TABLE DES MATIÈRES

LISTE DES FIGURES

CHAPITRE 1

INTRODUCTION

Actuellement, l'analyse, l'évaluation des performances et la garantie du bon fonctionnement des systèmes sont rendues des activités difficiles à cause de la complexité croissante des systèmes. Or, dans les dernières décennies, la traçabilité a démontré son utilité que ce soit pour identifier des problèmes de performance ou de fonctionnalité. Selon la norme ISO 8402, la traçabilité est : « l'aptitude à retrouver l'historique, l'utilisation ou la localisation d'un article ou d'activités semblables au moyen d'une identification enregistrée». Il s'agit donc d'une démarche qui consiste à donner la possibilité de retrouver la trace et les différents lieux de vie d'un produit. Parmi les fonctions de la traçabilité, la vérification des traces ou l'analyse des traces est considérée comme une étape cruciale.

Quel que soit le domaine, le principe de fonctionnement est toujours le même : on conserve la trace (aussi appelé *log* ou *journal*) de tous les événements produits par un système (transfert de fonds, entreposage ou transformation d'un item) pour éventuellement procéder à l'analyse du registre ainsi produit en cas de problème. Une trace informatique est l'ensemble des informations qui ont été conservées de l'exécution du système et qui permettent de savoir ce qui s'est produit. Elles peuvent être issues d'un traitement effectué par un utilisateur, un processus ou une application. La trace informatique prend des formes variées : lignes d'un fichier, événements d'un journal, entrées dans une base de données, informations dans l'application elle-même. Comme toute trace, sa qualité doit garantir une reconstitution exacte des événements et faire l'objet d'une analyse selon des critères. L'analyse des traces d'événements permet d'identifier des erreurs dans l'exécution du système ou la violation de certaines politiques.

1.1 Exemples

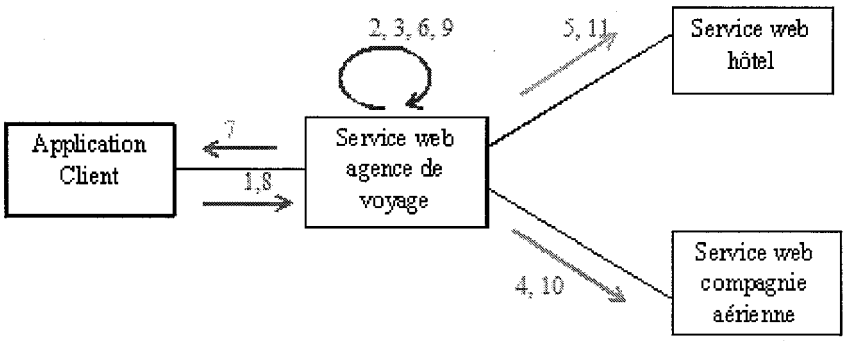
Pour illustrer l'intérêt de l'analyse de traces, nous allons donner deux exemples. Le premier exemple est celui des services web. Prenons l'exemple d'un individu qui désire effectuer un voyage. Pour ce faire, il voudra acheter tous les services nécessaires à ce périple sur la page web de son agence de voyages. Ce scénario se compose de quatre entités : un client, une agence de voyages, une compagnie aérienne et une chaîne d'hôtels. La Figure 1.1 représente ces entités, ainsi que leurs interactions possibles. Un client effectue d'abord une requête pour s'informer des vols et des chambres d'hôtel, puis valide une réservation. L'agence de voyages reçoit et traite cette requête et le transmet à la compagnie aérienne et à la chaine d'hôtels. Ces deux dernières répondent aux requêtes de l'agence de voyages pour réserver des places sur ces vols et des chambres dans ses hôtels. Dans ce scénario, une trace d'événements correspond aux différentes requêtes et réponses échangées entre ces entités. Plusieurs contraintes peuvent devoir s'appliquer dans ce scénario. Par exemple :

1- Si l'agent client demande un forfait, alors il recevra éventuellement la liste des vols et des hôtels.
2- Dans chaque exécution, il est toujours vrai que l'annuaire de l'hôtel est finalement interrogé.

Si ces deux contraintes sont violées, alors il existe une anomalie dans l'interaction entre les entités et un message d'erreur doit être envoyé au client.

Un deuxième exemple est considéré dans le même contexte, celui de l'interface Iterator de Java. Cette interface fournit des méthodes pour effectuer un parcours séquentiel d'une structure de données. L'interface offre principalement deux méthodes : *hasNext* et *next*. La première renvoie un booléen qui indique si le parcours de la structure de données est achevé. La seconde renvoie l'objet suivant de la structure de données. L'interface Iterator exige cependant que la méthode *hasNext* soit appelée et renvoie 'True' avant que la méthode *Next* ne soit appelée. La Figure 1.2 décrit une machine à états finis qui définit les

interactions possibles entre les états. À partir de l'état '*unknown*', il existe toujours une erreur en appelant la méthode *Next*. Si la méthode *hasNext* est appelée et renvoie' True', on peut dans ce cas appeler la méthode *Next*. Sinon, il n'y a plus des éléments et le moniteur passe à l'état '*none*'. Dans les états '*more*' et '*none*', l'appel de la méthode *hasNext* ne fournit aucune nouvelle formation. À partir de l'état '*more*', il est acceptable d'appeler la méthode *next* et le moniteur retourne à l'état '*unkown*'. L'appel de la méthode '*Next*' à partir de l'état '*none*' rentre dans l'état '*error*'



1 : Demande d'une réservation

2 : Reçoit les requêtes des clients

3 : Estimer les prix et les déléguer à la compagnie aérienne et l'hôtel

4 : Rechercher un vol

5 : Rechercher une chambre

6 : coordonner entre l'hôtel et la compagnie aérienne

7 : Fournir les offres

8 : Réserver

9 : Réserve l'ordre

10 : Réserver un vol

11 : Réserver une chambre

Figure1.1: Événements observés dans l'exemple de l'agence de voyages

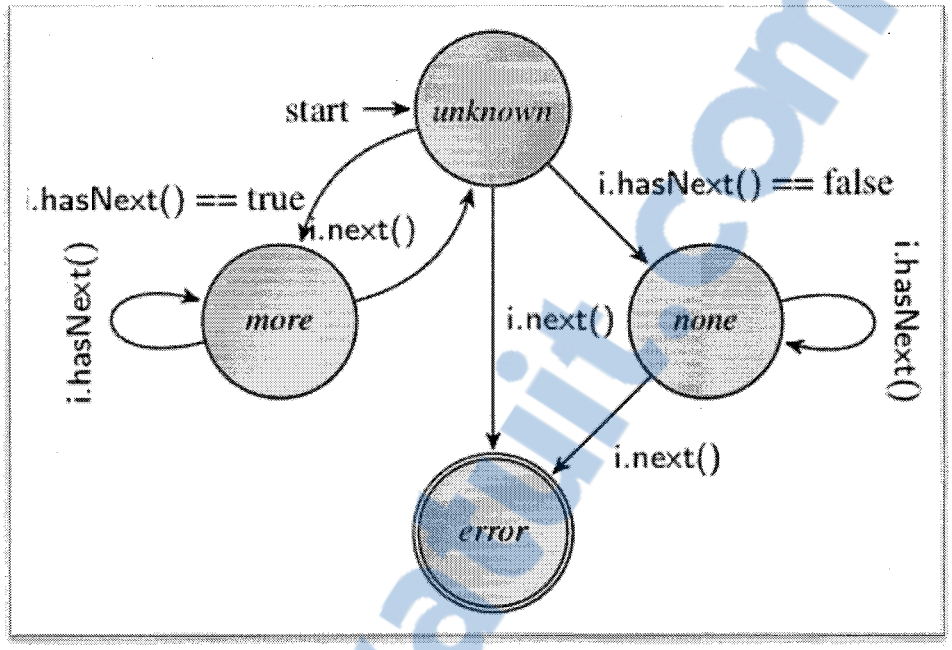


Figure 1.2: Propriété hasNext, (runtime verification, 2013)

Une contrainte qu'on pourra vérifier sur ce scénario est : tout appel de la méthode *next* doit être immédiatement précédé par un appel à la méthode *hasNext* qui retourne 'True'. Java lance d'ailleurs une exception dans le cas contraire.

1.2 Problématique et objectif

Les exemples précédents montrent l'importance de vérifier des propriétés sur des traces pour garantir le bon fonctionnement des systèmes. Or, l'analyse efficace de logs de grande taille est présentement un domaine incontournable, particulièrement si le contrat à vérifier inclut des contraintes sur les paramètres des événements. Mis à part quelques algorithmes ad hoc, il existe peu d'outils répandus permettant de vérifier des propriétés complexes sur des traces d'événements.

Un bon analyseur de traces doit répondre à un certain nombre de besoins. D'abord, il doit permettre de trouver des séquences complexes dans une trace, qui invoquent

plusieurs événements séparés dans le temps et plusieurs paramètres à l'intérieur de chacun. Son langage d'entrée doit donc être suffisamment riche pour permettre à l'utilisateur de spécifier ces séquences. Deuxièmement, l'analyseur de traces doit pouvoir fonctionner sur de grandes traces d'événements. Dans ce cas, il faut un système qui puisse traiter efficacement de grandes traces et qui se comporte de façon gracieuse par rapport à une augmentation de la quantité d'événements. Le temps et la mémoire requise pour traiter chaque événement individuel doivent demeurer à l'intérieur de bornes raisonnables.

Or, si le procédé de journalisation des événements est généralement bien compris, les produits actuels n'offrent que des moyens primitifs d'analyser et de vérifier les traces d'événements qu'ils produisent. On peut, au mieux, filtrer la liste des événements pour ne conserver que ceux qui répondent à certains critères. Les exemples mentionnés précédemment montrent que les contraintes à respecter vont au-delà de tels filtres simplistes.

Depuis une dizaine d'années, des logiciels d'analyses de traces et des prototypes ont été développés par des chercheurs. Notre revue de littérature a répertorié quatorze algorithmes et logiciels pouvant être employés à l'analyse de traces, et dont les domaines d'applications sont variés. Par exemple, BeepBeep (Hallé, 2008) est utilisé dans le monitoring des services web; Monid (Naldurg et al, 2004) est un outil de détection des intrusions; SPIN (Holzmann, 1997) est un outil logiciel pour la vérification des modèles qui a permis la vérification effective de protocoles de communication et de programmes C; NuSMV (Cimatti et al., 2002) est un model checker conçu pour faire une vérification fiable des modèles de taille industrielle. À la lumière de cette étude bibliographique, on distingue quatre problèmes.

1- Chaque outil utilise un format différent pour les traces et les propriétés. À titre d'illustration, BeepBeep prend en entrée une trace dans un fichier XML et une propriété décrite en logique temporelle linéaire de premier ordre (LTL-FO+); SPIN prend en entrée un modèle décrit par le langage PROMELA et une propriété LTL.

Aussi, NuSMV reçoit en entrée un modèle décrit par le langage SMV et les propriétés sont exprimées en logique temporelle (CTL, LTL).

De plus, la conversion d'un langage à un autre n'est pas triviale. La conversion n'est pas limitée à une simple traduction syntaxique, mais est plutôt une conversion complète qui prend en considération les propriétés les plus spécifiques pour chaque fichier. On ne peut pas simplement rechercher les mots clés d'un tel langage et les remplacer dans l'autre langage.

2- La plupart des solutions existantes sont mal documentées en ce qui a trait à leur performance. Parmi tous les outils répertoriés, seuls BeepBeep et SPIN fournissent des données empiriques. Il est donc très difficile de savoir à quel point tel ou tel outil fonctionne bien.
3- La plupart des solutions existantes ont été développées pour une utilisation particulière : par exemple, BeepBeep a été développé pour monitorer des services web; NuSMV vérifie des machines à états finis. Même si plusieurs solutions pourraient être appliquées dans un autre scénario que celui pour lequel elles ont été conçues, il n'existe pas de bancs d'essai avec lesquels comparer les solutions de toute façon.
4- Dans la plupart des solutions existantes, le temps et la mémoire consommée varient beaucoup d'un scénario à l'autre, sans qu'il existe un consensus à savoir quelle méthode fonctionne « le mieux ».

L'objectif de la présente recherche consiste à pallier à ce manque en faisant une étude comparative des outils et des algorithmes de vérification des traces.

1.3 Méthodologie

Pour atteindre l'objectif de la recherche, trois étapes ont été définies :

1- Inventorier les outils de validation des traces. Une revue exhaustive de la littérature scientifique dans le domaine de la vérification et la validation des traces est faite

afin de dégager les principaux outils et algorithmes développés. En plus des logiciels et algorithmes « traditionnels », nous avons ajouté un moteur de bases de données relationnelles et deux model checkers.

2- Développer un banc d'essai permettant de comparer leur performance. Ceci implique de :
   - Comprendre le fonctionnement de chaque outil,
   - Développer un modèle de traces indépendant de la représentation de chacun,
   - Traduire de /vers les formats d'entrées de chacun à partir de cette représentation.

3- Tester le banc d'essai en développant une banque des traces et des propriétés provenant de scénarios variés et réalistes.

1.4 Contexte de la recherche

Ce travail a été réalisé en collaboration avec l'entreprise Novum Solutions, établie à Lévis. Cette entreprise, qui se spécialise dans le développement de produits de gestion d'inventaire basés sur la technologie RFID (Radio Frequency IDentification), veut ajouter la valeur à ses produits en lui adjoignant des fonctions d'analyse des traces. Dans ce scénario particulier, les événements sont des commandes de configuration d'un lecteur RFID et des entrées-sorties d'un article dans un entrepôt. Par exemple, le système pourrait s'assurer que tous les items observés dans un entrepôt sont bien réapparus par la suite dans l'inventaire à leur destination. D'une manière plus générale, l'utilisateur du système pourrait soumettre n'importe quel patron de recherche, représentant une séquence d'événements correspondant à un scénario précis, et obtenir du système les éléments qui y correspondent, s'il y a lieu. Les résultats obtenus vont donc permettre à Novum Solutions d'intégrer, par la suite, l'outil le plus performant d'après les différents tests effectués durant le projet.

1.5 Organisation du document

Le présent mémoire est composé de six chapitres. Après ce premier chapitre d'introduction générale, qui vise à situer le mémoire dans son contexte et à présenter les différentes étapes de la méthodologie de recherche employée dans le cadre de ce projet, le chapitre 2 est consacré à la présentation des scénarios où la vérification des traces est nécessaire. On décrit en détail cinq scénarios tirés de la littérature, pour lesquels la vérification de traces est importante. De plus, on énonce un total de neuf propriétés à vérifier dans ces scénarios. Le chapitre 3 présente les différentes formes des traces. Il définit un modèle formel pour représenter les traces. Aussi, il présente des langages pour exprimer les propriétés. À la fin du chapitre, on revient sur les exemples de propriétés décrites dans le chapitre 2 et on les exprime suivant le langage de spécification approprié. Le chapitre 4 décrit les principaux outils d'analyse des traces d'événements. Cette introduction est nécessaire dans le développement du banc d'essai. Dans le chapitre 5, les outils et les algorithmes cités sont montés dans un banc d'essai (*benchmark*) où ils sont comparés à travers une série de tests représentatifs des scénarios présentés dans le chapitre 2. En particulier, on explique comment un problème de validation de traces pour un outil peut être traduit en un problème équivalent pour un autre. Une partie est réservée pour la discussion des résultats obtenus. Nous terminons ce travail de recherche par une conclusion générale dans le chapitre 6 qui récapitule les différentes contributions apportées par ce mémoire, démontre l'atteinte de l'objectif et propose diverses perspectives de recherche future. Les principales conclusions de ce travail sont :

- Le temps alloué pour la vérification des traces est proportionnel aux longueurs des traces.
- À travers les différents tests que nous avons faits, les meilleures performances sont obtenues avec l'outil MonPoly.

Les résultats préliminaires de ce projet de recherche ont fait l'objet d'un article scientifique publié à la conférence "3rd International Conference on Runtime Verification

2012" (Mrad et al., 2012): "A Collection of Transducers For Trace Validation", RV 2012, Istanbul, Turquie 25 – 28 sept 2012. Un second article sera soumis à la conférence EDOC 2013.

CHAPITRE 2

SCÉNARIOS DE VÉRIFICATION DES TRACES

Ce chapitre est une description de tous les scénarios et leurs contraintes qui seront utilisées ultérieurement dans le jeu de tests de notre banc d'essai. Nous allons considérer cinq scénarios tirés de la littérature:

- Bookstore: scénario d'un achat de livre, tiré de (Van der Aalst, 2011), utilisé pour la surveillance des flux de production, la gestion des processus d'affaires, la modélisation et l'analyse des processus.
- Amazon ECS: service fournissant un accès à des données d'Amazon et des données de commerce électronique, tiré de (Amazon e-commerce service, 2005).
- LLRP: protocole de configuration des lecteurs *RFID*, tiré de (EPC-Globel, 2007).
- Cycle générateur: scénario de vérification des étiquettes *RFID* (aussi appelé *Tags RFID*), tiré de la gestion des stocks en entrepôts.
- Nombre aléatoire, scénario synthétique développé pour valider les résultats obtenus avec les derniers scénarios.

2.1 Bookstore

Le processus bookstore (Van der Aalst, 2011) décrit un scénario d'achat de livres. Il a été utilisé dans les travaux de Van der Aalst, professeur à l'université de technologie Eindhoven, pour la spécification et la surveillance des flux de production et pour la gestion des processus d'affaires. Le réseau de Pétri de la Figure 2.1 décrit le scénario de bookstore. Ce scénario se compose de quatre entités : le client (Customer), la librairie (Bookstore), l'expéditeur (Shipper) et l'éditeur (Publisher). Dans ce scénario, les événements sont les différents messages d'interactions entre ces quatre entités. Ce processus est initié par un client passant une commande (événement `place_c_order`), cette commande est envoyée et gérée par la librairie (événement `handle_c_order`). La librairie transfère par

la suite la commande du livre de votre choix à un éditeur (événement `place_b_order`). L'éditeur évalue la commande de la librairie (événement `eval_b_order`) ; cette commande est soit acceptée (`b_accept`), soit rejetée (`b_reject`). Dans les deux cas, une réponse est envoyée à la librairie. Si la librairie reçoit une réponse négative, elle décide (événement `decide`) soit de rechercher un éditeur alternatif (événement `alt_publ`), soit de rejeter la commande du client (événement `c_reject`). Si la librairie recherche un éditeur alternatif, une nouvelle commande est envoyée au nouvel éditeur. Si le client reçoit une réponse négative (événement `rec_decl`), alors le processus est terminé. Si la librairie reçoit une réponse positive (événement `c_accept`), le client est informé (événement `rec_acc`) et la librairie continue le traitement de la commande du client. La librairie envoie une requête à un expéditeur (événement `req_shipment`) et l'expéditeur évalue la demande (événement `eval_s_req`), soit il accepte (événement `s_accept`), soit il rejette la demande (événement `b_reject`). Si la librairie reçoit une réponse négative, elle cherche un autre expéditeur. Ce processus est répété jusqu'à ce que l'expéditeur accepte.

Une fois l'expéditeur trouvé, l'éditeur est informé (événement `inform_publ`), l'éditeur prépare le livre pour l'expédition (événement `prepare_b`) et le livre est envoyé de l'éditeur à l'expéditeur (événement `send_ book`). L'expéditeur prépare l'envoi au client (événement `prepare_s`) et envoie le livre au client (événement `ship`). Le client reçoit le livre (événement `rec_book`) et l'expéditeur avise la librairie (événement `notify`). La librairie envoie la facture au client (événement `send_bill`). Après avoir reçu à la fois le livre et la facture (événement `rec_bill`), le client effectue un paiement (événement `pay`). Puis la librairie traite le paiement (événement `handle_payment`) et le processus se termine.

Plusieurs contraintes sur l'ordre possible des événements peuvent se vérifier sur ce scénario. Par exemple:

P1 : L'événement « `rec_acc`» ne peut pas se produire à moins qu'un événement « `place_c_order` » n'ait été vu précédemment.

P2 : Au moins une commande du client doit être éventuellement acceptée par la librairie.

2.2 Amazon E-Commerce web service

Les Amazon web services sont des services Amazon qui permettent d'accéder directement à la plateforme technologique Amazon. Les utilisateurs du service demandent des données à l'aide du protocole SOAP et les données sont renvoyées par le service sous la forme d'un flux de texte au format XML. Parmi les services web d'Amazon, on compte *S3 (Amazon Simple Storage Service)* fournissant un stockage basé sur les services web, *SES (Amazon Simple Email Service)*, service d'envoi en nombre et transactionnel d'emails, *SQS (Amazon Simple Queue Service)*, fournissant une file de messages hébergé pour les applications web, etc. Le scénario qui nous intéresse ici s'appelle l'Amazon E-Commerce Service (AWS-ECS). C'est un service fournissant un accès aux données produits d'Amazon et des données de commerce électronique. Ce scénario a été utilisé dans les travaux de Hallé (Hallé et al.,2010). On considère le processus d'un panier de la Figure 2.2: Figure 2.2: Événements observés dans le scénario AWS – ECS. Ce scénario est initié par la création d'un panier (événement `CREATE CART`). Ensuite, des éléments peuvent être ajoutés dans ce panier (événement `CART ADD`). Enfin, ce panier peut être vérifié (événement `CHECK OUT`). La Figure 2.2 décrit un scénario simplifié d'AWS-ECS. Généralement, le scénario AWS-ECS contient six opérations principales :

1- `ItemSearch` : rechercher, en utilisant un mot clé, dans la base de données d'Amazon les articles rapprochés et renvoie une liste de produits correspondant aux critères de recherche.

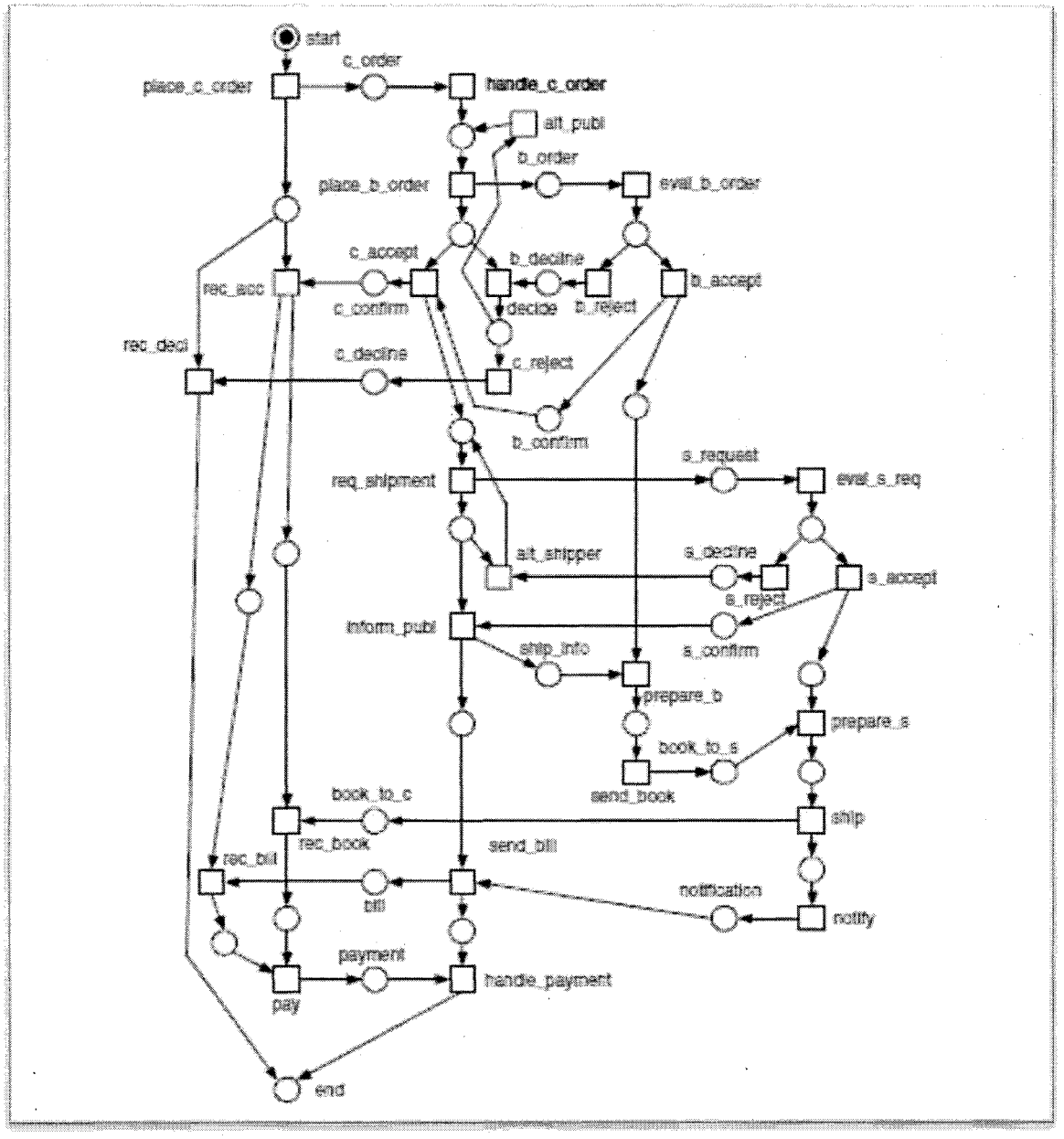


Figure 2.1: Réseau de Pétri décrivant le processus de bookstore, (Van der Aalst, 2011)

2- `CartCreate` : prend en entrée un ASIN (un identificateur unique pour un élément dans la base de données d'Amazon) et un entier ' *n* ' positif, et crée un nouveau panier d'achats qui représente une demande de *n* copies de cet élément.

Amazon stocke et gère le contenu du panier. Les opérations SOAP référent au panier en passant son identifiant unique.

3- `Cart Clear` : vide le panier avec un ID donné.

4- `Cart Add` : ajoute une nouvelle ligne à un panier donné demandant n copies de certains ASIN.

5- `Cart Remove`: supprimer tous les articles d'un panier donné.

6- `Cart Edit`: modifie la quantité d'un certain article dans un panier donné.

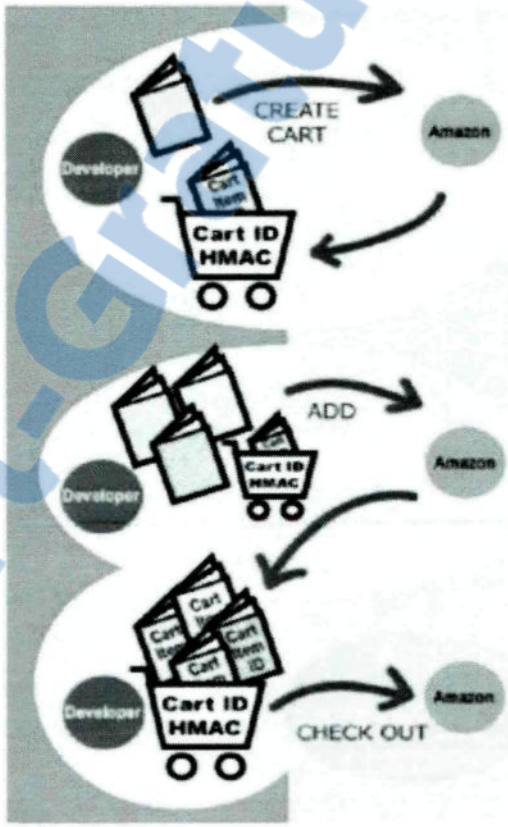


Figure 2.2: Événements observés dans le scénario AWS – ECS, (Marconi, M.Pistore, 2009)

Plusieurs contrats d'interface peuvent être vérifiés. Par exemple, si le panier n'est pas encore créé alors aucun item ne peut être ajouté au panier. On note cette contrainte par :

1- **P3** : Aucune `CartAdd` ne peut se produire avant `CartCreate`.

Une deuxième contrainte peut-être vérifiée sur ce scénario est :

2- **P4** : Pour ajouter un article à un panier, il faut spécifier son numéro d'identification CartID.

Plusieurs incidents peuvent se produire en cas de violation de ces contraintes. Ainsi, le serveur peut interrompre la communication et renvoyer un message d'erreur. Aussi, il peut continuer la conversation sans avertissement, mais envoyer des données erronées. Toutefois, il est aussi possible que le serveur ne corresponde pas tout à fait avec sa documentation, ce qui rend encore plus important de vérifier les contraintes.

2.3 LLRP

La radio-identification ou RFID (Radio Frequency Identification) est une technologie sans fil qui permet de mémoriser et de récupérer des données à distance en utilisant des marqueurs appelés « radioétiquettes » (ou « RFID tag »). Les tags RFID sont des balises métalliques qui peuvent être collées ou incorporées dans des produits et qui sont composées d'une antenne et d'une puce électronique, qui réagissent aux ondes radio et qui transmettent des informations à distance. Un système de radio-identification se compose de deux entités : les tags RFID et un ou plusieurs lecteurs. Les lecteurs sont des dispositifs, émetteurs de radiofréquences qui vont activer les tags. La configuration des lecteurs RFID se fait via un protocole qui s'appelle LLRP (Low Level Reader Protocol) (EPC-Globel, 2007). Il fournit des moyens de contrôler les aspects du fonctionnement des tags, y compris les paramètres de protocole et les paramètres d'individualisation.

Une séquence typique LLRP entre le client et le lecteur RFID suit le processus suivant :

1- Vérification des capacités du lecteur par le client
2- Réglage de la configuration du client

3- Envoi des commandes des configurations du lecteur. Ces commandes sont appelées spécifications de fonctionnement du lecteur (*ROSpecs : Reader Operation Specifications*). Elles contiennent une liste de commandes séquentielles des opérations d'inventaire appelé *AISpec (Antenna Inventory Specifications)*
4- Envoyer les spécifications d'accès du lecteur (*AccessSpec*)
5- Obtenir le rapport du lecteur

Les événements dans ce scénario sont les états des transitions d'un ROSpec. La Figure 2.3 illustre les états des transitions d'un ROSpec. Le ROSpec possède trois états : « Disabled », « Active » et « Inactive ». Le client configure une nouvelle ROSpec en utilisant le message `ADD_ROSPEC` pour le ROSpec. Le ROSpec commence à l'état « Disabled » en attendant le message « `ENABLE_ROSPEC` » pour le ROSpec du client sur lequel il passe à l'état « Inactive ». À l'état « Disabled », le ROSpec ne répond pas aux demandes d'arrêt ou de démarrage des déclencheurs. Le client désactive un ROSpec grâce au message « `DISABLE_ROSPEC` » pour le ROSpec. Les transitions de l'état « Inactive » à l'état « Active » de ROSpec surviennent lorsqu'un message « `ROSpecStartCondition` » arrive pour le ROSpec. Les transitions ROSpec reviennent à l'état « Inactive » lorsqu'un message « `ROSpecDoneCondition` » arrive. Lorsque le ROSpec est indéfini, il n'est plus considéré pour l'exécution. Le client annule le ROSpec avec le message « `DELETE_ROSPEC` ».

Plusieurs contraintes peuvent être vérifiées sur ce scénario :

P5 : Aucun message « `ENABLE_ROSPEC` » ne peut se produire avant un message « `ADD_ROSPEC` ».

P6 : L'événement « `DISABLE_ROSPEC`» ne peut pas se produire à moins qu'un événement «`ENABLE_ROSPEC` » ait été vu précédemment.

En cas de violation d'un contrat, un message d'erreur est envoyé au client et la communication entre le client et le lecteur est interrompue.

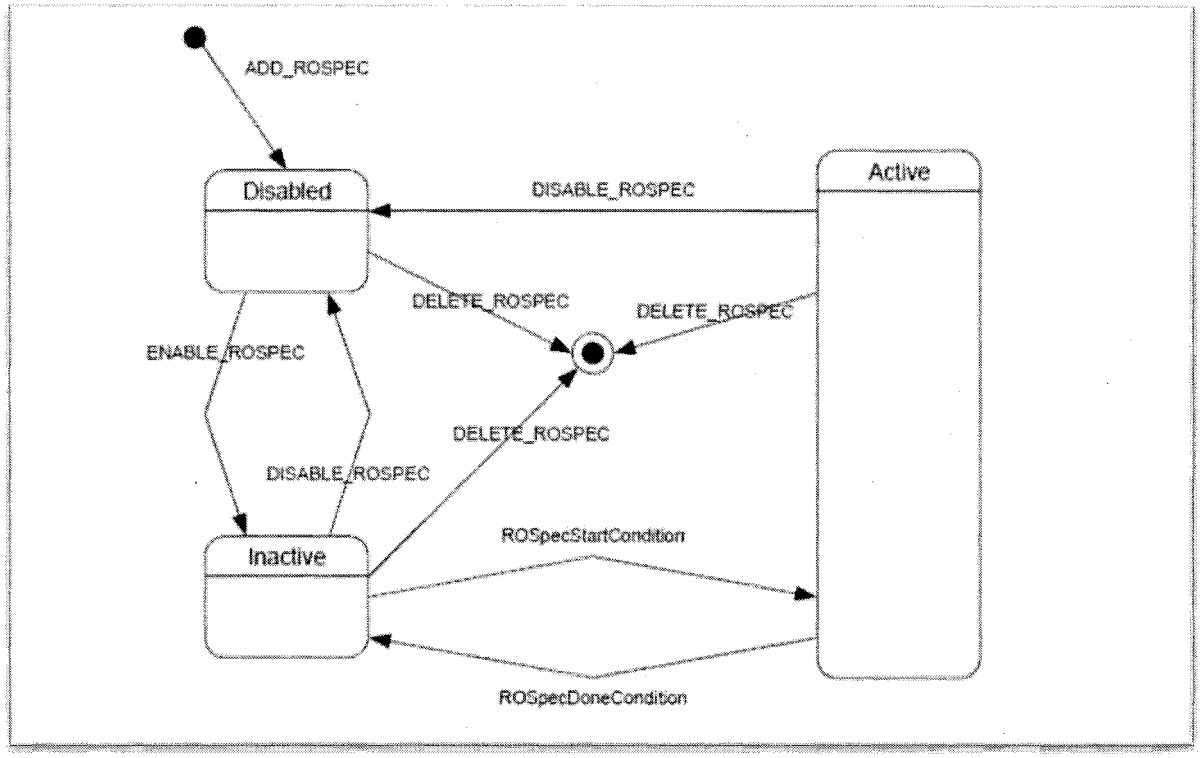


Figure 2.3: Diagramme d'états d'un ROSpec, (EPC-Globel, 2007)

2.4 Cycle générateur

Le scénario du cycle générateur est un scénario simple qui est lui aussi lié à l'utilisation d'un lecteur RFID. Cycle générateur est un scénario particulier de Novum Solutions. Il est tiré de la gestion des stocks en entrepôt. Chaque élément de l'entrepôt est identifié par un tag et des capteurs identifiant les objets entrant et sortant de l'entrepôt. Dans ce scénario, une trace est une séquence d'identifiants de tags RFID. Les «contraintes» correspondant ici à des patrons d'événements sur les tags qu'un utilisateur souhaite rechercher dans une trace. Il existe plusieurs contraintes à valider. Parmi lesquels, on souligne :

P7 : Globalement, pour tout tag dont la valeur est égale à 1 alors le prochain tag est de valeur égale à 2.
Cette contrainte pourrait représenter, par exemple, le fait que deux tags se suivent dans un entrepôt «artificielle».

2.5 Nombres aléatoires

Le scénario des nombres aléatoires est un ensemble des traces dont les événements ne sont qu'une suite des paramètres abstraits (p0, p1, ...) possédant une valeur numérique aléatoire. Ce scénario est utilisé pour vérifier et valider les résultats obtenus avec les autres scénarios. Aussi, sur ce scénario on peut vérifier des contraintes. Par exemple :
P8: Globalement, il existe une variable p0 qui égale à 0.
P9 : Pour toute variable p0 dont la valeur est égale à 0 alors, éventuellement, il existe une variable p0 dont la valeur est égale à 1.

CHAPITRE 3

FORMALISATION

Dans ce chapitre, on définit formellement ce qu'est une trace et ce qu'est une propriété sur une trace. On reviendra sur les exemples de propriétés décrites dans le chapitre 2 et on les exprimera suivant le langage de spécification approprié.

3.1 Modèle de base: XML

XML (*eXtensible Markup Language)* est un langage, standardisé par le W3C (World Wide Web Consortium), permettant de mettre en forme des documents grâce à des balises (markup) et de décrire des données de domaines variés. Plusieurs éléments motivateurs ont permis d'adopter XML. Ces éléments reposant principalement sur :

- L'intégrabilité : un document XML est utilisable par toute application pourvue d'un parseur;
- L'extensibilité : un document XML doit pouvoir être utilisable dans tous les domaines d'applications;
- La lisibilité : un document XML est facile à comprendre. Aucune connaissance ne doit être nécessaire pour comprendre son contenu;
- Une structure arborescente : permettant de modéliser la majorité des problèmes informatiques.

Un document XML se compose de trois parties : prologue, arbres d'éléments et commentaires. Le prologue contient une déclaration de la version du XML et le codage de caractères utilisés. Il peut aussi donner une référence à un autre document, la *grammaire*, qui déclare la structure attendue du document. L'arbre d'éléments est la partie la plus importante des documents XML, c'est celle qui contient les données à proprement parler. Un élément se note *<a> .....*, où a est le nom de l'élément. Son nom est délimité par

des balises ouvrante et fermante. Il existe toujours un élément «racine» qui contient tous les autres. Un élément peut contenir d'autres éléments et doit se situer à l'intérieur d'un seul élément. Les commentaires, dont la présence est facultative, sont encadrés par les marques de début <!-- et de fin --> de commentaire.

Soit le fichier XML « amazon.xml » suivant tiré du scénario Amazon E-commerce web service. Le prologue dans l'exemple est « <?xml version="1.0" encoding="UTF-8"?> ». Il permet d'identifier le jeu de caractères utilisé en utilisant le mot clé « encoding ». Dans la suite, on omettra le prologue dans les exemples. Dans cet exemple, le jeu de caractères est l'Unicode avec encodage UTF-8. L'arbre des éléments est tout le bloc « Trace ». Il contient deux événements, identifiés par l'élément <message>. On voit que chaque événement possède des paramètres comme SessionKey, CartId ou Price.

```
<?xml version="1.0" encoding="UTF-8"?>
<Trace>
        <message>
                <SessionKey>1234</SessionKey>
                <Action>CartCreate</Action>
                <Items>
                        <Item>
                                <ItemId>7</ItemId>
                                <Price>35</Price>
                        </Item>
                </Items>
        </message>
        <message>
                <SessionKey>1234</SessionKey>
                <Action>CartCreateResponse</Action>
                <CartId>21</CartId>
                <Items>
                        <Item>
                                <ItemId>7</ItemId>
                                <Price>35</Price>
                                <Quantity>1</Quantity>
                        </Item>
                </Items>
</message>
</Trace>
```

3.2 Un modèle formel des traces d'événements

Généralement, une trace est une suite d'événements produits par un système. On note par $(\sigma_1,\sigma_2,\ldots)$ la trace formée par les événements σ_1, σ_2,..... Pour toute valeur de $i \geq 1$, on écrit σ_i pour désigner le i-ème événement de la trace σ et σ^i pour désigner la trace obtenue à partir de σ en commençant par le i-ème événement.

On distinguera deux types de traces : *finie* et *infinie.* Dans une trace finie, l'exécution du système est considérée terminée et tous les événements sont donc connus. Au contraire, dans une trace infinie, on peut toujours ajouter un événement à la suite. Dans la pratique, si la trace est infinie, on analyse un préfixe et on tente de tirer des conclusions qui seront valides quel que soit le suffixe qu'on pourrait lui ajouter.

À la lumière de l'étude bibliographique, nous pouvons classer les traces générées par les différents scénarios dans quatre catégories.

3.2.1 Trace multi valeurs

Elle correspond à la forme la plus générique des traces où chaque événement est considéré comme une fonction σ_i: $P \rightarrow 2^V$ associées au paramètre $p \in P$ où P est l'ensemble des paramètres et V est l'ensemble des valeurs possibles pour ces paramètres. On associe, par la suite, au paramètre p les valeurs rencontrées dans les événements. On considère l'exemple des nombres aléatoires suivant (voir Figure 3.1):

```
<Trace>
        <message>
                <p3>2</p3>
                <p0>2</p0>
                <p3>3</p3>
                <p0>3</p0>
                <p3>0</p3>
        </message>
        <message>
                <p3>0</p3>
                <p2>2</p2>
                <p2>3</p2>
        </message>
</Trace>
```

Figure 3.1: Trace multi valeurs

Cette trace contient deux événements. Dans cet exemple, P est l'ensemble des paramètres {p0, p2, p3}, V= {0, 2, 3} et la fonction σ associée au premier événement est définie par σ_1 (p3) = {2, 3, 0} et σ_1(p0) = {2, 3}. Pour le second événement, la fonction est définie par σ_2 (p2) = {2, 3}, σ_2 (p3) = {2, 3}, et σ_2 (p0) = Ø. On utilise le symbole Ø pour dénoter l'absence de valeur.

On peut généraliser σ et remplacer P par Π, l'ensemble de chemins de la racine à un élément de l'événement. Ce chemin est défini en utilisant le langage *XPath* (Clark et al., 1999). XPath est un langage permettant d'adresser les différents nœuds ou groupes de nœuds d'un document XML. Par exemple, si p ∈ Π une formule particulière qui est la suivante : ''/message/Items/Item/ItemID'' et σ_1 ∈ M l'événement suivant pris du scénario Amazon web service (voir Figure 3.2):

```
<message>
<SessionKey>0</SessionKey>
<Action>ItemSearchResponse</Action>
<Items>
        <Item>
                <ItemId>5</ItemId>
                <Price>35</Price>
        </Item>
        <Item>
                <ItemId>6</ItemId>
                <Price>7</Price>
        </Item>
</Items>
</message>
```

Figure 3.2: Exemple de trace du scénario AWS

Alors $\sigma_1(p) = \{5, 6\}$.

Si p_1="/message/Items/Item/Price", $\sigma(p_1) = \{35, 7\}$.

3.2.2 Trace à une seule valeur

Dans cette catégorie des traces, chaque événement est considéré comme une fonction σ_i: $P_i \rightarrow V$ avec les mêmes ensembles que précédemment. L'ensemble P_i dépend des événements du fichier XML. Chaque paramètre peut donc survenir au plus une fois dans chaque événement ; formellement, ceci impose que $|\sigma_i(p)| = 1$, pour tout p $\in P_i$. Le format XES (Extensible Event Stream) (Günther, 2009) est une notation possible pour ce genre des événements. Prenons l'exemple des nombres aléatoires suivant (voir Figure 3.3):

```
<Trace>
        <message>
                <p3>5</p3>
                <p0>2</p0>
                <p1>2</p1>
                <p2>3</p2>
        </message>
</Trace>
```

Figure 3.3: Trace à une seule valeur

Dans cet exemple, la trace contient un seul événement, σ_1 (p3) = {5}, σ_1 (p0) = {2}, σ_1 (p1) = {2} et σ_1 (p2) = {3}.

3.2.3 Trace à un schéma fixe

Chaque événement est considéré comme une fonction σ_i: P→V définie comme précédemment. Cependant, dans cette forme de trace, tous les messages contiennent les mêmes paramètres et ceux-ci ne seront différents que par leurs valeurs. On considère comme exemple les fichiers CSV, ce sont des fichiers tableurs, contenant des données sur chaque ligne séparée par le caractère de séparation qui est la virgule. En général, toutes les données extraites d'une table de base de données relationnelle créeront des traces de ce type, comme la montre Figure 3.4.

P0	P1	P3
8	6	2
4	10	3
2	8	4

Figure 3.4: Trace à un schéma fixe

3.2.4 Trace atomique

Chaque événement est considéré comme une fonction $\sigma_i = a$, où a est un simple symbole appelé *atome* pris dans un ensemble A, l'*alphabet*. Ce type de traces est approprié lorsque les événements n'ont pas de paramètres. Le symbole représente donc le nom de l'événement. C'est le cas, entre autres, du scénario Bookstore. Les exemples incluent aussi des séquences d'appels de méthodes sur des objets de programme (en rejetant leurs arguments) (F. Chen et al., 2006) et quelques journaux de processus d'affaires.

Par la suite, on intègrera ces traces à des documents XML dont les événements n'ont qu'un seul paramètre, appelé arbitrairement <name>. La Figure 3.5 montre une telle trace tirée du scénario Bookstore.

```
<Trace>
        <message>
                <name>place_c_order</name>
        </message>
        <message>
                <name>handle_c_order</name>
        </message>
        <message>
                name>place_b_order</name>
        </message>
        <message>
                <name>eval_b_order</name>
        </message>
        <message>
                <name>b_reject</name>
        </message>
        <message>
                <name>decide</name>
        </message>
        <message>
                <name>c_reject</name>
        </message>
        <message>
                <name>rec_decl</name>
        </message>
</Trace>
```

Figure 3.5: Exemple de trace atomique

3.3 Propriétés sur les traces d'événements

De la même façon qu'il existe plusieurs types de traces d'événements, les propriétés sur les traces peuvent elles aussi être exprimées formellement de plusieurs manières. Plusieurs formalismes, d'expressivité variable, existent pour exprimer ces propriétés. Parmi ces formalismes on distingue : les automates, les expressions régulières et les différentes formes de logique temporelle. Lorsqu'une propriété φ énonce quelque chose de vrai pour une trace σ donnée, on dit qu'une trace σ satisfait une propriété φ et on la note $\sigma \models \varphi$.

3.3.1 Les automates

Les automates sont un modèle très reconnu dans le domaine de la spécification formelle des systèmes. Ils ont été appliqués dans plusieurs domaines tels que la théorie combinatoire des groupes, la compilation, la modélisation et la vérification de protocoles.
Formellement, un automate est représenté par un 5-uplet $(Q, \sum, \delta, q_0, F)$ où

- Q est un ensemble d'états fini ou infini
- $\sum$ est un ensemble fini de symboles, appelé alphabet
- δ est la fonction de transition, telle que $\delta : Q \times \sum \rightarrow \sum$
- q_0 est l'état initial, $q_0 \subseteq Q$
- F est un ensemble d'états (sous-ensemble de Q), appelé états accepteurs ou états « finaux ».

Dans le contexte des automates, une séquence de symboles de $\sum$ est appelée un mot. Un automate prend en entrée un mot qui est un mot $\subseteq \sum^*$ (où $\sum^*$ est l'ensemble des mots sur Σ), et non $\subseteq \sum$ reconnue par l'automate. En fonction de l'état final, on dit qu'un automate accepte ou rejette le mot en entrée. Plus précisément, lorsqu'un parcours de l'automate aboutit dans un état final alors on dit que le mot est accepté, sinon il est rejeté. Par exemple, soit t un mot donné : chemin étiqueté par t depuis un état initial. On dit que le mot t est accepté si au moins une exécution sur t va à un état acceptant.
Pour exprimer une propriété dans une trace, il suffit d'écrire un automate A tel que $\sigma \models \varphi$ si et seulement si σ est acceptée par A. Par exemple, la propriété P5 de scénario LLRP est exprimée sous forme d'un automate (voir Figure 3.6).

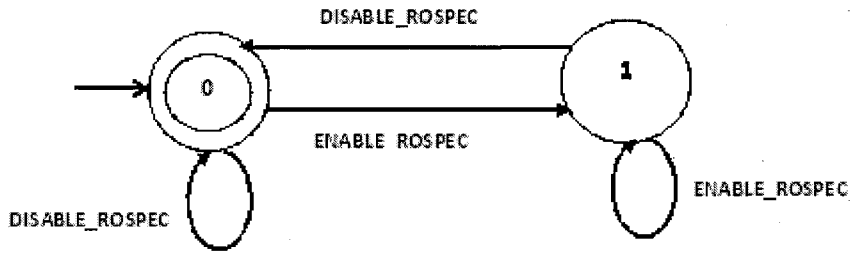


Figure3.6: Automate spécifiant la propriété P5

Soit p = ENABLE_ROSPEC et ¬ p= DISABLE_ROSPEC,

On interprète cet automate comme suit :

- tant que ¬p est vraie, l'automate reste dans l'état 0,
- si on rencontre un état où p est vraie (l'automate passe dans l'état 1), elle n'est plus acceptée; pour l'être à nouveau, l'état suivant doit satisfaire ¬p (retour à l'état acceptant 0).

On voit bien que cet automate accepte exactement les traces qui satisfont la propriété P5.

3.3.2 Les expressions régulières

Une expression régulière est un motif qui décrit un ensemble de chaînes de caractères possibles selon une syntaxe précise. Leur origine se situe dans la théorie des automates et des langages formels. C'est une autre manière de spécifier un automate. Formellement, étant donné un alphabet $\sum$, on appelle expression régulière une expression construite par les règles suivantes :

- Tout élément de $\sum$ est une expression régulière;
- Si e_1 et e_2 deux expressions régulières alors la concaténation $e_1 e_2$ est une expression régulière;
- Le mot vide, noté ε, est une expression régulière;

- Si e_1 et e_2 deux expressions régulières alors l'alternative $e_1|e_2$ est une expression régulière;
- Si e est une expression régulière alors e^* est une expression régulière.

Soit $\sum = \{a, b, c, d, f, e, f\}$.

1- L'expression régulière *abc / def* dénote l'ensemble des chaînes qui ont soit un *a* suivis d'un *b* et *c*, soit un *d* suivis d'un *e* et *f*.
2- L'expression régulière $(ab^* \mid c)^*$ dénote la concaténation de séquences de lettres *c* ou *a* suivi d'un nombre nul ou fini de *b*.
3- L'expression régulière $(a \mid b)^*$ dénote l'ensemble de toutes les chaînes constituées d'un nombre quelconque (éventuellement nul) de *a* ou de *b*.

L'expression régulière associée à l'automate représenté dans la Figure 3.6 est :
$((\neg p)^*(p^*))^*$.

3.3.3 LTL

La logique LTL (Logique temporelle linéaire) a été proposée par Manna et Pnueli (Pneuli, 1977). Elle permet la description qualitative et le raisonnement de changements du comportement des systèmes à travers le temps. Elle a été introduite pour exprimer des propriétés sur des états et des séquences d'états dans les systèmes appelés structures de Kripke (Clarke et al., 2000). Elle permet la représentation du temps en fournissant une variété d'opérateurs propositionnels ou connecteurs booléens et des combinateurs temporels.

Syntaxe

Soit $p \in \Pi$ et $v \subseteq V$, tels que définis précédemment. Une formule φ appartient à LTL si et seulement si elle est formée à partir de la grammaire BNF suivante :

$\varphi \equiv p = v \mid \neg \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \mathbf{G}\varphi \mid \mathbf{F}\varphi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \varphi$.

LTL utilise les connecteurs booléens classiques : $\neg$ (négation), $\wedge$ (conjonction : 'et' logique), $\vee$ ('ou' logique) et $\rightarrow$ (implication).

Les combinateurs temporels décrits dans la grammaire BNF signifient :

- **X** φ **:** φ est vraie à l'événement suivant (ne**X**t).
- **G** φ *:* φ est toujours vraie à partir de l'événement courant (**G**lobally).
- **F** φ : φ est vraie dans un certain événement (**F**inally).
- φ **U** ψ : φ est vraie dans tous les événements du futur jusqu'à un événement où ψ est vraie (**U**ntil).

Sémantique

En LTL, une trace des messages σ satisfait φ (i.e. $\sigma \vDash \varphi$) si et seulement si elle respecte les règles suivantes :

$\sigma \vDash p = \text{v} \Leftrightarrow \sigma_1(p) = \text{v}$.

$\sigma \vDash \neg \varphi \Leftrightarrow \sigma \nvDash \varphi$

$\sigma \vDash \varphi \wedge \psi \Leftrightarrow \sigma \vDash \varphi$ et $\sigma \vDash \psi$

$\sigma \vDash \varphi \vee \psi \Leftrightarrow \sigma \vDash \varphi$ ou $\sigma \vDash \psi$

$\sigma \vDash \mathbf{X}\varphi \Leftrightarrow \sigma^2 \vDash \varphi$

$\sigma \vDash \mathbf{G}\varphi \Leftrightarrow \sigma \vDash \varphi$ et $\sigma^2 \vDash \mathrm{G}\varphi$

$\sigma \vDash \mathbf{F}\varphi \Leftrightarrow \sigma \vDash \varphi$ ou $\sigma^2 \vDash \mathrm{F}\varphi$

$\sigma \vDash \varphi \, \mathbf{U} \, \psi \Leftrightarrow$ il existe un entier k tel que $\sigma^k \vDash \psi$ et $\sigma^i \vDash \varphi$ pour tout $i < k$

Équipée de cette logique, il existe des contraintes, parmi celles qu'on a vues dans le chapitre 2, qui peuvent être formalisées dans LTL. On note que :

- Dans le scénario « Booksotre », la contrainte P1 en LTL s'écrit :

$$\mathbf{F}\ \text{rec\_acc} \rightarrow (\neg\ \text{rec\_acc}\ \mathbf{U}\ \text{place\_c\_order})$$

Cette formule exprime qu'éventuellement l'événement rec_acc ne peut pas se produire à moins que l'événement place_c_order n'ait été vu précédemment.

Aussi, la contrainte P2 peut être exprimée en LTL suivant cette formule :

$$\mathbf{G}\ (\text{place\_c\_order} \rightarrow\ \mathbf{F}\ \text{rec\_acc})$$

Cette formule exprime que globalement après un message place_c_order, il y a éventuellement un message rec_acc.

La propriété P3 du scénario Amazon E-commerce web service en LTL est :

$$\neg(/\text{Action} = \text{CartAdd})) \; U \; (/\text{Action} = \text{CartCreate}))$$

Dans le scénario LLRP, la propriété P5 s'exprime de la façon suivante :

$$\mathbf{G} \; (\; \text{ENABLE\_ROSPEC} \rightarrow \mathbf{F} \; \text{DISABLE\_ROSPEC})$$

Globalement après un événement ENABLE_ROSPEC, il y a éventuellement un événement DISABLE_ROSPEC qui se déclenche.

Finalement, la propriété P6 peut être exprimée en LTL suivant cette syntaxe :

$$\mathbf{F} \; \text{ENABLE\_ROSPEC} \rightarrow (\neg \; \text{ENABLE\_ROSPEC} \; \mathbf{U} \; \text{DISABLE\_ROSPEC})$$

Exemple d'évaluation d'une propriété LTL

Les règles sémantiques précédentes nous donnent une manière d'évaluer les propriétés.

Soit la trace $\sigma =$ `aab` et la propriété $\varphi =$ `F(a→Fb)`.

La Figure 3.7 décrit un exemple de vérification d'une propriété LTL sur une trace σ.

`(aab)` $\models$ `F(a→Fb)` $\Leftrightarrow$ `aab` $\models$ `a→Fb` ou ab $\models$ F(a→Fb)

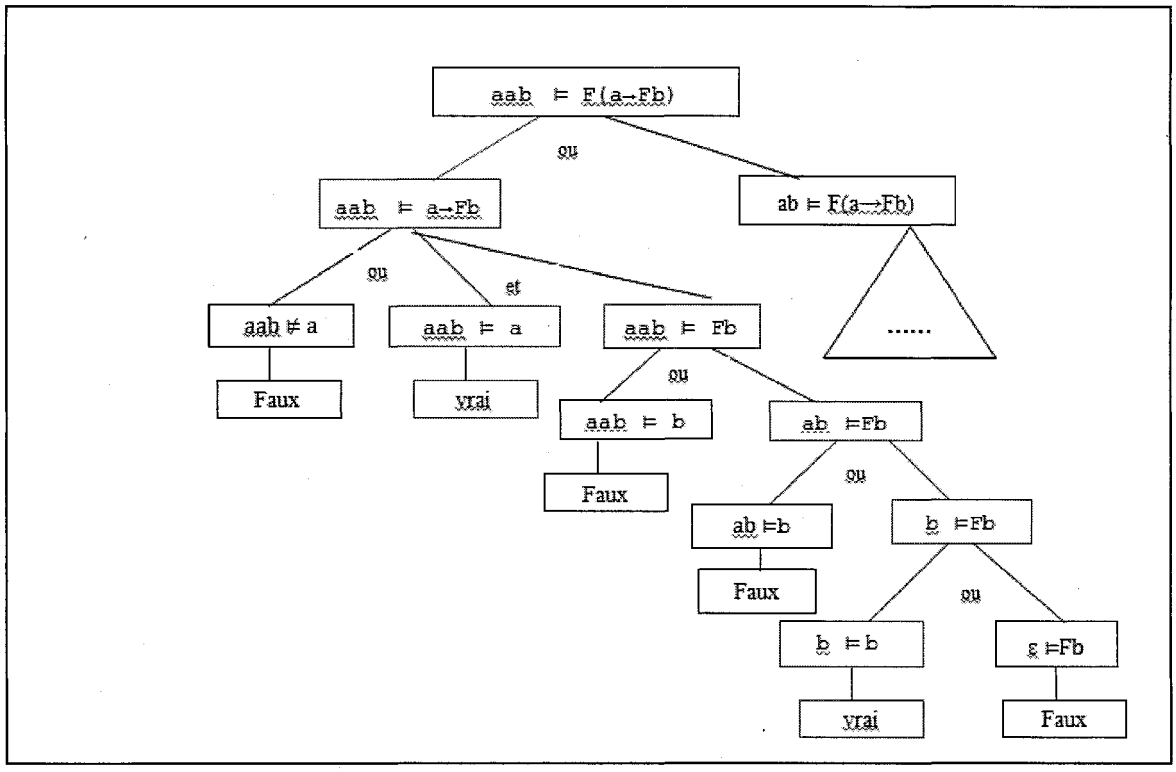


Figure 3.7: Étapes de vérification de $\sigma \models \varphi$

On divise la formule en deux sous formules et la trace σ doit satisfait la première sous formule et σ^i , avec i ≥ 1, satisfait la deuxième sous formule. On répète le même processus jusqu'à on arrive à la vérification des atomes.

3.3.4 LTL-FO+

Cependant, il y a des contraintes qu'on ne peut pas exprimer facilement avec LTL. Prenons l'exemple de la propriété P8, il faut que cette propriété soit vraie pour toutes les valeurs, ce qu'on ne peut dire avec LTL. C'est pour cette raison qu'on a besoin d'autres logiques pour les exprimer.

On peut étendre les propriétés LTL avec des quantificateurs de logique du premier ordre pour pouvoir également spécifier des relations entre les paramètres des différents

événements. Des travaux récents (Hallé et al, 2008) ont montré comment une telle logique est suffisante pour décrire des contraintes dans de nombreux scénarios du monde réel. Cette logique est appelée LTL-FO+ «Linear Temporal Logic with Full First-order Quantification».

LTL –FO+ est nécessaire dans deux cas :

1. Traces multi valeurs : On peut vouloir qu'une propriété soit vraie pour toutes les valeurs d'un paramètre p, ou pour au moins une.
2. Pour corréler la valeur de plusieurs paramètres dans des événements différents.

LTL-FO+ est une extension de la logique LTL. Pour accéder au contenu des messages, on utilise les quantificateurs du premier ordre (quantification universelle ''∀'' et la quantification existentielle ''∃''), définis comme suit:

1. $\forall \mathbf{x} \in p$: signifie *pour toute* valeur x dans *p*. Dans ce cas *x* est une variable et *p* est une expression XPath utilisée pour trouver toutes les valeurs possibles de x. Par exemple si φ est une formule qui contient une variable x, alors, l'expression suivante $\forall \mathbf{x} \in$ **/ tag1 / tag2 :** φ signifie que toute valeur à la fin de / tag1 / tag2 satisfait φ.
2. $\exists$ **x** $\in$ **p** : signifie il existe x dans **p**. Dans ce cas l'expression $\exists$ **x** $\in$**/tag1 / tag2 :** φ signifie qu'il existe au moins une valeur qui satisfait φ.

On dit qu'une trace σ satisfait une formule LTL-FO+ φ et on la note : $\sigma \vDash \varphi$ si et seulement si elle respecte les règles suivantes : si φ est de la forme $\neg \psi$, $\psi \vee \psi'$, $\psi \vee \psi'$, $\mathbf{F}\psi$, $\mathbf{X}\psi$, $\psi \, \boldsymbol{U} \, \psi'$. La sémantique est identique à LTL. Soit q $\in \Pi$: un chemin. Les deux cas manquants sont définis comme suit :

$$\sigma \vDash \exists x_i \in q: \varphi \Leftrightarrow \sigma \vDash \varphi[b/x_i] \text{ pour au moins une valeur b } \in \sigma_1(q).$$

$$\sigma \vDash \forall x_i \in q: \varphi \Leftrightarrow \sigma \vDash \varphi[b/x_i] \text{ pour toutes valeurs de b } \in \sigma_1(q).$$

L'expression $\varphi[b/x_i]$ signifie que les occurrences de x_i sont remplacées par la constante b dans φ.

Avec la logique LTL-FO+, on peut exprimer les contraintes vues dans le chapitre 2. Par exemple, la propriété P4 s'écrit :

$$G\ (\forall x \in /message/Action:\ (x=CartAdd) \rightarrow \exists y \in /CartAdd/CartID:\ (True))$$

Les deux propriétés du scénario des nombres aléatoires s'écrivent en LTL-FO+ suivant cette forme.

P8 : $G\ (\forall x \in /message/p0:\ (x=0))$: cette propriété signifie que globalement p0=0.

P9 : $G\ (\forall x \in /message/p0:\ (x=0) \rightarrow \exists y \in /message/p0:\ (x=1))$: cette propriété signifie que globalement pour tout valeur p0=0 alors éventuellement il existe p0=1

La propriété P7 s'écrit en LTL-FO+ suivant cette forme :

$$G\ (\forall x1 \in /message/Tag :\ (x1 = 1) \rightarrow X(\exists x2 \in /message/Tag:\ (x2= 2)))$$

3.3.5 MFOTL

MFOTL (Metric First Order Temporal logic) est une généralisation de LTL-FO+. Dans cette logique, les combinateurs temporels sont paramétrés par I (un intervalle de temps). MFOTL est conçue, entre autres, pour exprimer des retards dans les contrats commerciaux. Les événements sont associés à un nombre réel qui indique le moment où ils se produisent, considéré comme une « horloge ».

On considère l'exemple d'un feu de circulation. La formule 'G (vert→vert $U_{(0,5)}$ rouge)' signifie que si le feu est vert il reste vert jusqu'à ce qu'il passe au rouge après au maximum 5 unités de temps.

Aucun des scénarios considérés au chapitre 2 ne nécessite MFOTL puisqu'aucune propriété ne requiert l'usage d'intervalles de temps. Cependant, comme cette logique est le langage d'entrée d'un des outils considérés dans ce travail, on a préféré le mentionner dans l'inventaire des langages possibles.

CHAPITRE 4

OUTILS D'ANALYSE DES TRACES D'ÉVÉNEMENTS

Pour garantir le bon fonctionnement d'un système, plusieurs techniques ont été développées au fil du temps. Les outils d'analyses des traces étudiées dans ce mémoire se divisent en trois catégories: l'analyse des logs, le model checking et la vérification au moment de l'exécution.

4.1 Analyse des logs

L'analyse des logs consiste à enregistrer la séquence d'événements dans un *log* sur un support persistant et à chercher dans la trace résultante une suite d'événements violant la spécification formelle. Plusieurs outils sont dédiés à la validation des traces. Parmi ceux-ci, on note Maude et Saxon. À cette liste, on ajoute le moteur de base de données MySQL, qui peut lui aussi être utilisé comme validateur de traces.

4.1.1 Maude

Maude est un langage de spécification et de programmation basé sur une théorie mathématique de la logique de réécriture, développé par Jose Meseguar et son groupe dans le laboratoire d'informatique en SRI Inetrnational (Meseguer, 1992),. La logique de réécriture est une logique qui décrit un système concurrent qui a des états et qui évolue en termes de transitions. Elle est basée sur des simples règles de déduction. Elle est représentée par la théorie de réécriture $T = (\sum, E, R)$. Dans cette logique, la structure statique du système est décrite par la signature $(\sum, E)$ qui représente les états d'un système (où $\sum$ est un ensemble de sortes et d'opérateurs, et E est un ensemble d'équations). L'aspect dynamique représenté par les règles de réécriture (R) décrivant les transitions

possibles entre les états du système concurrent, où R est de la forme $r : [t] \rightarrow [t']$, où t et t' sont des expressions dans un langage quelconque donné.

On considère la théorie de réécriture suivante exprimant l'ensemble des booléens tels que :

$\sum$ = {{bool}, {T, F : $\rightarrow$ bool, _and_ : bool × bool $\rightarrow$bool}}

E = {x and y = y and x, x and (y and z) = (x and y) and z}

R = {r_0: [T] $\rightarrow$ [T and T], r_1: [F] $\rightarrow$ [F and T]

Étant donné l'expression F and F, est- il possible d'arriver à F and T and F and T and T?

En appliquant des éléments de cette théorie, par exemple la règle r_1, on peut prouver que [F and F] $\rightarrow$ [F and T and F and T and T].

Maude permet de programmer à deux niveaux différents : Core Maude et Full Maude.

- Core Maude : programmé en C++, il implémente les fonctionnalités de base du logiciel et il intègre les modules fonctionnels et les modules systèmes.
- Full Maude : est une extension du premier niveau, programmé en Core Maude. Dans Full Maude, les modules de Maude peuvent être combinés pour progresser le développement.

Maude regroupe trois types de modules : les modules *fonctionnels*, les modules *systèmes* et les modules orientés *objets*. Dans notre mémoire, on s'intéresse seulement aux modules fonctionnels. Les modules fonctionnels sont utilisés pour la définition des types de données et des opérations sur ces données par le biais des théories équationnelles. Un module fonctionnel est déclaré suivant cette syntaxe :

fmod (nom de module) *is* (declaration et instructions) *endfm*.

Le corps du module fonctionnel se compose d'une collection de déclarations. Ces déclarations sont une succession d'importation, *sorts*, *subsorts*, *opérations*, *variables* et *équations*.

Des travaux ont montré qu'on peut écrire des règles de réécriture qui permettent d'effectuer la validation des traces. Plusieurs modules fonctionnels ont été utilisés pour le

monitoring (Rosu et al., 2005). Le module *ATOM* définit les propositions atomiques comme un type abstrait des données ayant un type, *Atom,* aucune opération ou contrainte est nécessaire pour ce module. Les noms des propositions atomiques correspondant aux événements seront automatiquement générés dans un autre module qui s'étend ATOM, comme des constantes de sorte Atom. Ceux-ci seront générés par l'observateur lors de l'initialisation de la surveillance, des propriétés réelles que l'on veut surveiller. Le deuxième module fonctionnel est appelé *TRACE*. Une trace d'exécution se compose d'une liste finie d'événements.

Un troisième module, appelé *LOGICS-BASIC,* est défini. Ce module introduit les ingrédients de base de la logique de monitoring, logique utilisée pour spécifier les exigences de la surveillance. Le module *PROP-CALC* présente le calcul propositionnel. Il définit les connecteurs usuels ('et' logique, ou logique, implication, équivalence et négation). Le dernier module, appelé *LTL*, définit la sémantique des combinateurs temporels. La définition complète de ces modules est donnée à l'annexe 1.

On considère l'exemple de feu de circulation qui change entre les couleurs vert, jaune et rouge. La propriété LTL qu'on peut vérifier sur ce scénario est après une couleur verte, il déclenche une couleur jaune. Cette propriété est exprimée en Maude selon la syntaxe suivante :

```
reduce vert, jaune, rouge, vert, jaune, rouge, vert, jaune,
rouge, rouge ⊨ [] (vert →!rouge U jaune) .
```

Pour vérifier cette propriété, il faut ajouter un module fonctionnel appelé *MY-TRACE.* Ce module importe le module LTL. Il contient aussi la déclaration des atomes.
Le module fonctionnel pour cet exemple est :

```
fmod MY-TRACE is
    extending LTL .
    ops rouge vert jaune : → Atom .
```

endfm

La notation des connecteurs dans Maude est un peu différente. Par exemple, pour la négation, on note !, globalement ([]), éventuellement (<>), ET logique (/ \ ), Ou logique (\ /) et NEXT (o).

4.1.2 Saxon

Le logiciel Saxon est un ensemble d'outils pour le traitement des documents XML. Il permet entre autres d'interroger le contenu d'un document au moyen de deux langages, XPath et XQuery. XPath est un langage utilisé pour sélectionner des parties d'un document XML. Une expression XPath peut être vue comme un chemin de localisation de un ou plusieurs nœuds (éléments, attributs, ...). Le résultat d'une expression XPath peut être un ensemble de nœuds, une chaine de caractères, un nombre ou un booléen. Prenons par exemple l'expression XPath suivante : /Trace/message/Items/Item[Price>20]. Cette expression permet de sélectionner tous les éléments Item de l'élément Items de l'élément message qui ont un prix supérieur à 20. Si on évalue cette requête sur le document « Amaxon.xml » suivant (voir Figure 4.1), on obtient :

```
<Item>
        <ItemId>13</ItemId>
        <Price>35</Price>
</Item>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<Trace>
        <message>
                <SessionKey>1234</SessionKey>
                <Action>CartCreate</Action>
                        <Items>
                                <Item>
                                        <ItemId>13</ItemId>
                                        <Price>35</Price>
```

```
                    </Item>
                </Items>
        </message>
        <message>
                <SessionKey>1234</SessionKey>
                <Action>CartCreateResponse</Action>
                <CartId>13</CartId>
                <Items>
                        <Item>
                                <ItemId>13</ItemId>
                                <Price>35</Price>
                                <Quantity>1</Quantity>
                        </Item>
                </Items>
        </message>
        <message>
                <SessionKey>1234</SessionKey>
                <Action>CartClear</Action>
                <CartId>3</CartId>
        </message>
        <message>
                <SessionKey>1234</SessionKey>
                <Action>CartClearResponse</Action>
                <CartId>3</CartId>
        </message>
</Trace>
```

Figure 4.1: Document « amazon.xml »

De son côté, XQuery est lui aussi un langage d'interrogation de données XML. On peut caractériser XQuery comme étant le « SQL de XML ». Il permet de faire des requêtes selon la structure ou encore les contenus en se basant sur des expressions XPath. Généralement, XQuery permet d'extraire des fragments XML, d'y effectuer des recherches et de générer des fragments XML. Soit l'exemple suivant :

```
for $t in document("amazon1.xml")//Trace/message
  return if ($x/@Action="CartCreateResponse")
  then <Cart>{data($x/CartId)}</Cart>
```

Cette requête permet de chercher tous les nœuds CartId avec le chemin //Trace/message dans amazon.xml dont l'Action est « CartCreateResponse ». Si on évalue cette requête sur le document « amazon.xml » précédant (voir Figure 4.2), on obtient :

```
<Cart>
        <CartId>13</CartId>
</Cart>
```

Les travaux de Hallé et al. (Hallé et Villemaire, 2010) ont tenté d'exploiter les engins de requête XML pour effectuer la tâche de validation des traces. En encodant les événements observés dans une structure XML, une trace peut être vue comme un grand document XML qui peut ensuite être interrogé au moyen des langages de requête comme XPath et XQuery. Il suffit pour ce faire de trouver un moyen de traduire une expression LTL quelconque en une requête XQuery équivalente ce que nous allons expliquer au chapitre 5. Les résultats initiaux montraient qu'il était possible d'utiliser un moteur XML commercial et de le transformer en validateur de traces.

4.1.3 MySQL

Les systèmes de gestion de la base de données (SGBD) sont des systèmes mûrs et optimisés permettant d'exécuter des requêtes sur de très grandes bases de données. MySQL est un SGBD open-source maintenu par Oracle. Il est généralement utilisé pour la gestion de la base de données avec une architecture client-serveur. Les bases de données sont interrogées au moyen de SQL (Structured Query language), un langage de manipulation et de contrôle de données pour les bases de données relationnelles. On considère l'exemple suivant correspondant à la propriété P8 du scénario Nombres aléatoires.

```
SELECT SQL_NO_CACHE msgno FROM (SELECT DISTINCT traceA0.msgno
FROM trace AS traceA0 JOIN (SELECT msgno FROM trace AS trace1 WHERE
`p0` = "0") AS traceB0 WHERE traceA0.msgno <= traceB0.msgno) AS tracefinal
                    WHERE `msgno` = 0;
```

De la même façon qu'avec Saxon, il est possible d'utiliser un moteur de base de données tel que MySQL et de le transformer en validateur de traces. Pour ce faire, on a procédé par l'enregistrement d'une trace d'événements d'une manière efficace dans une base de données. Par la suite, une traduction du contrat d'interface LTL en une requête SQL équivalente a été développée.

4.2 Model checking

La seconde feuille d'outils considérée regroupe des logiciels dits de « model checking ». Le model checking est un ensemble de techniques de vérification automatique des propriétés temporelles sur des systèmes. Il consiste à encoder le système dans un modèle mathématique ainsi que la propriété à vérifier, et ensuite à utiliser des méthodes mathématiques pour s'assurer que le modèle satisfait la propriété énoncée. Schématiquement, l'algorithme de model checking prend en entrée un système de transitions à état fini et une formule LTL et répond si toutes les exécutions possibles du système satisfont la spécification. Il retourne un contre-exemple si la propriété n'est pas vérifiée (Voir Figure 4.2). Ce contre-exemple donne une explication permettant de justifier pourquoi le modèle est erroné et permet de repérer l'emplacement de cette erreur dans le modèle ou dans la formule.

Il existe plusieurs outils académiques pour le model checking. Les deux outils les plus connus sont SPIN développé aux Bell Labs en 1980, et NuSMV développé à l'université Carnegie Mellon. Dans ce qui suit, nous détaillerons davantage la fonction de ces deux outils.

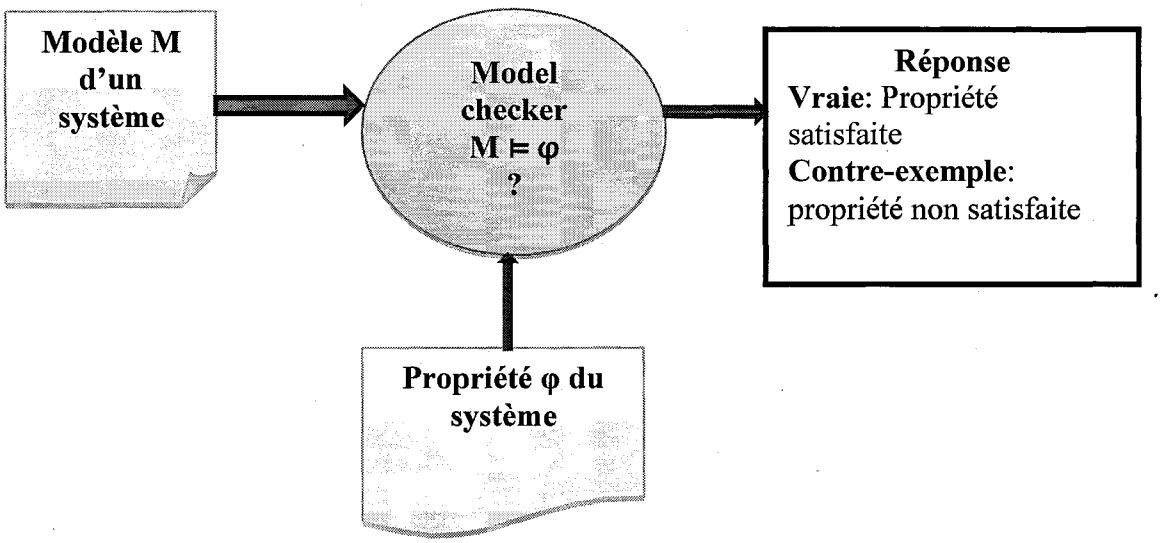


Figure 4.2: Le model checking, (Palshikar, 2005)

4.2.1 NuSMV

SMV (Symbolic Model Verifier) est le premier model checker symbolique, conçu par K. McMillan à l'université de Bologne. NuSMV est l'amélioration du model checker SMV. Il est une plateforme robuste, bien structurée et flexible pour le model checking symbolique. Il fut d'abord conçu pour être applicable à des projets de transfert de technologies (Cimatti et al., 2002). Il permet de modéliser un système en un réseau d'automates qui coopère par des variables partagées.

Le modèle NuSMV contient un module principal appelé main () (NuSMV, 2002). Le contenu de ce module est une succession de sections de type «VAR», «ASSIGN», «DEFINE» et «SPEC». On considère l'exemple suivant, celui de Bookstore de la Figure4.3. Ce modèle contient deux variables :

- m_num, qui peut prendre une valeur entière comprise entre 0 et 8.
- name, qui peut prendre le nom d'un événement de ce scénario.

Ces deux variables sont déclarées dans la section «VAR». On spécifie les valeurs initiales (avec init(variable)) ainsi que les valeurs à l'instant suivant (avec next(variable)) dans la section «ASSIGN». La valeur initiale de la variable m_num est 1 et place_c_order pour la variable name. Ces variables sont non seulement déclarées, mais on peut aussi spécifier leurs valeurs initiales ainsi que leurs évolutions dans le temps. Cette évolution est déclarée dans la section « TRANS ». La valeur suivante de m_num s'incrémente de 1 en passant à l'événement suivant et la variable name prend le nom de cet événement. Pour ce faire, on écrit une expression φ faisant intervenir les variables à l'état courant (X) et celles de l'état suivant (next(X)). Le passage X à next(X) est possible si φ(X, next(X)) = vrai. Si plusieurs paires X, next(X) sont possibles, une est choisie arbitrairement.

Les propriétés sont écrites en CTL, PSL (Property Specication Language) ou LTL. Dans ce mémoire, nous nous intéressons aux formules écrites en LTL. La propriété à vérifier est déclarée dans la section «LTLSPEC». Il s'agit ici de la propriété P2, exprimée en LTL.

```
-- Trace file automatically generated by
-- Event Trace Converter

MODULE main
VAR
  m_num : 0..8;
  name                                                                                    :
{decide,eval_b_order,place_c_order,c_reject,handle_c_order,rec_decl,b_reject,place_b_order,UND
EF};
ASSIGN
INIT
m_num = 0 & (
name = place_c_order)

TRANS
(next(m_num) = m_num + 1 | (m_num = 8 & next(m_num) = m_num)) & (
(next(m_num) = 0 ->
  (next(name) = place_c_order))
&
(next(m_num) = 1 ->
  (next(name) = handle_c_order))
```

```
&
(next(m_num) = 2 ->
  (next(name) = place_b_order))
&
(next(m_num) = 3 ->
  (next(name) = eval_b_order))
&
(next(m_num) = 4 ->
  (next(name) = b_reject))
&
(next(m_num) = 5 ->
  (next(name) = decide))
&
(next(m_num) = 6 ->
  (next(name) = c_reject))
&
(next(m_num) = 7 ->
  (next(name) = rec_decl))& (next(m_num) = 8 ->
  (next(name) = UNDEF)))
LTLSPEC
G ((name = place_c_order) -> (F (name = rec_acc)))
```

Figure 4.3: Une trace du scénario Bookstore écrite sous forme d'un module NuSMV

4.2.2 SPIN/Promela

SPIN (Simple Promela Interpreter) est un outil logiciel pour la vérification des modèles, développé par une équipe dirigée par Gerard J. Holzmann des laboratoires Bell (Holzmann, 1997). Il permet de vérifier l'exactitude des modèles de logiciels distribués d'une manière rigoureuse et largement automatisée. Il permet de vérifier des propriétés exprimées en logique temporelle sur des modèles décrits dans le langage PROMELA (Process Meta Language). L'outil SPIN comporte deux modes : un mode de simulation qui permet de simuler les comportements possibles du système et un mode de vérification qui détermine si le modèle satisfait ou non la propriété LTL.

Pour la vérification des propriétés LTL, l'outil SPIN utilise une méthode basée sur la construction du produit synchronisé du système à vérifier et d'un automate permettant de reconnaitre des séquences infinies validant une propriété LTL (Vardi et Wolper, 1986). Les

principes de cette vérification reposent sur le fait que l'ensemble des mots infinis satisfaisant une formule LTL forme un langage régulier et est représentable par un ω-automate. On résume cette méthode dans les étapes suivantes comme la montre la Figure4.4:

1- Traduction du modèle programmé initialement dans Promela en un premier automate de Büchi A;
2- Négation de formules logiques LTL en un second automate de Büchi $B_{\neg\varphi}$ permettant de reconnaitre un contre-exemple de la propriété φ;
3- Calcul du produit synchronisé de ces deux automates $A \otimes B_{\neg\varphi}$ qui est aussi un automate de Buchi. Donc, tester si un modèle $M \models \varphi$ se ramène à un problème sur les automates : tester si $L(A) \cap L(B_{\neg\varphi}) = \emptyset$ c.-à-d. n'accepte pas de mot infini. Autrement, s'il n'existe pas un cycle d'acceptation dans $A \otimes B_{\neg\varphi}$, alors le modèle satisfait la propriété φ.

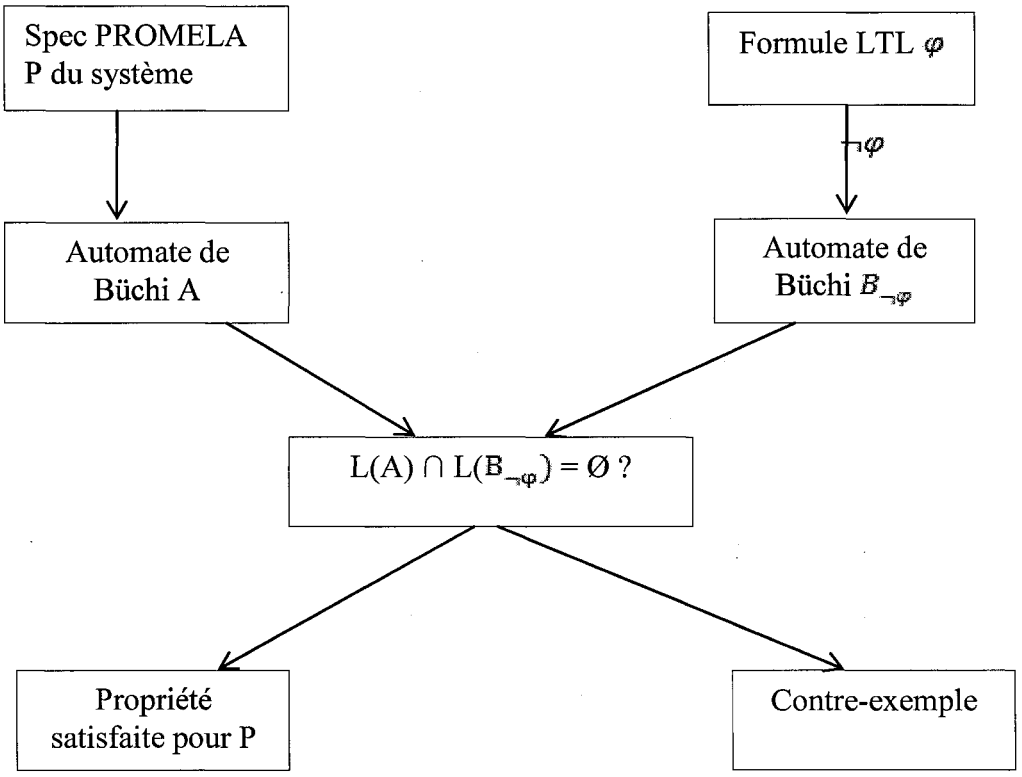


Figure 4.4: Étapes de vérifications d'une propriété avec SPIN

Promela (PROcess MEta LAnguage) est un langage de spécification de systèmes parallèles asynchrones. Il ressemble au langage C. Un modèle Promela est constitué des processus, des variables et des canaux de communications servant à transmettre des messages (Holzmann, 2000).

- Un processus est défini par ses éventuels paramètres et variables locales et un ensemble d'instructions. Un modèle Promela doit contenir au moins un processus. Le langage Promela prévoit l'emploi d'un processus principal (init). Ce processus est unique et est considéré comme initialement actif. On déclare un processus avec le mot clé «proctype».
- Les types de données de base en Promela sont bit, bool byte, mtype, short, et int. Les variables peuvent être déclarées aussi comme des tableaux d'une seule dimension. On peut définir aussi des structures similaires à celle du langage C avec le mot clé *typedef.*

- Les canaux de communications sont utilisés pour modéliser le transfert de données d'un processus à un autre. Plusieurs problèmes surviennent lors de l'interaction entre les processus. On note à titre d'exemple, l'interblocage, les modifications inattendues de valeurs des variables.

Promela, comme les langages structurés, dispose de différents types d'instructions. De plus, on peut utiliser les séquences atomiques pour éviter les problèmes d'accès à une variable globale. Pour ce faire, il faut regrouper plusieurs instructions en une seule séquence en utilisant le mot clé *atomic*. Cette séquence sera exécutée comme une seule instruction qui ne peut pas être interrompue, sans que d'autres instructions venant d'autres processus s'intercalent entre eux. On considère l'exemple suivant (voir Figure 4.5):

```
string name;
int msgno = 0;
active proctype A(){
do
::(msgno  == 0) ->
atomic {
 msgno = msgno + 1;
        name = place_c_order ;
}
::(msgno  == 1) ->
atomic {
 msgno = msgno + 1;
        name = handle_c_order ;
}
::(msgno  == 2) ->
atomic {
 msgno = msgno + 1;
        name = place_b_order ;
```

```
}
::(msgno == 3) ->
atomic {
 msgno = msgno + 1;
        name = eval_b_order ;
}
::(msgno == 4) ->
atomic {
 msgno = msgno + 1;
        name = b_reject ;
}
::(msgno == 5) ->
atomic {
 msgno = msgno + 1;
        name = decide ;
}
::(msgno == 6) ->
atomic {
 msgno = msgno + 1;
        name = c_reject ;
}
::(msgno == 7) ->
atomic {
 msgno = msgno + 1;
        name = rec_decl ;
}
od;
}
{run A();}
```

Figure 4.5: Une trace du scénario Bookstore écrite sous la forme d'un module PROMELA

Dans cet exemple, deux variables globales msgno et name sont déclarées : la variable msgno prend une valeur entière comprise entre 0 et 8 et la variable name, peut prendre le nom d'événement de ce scénario. Un processus, appelé A est défini. Ce processus est instancié par le mot clé *init*.

4.2.3 Autres outils

Plusieurs autres outils de vérification sont dédiés à l'analyse des traces en utilisant la technique de model checking. Parmi ces outils, on cite UPPAAL (UPPAAL CoVer, 2011), CPN Tools (CPN, 2010) et CPAchecker (CPAchecker, 2007). UPAAL est un outil qui permet de modéliser, simuler et vérifier les systèmes temps réel. Il reçoit en entrée un modèle formé par un réseau d'automates temporisé. Il est différent des autres outils à cause qu'il ne fournit pas un contre-exemple lorsque la propriété est violée. Ces propriétés sont exprimées en TCTL (Temporal Computation Tree Logic). CPN Tools (Colored Petri Net Tools) est un outil d'édition, de simulation et d'analyse des réseaux de Pétri. Il reçoit en entrée un réseau de Pétri basique, temporel ou coloré. Lors de la vérification, CPN Tools combine la capacité de réseaux de pétri colorés et la capacité d'un langage fonctionnel.

4.3 Vérification au moment de l'exécution (runtime monitoring)

Le runtime monitoring est un système d'analyse informatique, basé sur l'extraction d'informations à partir d'un système en cours d'exécution. Il utilise ces informations pour détecter et réagir aux comportements observés qui satisfont ou violent certaines propriétés. Il est utilisé pour plusieurs objectifs comme la sécurité ou le suivi des politiques de sécurité, le test, la vérification et la modification des comportements.

Le runtime monitoring évite la complexité des techniques traditionnelles de la vérification formelle, tel que le model checking, en travaillant directement avec le système en cours. Il évite l'étape fastidieuse de la modélisation formelle du système. Plusieurs outils et algorithmes ont été développés pour le runtime monitoring, parmi lesquels, on cite BeepBeep, Monpoly, Orchids, Monid et Logscope.

4.3.1 BeepBeep

Le BeepBeep Monitor (Hallé, 2008) est un moniteur pour les applications web AJAX et les programmes Java, développé par Sylvain Hallé. Une application standard Ajax communique avec un service Web en envoyant et recevant des messages SOAP via l'objet standard XMLHttpRequest fourni par le navigateur local. Ainsi, BeepBeep peut être utilisé sur le côté client, à l'intérieur de ces applications Web Ajax (voir Figure 4.6).

BeepBeep vérifie de façon transparente et en temps réel si les messages et les événements reçus et envoyés par une application satisfaire un contrat d'interface prédéfinie. Ces contrats d'interfaces sont exprimés en LTL-FO+. Cette description peut imposer des contraintes sur l'ordre des messages, les valeurs de données à l'intérieur d'un ou de plusieurs messages, ou une combinaison entre les deux. L'utilisation de BeepBeep empêche une application AJAX d'envoyer des messages erronés tout le long du chemin vers le serveur, en économisant la bande passante et le temps de traitement du serveur.

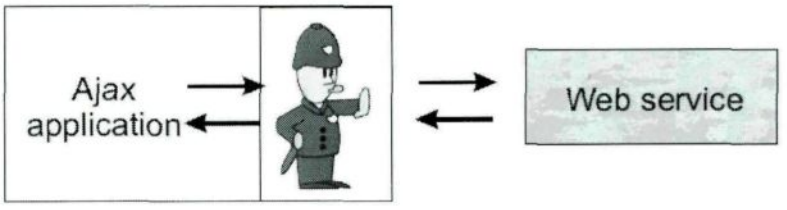


Figure 4.6: Le moniteur BeepBeep (Hallé et al., 2010)

BeepBeep surveille les conversations spécifiées au niveau des messages XML. Il est indépendant de toute implémentation client et ne fait pas référence à une variable interne du code source du client. Il est donc non invasif et peut faire respecter les spécifications de manière transparente avec des modifications mineures au code des applications.

Pour évaluer une formule LTL-FO+, BeepBeep utilise un algorithme de manipulation symbolique (Hallé et Villemaire, 2011) presque identique à la Figure 3.7.

4.3.2 Monpoly

Monpoly (Basin et al., 2012) est un moniteur pour le contrôle de conformité des fichiers logs en rapport avec les politiques. Il vise une vérification automatique de la conformité dans les systèmes d'informations où les actions sont effectuées par des composants de systèmes distribués et hétérogènes. Il traite un flux d'événements système avec des identificateurs représentant les données concernées et rapporte la politique des violations.

Monpoly prend en entrée un fichier log, un fichier signature et un fichier politique. Il émet des violations de la politique spécifiée.

Le fichier log représente une séquence d'événements horodatés, qui sont ordonnés par leurs « timestamp ». On considère l'exemple de fichier log suivant :

@1307532861 approve (52)

@1307955600 approve (63)

publish (60)

@1308477599 approve (87)

publish (63) (52)

Selon ce fichier log, le rapport avec le numéro 52 a été approuvé au moment du point 0 (time-point 0) avec un timestamp 1307532861(2011-06-08, 11:34:21 en temps UNIX). Et il a été publié au moment du point 2 (time-point 2) avec un timestamp 1308477599 (en 2011-06-19). Le fichier politique représente les propriétés à vérifier. Ces propriétés sont spécifiées en MFOTL (metric first order temporel logic). Par exemple, on veut vérifier que tous les rapports financiers doivent être approuvés au plus une semaine avant leur publication. En MFOTL, cette propriété est la suivante :

$$\text{publish}(?r) \rightarrow \text{ONCE}[0,7d]\ \text{approve}(?r).$$

Monpoly accepte les connecteurs booléens écrits en toutes lettres. Par exemple, pour 'implique', on écrit « IMPLIES » et pas « → ». La propriété dans ce cas devient :

publish(?r) IMPLIES ONCE[0,7d] approve(?r).

Les arités des prédicats et les types d'arguments sont spécifiés dans le fichier signature. Le fichier signature correspondant à cet exemple est :

publish (x :int)

approve (x : int)

Monpoly traite le fichier log progressivement et il renvoie pour chaque time-point toutes les violations de la politique.

Monpoly est écrit dans un langage de programmation appelé OCaml.

4.3.3 Autres outils

Plusieurs autres outils de runtime monitoring ont été développés pour la surveillance des services web. Parmi ces outils, on trouve Orchids (Olivain et Goubault-Larrecq, 2005), Monid (Naldurg et al., 2004), Logscope (Barringer et al., 2004) et RuleR (Barringer et al., 2010).

Orchids est un système de détection des intrusions en temps réel basé sur la logique temporelle LTL, développé par Julien OLIVAIN dans le laboratoire LSV (Laboratoire Spécification et Vérification) de l'ENS Cachan. Monid est semblable à Orchids, basé sur EAGLE (Barringer et al., 2004). Les propriétés sont spécifiées par des systèmes avec des équations paramétrées par des opérateurs booléens et temporels et une sémantique, appelée Xpoint. Finalement, Logscope peut être considéré comme une restriction de RuleR. Ces propriétés sont exprimées sous forme d'une conjonction des paramètres temporels et de machines à états finis.

CHAPITRE 5

EXPÉRIMENTATION

Comme on l'a mentionné plus tôt, il n'existe, à ce jour, aucune comparaison empirique des outils de vérification de traces décrits au chapitre précédent. Ce chapitre présente BabelTrace, un environnement d'exécution de type « benchmark » visant à combler ce manque. Le code source de BabelTrace est disponible sous licence GPL : https://github.com/sylvainhalle/TraceAdapter. Dans la première partie, on décrit l'architecture BabelTrace. La deuxième partie introduit l'ensemble des générateurs des traces développées, dont le but de mesurer les performances des outils décrits dans le chapitre précédent. Ensuite, nous expliquons comment un problème de validation de traces pour un outil peut être traduit en un problème équivalent pour un autre. Enfin, on présente les résultats obtenus.

Pour fixer les idées, nous allons illustrer le fonctionnement de BabelTrace en utilisant la même trace et la même propriété pour tous les outils, soit la propriété P2 et la trace suivante :

```
<?xml version="1.0" encoding="UTF-8"?>

<Trace>
<message>
        <name>place_c_order</name>
</message>
<message>
        <name>handle_c_order</name>
</message>
<message>
        <name>place_b_order</name>
</message>
<message>
        <name>eval_b_order</name>
</message>
<message>
        <name>b_reject</name>
</message>
```

```
<message>
        <name>decide</name>
</message>
<message>
        <name>c_reject</name>
</message>
<message>
        <name>rec_decl</name>
</message>
</Trace>
```

5.1 Implémentation

La Figure 5.1 résume l'architecture « BabelTrace » mise en œuvre. Cette architecture permet le traitement uniforme des traces d'événements indépendamment de la syntaxe propre à chacun des outils et du format de fichier représentant les traces à valider.

BabelTrace contient un ensemble des classes permettant le traitement de fichier d'entrée et la récupération de trois paramètres (le résultat de vérification de la propriété, le temps alloué pour la vérification de la propriété et la mémoire utilisée). Ces classes sont :

1- La classe lecteur (ou Reader) : prend en entrée une trace d'événements d'un scénario réaliste et la convertit sous la forme des traces présentées dans le chapitre 3. Actuellement, l'architecture supporte trois formats de fichier d'entrée : le format XML, le format XES et le format CSV.

2- La classe traducteur (ou Translator) : prend en entrée une trace τ et une propriété φ et génère les fichiers d'entrée dans la syntaxe propre à chaque outil.

3- La classe lanceur (ou Launcher) : permet de construire la syntaxe de la ligne de commande des outils à exécuter. Ensuite, elle analyse le résultat d'exécution et convertit la réponse pour décider si la trace τ satisfait la propriété φ. Enfin, elle récupère le temps d'exécution et la mémoire allouée pour la vérification de la propriété pour chacun des outils sélectionnés.

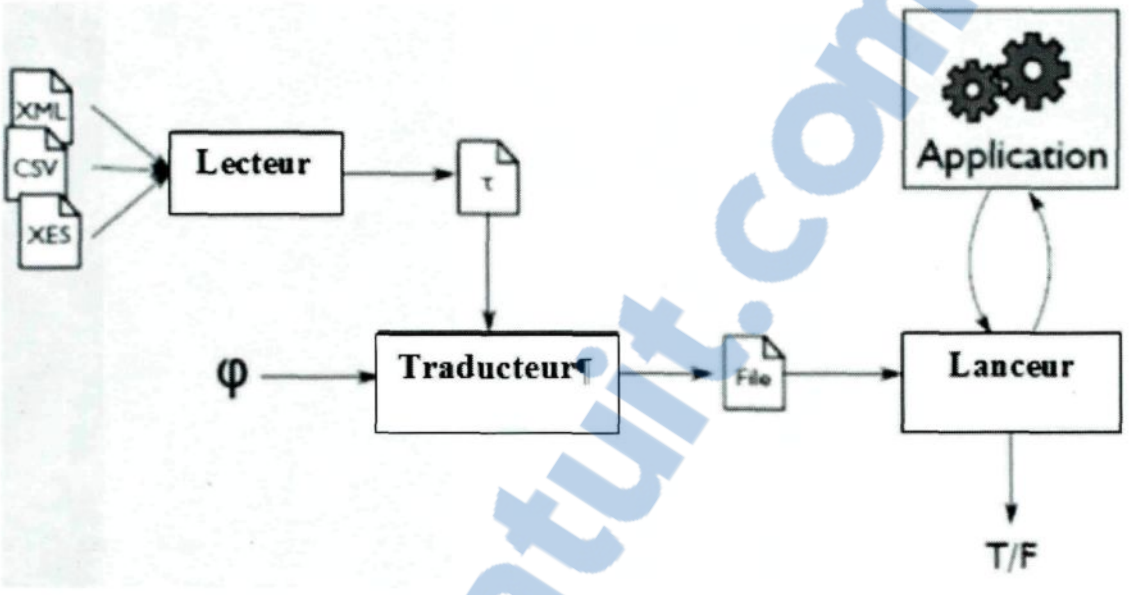


Figure 5.1: Architecture de BabelTrace

Une interface graphique a été élaborée pour faciliter le lancement des tâches de traduction et d'évaluation des performances de chaque outil. Cette interface est montrée dans la Figure 5.2 dans laquelle on distingue trois onglets : Trace Translator, Trace Generator et Runtime.

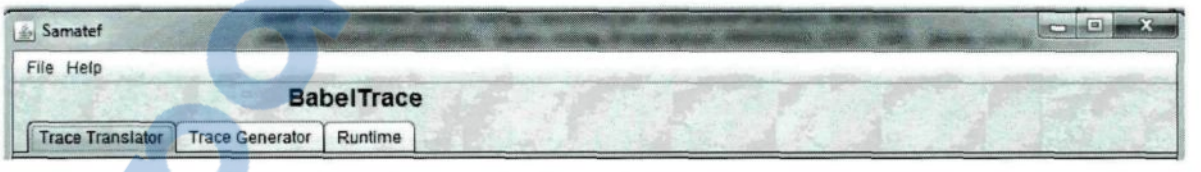


Figure 5.2: Aperçu de l'interface graphique principale

- ***Trace Generator*** (voir Figure 5.3): cet onglet permet de générer des traces qui décrivent l'ensemble des scénarios présentés dans le chapitre 2. Elle contient six générateurs : « *Amazon* », « *Random Trace* », « *Book strore* », « *Cycle generator* », « *LLRP generator* » et « *Petri Net* ». De plus, cet onglet permet de saisir le nombre de traces à générer et de les sauvegarder dans l'emplacement désiré. La section suivante détaillera les caractéristiques de chacun de ces six générateurs.

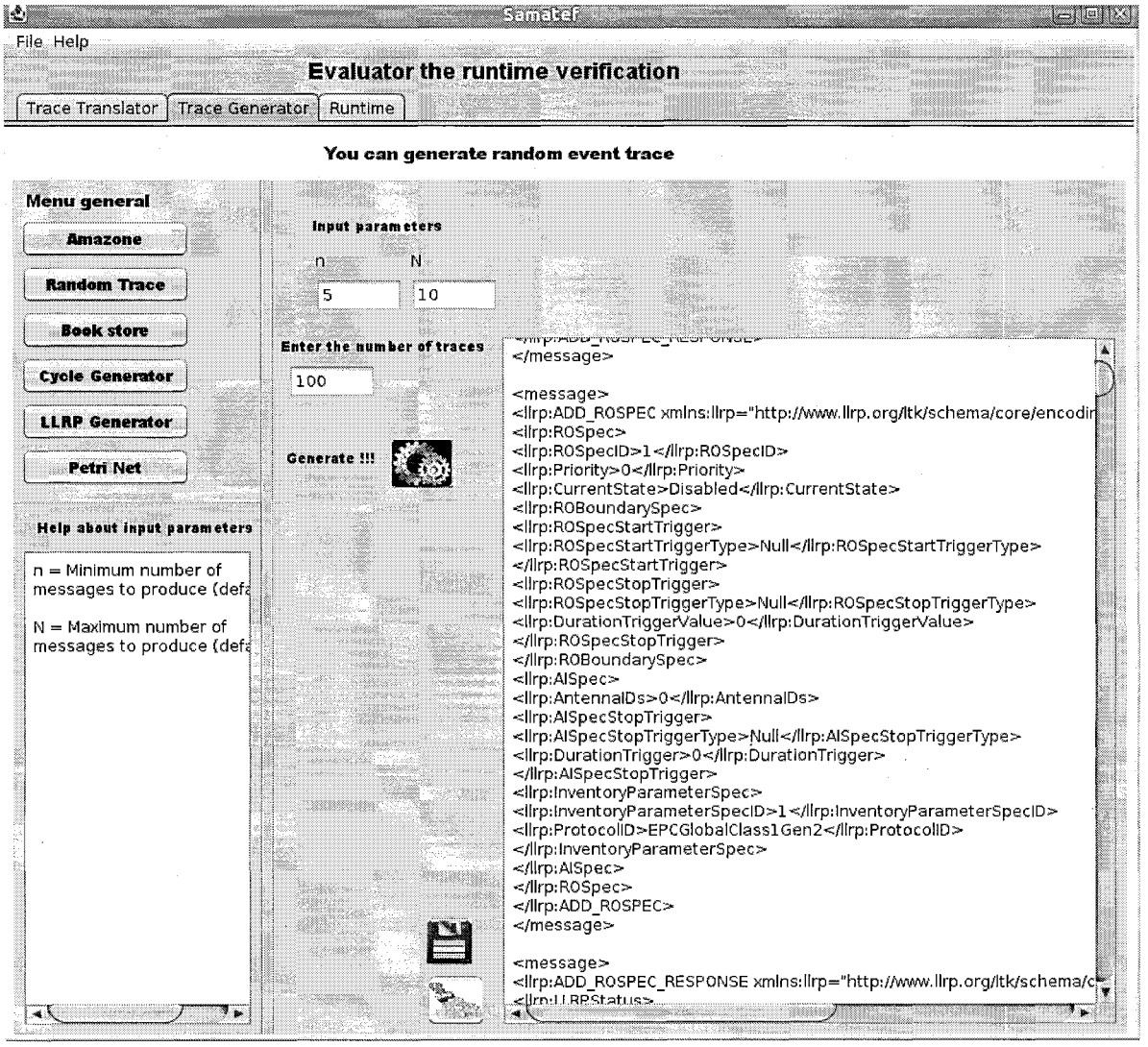


Figure 5.3: Aperçu de l'onglet « Trace generator »

- ***Trace Translator*** (voir Figure 5.4): cet onglet permet de prendre en entrée un fichier de trace et la propriété LTL à vérifier. Ces deux entrées sont, par la suite, traduites suivant la syntaxe propre à chacun des outils et des algorithmes présentés dans le chapitre précédent. Suite à cette traduction, les données générées seront sauvegardées dans un fichier dont l'extension varie selon l'outil sélectionné.

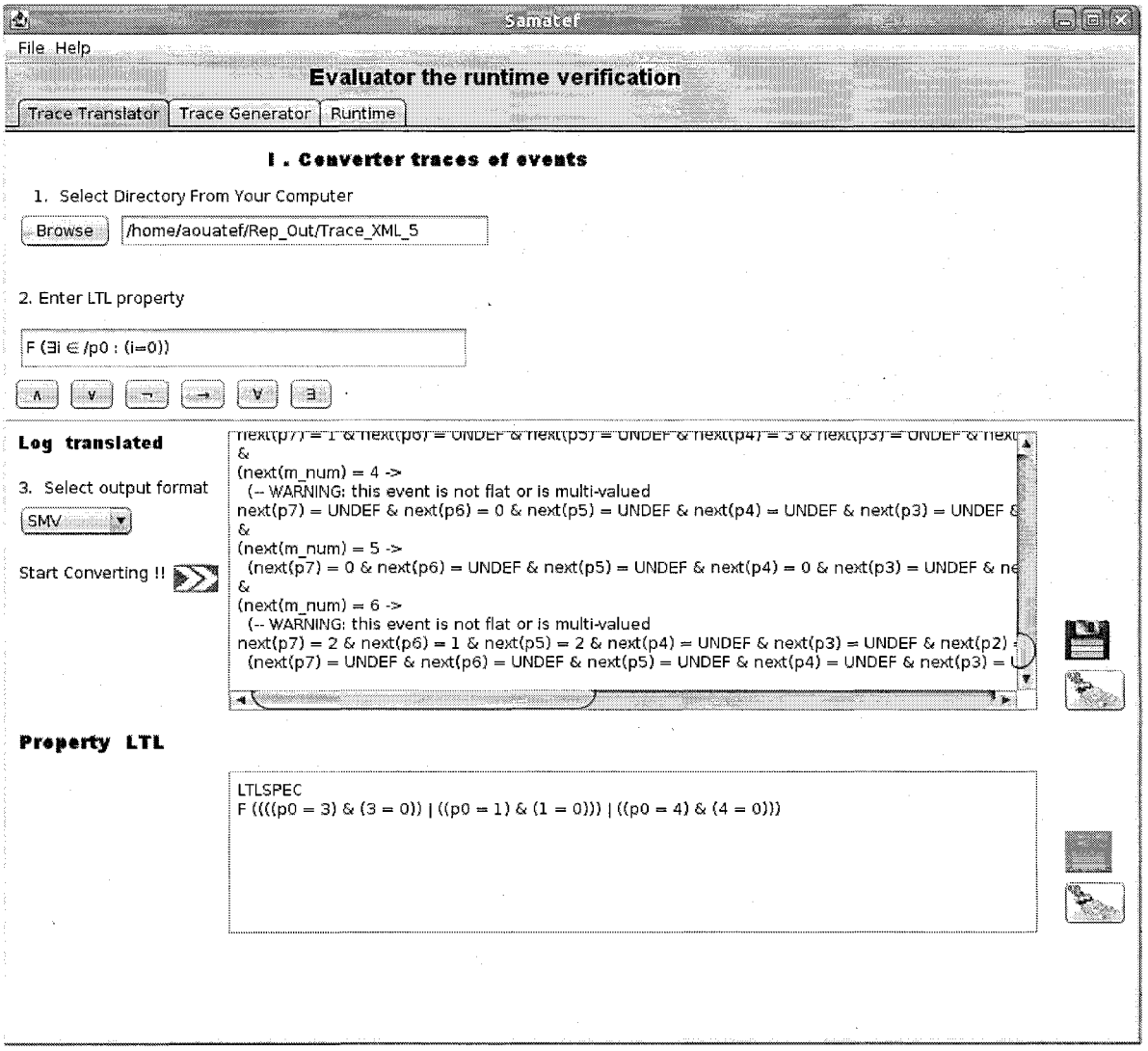


Figure 5.4: Aperçu de l'onglet « Trace translator »

- ***Runtime*** (voir Figure 5.5): l'entrée de cet onglet est un dossier qui contient l'ensemble des traces à valider. L'architecture BabelTrace supporte actuellement les sept outils présentés dans le chapitre précédent. Elle permet de sélectionner les outils dont on veut évaluer les performances. La sortie de cet onglet est un fichier CSV. Pour chacun de ces outils et chaque trace, y donnent le temps d'exécution, la mémoire consommée et le résultat de la vérification. De plus, un graphique, décrivant le temps d'exécution en

fonction de nombre d'événements, est généré en temps réel lors de la vérification pour l'ensemble des outils.

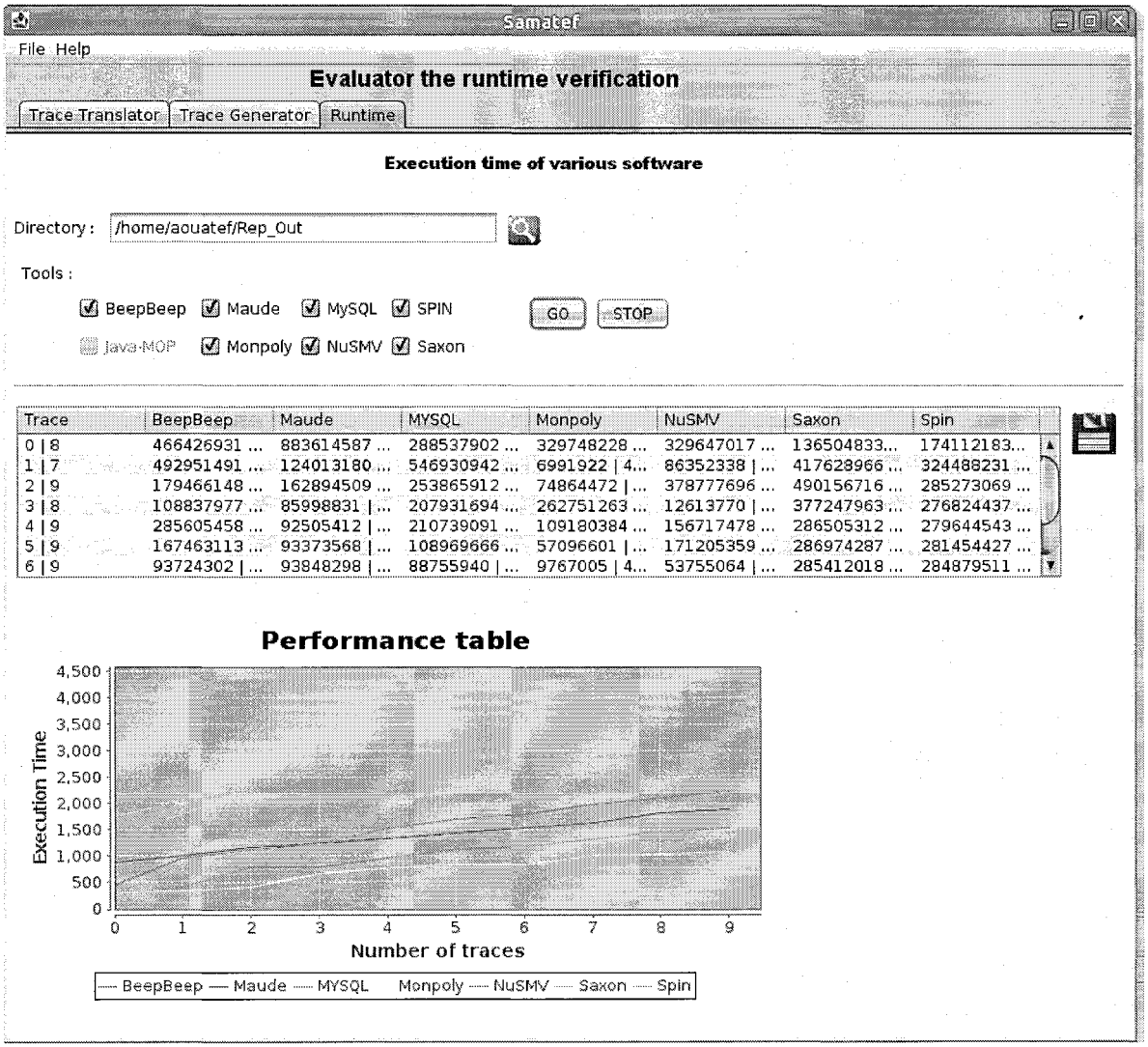


Figure 5.5: Aperçu de l'onglet « Runtime »

5.2 Les générateurs des traces

Pour chacun des cinq scénarios décrits dans le chapitre 2, un générateur a été développé. L'ensemble de ces générateurs se base principalement sur une fonction

aléatoire. Les points communs entre ces générateurs sont le nombre de traces et l'intervalle de nombres des événements à produire. Chaque générateur possède ses propres paramètres d'entrée et de sortie. Les traces d'événements valides sont générées à l'intérieur de ces paramètres.

5.2.1 Amazon

Le générateur des traces Amazon permet de générer des traces aléatoires pour le service web Amazon ECS. Les opérations possibles sur les paniers sont : *CartAdd, CartCreate, CartClear, CartRemove, CartEdit* et *ItemSearch*. Pour générer des traces, il faut définir les paramètres suivants:

- Le nombre maximal de paniers
- Le nombre minimum des messages à produire
- Le nombre maximal des messages
- La taille maximale de chaque panier
- La taille du catalogue du magasin
- Le nombre de traces

5.2.2 Bookstore

Le générateur Bookstore génère des traces aléatoires de requêtes et de réponses pour la librairie Amazon. Les paramètres d'entrées de ce générateur sont :

- Le nombre minimum des messages à produire
- Le nombre maximal des messages
- Le nombre des traces à générer

5.2.3 Cycle Generator

Le générateur Cycle Generator génère des traces d'événements de façon aléatoire pour les tags RFID. Il prend comme paramètres :

- l'intervalle de nombre des messages à produire
- Le nombre maximal des tags

5.2.4 LLRP generator

Le ROSpec est une structure des données qui définit le LLRP spec. On rappelle qu'un ROSpec est un processus simple qui se déplace entre trois états (« Disabled », « Active » et « Inactive ») comme il a été expliqué dans le chapitre 2. Le générateur LLRP génère, alors, des traces dont les événements sont les transitions possibles entre ces trois états. Ce dernier prend comme paramètre :

- L'intervalle de nombre de messages à produire
- Le nombre de traces à générer

5.2.5 Random generator

Le générateur Random produit des traces d'événements de façon aléatoire en se basant sur un certain nombre de paramètres :

- L'intervalle pour le nombre de messages à produire
- L'arité maximale de chaque message
- Le nombre des noms des paramètres disponibles
- La taille maximale de domaine pour tous les paramètres
- Si les événements sont multivaleurs ou non

5.3 Traductions et transductions

Les outils et les algorithmes d'analyse des traces cités possèdent des formats d'entrées différents, comme le montre la Figure 5.6. On peut classer ces outils en six classes selon leurs formats d'entrées:

- La première classe contient les deux outils NuSMV et SPIN. Elles prennent des traces à une seule valeur et des propriétés exprimées en LTL.
- La deuxième catégorie comporte MySQL. Ce dernier prend en entrée une propriété exprimée en SQL et une trace à un schéma fixe.
- La troisième catégorie contient Maude, celui-ci prend en entrée une trace atomique et une propriété LTL.
- La quatrième classe contient BeepBeep. Ce dernier prend entrée une trace multi valeurs et une propriété LTL-FO+.
- La cinquième classe comporte Saxon, il prend en entrée une trace multi valeurs et une propriété exprimée en XQuery.
- La dernière catégorie comporte Monpoly, celui-ci prend en entrée une trace multi valeurs et une propriété MFOTL.

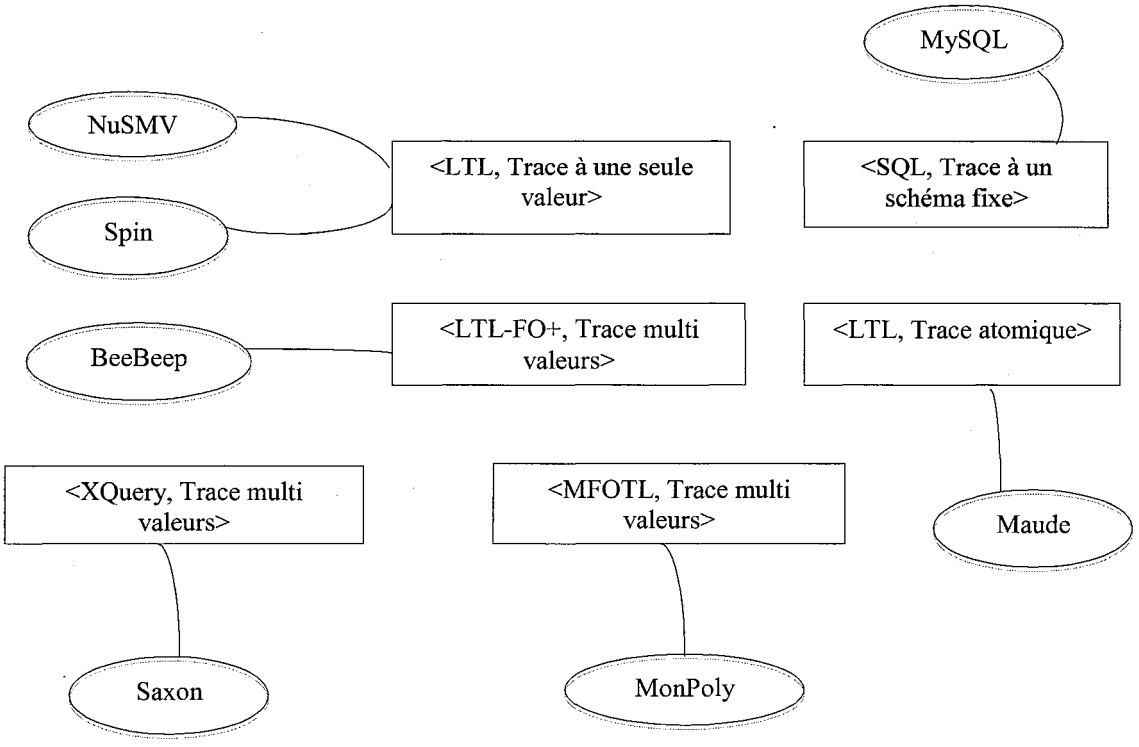


Figure 5.6: Formats d'entrées des outils

Il est impossible dans ce cas de mesurer directement la performance de ces outils. Pour résoudre ce problème, on a donc dû développer des traducteurs. Chaque outil est associé à une paire de trace/langage, noté $<\mathcal{T}, \mathcal{L}>$. En se basant sur la dernière classification des outils, un problème de validation des traces est donc une paire (τ, φ) où τ est une trace exprimée au format $\mathcal{T}$ et φ une spécification exprimée en langage $\mathcal{L}$.

Les transducteurs permettent de transformer un exemple de problème $(\tau, \varphi) \in <\mathcal{T}, \mathcal{L}>$ en un nouveau problème $(\tau', \varphi') \in <\mathcal{T}', \mathcal{L}'>$, de telle sorte que $\tau \models \varphi$ si et seulement si $\tau' \models \varphi'$. La Figure 5.7 est une carte des outils, des formats d'entrée et de transducteurs inclus dans BabelTrace. Dans cette figure, une flèche étiquetée par $A \xrightarrow{\subseteq} B$ indique que la transduction de A à B est trivialement implémentée puisque A est un cas particulier de B (par exemple la traduction de LTL vers LTL-FO+). Une flèche étoilée indique que la transduction est l'équivalent d'un fragment du langage saisi (par exemple la traduction d'une trace multi-valeurs et d'une propriété LTL-FO+ vers une trace à une seule valeur avec la même propriété).

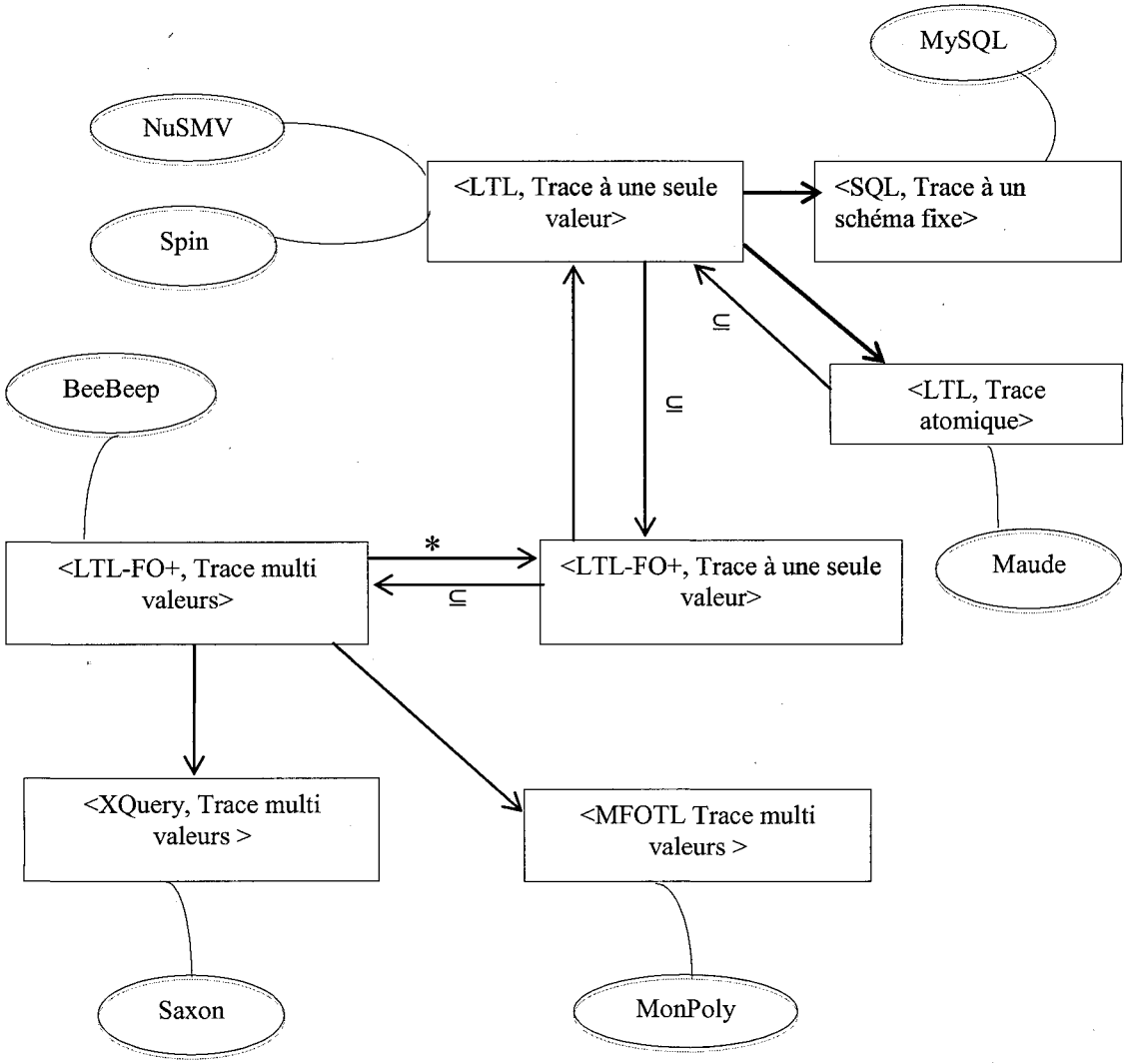


Figure 5.7: Carte des outils, des formats d'entrée et de transducteurs inclus dans BabelTrace

5.3.1 Transduction de LTL à des atomes

Cette transduction récupère toutes les valeurs requises par la formule LTL et crée un symbole atomique pour chaque combinaison de ces valeurs. Les prédicats sur les valeurs des paramètres de la formule LTL sont ensuite remplacés par des expressions booléennes sur ces symboles.

Le résultat de traduction de la propriété P2 et la trace donnée au début de ce chapitre est la suivante.

```
in ltl.maude
fmod MY-TRACE is
 extending LTL-REVISED .
 ops  e3 e4 e1 e2 e0 e7 e6 e5  : -> Atom .
endfm
reduce e4,e2,e5,e3,e7,e0,e6,e1 |- [] ((e4) -> (<> (false))).
 quit
```

5.3.2 Traduction de NuSMV

Elle se fait par la construction d'un système de transition complètement déterministe dont la seule exécution possible est la trace que l'on veut valider. La validation de trace est donc reformulée comme un cas particulier de model checking. La traduction de la trace énoncée au début de ce chapitre et de la propriété P2 est la suivante :

```
-- Trace file automatically generated by
-- Event Trace Converter

MODULE main

VAR
 m_num : 0..8;
 name                                                                                    :
{decide,eval_b_order,place_c_order,c_reject,handle_c_order,rec_decl,b_reject,place_b_order,UND
EF};

INIT
m_num = 0 & (
 name = place_c_order)

TRANS
(next(m_num) = m_num + 1 | (m_num = 8 & next(m_num) = m_num)) & (
```

```
(next(m_num) = 0 ->
  (next(name) = place_c_order))
&
(next(m_num) = 1 ->
  (next(name) = handle_c_order))
&
(next(m_num) = 2 ->
  (next(name) = place_b_order))
&
(next(m_num) = 3 ->
  (next(name) = eval_b_order))
&
(next(m_num) = 4 ->
  (next(name) = b_reject))
&
(next(m_num) = 5 ->
  (next(name) = decide))
&
(next(m_num) = 6 ->
  (next(name) = c_reject))
&
(next(m_num) = 7 ->
  (next(name) = rec_decl))& (next(m_num) = 8 ->
  (next(name) = UNDEF)))
LTLSPEC
G ((name = place_c_order) -> (F (name = rec_acc)))
```

5.3.3 Traduction de SPIN

La traduction d'une trace XML en un fichier PROMELA suit les mêmes étapes que celle utilisées dans NuSMV. Cette traduction se base principalement sur l'utilisation des séquences atomiques. Chaque trace est traduite en un seul processus.

```
-- Trace file automatically generated by
-- Event Trace Converter

string name;
int msgno = 0;
```

```
active proctype A(){
do
::(msgno == 0) ->
atomic {
 msgno = msgno + 1;
        name = place_c_order ;
}
::(msgno == 1) ->
atomic {
 msgno = msgno + 1;
        name = handle_c_order ;
}
::(msgno == 2) ->
atomic {
 msgno = msgno + 1;
        name = place_b_order ;
}
::(msgno == 3) ->
atomic {
 msgno = msgno + 1;
        name = eval_b_order ;
}
::(msgno == 4) ->
atomic {
 msgno = msgno + 1;
        name = b_reject ;
}
::(msgno == 5) ->
atomic {
 msgno = msgno + 1;
        name = decide ;
}
::(msgno == 6) ->
atomic {
 msgno = msgno + 1;
        name = c_reject ;
}
::(msgno == 7) ->
atomic {
 msgno = msgno + 1;
        name = rec_decl ;
```

```
}
od;
}
[] ((name==place_c_order) <-> (<> (name==rec_acc)))
```

5.3.4 Transduction de LTL à SQL

Cette transduction transforme une trace en une base de données, et une formule LTL en une requête SQL. La validation de trace est donc reformulée comme un cas particulier d'évaluation des requêtes SQL. La première étape de cette transduction consiste à enregistrer la trace dans une base de données. Nous avons créé une table, appelée T, dont les attributs sont les paramètres apparaissant dans les événements. Chaque événement d'une trace devient un tuple de T. Par exemple, le tableau suivant montre les premiers événements de la trace du scénario AWS-ECS.

n	Action	SessionKey	CartID	ItemID
0	CartCreate	124	null	null
1	CartCreateResponse	124	4321	null
2	CartAdd	124	4321	5
3	CartClear	124	4321	null

Puisque, les tuples dans une relation sont non ordonnés par définition, on ajoute un attribut, appelé n, qui contient un numéro de séquence conservant la relation d'ordre entre les événements.

La deuxième étape consiste à traduire une propriété LTL en une requête SQL équivalente. Pour ce faire, nous avons développé une fonction w_1 de traduction récursive ; chaque application de cette fonction construit une instruction SQL calculant une table T' avec un seul attribut n. Intuitivement, pour une formule LTL φ, la table résultante de

l'application de w_l (φ) contient exactement les n valeurs telle que $\sigma^n \models \varphi$. La fonction comporte également un paramètre *l*, qui est utilisé pour que chaque appel récursif de *w* peut générer des alias pour T. Lorsqu'on commence la traduction, on met *l* à 0.

Connecteurs booléens : il suffit de définir une fonction w_l pour chaque opérateur. Par exemple dans le cas des égalités, la traduction de ces termes revient à chercher les numéros d'événement où cette égalité est vraie:

$$w_l(\text{x= y}) \equiv \textbf{SELECT } n \textbf{ FROM } T \textbf{ WHERE } \text{x=y}$$

La conjonction et la disjonction logique sont traduites respectivement en union et intersection.

$$w_l(\varphi \wedge \psi) \equiv w_{l+1}(\varphi) \textbf{ INTERSECT } w_{l+1}(\psi)$$

$$w_l(\varphi \vee \psi) \equiv w_{l+1}(\varphi) \textbf{ UNION } w_{l+1}(\psi)$$

Cependant, puisque l'intersection n'est pas prise en charge par les moteurs de base de données (y compris MySQL), la traduction de conjonction logique doit être remplacée par une construction plus complexe simulant l'intersection avec une jointure interne:

$$w_l(\varphi \wedge \psi) \equiv \textbf{SELECT } T_{A,l}.*$$

$$\textbf{FROM } w_{l+1}(\varphi) \textbf{ AS } T_{A,l} \textbf{ INNER JOIN } w_{l+1}(\varphi) \textbf{ AS } T_{B,l}$$

$$\textbf{ON } T_{A,l}.\text{n} = T_{B,l}.\text{n}$$

La négation est la différence ensembliste :

$$w_l(\neg \varphi) \equiv \text{T } \textbf{MINUS } w_{l+1}(\varphi)$$

En tant que première optimisation, toutes les expressions booléennes qui ne contiennent pas d'opérateurs temporels s'appliquent à un seul événement dans la trace. Par conséquent, ils peuvent être traduits comme une seule instruction SELECT avec toute l'expression booléenne écrite dans la clause WHERE.

Opérateurs temporels : pour la traduction de l'opérateur Next (**X** φ), il suffit déplacer les valeurs de *n* contenues dans $w_{i+1}(\varphi)$ par une seule position. Autrement dit, si l'événement *k* satisfait φ alors l'événement *k*-1 satisfait **X** φ.

$$w_i(\mathbf{X}\ \varphi) \equiv \textbf{SELECT n-1 FROM } w_{i+1}(\varphi)$$

Selon la sémantique de LTL, une formule de la forme **F** φ est vraie si et seulement si φ est vrai plus tard dans au moins un événement de l'exécution. On obtient donc :

$$w_i(\mathbf{F}\ \varphi) \equiv \textbf{SELECT DISTINCT } T_{A,i}.\text{n}$$
$$\textbf{FROM T AS } T_{A,i} \textbf{ JOIN } w_{i+1}(\varphi) \textbf{ AS } T_{B,i}$$
$$\textbf{WHERE } T_{A,i}.\text{n} \leq T_{B,i}.\text{n}$$

La traduction de l'opérateur Until (φ **U** ψ) se fait en deux étapes :

1- Pour chaque état qui contient ψ, nous calculons l'état le plus proche dans le passé où φ n'apparait pas. Ceci crée une relation T_i' temporaire :

$T_i' \equiv$ **SELECT MAX** ($T_{A,i}$.n) **AS** n_1, $T_{B,i}$.n **AS** n_2

FROM $w_{i+1}(\neg\varphi)$ **AS** $T_{A,i}$ **JOIN** $w_{i+1}(\psi)$ **AS** $T_{B,i}$

WHERE $n_1 < n_2$ **GROUP BY** $T_{B,i}$.n

2- Les nœuds où φ **U** ψ est vraie sont ceux qui appartiennent à l'intervalle

[n_1+ 1, n_2], pour chaque tuple (n_1, n_2) de T'. La traduction de l'opérateur **U** devient :

w_i (φ **U** ψ) ≡ **SELECT** $T.n$ **FROM** T **JOIN** T_i'

WHERE $T.n > T'.n_1$ **AND** $T.n \leq T'.n_2$

Les opérateurs temporels restants sont traduits en les convertissant à l'un des cas précédents, grâce à l'application des identités standard (par exemple **G** φ ⇔ ¬ **F** ¬φ).

La traduction de la trace BookStore.xml et la propriété P2 est donnée dans la figure suivante.

```
USE TraceAdapter;

INSERT INTO trace (`msgno`,`name`)

VALUES

(0, "place_c_order"),

(1, "handle_c_order"),

(2, "place_b_order"),

(3, "eval_b_order"),

(4, "b_reject"),

(5, "decide"),

(6, "c_reject"),

(7, "rec_decl");


SELECT SQL_NO_CACHE msgno FROM (SELECT msgno FROM trace AS trace0 WHERE msgno NOT IN (SELECT DISTINCT traceA1.msgno FROM trace AS traceA1 JOIN (SELECT msgno FROM trace AS trace2 WHERE msgno NOT IN (SELECT msgno FROM ((SELECT msgno FROM trace AS trace5 WHERE `name` != "place_c_order") UNION (SELECT DISTINCT traceA4.msgno FROM trace AS traceA4 JOIN (SELECT msgno FROM trace AS trace5 WHERE `name` = "rec_acc") AS traceB4 WHERE traceA4.msgno <= traceB4.msgno)) AS trace3)) AS traceB1 WHERE traceA1.msgno <= traceB1.msgno)) AS tracefinal WHERE `msgno` = 0;
```

- **Traduction de SQL à SQL optimisé**

Afin d'éliminer les jointures liées aux requêtes SQL, une version optimisée a été mise en place. Cette version permet d'optimiser les traductions et ainsi minimiser le temps d'analyse dû à ces jointures. En utilisant la version optimisée pour la propriété P2 et la trace énoncée au début de ce chapitre, on obtient le fichier SQL suivant :

```
USE TraceAdapter;
```

```
INSERT INTO trace (`msgno`,`name`)
VALUES
(0, "place_c_order"),
(1, "handle_c_order"),
(2, "place_b_order"),
(3, "eval_b_order"),
(4, "b_reject"),
(5, "decide"),
(6, "c_reject"),
(7, "rec_decl");

SELECT SQL_NO_CACHE msgno FROM (SELECT msgno FROM trace AS trace0
WHERE msgno NOT IN (select msgno FROM trace WHERE msgno <= (SELECT
MAX(msgno) FROM (SELECT msgno FROM trace AS trace2 WHERE msgno NOT IN
(select msgno FROM trace WHERE (msgno IN (SELECT msgno FROM trace AS trace5
WHERE `name` != "place_c_order") OR msgno IN (select msgno FROM trace WHERE
msgno <= (SELECT MAX(msgno) FROM (SELECT msgno FROM trace AS trace5
WHERE `name` = "rec_acc") AS trace4))))) AS trace1))) AS tracefinal WHERE `msgno`
= 0;
```

5.3.5 Transduction de LTL-FO+ à XQuery

Cette transduction est une implémentation d'un résultat démontré dans (Hallé et Villemaire, 2008). La validation des traces est reformulée comme un cas particulier du traitement des requêtes XML.

Tout comme pour SQL, une fonction récursive w_ρ de traduction a été développée. Cette fonction prend en entrée une propriété LTL-FO+ et la traduit en une expression Xquery équivalente. Cette fonction comporte un paramètre ρ (un pointeur vers la racine de premier événement de la trace actuelle). Si on commence la traduction, ρ doit pointer le

premier événement du document trace. L'expression Xpath */trace/message[1]* peut être utilisée pour désigner le premier événement. Il suffit, par la suite, de définir une fonction w_ρ pour chaque opérateur LTL.

La traduction des expressions d'un schéma XML est directe: pour p un paramètre dans un événement et $v \in V$ une valeur à la fin du chemin, on a :

$$w_\rho\ (p = v) \equiv \rho/p = v$$

Le chemin p est relatif à l'événement actuel de la trace ; c'est pourquoi p doit être joint à ρ. XQuery supporte tous les connecteurs logiques (¬, ∧ et ∨). On traduit ces connecteurs de la manière suivante :

$$w_\rho\ (\neg\varphi\ ) \equiv \textbf{not}(w_\rho\ (\varphi\ ))$$

$$w_\rho\ (\varphi \wedge \psi) \equiv (\ w_\rho\ (\varphi)\ \textbf{and}\ w_\rho\ \psi))$$

$$w_\rho\ (\varphi \vee \psi) \equiv (\ w_{\$x}\ (\varphi)\ \textbf{or}\ w_\rho\ (\psi))$$

$$w_\rho\ (\varphi \rightarrow \psi) \equiv (\textbf{not}\ (w_\rho\ (\varphi))\ \textbf{or}\ w_\rho\ (\psi))$$

Selon la sémantique de LTL, une propriété de la forme G φ est vraie à partir de l'événement courant, si et seulement si, tous les événements suivants satisfont φ. Puisque ρ est un pointeur vers l'événement actuel, alors l'expression XQuery « ρ/following-sibling::* » désigne tous les événements qui suivent ρ. Pour obtenir les événements désirés, il faut réaliser l'union entre ces deux ensembles : ρ union ρ/following-sibling::*. Alors pour que G φ soit vraie, alors il faut que tout élément dans l'ensemble ρ union ρ/following-sibling::* doit satisfait φ.

La traduction de la propriété G φ est donc la suivante :

$$w_\rho\ (\mathbf{G}\ \varphi) \equiv \textbf{\textit{every \$x in (}}\rho\ \textit{union}\ \rho\textit{/following-sibling::*)}\ \textbf{\textit{satisfies}}\ w_{\$x}\ (\varphi\ )$$

Dans la traduction, le variable $x est introduite pour désigner chaque événement dans l'ensemble des événements.

5.4 Résultats et discussion

Muni de cet environnement, nous pouvons maintenant comparer les performances de l'ensemble des outils cités précédemment. Les résultats comparatifs ont été obtenus sur un système d'exploitation Ubuntu 11.10 avec un processeur Intel Pentium IV cadencé à 3100 MHz et d'une mémoire totale de 4 GB de RAM. Les calculs durant plus d'une journée où exigeant plus qu'un Go de mémoire ont été interrompus. Les temps d'exécutions sont donnés en secondes et la mémoire utilisée est exprimée en MB.

Les deux Figures 5.8 et 5.9 présentent respectivement le temps d'exécution pour les propriétés P1 et P2. Au vu de ces résultats, nous pouvons tirer les conclusions suivantes. Saxon est l'outil le plus lent et les meilleures performances sont obtenues en utilisant Monpoly. Il est très clair que dans ces deux figures que Saxon a pris plus de temps que les autres outils pour vérifier la propriété.

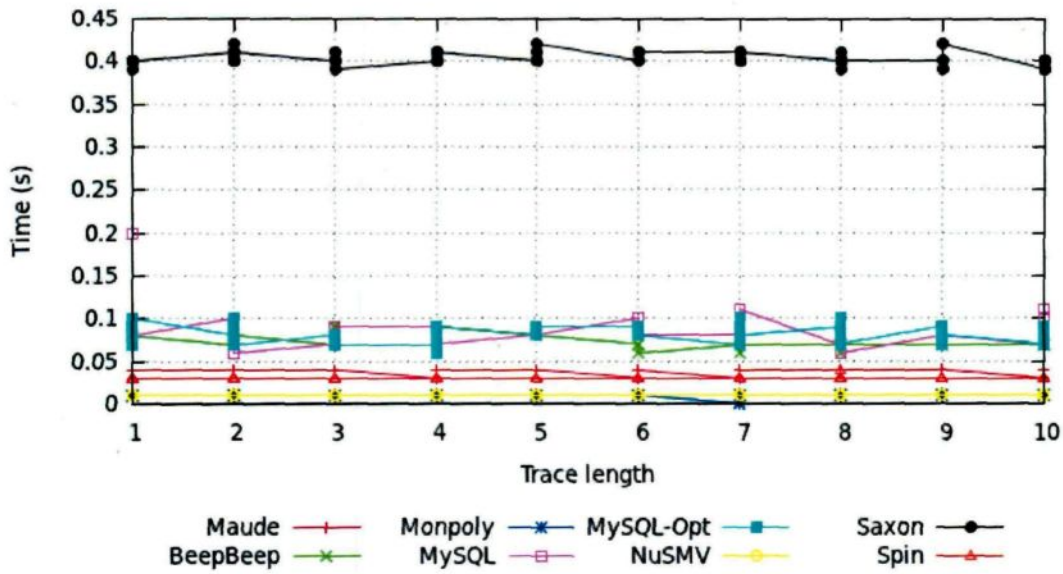


Figure 5.8: Temps d'exécution pour la vérification de la propriété P 1

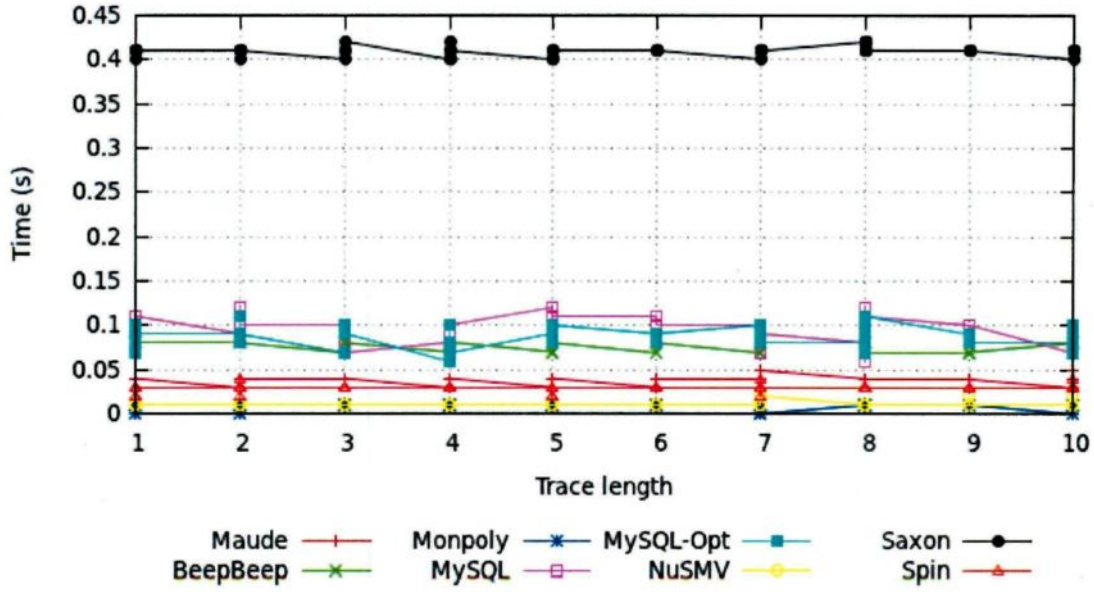


Figure 5.9: Temps d'exécution pour la vérification de la propriété P 2

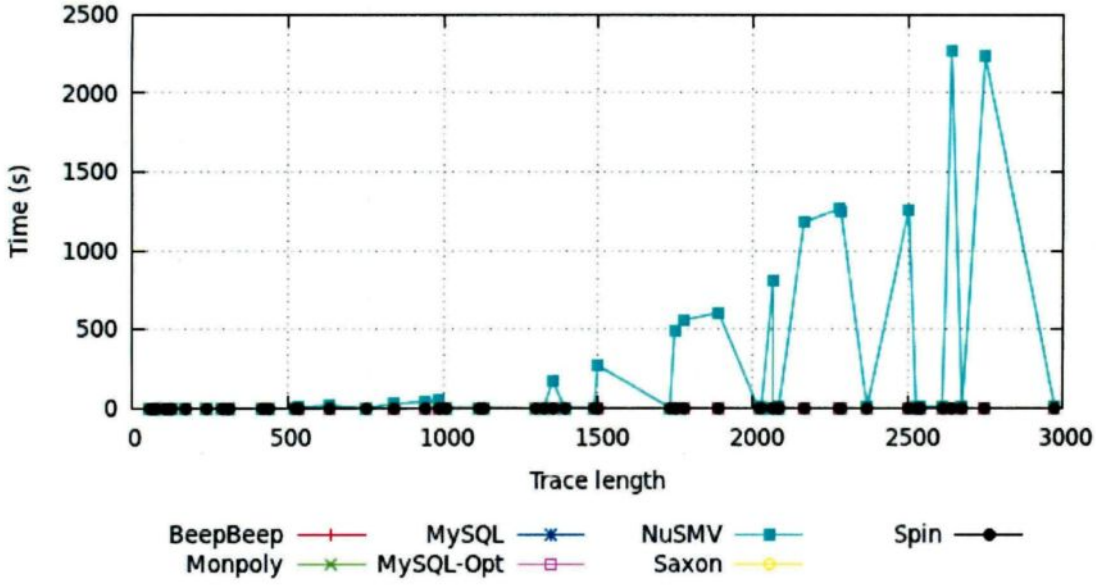


Figure 5.10: Temps d'exécution pour la vérification de la propriété P3

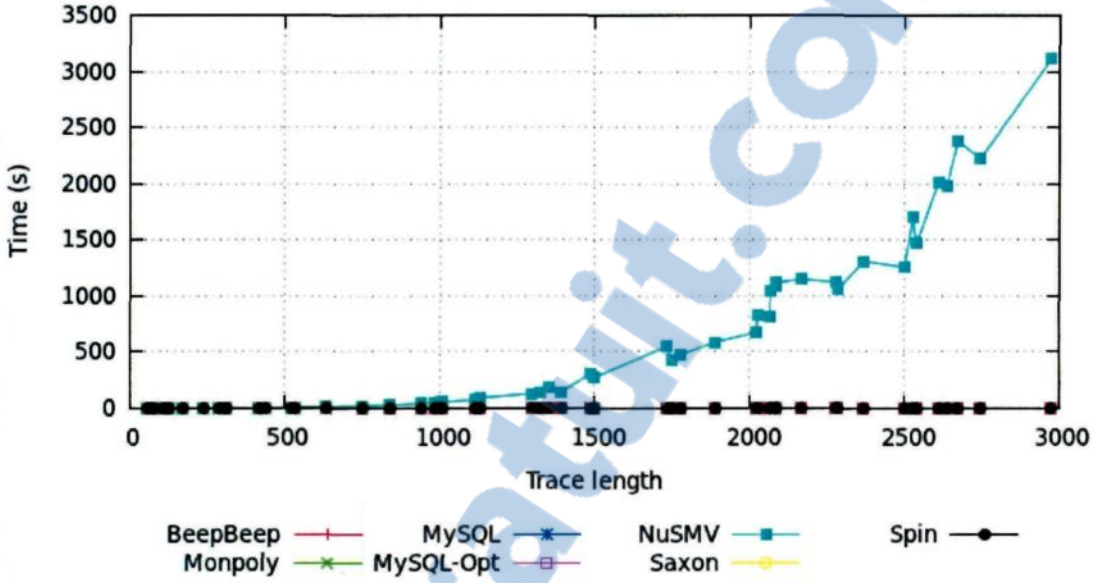


Figure 5.11: Temps d'exécution pour la vérification de la propriété P4

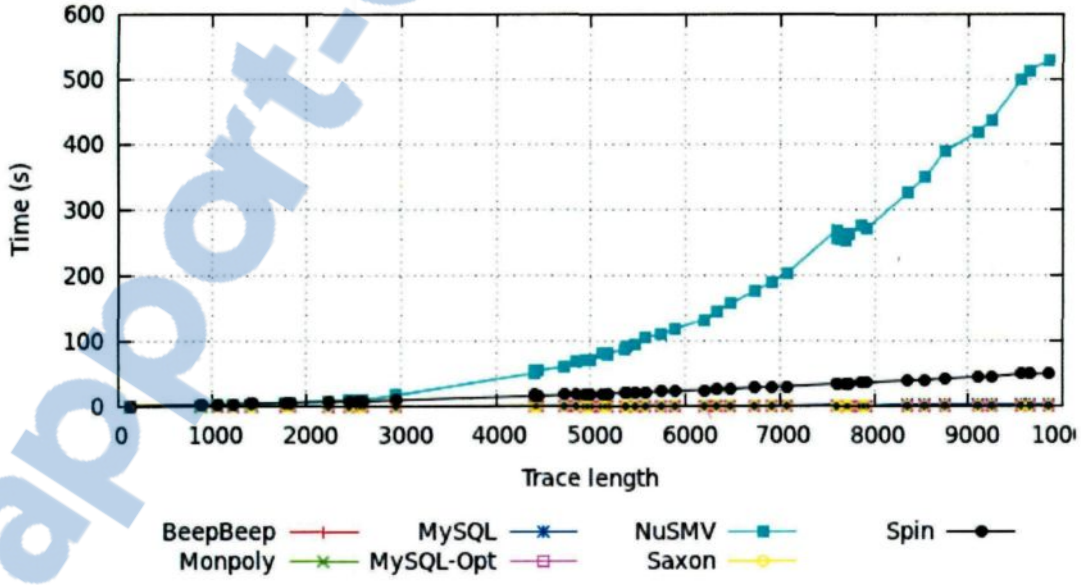


Figure 5.12: Temps d'exécution pour la vérification de la propriété P8

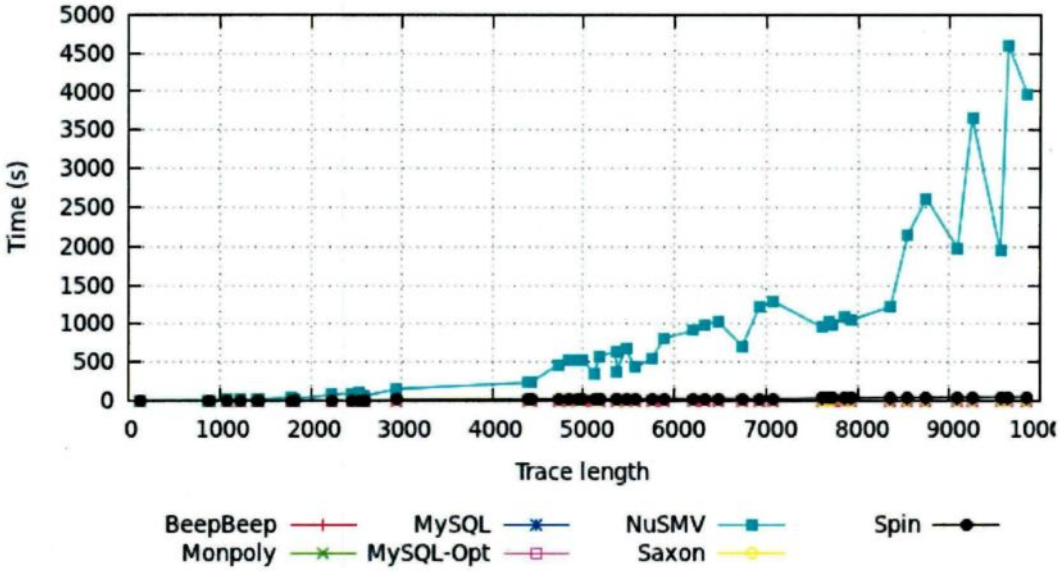


Figure 5.13: Temps d'exécution pour la vérification de la propriété P9

Comme nous pouvons le constater à travers ces résultats, l'outil NuSMV est l'outil le plus lent dans tout l'ensemble des outils étudiés. Il possède un temps d'exécution beaucoup plus élevé que les autres outils.

L'ampleur de nos graphiques ne permet pas une discrimination évidente des outils. Cependant, un examen plus approfondi des données révèle que, globalement, Monpoly est l'outil le plus performant. Il semble donner d'assez bons résultats même sur de grandes traces. De plus, BeepBeep et Saxon ont un temps de fonctionnement comparable. Le model checker NuSMV prend aussi un temps important pour la vérification. Par exemple, pour vérifier la propriété P3 sur une trace de 7855 événements, NuSMV prend un temps de 1131.25s. Nous pouvons dire que le temps de vérification d'une propriété augmente en fonction de nombre d'événements dans chaque trace.

On observe dans les Figures 5.10 à 5.13, l'absence de l'outil Maude. Si on compare ce dernier avec l'outil NuSMV, on conclut que c'est le plus long dans la vérification des propriétés sur de très grandes traces. Il a pris plus de 4 heures pour vérifier une propriété et plante très souvent après avoir épuisé le mémoire réservée dans sa pile.

On observe également que l'utilisation d'un moteur de base des données relationnelles se révèle être une voie prometteuse pour vérifier des propriétés temporelles sur les traces d'événements. Les résultats expérimentaux montrent qu'un moteur de base de données MySQL surpasse les deux models chekers NuSMV et Spin et aussi BeepBeep et Saxon, en terme de la durée de temps d'exécution et de mémoire consommée. On conclut que les systèmes de gestion de base de données sont des systèmes mûrs et performants.

Une étude approfondie des résultats obtenus montre une grande amélioration dans la mémoire consommée et dans le temps de vérification d'une propriété sur des grandes traces entre MySQL et MySQL optimisé a permis de minimiser le temps d'exécution dû à l'utilisation des jointures. On conclut que les systèmes de gestion de base de données sont des systèmes mûrs et performants.

5.5 Démarches pour obtenir les outils

SEQ.OPEN et SPIN ont été obtenus en vertu d'une licence universitaire. Le code pour Maude, donné dans (Rosu et al., 2005), est supposé être public. Maude est distribuée sous la licence GPL, comme c'est le cas pour BeepBeep, Monpoly et MySQL. NuSMV est sous la licence LGPL. L'outil Logscope n'a pu être obtenu en raison de contraintes de licence de la NASA. Pour Monid et RuleR, nos multiples tentatives de contact avec les auteurs sont restées sans réponse.

CHAPITRE 6

CONCLUSION

Ce projet de recherche apporte des solutions à la problématique énoncée en introduction qui porte sur l'analyse des traces d'événements. L'analyse des traces d'événements permet d'identifier des erreurs dans l'exécution du système ou la violation de certaines politiques. Le principe de cette analyse consiste à vérifier si un système respecte un contrat donné.

Plusieurs outils ont été développés pour résoudre ce problème. Ces outils sont mal documentés en ce qui a trait à leur performance. De plus, la plupart des solutions existantes ont été développées pour une utilisation particulière. Aussi, dans la plupart des scénarios, le temps et la mémoire consommée varient beaucoup d'un scénario à l'autre sans qu'il existe un consensus à savoir quelle méthode fonctionne le mieux.

Les résultats expérimentaux ont montré que le système de gestion de base de données MySQL pourra être utilisé comme un analyseur de trace. En effet, on a enregistré d'une manière efficace la trace d'événements dans une base de données. De plus, on a développé un traducteur qui permet de traduire une propriété LTL en une requête SQL.

Ce mémoire a donné naissance à notre benchmark, appelé BabelTrace. Ce dernier supporte sept outils d'analyse des traces d'événements. Il est en cours d'adaptation pour une utilisation en ligne via un portail web. Il vise à faciliter l'accès d'un grand ensemble d'outils de validation de traces et la conversion d'un problème dont les formats d'entrée sont différents.

Ce projet de recherche a fait l'objet d'une publication scientifique importante dans le cadre de la conférence internationale *Runtime Verification RV 2012* (Mrad et al., 2012) : "A Collection of Transducers For Trace Validation",3rd International Conference on Runtime Verification RV 2012, Istanbul, Turquie 25 – 28 sept 2012. Le taux d'acceptation était de 40%. Un second article sera soumis à la conférence EDOC 2013.

La liste des formats, des spécifications et des outils est en pleine croissance. Nous ajouterons à cette liste SEQ.OPEN, ProM, Orchids et d'autres outils pour des travaux futurs. De plus, le succès de l'approche de base de données par rapport à d'autres outils ouvre la voie à des améliorations sur la fonction de traduction et pour une étude plus approfondie en utilisant des langages de requêtes temporelles telle que TSQL2 (Steiner, 1998).

BIBLIOGRAPHIE

(Amazon e-commerce service, 2007) Amazon e-commerce service, http://en.wikipedia.org/wiki/Amazon_Product_Advertising_API (2007)

(Barringer et al., 2004) H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In Proc. of the 5th Int. Conf. on Verification, Model Checking and Abstract Interpretation, vol. 2937 of LNCS, pp. 44-57. Springer, 2004.

(Barringer et al., 2010) H. Barringer, D. E. Rydeheard, and K. Havelund : Rule systems for run-time monitoring: From Eagle to RuleR. J. Logic Comput., 20(3), pp. 675-706, 2010.

(Basin et al., 2012) D. Basin, F. Klaedtke, M. Harvan, E. Zalinescu. MONPOLY: Monitoring Usage-control Policies. In the Proceedings of the 2nd International Conference on Runtime Verification (RV'11). Volume 7186 of Lecture Notes in Computer Science, pages 360--364, Springer-Verlag, 2012.

(Chen et al., 2006) Chen, F., d'Amorim, M., & Roşu, G. (2006). Checking and correcting behaviors of java programs at runtime with java-mop. Electronic Notes in Theoretical Computer Science, 144(4), 3-20.

(Cimatti et al., 2002) Cimatti, Alessandro; Clarke, Edmund M.; Giunchiglia, Enrico; Giunchiglia, Fausto; Pistore, Marco; Rover, Marco; Sebastiani, Roberto; and Tacchella, Armando, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking" (2002). Computer Science Department. Paper 430.

(Clark et DeRose, 1999) J. Clark et S.DeRose : W3C : XML Path Language (XPath) Version 1.0: http://www.w3.org/TR/xpath/

(Clarke et al., 2000) E. Clarke, O. Grumberg et D. Peled. « Model Checking ». MIT Press (2000).

(CPAchecker, 2007) CPAchecker: http://cpachecker.sosy-lab.org/ , 2007.

(CPN, 2010) CPN Group. http://cpntools.org/ , 2010.

(EPC-Global, 2007) EPC-Global : Low Level Reader Protocol (LLRP), Version 1.0.1. Ratified Standard with Approved Fixed Errata, August 13, 2007.

(Günther , 2009) C. W. Günther. Extensible event stream standard definition 1.0. Technical report, 2009. http://www.xes-standard.org/. 83

(Hallé, 2008) S.Hallé http://www.beepbeep.sourceforge.net/ (2008).

(Hallé et Villemaire, 2008) S.Hallé et R. Villemaire : XML methods for validation of temporal properties on message traces with data. Lecture Notes in Computer Science, 5331, p. 337-353, 2008.

(Hallé et al., 2010) Hallé, S., Bultan, T., Hughes, G., Alkhalaf, M., & Villemaire, R. (2010). Runtime verification of web service interface contracts. Computer, 43(3), 59-66.

(Hallé et Villemaire, 2010) S.Hallé and R. Villemaire : XML Query Evaluation in Validation and Monitoring of Web Service Interface Contracts, Advanced Applications and Structures in XML Processing: Label Streams, Semantics Utilization and Data Query Technologies, pages 406-424, 2010.

(Hallé et Villemaire, 2012) Hallé, S., et Villemaire, R. (2012). Runtime enforcement of web service message contracts with data. Services Computing, IEEE Transactions on, 5(2), 192-206.

(Holzmann, 1997) G.J. Holzmann. 1997. « The Model Checker Spin ». IEEE Trans. on Software Engineering, vol. 23, no. 5 (mai), pages: 279-295.

(Holzmann, 2000) G.J. Holzmann : « Using Spin ». Plan 9 Programmer Manual Documents, pages: 353-382, Vita Nuova Holdings Ltd, York England. 2nd edition.

(Marconi et Pistore, 2009) A.Marconi et M.Pistore : Synthesis and Composition of Web Services. Formal Methods for Web Services. Lecture Notes in Computer Science Volume 5569, 2009, pages 89-157.

(Meseguer, 1992) J. Meseguer. Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science, 96(1) :73–155, 1992.

(Mrad et al., 2012) A. Mrad, S. Hallé, E. Beaudet. (2012). A Collection of Transducers For Trace Validation. Proc. 3rd International Conference on Runtime Verification, Istanbul, Turkey, September 2012. Springer: Lecture Notes in Computer Science 7687, 126-130.

(Naldurg et al, 2004) P. Naldurg, K. Sen, et P. Thati. A temporal logic based framework for intrusion detection. In Proc. of the 24th IFIP Int. Conf. on Formal Techniques for Networked and Distributed Systems, vol. 3235 of LNCS, pp. 359-376. Springer, 2004.

(Olivain et Goubault-Larrecq, 2005) J. Olivain and J. Goubault-Larrecq. The Orchids intrusion detection tool. In Proc. of the 17th Int. Conf. on Computer Aided Verification, vol. 3576 of LNCS, pp. 286-290. Springer, 2005. 84

(Palshikar, 2004) Palshikar, Girish Keshav. "An introduction to model checking." Embedded Systems Programming 17.3 (2004): 22-29.

(Pneuli, 1977) Pnueli, A. 1977. The temporal logic of programs. In Symposium on the Foundations of Computer Science (FOCS). IEEEComputerSocietyPress, Providence, RhodeIsland, 46–57.

(Rosu et Havelund, 2005) G. Rosu, K. Havelund: Rewriting-based techniques for runtime verification. Automated Software Eng. 12(2), 151–197 (2005)

(Runtime verification, 2012) Runtime Verification. (2012). http://en.wikipedia.org/wiki/Runtime_verification

(Steiner, 1998) A. Steiner. A Generalisation Approach to Temporal Data Models and their Implementations. PhD thesis, ETH Zurich, 1998.

(Uppaal CoVer, 2011) Uppaal CoVer. http://uppaal.org/, February 2011.

(Vardi et Wolper, 1986) M.Y. Vardi et P. Wolper. Automata theoretic techniques for modal logics of programs. Journal of Computer and System Sciences, 32(2) :183–221, April 1986.

(Van der Aalst, 2011) Van der Aalst, W. M. (2011). Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer.

ANNEXE A : LES MODULES FONCTIONNELS DE MAUDE

Cette annexe contient les modules fonctionnels de Maude pour la validation des traces **(G. Rosu et al., 2005)**.

```
fmod ATOM is
    sort Atom .
endfm

fmod TRACE is
    protecting ATOM .
    sorts Event Event* Trace .
    subsorts Atom < Event < Event* < Trace .
    op empty : -> Event .
    op __ : Event Event -> Event [assoc comm id: empty prec 23] .
    var A : Atom .
    eq A A = A .
    op _* : Event -> Event* .
    op _,_ : Event Trace -> Trace [prec 25] .
endfm

fmod LOGICS-BASIC is
    protecting TRACE .
    sort Formula .
    subsort Atom < Formula .
```

```
    ops true false : -> Formula .
    op [_] : Formula -> Bool .
    eq [true] = true .
    eq [false] = false .
    var A : Atom .
    var T : Trace .
    var E : Event .
    var E* : Event* .
    op _{_} : Formula Event* -> Formula [prec 10] .
    eq true{E*} = true .
    eq false{E*} = false .
    eq A{A E} = true .
    eq A{E} = false [owise] .
    eq A{E *} = A{E} .
    op _|=_ : Trace Formula -> Bool [prec 30] .
    eq T |= true = true .
    eq T |= false = false .
    eq E |= A = [A{E}] .
    eq E,T |= A = E |= A .
endfm

fmod PROP-CALC is
    extending LOGICS-BASIC .
    *** Constructors ***
    op _/\_ : Formula Formula -> Formula [assoc comm prec 15] .
    op _++_ : Formula Formula -> Formula [assoc comm prec 17] .
    vars X Y Z : Formula .
    eq true /\ X = X .
```

```
eq false /\ X = false .
eq X /\ X = X .
eq false ++ X = X .
eq X ++ X = false .
eq X /\ (Y ++ Z) = X /\ Y ++ X /\ Z .
*** Derived operators ***
op _\/_ : Formula Formula -> Formula [assoc prec 19] .
op !_ : Formula -> Formula [prec 13] .
op _->_ : Formula Formula -> Formula [prec 21] .
op _<->_ : Formula Formula -> Formula [prec 23] .
eq X \/ Y = X /\ Y ++ X ++ Y .
eq ! X = true ++ X .
eq X -> Y = true ++ X ++ X /\ Y .
eq X <-> Y = true ++ X ++ Y .
*** Finite trace semantics
var T : Trace .
var E* : Event* .
eq T |= X /\ Y = T |= X and T |= Y .
eq T |= X ++ Y = T |= X xor T |= Y .
eq (X /\ Y){E*} = X{E*} /\ Y{E*} .
eq (X ++ Y){E*} = X{E*} ++ Y{E*} .
eq [X /\ Y] = [X] and [Y] .
eq [X ++ Y] = [X] xor [Y] .
```

endfm

fmod LTL is

```
extending PROP-CALC .
*** syntax
```

```
    op []_ : Formula -> Formula [prec 11] .
    op <>_ : Formula -> Formula [prec 11].
    op _U_ : Formula Formula -> Formula [prec 14] .
    op o_ : Formula -> Formula [prec 11] .
    *** semantics
    vars X Y : Formula .
    var E : Event .
    var T : Trace .
    eq E |= o X = E |= X .
    eq E,T |= o X = T |= X .
    eq E |= <> X = E |= X .
    eq E,T |= <> X = E,T |= X or T |= <> X .
    eq E |= [] X = E |= X .
    eq E,T |= [] X = E,T |= X and T |= [] X .
    eq E |= X U Y = E |= Y .
    eq E,T |= X U Y = E,T |= Y or E,T |= X and T |= X U Y .
endfm
```