sequenceDSL := {} ||
54-   A_nullmessage_marker := {} ||
55-   A_marker_voidmessage := {} ||
56-   A_markers_sequenceDSL := {}
57-
58- OPERATIONS
59-   SequenceDSL_NEW(aSequenceDSL) =
60-   PRE aSequenceDSL : SEQUENCEDSL &
61-   aSequenceDSL /= SequenceDSL
62-
63-   THEN SequenceDSL := SequenceDSL \ {aSequenceDSL}
64-   END;
65-

```

Figure 9 – début de la machine B générée par Meeduse

Sans rentrer dans les détails de la syntaxe B, on voit les différentes sections, et si on se concentre sur l'opération définie tout en bas, on a un exemple de l'utilisation du mot-clef PRE. Il va vérifier que la variable `aSequenceDSL` appartient bien à l'ensemble `SEQUENCEDSL` et n'appartient pas à l'ensemble `SequenceDSL`. Si c'est le cas, on ajoute `aSequenceDSL` dans l'ensemble `SequenceDSL` en utilisant l'union ensembliste et une substitution simple qui va remplacer l'ancien `SequenceDSL` par $\text{SequenceDSL} \cup \{a\text{SequenceDSL}\}$.

```

160 addEntryLambda =
161   ANY re, pr WHERE
162     state = 1 &
163     re : Letter &
164     pr : S_prefix_closed &
165     pr = {} &
166     (pr <- re) /: entries
167   THEN
168     entries := entries \ / {(pr <- re)}
169   END;

```

Figure 10 – une des opérations de notre raffinement

Dans l'exemple ci dessus tiré de notre raffinement de la machine B de MAS (voir plus loin), on voit un cas d'usage de ANY ... WHERE. Ici, l'interpréteur B va chercher des valeurs de `re` et `pr` telles que la condition en dessous soit vérifiée. Si c'est le cas, il ajoute la concaténation des deux `re.pr` à l'ensemble `entries`.

Pour les sections suivantes, le code source écrit pendant l'IRL est disponible à <https://gricad-gitlab.univ-grenoble-alpes.fr/julouj/irl>

iii. Implémentation de L^*

Avant de s'attaquer à MAS, l'algorithme plus simple L^* a d'abord été implémenté en B.

Après plusieurs itérations, pour rendre le test plus facile, on a implémenté l'enseignant comme étant une base de donnée donnée initialement contenant une liste de mots acceptés et refusés.

Le DSL en entrée comprend donc une section où on précise l'alphabet. Ensuite on donne un ensemble de mots acceptés, puis un ensemble de mots refusés, construits sous forme de listes chaînées comme il n'y a pas de séquence en Xtext.

Au tout début on initialise S et E à λ . Pour assurer la séquentialité, l'algorithme est séparé en 8 états. On utilise une variable entière pour se repérer.

- **initialisation** : on crée les ensembles de mots connus acceptés et refusés à partir du DSL d'entrée.
- **ajout d'entries** : on ajoute des éléments de S.A dans la table d'observation.
- **fermeture et consistance** : on regarde si la table est fermée et consistante. Si ce n'est pas le cas, on applique la méthode de correction adaptée. Le ANY...WHERE est très utile ici comme dans L^* à cette étape il faut trouver des éléments fautifs pour appliquer les corrections.
- **membership query** : on garde un ensemble des mots déjà vus. Si il existe un élément de S.A qui n'est pas dedans, on va poser une membership query. On peut accepter et refuser automatiquement en regardant dans les ensembles de mots connus par l'enseignant. Il reste une réponse par défaut, qui est ici de refuser. On retourne à l'étape de fermeture et consistance quand tout est déjà vu.
- **génération de l'automate** : Une fois que la table est fermée et consistante, on applique à la lettre les règles de génération de l'automate. On obtient donc à la fin les ensembles Q et F, l'état q_0 , et la fonction de transition δ .
- **vérification de l'automate** : Ici on vérifie que l'automate accepte bien les mots connus pour être acceptés et refuse bien ceux que l'on sait refusés. Si on trouve une erreur, elle devient un contre-exemple.
- **contre-exemple** : On dépile les lettres du contre-exemple pour ajouter tous ses préfixes à S.
- **jeu avec l'automate** : On peut parcourir l'automate avec des opérations permettant de prendre les transitions entre états.

Dans cette implémentation, `row` est codé comme une relation entre éléments de S et éléments de E. Ainsi le `row` que l'on a défini dans L^* pour un élément s est la classe d'équivalence dans E par rapport à s pour cette relation.

On définit la fonction T à partir de `row` en utilisant l'équivalence entre l'appartenance de e à `row(s)` et $T(s.e) = 1$.

iv. Implémentation de MAS

Les figures données auparavant sur l'exemple de l'humain qui éteint la lumière en sortant sont toutes (sauf le diagramme de séquence initial) des représentations graphiques propres de l'état de cette implémentation de MAS une fois l'automate fini et accepté par l'enseignant, lu dans les "state properties" sur ProB.

On change le DSL d'entrée pour correspondre à l'entrée de MAS. Plutôt qu'une section où on précise l'alphabet, on donne ici les messages possibles au début. Ensuite, il faut donner une trace d'exécution sous forme de mot sur l'alphabet dans le formalisme décrit page 6. On précise à la fin, dans la section "MARKERS", quels messages correspondent à NULL et VOID, pour simplifier leur usage ensuite.

On garde l'idée de l'enseignant sous forme de base de données, il y a donc aussi un champ pour des mots connus comme refusés et acceptés dans le DSL. Cependant, cette fois-ci, on laissera le choix d'avoir un enseignant partiellement humain pour les membership queries ou pas, réglable en modifiant la ligne 544 du code source du raffinement. Cependant les contre-exemples seront forcément tirés de la base de données initiale. Cet aspect devrait pouvoir être modifié pour permettre un enseignant partiellement humain sur les contre-exemples aussi.

Des variables sont rajoutées pour assurer le state determinism. Un ensemble *symbol_pairs* des paires de lettres (elles-même des couples de messages) apparaissant dans des mots acceptés. On garde aussi un ensemble *leads_to_NULL* de tous les premiers messages pour lesquels le second est NULL.

remarque Il y a en fait une optimisation qui manque dans l'implémentation actuelle. Plutôt que de seulement conserver les premiers messages qui amènent à NULL, il faudrait aussi conserver dans un ensemble *doesn't_lead_to_NULL* conservant les messages qui amènent à autre chose que NULL (dont on précise dans l'invariant que son intersection avec *leads_to_NULL* est vide) pour bien pouvoir vérifier ensuite que l'on ne peut pas ajouter un mot qui contient une lettre contenant un NULL alors qu'elle ne devrait pas.

La structure de l'implémentation reste globalement la même, mais la partie concernant les membership queries est très fortement développée. On rajoute donc des états de l'algorithme supplémentaires à la place de l'état "membership query".

- **accepter la membership query** : le mot demandé se trouve dans la base de donnée des mots connus et acceptés, on l'accepte donc sans hésiter.

Cependant en plus de cela, on effectue deux passes de décomposition du mot. Elles consistent grossièrement à dépiler une à une les lettres du mot. Pendant la première on ajoute tous les couples de lettres successives du mot dans *symbol_pairs* sous la forme (première lettre, premier message de la seconde lettre). Pendant la seconde, si une lettre contient NULL, on ajoute son premier message à *leads_to_NULL*. Sinon, il faudrait l'ajouter à *doesn't_lead_to_NULL*.

- **refuser la membership query à cause d'un double void** : le mot demandé contient 2 fois VOID, ce qui est illégal. On le refuse donc.
- **refuser la membership query** : le mot demandé se trouve dans la base de donnée des mots connus et refusés, on le refuse donc sans hésiter.
- **refuser la membership query à cause de l'absence de void** : le mot demandé ne contient pas de VOID dans sa dernière lettre. Il ne peut donc représenter une trace d'exécution. On le refuse.
- **tester le state determinism** : avant de proposer un choix à l'enseignant, on teste si le mot vérifie le state-determinism. on effectue donc aussi deux passes de décomposition du mot.

Pendant la première, on vérifie qu'il n'y a pas de paires dont la première lettre est dans un des couples de *symbol_pairs* et où la deuxième ne commence pas par un message qui est le deuxième élément d'une d'un de ces couples. Si une telle paire existe, on refuse le mot.

Pendant la seconde, on vérifie qu'il n'y a pas de lettre contenant autre chose qu'un NULL telle que le premier message de la lettre est dans *leads_to_NULL*. Il faudrait aussi vérifier qu'il n'y a pas de lettre contenant un NULL telle que le premier message de la lettre est dans *doesn't_lead_to_NULL*.

- **cas par défaut** : On laisse l'enseignant choisir si le mot satisfaisant le state-determinism est refusé ou accepté. Dans la version sans enseignant humain, on accepte forcément.

VI. CONCLUSION

i. Résumé

Constituant une introduction à la fois au domaine de l'inférence de modèles et de la méthode B, Cette IRL aura eu une très importante composante bibliographique.

Au final, des pistes d'implémentation de L^* et de MAS en B auront été explorées, même si les machines B en l'état ne sont très probablement pas utilisables pratiquement.

La partie preuve a été négligée pour se concentrer sur l'obtention d'un algorithme fonctionnel.

ii. Bilan et perspectives

ii.1 performances

L'implémentation B tourne assez lentement. Il faut attendre un peu plus d'une minute pour faire tourner l'exemple de l'humain qui éteint la lumière. Cela est dû d'une part au fait que l'on utilise ici du B non compilé comme toutes les étapes de raffinement n'ont pas été faites. Un autre problème est l'usage de condition en $\forall a, P(a)$ assez coûteuses entre certaines étapes pour assurer la séquentialité de l'algorithme.

Au delà de ce problème, il manque encore l'automatisation de la conversion de l'automate en statechart diagram. Cela signifie que la conversion doit être faite à la main pour le moment. Pour l'exemple, il a fallu faire cela 3 fois pour ajouter des contre-exemples supprimant les généralisations indésirables et obtenir quelque chose qui approchait l'objectif : avoir la lumière éteinte en sortant.

ii.2 preuve de l'algorithme implémenté

L'implémentation ne passe pas les preuves automatiques de AtelierB pour l'invariant. La façon dont elle est écrite, avec différents états numérotés, se prête en fait mal à la preuve, car il faut remonter à l'opération qui a modifié l'état pour savoir quelles propriétés garantissent la présence dans cet état.

A part cette preuve de l'invariant, on pourrait chercher à montrer que l'automate accepte les mots que l'enseignant lui a dit d'accepter et refuse ceux que l'enseignant a dit de refuser, ou encore la terminaison de l'algorithme en montrant que l'on atteint forcément le dernier état.

On a pas de preuve statique, mais en exécutant dans ProB, si on atteint la fin, et si l'invariant est encore valide à la fin, on peut être certain que pour ce résultat, l'invariant a toujours été respecté, et que l'automate accepte et refuse bien les mots connus à cause des conditions de passage d'un état à un autre. C'est déjà une garantie supplémentaire par rapport à une implémentation directement dans un langage de programmation.

ii.3 autres algorithmes possibles

MAS se base sur L^* et convertit l'automate accepteur en diagramme état-transition après. Il existe cependant une variante de L^* qui infère des automates de Mealy, L_M^* [5]. On pourrait imaginer une version modifiée de L_M^* qui infère directement un statechart diagram, et alors il serait peut-être possible d'utiliser des optimisations semblables au state determinism pour réduire le nombre de membership queries.

Il existe aussi l'algorithme Z-quotient, qui est bien expliqué dans [6], et qui se place en comparaison avec L_M^* . Il utilise une combinaison d'inférence passive et active, et on peut considérer que MAS aussi comme il nécessite une trace initiale. Il pourrait donc être un candidat potentiel pour inférer des modèles sur des traces d'exécution en adaptant MAS.

Remerciements. Je tiens à remercier Akram Idani pour m'avoir proposé ce sujet d'IRL, m'avoir appris à utiliser les outils de la méthode B et m'avoir patiemment aidé quand je faisais fausse route. Je remercie également Roland Groz pour son précieux apport sur l'inférence de modèles. Ce fut très intéressant de s'initier à ces deux domaines et d'essayer d'appliquer l'un à l'autre.

VII. BIBLIOGRAPHIE

RÉFÉRENCES

- [1] E. Mäkinen et T. Systä.
MAS - an interactive synthetizer to support
behavioral modeling in UML. 2001
- [2] E. Mäkinen et T. Systä.
Minimally adequate teacher designs software.
Technical Report A-2000-7, Dept. of Computer
and Information Science,
University of Tampere, Avril 2000.
- [3] E. Mäkinen et T. Systä.
Implementing minimally adequate synthetizer.
Technical Report A-2000-9, Dept. of Computer
and Information Science,
University of Tampere, Juin 2000
- [4] D. Angluin.
Learning regular sets from queries and
counterexamples. *Information and Computation*,
1987
- [5] M. Shahbaz.
Reverse Engineering Enhanced State Models
of Black Box Software Components to support
Integration Testing. Thèse. Laboratoire Infor-
matique de Grenoble, décembre 2008
- [6] A. Petrenko, K. Li, R. Groz, K. Hossen, C. Oriat.
Inferring Approximated Models for Systems
Engineering. *IEEE 15th International Symposium
on High-Assurance Systems Engineering*. 2014
- [7] D. Harel
Statecharts : A visual formalism for complex
systems. *Science of Computer Programming*. 1987
- [8] [https://www.transformation-tool-
contest.eu/2019/papers/TTC19_paper_5.pdf](https://www.transformation-tool-contest.eu/2019/papers/TTC19_paper_5.pdf)
- [9] [https://www.irif.fr/sighirea/cours/genielog/poly-
B-2007.pdf](https://www.irif.fr/sighirea/cours/genielog/poly-B-2007.pdf)