MASTER EN SCIENCES INFORMATIQUES

Approche déclarative pour la modélisation et l'exécution de processus métiers les workflow Saturn

Demeyer, Romain

Award date:
2009

Awarding institution:
Universite de Namur

Approche déclarative pour la modélisation et l'exécution de processus métiers : les *workflow Saturn*

Romain Demeyer

2008-2009

Mémoire présenté en vue de l'obtention du grade de Maître en informatique

Abstract. This work is within the framework of the modeling and management of business processes, a particularly important field in modern organizations. A business process consists of a collection of structured and related tasks that are performed in a certain order in order to produce a specific service or product. Business processes modeling is usually done by specifying a workflow using a workflow modeling language (like BPEL or YAWL) and management systems, specific to each language, are developed to monitor automatically and dynamically the execution of business processes. Languages currently used are of imperative nature, they describe in a step by step way how the business process has to be performed. This approach shows its limits, especially when it involves flexible business processes. The modeling of such processes using the current formalisms is tedious and complex, while their imperative nature forces designers to over-specify. To overcome these problems, we propose a paradigm shift by providing a more declarative approach. A declarative workflow defines constraints that must be respected through the execution of the modeled business processes and every execution that does not violate these constraints is allowed. We propose the declarative wokflow modeling language *Saturn*, based on temporal logic, and we'll see how we can monitor the execution of business processes modeled using this language.

keywords : business process, workflow management, dynamic workflow, declarative model specification, flexibility, temporal logic, constraint-based model

Résumé. Ce travail s'inscrit dans le cadre de la modélisation et de la gestion électronique des processus métiers, un domaine particulièrement important dans les organisations modernes. Un processus métier est constitué d'un ensemble de tâches qui sont réalisées dans un certain ordre et selon certaines dépendances dans le but de produire un service ou un produit spécifique. La modélisation de processus métiers se fait généralement en spécifiant un *workflow* exprimé à l'aide d'un langage de spécification de *workflow* (comme BPEL ou YAWL) et des systèmes associés à chaque langage sont développés pour contrôler automatiquement et dynamiquement l'exécution de processus métiers selon leur modélisation. Les langages utilisés actuellement sont de nature impérative, les *workflow* définissent ainsi de manière stricte comment le processus métier doit s'exécuter. Cette approche montre actuellement des limites, notamment quand il s'agit de spécifier des processus métiers flexibles. La modélisation de tels processus à l'aide des formalismes actuels est en effet fastidieuse et complexe, tandis que leur nature impérative force les concepteurs à sur-spécifier les processus métiers. Pour pallier à ces problèmes, nous proposons dans ce mémoire un changement de paradigme en introduisant une méthode dite déclarative. Un *workflow* déclaratif définit des contraintes qui doivent être respectées par l'exécution du processus métier modélisé et tous les comportements ne violant pas ces contraintes sont autorisés. Il précise ainsi ce qui doit être effectué sans décrire comment cela doit être effectué. Nous proposons le langage de spécification de *workflow* déclaratif *Saturn*, basé sur une forme de logique temporelle, et nous allons voir comment on peut contrôler l'exécution des processus métiers modélisés à l'aide de ce langage.

Avant-propos

Ce mémoire constitue le fruit d'un travail entamé lors d'un stage dans l'entreprise Mission Critical [1]. Celle-ci offre des solutions informatiques de haute qualité à ses clients et a la particularité d'utiliser un langage de programmation déclaratif (*Mercury*, [2, 3]), nécessitant une grande rigueur mathématique mais permettant de réaliser des programmes performants, flexibles et d'une grande robustesse. L'article [4] de Michel Vanden Bossche présente les techniques employées par *Mission Critical*. Les membres de cette entreprise utilisent un formalisme (*YAWL*, [5]) basé sur les réseaux de Petri pour modéliser et exécuter les processus métiers mais ont pu constater les limites de cette approche impérative. Dans ce contexte, ils ont supporté activement les recherches et les développements qui ont permis la rédaction de ce mémoire.

Remerciements

Je voudrais remercier le promoteur de ce mémoire, Wim Vanhoof, pour son suivi et sa disponibilité exceptionnelle ainsi que pour les précieux conseils qu'il m'a prodigués durant la rédaction.

Je tiens à exprimer ma gratitude envers Michel Vanden Bossche qui m'a permis de travailler au sein d'une entreprise si compétente et visionnaire. Je remercie également tout le personnel de Mission Critical pour son accueil chaleureux. J'adresse une pensée particulièrement émue à la regrettée Vinciane Cappelle qui a veillé à mon bien-être durant mon séjour dans l'entreprise.

Je tiens à remercier tout spécialement Maxime Van Assche, Ludovic Langevine et Raphaël Collet qui m'ont fait bénéficier quotidiennement de leurs connaissances scientifiques et de leur expérience, tout en me guidant de manière avisée lors de ma recherche.

Je remercie enfin tout mon entourage, famille, amis et professeurs, pour avoir contribué à la rédaction de ce travail par leur enseignement, leur aide et leur soutien.

Table des matières

1 Introduction **5**

2 La Logique Temporelle Linéaire - LTL **10**
2.1 Définitions et illustrations 11
2.2 Utilité de LTL en informatique 19
2.3 Conversion d'expressions LTL en automates de Büchi 21
2.4 LTL et les séquences finies 37

3 Les *workflow* *Saturn* **43**
3.1 Présentation du langage déclaratif *Saturn* 44
3.2 Interprétation formelle des *workflow* *Saturn* 52
3.2.1 *Workflow* *Saturn* et PM constitué de tâches synchrones . . 52
3.2.2 *Workflow* *Saturn* et PM constitué de tâches asynchrones . 55
3.2.3 *Workflow* *Saturn* et PM sensible à un environnement . . . 60
3.2.4 *Workflow* *Saturn* et PM hiérarchique 63

4 Moteur de *workflow* *Saturn* **70**
4.1 Structure 71
4.2 Performances 78
4.2.1 Performances à l'initialisation 78
4.2.2 Performances à l'exécution 85

5 Conclusion et perspectives **89**

A Annexe **96**
A.1 Rappel des notions et notations principales 97
A.2 Témoignage d'expérience du langage *Mercury* 106

Chapitre 1

Introduction

Ce mémoire s'inscrit dans le domaine de la gestion électronique des processus métiers, et plus particulièrement dans le domaine des *workflow*[1]. Notons que ces différents termes ont tendance à être utilisés dans le monde de l'industrie et dans le monde académique pour désigner différents concepts, qui évoluent d'ailleurs au fil des années [6, 7]. Il est donc utile de les redéfinir par rapport au contexte de ce mémoire.

Un processus métier[2] désigne un ensemble d'activités (ou tâches) qui s'enchaînent de manière chronologique pour atteindre un objectif qui, dans le contexte d'une organisation de travail, est généralement la fourniture d'un service ou la production d'un bien. Une exécution d'un processus métier correspond à une réalisation particulière de ce processus et est composée d'une séquence de tâches (les tâches effectuées).

Un *workflow* correspond à la modélisation d'un processus métier. Il est généralement constitué d'une collection de tâches (les unités atomiques du *workflow*) organisées dans le but d'accomplir le processus métier auquel il fait référence. Il délimite ainsi un ensemble d'exécutions possibles pour la réalisation du processus métier.

Notons que dans le contexte des *workflow*, une tâche est effectuée par une entité autonome, qui peut aussi bien être constituée d'un ou plusieurs systèmes informatiques, d'une ou plusieurs personnes ou d'une combinaison de ceux-ci. Une tâche peut notamment être effectuée par une personne qui interagit avec un système informatique en y entrant des commandes ou, simplement, en indiquant l'état de progression de la tâche (exemple : la tâche est terminée). Les entités autonomes impliquées dans un *workflow* seront appelées les utilisateurs du *workflow*.

Le formalisme utilisé pour décrire un processus métier, ou autrement dit pour exprimer (ou spécifier) un *workflow*, est appelé un langage de spécification de *workflow*. Un des intérêts de la modélisation d'un *workflow* est que le langage de spécification utilisé puisse être compris et utilisé par les analystes non-informaticiens.

[1] Nous ferons usage de ce terme anglais reconnu et majoritairement utilisé dans les communautés scientifiques et informatiques francophones

[2] ou business process

On peut distinguer deux utilisations d'un *workflow*. La première utilisation est représentative. Le *workflow* permet en effet d'analyser le processus métier (et notamment son efficacité), de l'abstraire, de le comprendre, de le visualiser et de raisonner sur celui-ci (notamment dans le but de l'optimiser). La seconde utilisation est exécutive : il est directement utilisé par un système, appelé moteur de *workflow*, qui contrôle le déroulement du processus métier que le *workflow* modélise, en s'assurant que les utilisateurs effectuent les tâches dans un ordre qui est conforme à ce *workflow*. Dans ce contexte, il est important de distinguer le *workflow* et une instance de *workflow*. Une instance de *workflow* représente une seule exécution du processus modélisé par ce *workflow*. Ainsi, une instance d'un *workflow* est caractérisée par un état qui reflète les tâches préalablement effectuées. Suivant l'état de l'instance, le système peut indiquer aux utilisateurs quelles sont les tâches qui peuvent être effectuées et fait évoluer cet état lorsque l'une de ces tâches est effectuée. On dit que le système exécute l'instance.

Pour illustrer ces différents concepts, imaginons un site de vente de livres en ligne et le processus métier "vente d'un livre" qui se déroule comme suit : une commande provenant d'un client est reçue (tâche *receive_order*). La commande peut être soit acceptée (tâche *accept*), soit refusée (tâche *decline*). Si elle est refusée, le processus se termine. Si elle est acceptée, le livre et la facture sont envoyés[1] (tâches *send_book* et *send_bill*). Le paiement relatif au livre est alors perçu (tâche *receive_paym.*) et le processus de vente se termine.

Le *workflow* est donc constitué d'un ensemble de tâches $Tasks = \{receive\_order, accept, decline, send\_book, send\_bill, receive\_paym.\}$.

Le langage de spécification de *workflow* que nous allons utiliser pour modéliser le *workflow* est le réseau de Petri [8]. Un réseau de Petri consiste en un ensemble *Places* de places (représentées par des cercles) et un ensemble *Transitions* de transitions (représentées par des rectangles). Ces dernières représenteront les tâches du *workflow*, soit $Transitions = Tasks$. Les transitions et les places sont connectées par des arcs orientés. Les places peuvent contenir des jetons. Le réseau de Petri possède une place particulière qui n'a pas d'arc en entrée, appelée place initiale et une place particulière qui n'a pas d'arc en sortie, appelée place finale. Une transition possède au moins une place en entrée et au moins une place en sortie. Une transition est activée s'il y a un jeton dans chacune des places en entrée. Si la transition est activée, elle peut être effectuée en consommant un jeton dans chacune des places en entrée et en plaçant un jeton dans chacune des places en sortie.

Un *workflow* modélisant le processus métier "vente d'un livre" et exprimé sous forme de réseau de Petri est présenté à la figure 1.1.

Chaque fois qu'un client veut acheter un livre, une instance du *workflow* est créée en associant au *workflow* un état particulier. L'état d'un réseau de Petri peut être représenté par une fonction $state : Places \to \mathbb{N}$ qui indique pour chaque place le nombre de jetons qu'elle contient. Au début de l'exécution, seule la place initiale contient un jeton. Le moteur de *workflow* peut, grâce à l'état du réseau de Petri, indiquer quelles sont les tâches qui peuvent être effectuées (représentées par les transitions activées) et lorsqu'une tâche est effectuée, il peut faire évoluer l'état en conséquence (en effectuant la transition correspondante).

[1] mais pas nécessairement en même temps, le livre et la facture pouvant être envoyés au client par des canaux différents

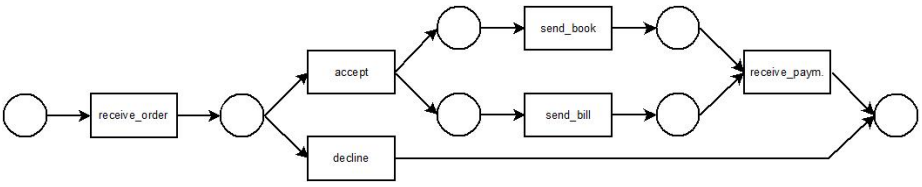


FIG. 1.1 – Un réseau de Petri modélisant le processus métier "vente d'un livre"

Le processus de vente, et donc l'instance, se termine dés qu'il y a un jeton dans la place finale.

Actuellement, les langages de spécification de *workflow* utilisés sont souvent de nature impérative, ce qui implique que les *workflow* modélisés en utilisant ces langages décrivent les processus métiers en définissant comment les tâches doivent s'enchaîner. En effet, les formalismes utilisés, à l'image des réseaux de Petri de la figure 1.1, forcent à définir un "algorithme" pour l'exécution du processus. Le moteur de *workflow* utilise alors cet algorithme comme une "recette" pour déterminer la séquence des tâches à exécuter. Les langages BPEL [9] (Business Process Execution Language) et YAWL [5] (Yet Another Workflow Language) s'inscrivent dans ce cadre et sont employés avec un certain succès dans les organisations. Néanmoins, cette façon de procéder n'est pas toujours adaptée à des processus métiers flexibles, où les tâches peuvent s'agencer de nombreuses façons différentes, ce qui est de plus en plus fréquent dans les architectures orientées service. La réalisation d'un *workflow* pour ce type de processus peut, en raison de la nature impérative du langage utilisé, rapidement s'avérer fastidieuse. Le *workflow* perd en clarté et en lisibilité et s'avérera par la suite peu flexible car les changements seront difficilement intégrables. Ces points ont notamment été relevés dans [10, 11, 12, 13]. Vu la nature impérative des langages de spécification et vu que la modélisation s'avère fastidieuse lorsque le processus métier est flexible, les développeurs et analystes ont tendance à sur-spécifier le *workflow* [10, 11, 12, 13]. Il en résulte que l'exécution de celui-ci est plus rigide que nécessaire, ce qui limite inutilement les utilisateurs du *workflow* dans leur tâche et peut rapidement s'avérer inefficace.

Pour l'illustrer, considérons un processus métier qui est composé de deux tâches A et B. La seule exigence est que si la tâche A est réalisée au moins une fois, la tâche B doit également être réalisée au moins une fois et vice-versa. L'ordre dans lequel elles sont réalisées n'a pas d'importance et le nombre de réalisations de chacune des tâches peut être différent. On constate que modéliser de manière complète et concise un tel processus métier, pourtant très simple, à l'aide d'un formalisme impératif est difficile. La figure 1.2 montre une tentative de modélisation de ce processus à l'aide des réseaux de Petri. Comme on peut le constater, elle semble déjà complexe et redondante. Elle fait intervenir une tâche supplémentaire (C) qui correspond au choix. Pire, c'est n'est qu'une modélisation partielle du processus métier. Par exemple, ce *workflow* ne validerait pas (vu qu'il n'y aurait pas de jeton dans la place finale) une exécution pourtant correcte constituée de la réalisation de la tâche A, suivi de la tâche B puis d'une nouvelle réalisation de la tâche A. Ceci illustre bien que ce type de formalisme a tendance à inciter les concepteurs à sur-spécifier les processus métiers.

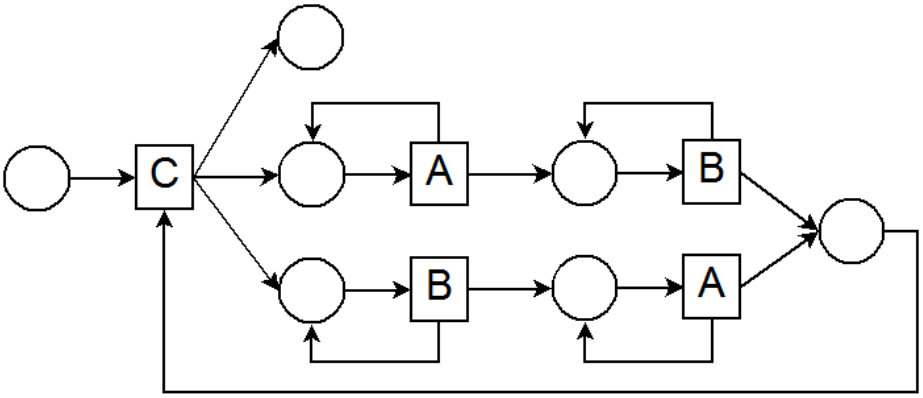


FIG. 1.2 – Un réseau de Petri ne modélisant que partiellement un processus métier simple mais flexible

Pour résoudre ces problèmes, nous allons proposer une approche déclarative pour spécifier les *workflow*. Intuitivement, une méthode déclarative spécifie ce qui doit être fait, le quoi, plutôt que spécifier comment cela doit être fait. Le développement de cette approche introduit donc une rupture fondamentale dans la manière de décrire des processus métiers au travers de *workflow* et est résolument nouvelle. A notre connaissance, les seuls travaux de recherche effectués dans ce domaine ont été réalisés dans le cadre du projet Declare ([10, 11, 12, 13]. Concrètement, un langage de spécification de *workflow* déclaratif consiste, en plus de la liste des tâches susceptibles d'être exécutées au cours du processus métier, en un ensemble de contraintes qui doivent être respectées au cours de l'exécution du *workflow*. Ces contraintes portent sur l'occurence, les relations et l'ordre des différentes tâches. Par exemple, une contrainte peut imposer qu'une tâche X soit effectuée au moins une fois durant le processus, imposer qu'une tache X ne soit jamais effectuée au cours d'un processus durant lequel une tâche Y est effectuée ou encore que lorsqu'une tâche X est effectuée, une tâche Y doit être effectuée par la suite durant le même processus. Nous verrons qu'il est ainsi très simple de représenter de manière complète des processus métiers comme celui que nous décrivons au paragraphe précédent .

De façon très intuitive, on peut voir l'approche déclarative comme une approche duale de l'approche impérative. Pour mettre en évidence l'opposition entre les deux approches, analysons la démarche entreprise pour spécifier un *workflow* pour chacune d'entre elles. Le *workflow* impératif le plus "épuré" décrit généralement le processus "vide" au cours duquel aucune tâche n'est exécutée. Dans le cas d'un réseau de Petri, le *workflow* le plus épuré serait en effet constitué d'une seule place, à la fois initiale et finale, et d'aucune transition. Pour augmenter les possibilités d'exécution, on utilise les différents éléments du langage de spécification impératif. Ainsi, comme on peut le constater pour les réseaux de Petri, plus on développe le *workflow* en y ajoutant des structures de plus en plus complexes constituées de transitions et de places, plus le *workflow* offre des possibilités d'exécution. Inversement, le *workflow* déclaratif le plus épuré ne possède pas de contrainte et est seulement constitué de l'ensemble des tâches qui peuvent être exécutées au cours du processus métier. Vu qu'il n'y a

aucune contrainte, ces tâches peuvent être effectuées dans n'importe quel ordre et autant de fois qu'on le désire. Autrement dit, les possibilités d'exécution sont maximales. Plus on développe le *workflow* en y ajoutant des contraintes, plus ces possibilités d'exécutions se réduisent. La figure 1.3 illustre cette opposition entre les approches impérative et déclarative. Il est important de remarquer que ces deux approches sont complémentaires : l'approche la plus appropriée sera déterminée selon la nature du processus métier ou de la partie de processus métier à modéliser.

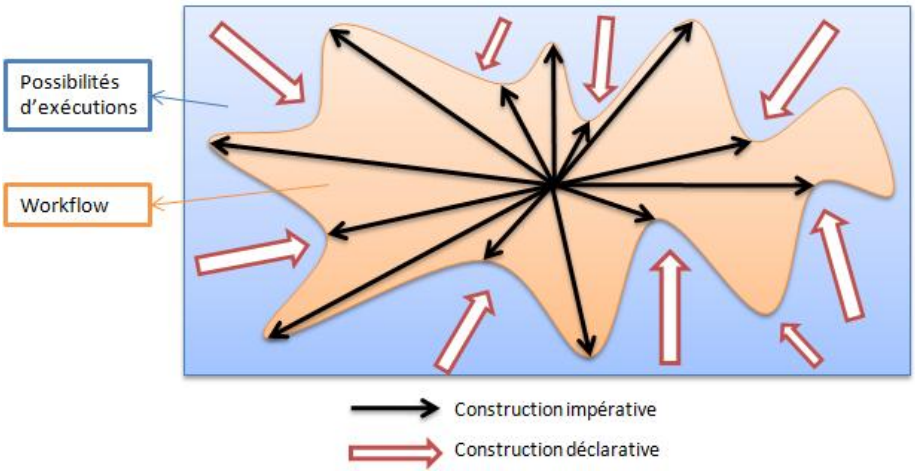


FIG. 1.3 – L'approche impérative construit progressivement le *workflow* en augmentant ses possibilités d'exécutions tandis que l'approche déclarative le construit en réduisant ses possibilités d'exécutions

L'objectif de ce mémoire est de présenter un formalisme qui permet de décrire des processus métiers de manière déclarative. Ce formalisme sera basé sur la logique temporelle linéaire [14, 15], qui sera présentée en détail au cours du chapitre 2. Nous présenterons un langage de spécification de *workflow* déclaratif, le langage *Saturn*, au cours du chapitre 3 et nous verrons comment les *workflow* exprimés dans ce langage, les *workflow Saturn*, sont interprétés. Nous présenterons enfin, au cours du chapitre 4, un moteur de *workflow* qui permet de contrôler dynamiquement l'exécution d'un processus métier conformément au *workflow Saturn* qui le modélise. Le chapitre 5 reviendra en guise de conclusion sur les contributions et les perspectives associées à ce mémoire. Notons que pour faciliter la lecture et la compréhension du texte, les différentes notions et notations abordées sont reprises dans l'annexe A.1.

Chapitre 2

La Logique Temporelle Linéaire - LTL

Ce chapitre est dédié à la logique temporelle linéaire (LTL[1]) [14, 15] qui est, rappelons-le, à la base du langage déclaratif de spécification de *workflow* que nous présenterons au cours du chapitre 3. Au cours de la section 2.1, LTL sera introduite de manière intuitive puis définie formellement, et ce par rapport à des séquences infinies. Nous verrons au cours de la section 2.2 que LTL peut s'avérer très utile dans différents domaines informatiques, que ce soit pour la vérification de modèles ou, comme dans le cadre de ce mémoire, pour la spécification de programmes. Afin de pouvoir être évaluées algorithmiquement, les expressions LTL peuvent être converties en automates de Büchi comme nous le verrons au cours de la section 2.3. Enfin, nous proposerons durant la section 2.4 une utilisation particulière de LTL, en redéfinissant ses opérateurs par rapport à des séquences finies et nous verrons qu'il est alors possible de mettre en place un système qui permet de construire n'importe quelles séquences respectant une expression LTL.

[1] pour *Linear-time Temporal Logic*

2.1 Définitions et illustrations

Une logique temporelle pour étendre la logique propositionnelle

Cette sous-section a pour objectif de donner une idée intuitive de l'intérêt et de l'expressivité de la logique temporelle.

Considérons un système constitué d'un ensemble de propositions P qui sont soit vraies, soit fausses. Une configuration de ce système peut être représentée par un sous-ensemble des propositions $P' \subseteq P$. Les propositions qui font partie de ce sous-ensemble P' sont considérées comme vraies pour cette configuration, et les propositions qui ne font pas partie de ce sous-ensemble P' sont considérées comme fausses. L'ensemble de toutes les configurations possibles est noté 2^P.

Par exemple, si l'on considère l'ensemble de propositions $P = \{p, q\}$, l'ensemble des configurations 2^P est constitué de
- $\{\}$: p et q sont faux
- $\{p\}$: p est vrai et q est faux
- $\{q\}$: p est faux et q est vrai
- $\{p, q\}$ p et q sont vrais

On peut remarquer que la logique propositionnelle permet d'exprimer des propriétés sur une configuration d'un ensemble de propositions.

Exemple 2.1.1 *Soit l'ensemble de propositions $P = \{p, q, r\}$ tel que*
- *p signifie "Il pleut"*
- *q signifie "Il fait beau"*
- *r signifie "J'emporte mon parapluie",*

En effet, si l'on considère le système constitué de l'ensemble de propositions P de l'exemple 2.1.1, la logique propositionnelle permet d'exprimer les propriétés que doit respecter une configuration du système :
- $p \vee q \equiv$ Il pleut ou il fait beau
- $\neg p \wedge q \equiv$ Il ne pleut pas et il fait beau
- $p \Rightarrow r \equiv$ S'il pleut, j'emporte mon parapluie
- ...

Par contre, la logique propositionnelle ne permet pas d'exprimer le fait que ces propriétés peuvent évoluer au cours du temps. Or, il peut être intéressant (voir section 2.2) d'exprimer des relations sur les propriétés de différentes configurations successives du système. Par exemple, on aimerait pouvoir exprimer que

(a) Il va pleuvoir

(b) Il fait toujours beau

(c) Je prends mon parapluie jusqu'à ce qu'il fasse beau

(d) Il pleut infiniment souvent (il finit toujours bien par pleuvoir à un moment donné).

La logique temporelle [14] est un formalisme qui permet d'exprimer l'évolution de propriétés d'un système dans le temps, et ce en décrivant des relations entre les différentes configurations de ce système. Pour ce faire, elle étend la logique propositionnelle classique avec des modalités qui consistent en un certain nombre d'opérateurs temporels , que l'on peut voir comme les adverbes d'une grammaire. La logique temporelle, qui est donc une logique modale, utilise des opérateurs comme $\diamond$ (finalement) , $\circ$ (après) , $\Box$ (toujours) ou $\mathbb{U}$ (jusqu'à).

Ainsi, en considérant à nouveau l'ensemble de propositions P de l'exemple 2.1.1, la logique temporelle permet d'exprimer les faits ci-dessus :

(a) $\diamond p$

(b) $\Box q$

(c) $r \mathbb{U} q$

(d) $\Box(\diamond p)$

Si une formule de logique propositionnelle peut être vérifiée pour une configuration du système, une formule de logique temporelle est vérifiée sur une séquence de configurations du système.

Ainsi, en considérant l'ensemble de propositions $P = \{p, q, r\}$, la formule de logique propositionnelle $p \Rightarrow r$ est vérifiée par les configurations $\{\}$, $\{r\}$, $\{p, r\}$, $\{q\}$, $\{q, r\}$ et $\{p, q, r\}$ alors que les configurations $\{p\}$ et $\{p, q\}$ ne la vérifient pas. Une formule de logique temporelle $\diamond p$, qui spécifie que la proposition p doit être vraie à un moment donné, est vérifiée pour toutes séquences constituées d'au moins une configuration qui contient p.

Il peut être utile de commenter la notion de "temps" propre à la logique temporelle. Le temps est vu comme une séquence de configurations du système, et un instant dans le temps est vu comme une configuration particulière ("à un moment donné") de cette séquence. On considère donc un temps discret plutôt qu'un temps continu.

Différentes logiques temporelles

Il existe plusieurs logiques temporelles, qui diffèrent entre elles par les opérateurs qui les composent et par la sémantique de ces opérateurs. On distingue la logique en temps arborescent CTL (Computation Tree Logic) [14, 16, 17] de la logique en temps linéaire LTL (Linear-Time Logic)[14, 15], qu'on appelle aussi logique temporelle linéaire.

La logique en temps arborescent (CTL) modélise le temps comme une structure en arbre, dans laquelle le futur n'est pas déterminé puisqu'il y a plusieurs chemins possibles à partir d'une configuration donnée. Les expressions CTL sont exprimées à l'aide de quantificateurs en plus des opérateurs temporels, qui seront présentés par la suite, et des opérateurs de logique propositionnelle ($\vee$, $\wedge$, $\neg$). Les quantificateurs décrivent la structure arborescente alors que les opérateurs temporels classiques décrivent les propriétés d'un seul chemin de l'arbre. Ces quantificateurs permettent par exemple d'exprimer que tous les chemins

ou que certains chemins à partir d'une configuration donnée satisfont une propriété temporelle. Il existe plusieurs formes de logique en temps arborescent d'expressivité différente [14].

La logique en temps linéaire (LTL) modélise, comme son nom l'indique, le temps de façon linéaire et permet d'exprimer des propriétés sur des séquences de configurations, ce qui la rend plus naturelle et plus simple à comprendre que CTL. Intuitivement, on peut voir la logique en temps linéaire comme celle qui ne traite que d'un seul chemin d'une structure en arbre. Les expressions LTL sont uniquement exprimées à l'aide des opérateurs temporels et des opérateurs de la logique propositionnelle. Elles font toujours référence au futur (ou au présent). Notons néanmoins l'existence de PLTL (LTL avec passé) qui ajoute des opérateurs traitant du passé mais qui n'ajoute pas d'expressivité par rapport à la logique temporelle classique LTL [18].

Les différentes logiques temporelles ont des expressivités différentes [19, 14]. Notons que contrairement à ce que l'on peut croire, la logique temporelle linéaire n'est pas nécessairement moins expressive que logique en temps arborescent.

Dans la suite, nous allons présenter en détails et utiliser tout au long de ce mémoire la logique temporelle linéaire (LTL). Son expressivité, adéquate au contexte qui nous occupera par la suite, ainsi que sa simplicité sont les raisons de ce choix.

Introduction aux opérateurs temporels de LTL

Avant de les définir formellement, nous allons introduire les notions intuitives liées aux différents opérateurs temporels (modalités) qui constituent la logique temporelle linéaire. Pour cela, nous considérons l'ensemble de propositions $P = \{p, q\}$.

Pour illustrer la présentation des différents opérateurs temporels, nous allons utiliser des tableaux pour représenter des séquences de configurations. En voici un exemple :

P \ t	0	1	...	i	...
p	Faux	Vrai	—	—	—
q	Faux	Faux	Faux	Faux	Faux

Le tableau ci-dessus est à lire comme suit : A l'instant 0 (maintenant), p et q sont des propositions fausses (configuration $\{\}$). A l'instant 1, p est vrai et q est faux (configuration $\{p\}$). Pour tout instant entre 1 et un nombre $i \geq 1$, ainsi que pour l'instant i et tous les instants qui suivent, la valeur de p n'a aucune influence sur la vérification de l'expression[1] à laquelle fait référence cette séquence et la valeur de q est fausse (configurations $\{\}$ et $\{p\}$).

[1] Pour le lecteur averti, une expression LTL correspondante à cet exemple est $\neg p \wedge \circ p \wedge \Box \neg q$

Suivant ($\circ$) $\circ p$ signifie que la proposition p doit être vraie à l'instant suivant, comme l'illustre le tableau suivant. Remarquons que cette exigence ne porte que sur l'instant suivant et que sur la proposition p. Autrement dit, à tout autre moment, p peut être soit vrai, soit faux. La valeur de q, elle, n'a pas d'importance dans ce contexte.

P \ t	0	1	...	i	...
p	—	Vrai	—	—	—
q	—	—	—	—	—

Finalement ($\diamond$) $\diamond p$ signifie que la proposition p doit finalement être vraie, c'est-à-dire que p doit être vrai au moins à un moment dans le futur, comme illustré par le tableau suivant.

P \ t	0	1	...	i	...
p	—	—	—	Vrai	—
q	—	—	—	—	—

Remarquons que les deux séquences suivantes satisfont également $\diamond p$.

P \ t	0	1	...	i	...
p	Vrai	Faux	Faux	Faux	Faux
q	—	—	—	—	—

P \ t	0	1	...	i	...
p	Faux	Faux	Faux	Vrai	Vrai
q	—	—	—	—	—

Par contre, cette séquence, où p est faux tout le temps, ne satisfait pas $\diamond p$.

P \ t	0	1	...	i	...
p	Faux	Faux	Faux	Faux	Faux
q	—	—	—	—	—

Toujours ($\Box$) $\Box p$ signifie que la proposition p doit toujours être vraie comme illustré par le tableau suivant.

P \ t	0	1	...	i	...
p	Vrai	Vrai	Vrai	Vrai	Vrai
q	—	—	—	—	—

Il suffit que p soit faux à un seul instant pour que l'expression $\Box p$ ne soit pas vérifiée.

P \ t	0	1	...	i	...
p	Vrai	Vrai	Vrai	Faux	Vrai
q	—	—	—	—	—

Notons que ceci met en évidence l'équivalence suivante : $\Box p \equiv \neg(\Diamond \neg p)$. L'inverse est également vrai : $\Diamond p \equiv \neg(\Box \neg p)$.

Jusqu'à ($\mathbb{U}$) $p \mathbb{U} q$ signifie que la proposition p doit être vraie jusqu'à ce que la proposition q devienne vraie comme illustré par le tableau suivant.

P \ t	0	1	...	i	...
p	Vrai	Vrai	Vrai	—	—
q	—	—	—	Vrai	—

Remarquons que la séquence suivante satisfait $p \mathbb{U} q$.

P \ t	0	1	...	i	...
p	Faux	Faux	Faux	Faux	Faux
q	Vrai	Faux	Faux	Faux	Faux

Par contre, la proposition q doit absolument passer à vrai à un moment donné. Autrement dit, la séquence suivante ne satisfait pas $p \mathbb{U} q$.

P \ t	0	1	...	i	...
p	Vrai	Vrai	Vrai	Vrai	Vrai
q	Faux	Faux	Faux	Faux	Faux

Libération ($\mathbb{R}$) $p \mathbb{R} q$ signifie que la proposition q doit rester vraie jusqu'à ce que la proposition p devienne vraie et "libère" q comme illustré par le tableau suivant. Remarquons que $p \mathbb{R} q$ exige que q soit vrai jusqu'à ce que p devienne vrai, mais également à l'instant où q devient vrai (ce qui n'est pas le cas de $q \mathbb{U} p$). L'opérateur "libération" est l'opérateur dual de "jusqu'à".

P \ t	0	1	...	i	...
p	—	—	—	Vrai	—
q	Vrai	Vrai	Vrai	Vrai	—

De plus, p ne doit pas nécessairement devenir vrai. Autrement dit, la situation suivante, où p ne "libère" jamais q, respecte l'expression $p \mathbb{R} q$ (mais ne respecte pas l'expression $q \mathbb{U} p$).

P \ t	0	1	...	i	...
p	Faux	Faux	Faux	Faux	Faux
q	Vrai	Vrai	Vrai	Vrai	Vrai

Dans un souci de clarté, nous avons jusqu'ici introduit les opérateurs en considérant qu'ils prenaient des propositions comme arguments. Mais ces arguments peuvent en fait être constitués de n'importe quelle expression LTL (voir la définition 2.1.1), comme le suggère l'exemple 2.1.2.

Exemple 2.1.2 *Soit l'ensemble de propositions* $P = \{p, q\}$,
- $\square(\neg p \Rightarrow \circ q) \equiv$ *chaque fois que p est faux, q doit être vrai à l'instant suivant.*
- $\diamond(\square p) \equiv$ *à partir d'un moment, p est vrai indéfiniment.*

Voici un tableau récapitulatif des différents opérateurs temporels, φ, φ_1 et φ_2 étant des expressions LTL :

Notation	Nom anglais	Nom français	Signification
$\circ\varphi$	Next	Suivant	φ doit être vraie à l'instant suivant.
$\diamond\varphi$	Eventually	Finalement	φ doit être vraie à (au moins) un instant de la séquence.
$\square\varphi$	Always	Toujours	φ doit être vraie tout au long de la séquence.
$\varphi_1 \mathbb{U} \varphi_2$	Until	Jusqu'à	φ_2 doit être vraie à (au moins) un instant de la séquence. φ_1 doit être vraie jusqu'à cet instant (non compris).
$\varphi_1 \mathbb{R} \varphi_2$	Release	Libération	φ_2 doit être vraie jusqu'au premier instant où φ_1 est vraie, celui-ci y compris. Si un tel instant n'existe pas, φ_2 doit être vraie indéfiniment.

TAB. 2.1 – Tableau récapitulatif des opérateurs temporels de LTL

Définition formelle

Comme une expression LTL est vérifiée sur une séquence de configurations, nous définissons d'abord les concepts relatifs aux séquences avant de donner la définition formelle de la logique temporelle linéaire.

Définitions - Séquences

Une séquence σ est une liste ordonnée d'objets dans laquelle un même élément peut apparaître plusieurs fois à différentes positions. Une séquence peut être finie ou infinie. En général, nous représenterons une séquence finie σ de n objets par $\langle e_1, e_2, \ldots, e_n \rangle$. La longueur d'une séquence finie σ est notée $|\sigma|$. La séquence vide est notée ϵ. Nous représenterons une séquence infinie σ par $\langle e_1, e_2, \ldots \rangle$.

Pour une séquence σ, $\sigma(i)$ représente le i-ème élément et on dit que i est l'indice de $\sigma(i)$. σ^i représente le suffixe de σ obtenu en supprimant ses i premiers éléments. Ainsi, $\forall j : \sigma^i(j) = \sigma(i + j)$

La concaténation $\sigma\sigma'$ d'une séquence finie $\sigma = \langle e_1, e_2, \ldots, e_n \rangle$ avec une séquence (finie ou infinie) $\sigma' = \langle e'_1, e'_2, \ldots \rangle$ correspond à la séquence $\langle e_1, e_2, \ldots, e_n, e'_1, e'_2, \ldots \rangle$.

Une séquence σ d'éléments de l'ensemble E ou séquence σ sur E est une séquence σ telle que $\forall i : \sigma(i) \in E$.

Pour un ensemble E, $\Sigma^*(E)$ représente l'ensemble de toutes les séquences finies sur E et $\Sigma^\omega(E)$ représente l'ensemble de toutes les séquences infinies sur E.

Définitions - LTL

La syntaxe des expressions LTL est donnée par la définition 2.1.1.

Définition 2.1.1 *Soit P un ensemble de propositions,*
- ***true*** *et* ***false*** *sont des expressions* LTL *relatives à P.*
- *$\forall p \in P : p$ est une expression* LTL *relative à P.*
- *Si φ_1 et φ_2 sont des expressions* LTL *relatives à P, alors $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2$, $\circ\varphi_1$, $\diamond\varphi_1$, $\Box\varphi_1$, $\varphi_1 \mathbb{U} \varphi_2$ et $\varphi_1 \mathbb{R} \varphi_2$ sont des expressions* LTL *relatives à P.*

L'ensemble des expressions LTL *relatives à P est noté $\mathcal{LTL}(P)$*

Nous utilisons parfois "formules" comme synonyme d'expressions.

La sémantique d'une expression LTL est définie par rapport à une séquence infinie de configurations via la relation $\models_\omega$:

Définition 2.1.2 *Soit P un ensemble de propositions, $\sigma \in \Sigma^\omega(2^P)$ une séquence infinie et $\varphi \in \mathcal{LTL}(P)$ une expression* LTL*, σ satisfait φ si la relation $\sigma \models_\omega \varphi$, qui est définie ci dessous, est respectée :*
- $\sigma \models_\omega p$ *pour* $p \in P$ *ssi* $p \in \sigma(1)$
- $\sigma \models_\omega \neg\varphi$ *ssi* $\sigma \not\models_\omega \varphi$
- $\sigma \models_\omega \varphi_1 \vee \varphi_2$ *ssi* $\sigma \models_\omega \varphi_1 \vee \sigma \models_\omega \varphi_2$
- $\sigma \models_\omega \circ\varphi$ *ssi* $\sigma^1 \models_\omega \varphi$
- $\sigma \models_\omega \varphi_1 \mathbb{U} \varphi_2$ *ssi* $\exists i \geq 0 : \sigma^i \models_\omega \varphi_2 \wedge \forall j : 0 \leq j < i : \sigma^j \models_\omega \varphi_1$

On dira qu'une séquence finie σ satisfait partiellement une expression LTL φ si et seulement s'il existe une séquence infinie σ' telle que $\sigma\sigma'$ satisfait l'expression φ. Si une séquence σ ne satisfait pas partiellement une expression φ, on dit qu'elle viole l'expression φ. Cette distinction sera utile par la suite.

La sémantique des autres expressions LTL (2.1.1) n'est pas explicitement mentionnée car celles-ci sont dérivées des définitions suivantes :

$$
\begin{aligned}
\textbf{true} &\stackrel{def}{=} p \vee \neg p \\
\textbf{false} &\stackrel{def}{=} \neg\textbf{true} \\
\varphi_1 \wedge \varphi_2 &\stackrel{def}{=} \neg(\neg\varphi_1 \vee \neg\varphi_2) \\
\varphi_1 \Rightarrow \varphi_2 &\stackrel{def}{=} \neg\varphi_1 \vee \varphi_2 \\
\varphi_1 \mathbb{R} \varphi_2 &\stackrel{def}{=} \neg(\neg\varphi_1 \mathbb{U} \neg\varphi_2) \\
\diamond\varphi &\stackrel{def}{=} \textbf{true} \mathbb{U} \varphi \\
\Box\varphi &\stackrel{def}{=} \textbf{false} \mathbb{R} \varphi
\end{aligned}
$$

On peut facilement vérifier que :

$$
\begin{array}{lll}
\sigma & \vDash_\omega & \textbf{true} \\
\sigma & \nvDash_\omega & \textbf{false} \\
\sigma & \vDash_\omega & \varphi_1 \wedge \varphi_2 \text{ ssi } \sigma \vDash_\omega \varphi_1 \wedge \sigma \vDash_\omega \varphi_2 \\
\sigma & \vDash_\omega & \diamond\varphi \text{ ssi } \exists i \geq 0 : \sigma^i \vDash_\omega \varphi \\
\sigma & \vDash_\omega & \Box\varphi \text{ ssi } \forall i \geq 0 : \sigma^i \vDash_\omega \varphi \\
\sigma & \vDash_\omega & \varphi_1 \ \mathbb{R} \ \varphi_2 \text{ ssi } \forall i \geq 0 : \sigma^i \vDash_\omega \varphi_2 \vee \exists j : 0 \leq j < i : \sigma^j \vDash_\omega \varphi_1
\end{array}
$$

Nous pouvons encore mettre en évidence les équivalences suivantes :
- $\neg(\varphi_1 \ \mathbb{U} \ \varphi_2) \equiv \neg\varphi_1 \ \mathbb{R} \ \neg\varphi_2$
- $\neg(\varphi_1 \ \mathbb{R} \ \varphi_2) \equiv \neg\varphi_1 \ \mathbb{U} \ \neg\varphi_2$
- $\neg(\circ\varphi) \equiv \circ\neg\varphi$
- $\neg(\diamond\varphi) \equiv \Box\neg\varphi$
- $\neg(\Box\varphi) \equiv \diamond\neg\varphi$

Nous donnons également la définition d'un langage relatif à une expression LTL pour la relation $\vDash_\omega$, qui correspond à l'ensemble des séquences infinies qui satisfont l'expression considérée :

Définition 2.1.3 *Le langage d'une expression $\varphi \in \mathcal{LTL}(P)$ pour la relation $\vDash_\omega$ est défini par $\mathcal{L}_{\vDash_\omega}(\varphi) = \{\sigma \in \Sigma^\omega(2^P) \mid \sigma \vDash_\omega \varphi\}$*

2.2 Utilité de LTL en informatique

Vérification formelle de programmes informatiques

Actuellement, la logique temporelle linéaire (et la logique temporelle en général) a été développée et est principalement utilisée en *model checking* [20, 14, 21, 22] qui désigne une famille de méthodes formelles de vérification de programmes informatiques.

Un algorithme de *model checking* agit de la manière suivante : il prend en entrée un modèle, exprimé dans un formalisme imposé et représentant le comportement d'un système informatique, ainsi qu'une spécification qui exprime les propriétés que doivent vérifier certaines données du modèle. Il effectue ensuite un calcul et produit un résultat positif (toutes les exécutions prises en compte par le modèle satisfont la spécification) ou négatif (au moins une exécution prise en compte par le modèle ne respecte pas la spécification, dans quel cas une de ces exécutions est renvoyée comme contre-exemple). [18]

C'est pour la spécification des propriétés à vérifier que la logique temporelle linéaire va s'avérer utile. Il est en effet courant de vouloir exprimer des propriétés incluant des contraintes temporelles, en s'intéressant principalement à l'ordre dans lequel les évènements s'enchaînent.[18]

LTL permet notamment d'exprimer des propriétés qu'on peut caractériser selon deux types :
- les propriétés de sûreté (safety properties)
- les propriétés de vivacité (liveness properties)

Une propriété de sûreté d'un système informatique cherche à exprimer que "quelque chose de mauvais ne va pas arriver". On peut par exemple exprimer que la température d'un réacteur (variable $reactor\_temp$) ne doit jamais dépasser 1000 degrés par l'expression LTL suivante :

$$\Box\neg(reactor\_temp > 1000)$$

Une propriété de vivacité d'un système informatique cherche à exprimer que "quelque chose de bon va ou peut toujours se produire". On peut par exemple s'assurer qu'un programme va terminer (la proposition $end\_program$ représente cet évènement) grâce à l'expression LTL suivante :

$$\Diamond end\_program$$

Si l'on cherche à s'assurer que chaque socket s ouvert (proposition s_open) est fermé (proposition s_closed) par la suite, on peut également l'exprimer par une expression LTL :

$$\Box(s\_open \Rightarrow \Diamond s\_closed)$$

Notons qu'une même propriété peut regrouper des caractéristiques de sûreté et de vivacité. Les propriétés de sûreté et de vivacité sont discutées dans l'article [23].

L'utilisation de la logique temporelle linéaire en *model checking* est notamment exploitée avec succès par l'outil SPIN [24]. Cet outil permet de développer le modèle d'un système dans le langage de spécification de systèmes asynchrones Promela (PROcess MEta LAnguage) [24]. SPIN peut alors automatiquement vérifier, sur ce modèle, la satisfaction d'un ensemble de propriétés, exprimées en LTL.

Bien que ce ne soit pas dans le cadre de la vérification de modèles que nous allons utiliser la logique temporelle linéaire par la suite, les travaux réalisés à cette fin nous seront très utiles, notamment au travers de la transformation des expressions en LTL en automates, comme nous allons le voir dans la section 2.3.

Spécification formelle de systèmes

Dans le cadre de ce mémoire, nous allons quelque peu détourner LTL de son utilisation initiale. En effet, LTL va être exploitée, non pas pour vérifier les propriétés d'un système mais pour spécifier le comportement d'un système, dans une démarche qui se rapproche de [12, 11] et qui sera développée en détails au cours des chapitres suivants. Il s'agit d'une utilisation originale et très peu répandue de la logique temporelle linéaire.

Pour l'illustrer, imaginons un système composé d'un ensemble de tâches exécutables. LTL peut être utilisée pour organiser l'exécution successive de ces tâches en définissant des contraintes que la séquence des tâches exécutées doit respecter.

Par exemple, si nous désirons exprimer le fait qu'une tâche T_1 doit être exécutée durant le fonctionnement du système, nous pouvons utiliser l'expression LTL suivante :

$$\diamond T_1$$

Si nous voulons que le système n'autorise pas l'exécution d'une tâche T_1 tant que la tâche T_2 n'est pas exécutée, nous pouvons écrire l'expression LTL :

$$\neg T_1 \ \mathbb{U} \ T_2$$

Si nous désirons qu'une tâche T_2 soit toujours exécutée à la suite de la tâche T_1, nous pouvons utiliser l'expression LTL :

$$\square(T_1 \Rightarrow \diamond T_2)$$

2.3 Conversion d'expressions LTL en automates de Büchi

[Conversion LTL- automates de Büchi]
Cette section est consacrée à une méthode attractive et largement développée dans le domaine du *model checking* qui permet d'automatiser l'évaluation d'une expression LTL et qui consiste en la construction d'un automate qui accepte le même langage que l'expression considérée [22, 14, 25, 18]. Grâce à cette conversion, il est ainsi aisé, par exemple, de déterminer algorithmiquement si une séquence σ satisfait une expression LTL φ. Il suffit en effet de s'assurer que la séquence σ est acceptée par l'automate construit à partir de l'expression φ.

Nous allons dans un premier temps introduire la théorie des automates finis, notamment au travers des automates de Büchi. Nous aborderons alors le lien qu'il est possible de faire entre les expressions LTL définies sur des séquences infinies et les automates de Büchi. Nous illustrerons ce lien par quelques exemples. Nous verrons ensuite que la traduction des expressions LTL en automates est un domaine de recherche particulièrement florissant au travers d'un bref état de l'art. Nous conclurons enfin cette section en présentant un algorithme simple et facile à implémenter, qui permet de générer un automate de Büchi à partir d'une expression LTL.

Les automates finis

Un automate fini est une machine abstraite constituée d'un nombre fini d'états et de transitions entre ces états. Son comportement est dirigé par une séquence[1] fournie en entrée : l'automate passe d'état en état, suivant les transitions, à la lecture successive de chaque élément de la séquence fournie en entrée. L'automate est dit "fini" car il possède un nombre fini d'états distincts. Nous allons présenter formellement les automates finis sur des séquences finies, puis les automates finis sur des séquences infinies.

Les automates finis sur des séquences finies : les automates classiques

Formellement, un automate fini sur des séquences finies est un quintuplet $\mathcal{AC} = (\mathcal{A}, Q, \delta, Q^0, F)$ tel que :
- $\mathcal{A}$ est l'alphabet (fini)
- Q est l'ensemble (fini) d'états
- $\delta \subseteq Q \times \mathcal{A} \times Q$ est la relation de transition
- $Q^0 \subseteq Q$ est l'ensemble des états initiaux
- $F \subseteq Q$ est l'ensemble des états finaux

Un tel automate sera appelé par la suite automate classique.

Un automate peut être représenté par un graphe orienté, dans lequel l'ensemble des noeuds est Q et les liens orientés sont donnés par la relation de transition δ. Les états initiaux sont marqués par une flèche entrante et les états finaux par un double cercle.

Exemple 2.3.1 *La figure 2.1 représente l'automate* $\{\mathcal{A}, Q, \delta, Q^0, F\}$ *tel que :*

[1] en théorie des automates, on utilise plus généralement le terme "mot"

- $\mathcal{A} = \{a, b\}$
- $Q = \{q_1, q_2\}$
- $\delta = \{(q_1, b, q_2), (q_1, a, q_1), (q_2, a, q_1), (q_2, b, q_2)\}$
- $Q^0 = F = \{q_1\}$

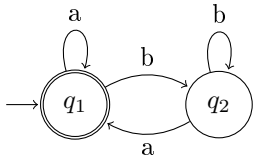


FIG. 2.1 – Un automate fini

Un état $q \in Q$ de l'automate $\mathcal{AC} = (\mathcal{A}, Q, \delta, Q^0, F)$ est accessible via une séquence finie $\sigma \in \Sigma^*(\mathcal{A})$ de longueur $n \geq 0$ si et seulement s'il existe une séquence finie d'états $\rho \in \Sigma^*(Q)$ de longueur $n+1$ telle que :

- $\rho(1) \in Q^0$ (le premier état est un état initial)
- $\forall i : 1 \leq i \leq n : (\rho(i), \sigma(i), \rho(i+1)) \in \delta$ (la suite des états est construite selon la relation de transition et la séquence σ)
- $\rho(n+1) = q$ (le dernier état est q)

Le statut de l'automate $\mathcal{AC} = (\mathcal{A}, Q, \delta, Q^0, F)$ pour une séquence finie $\sigma \in \Sigma^*(\mathcal{A})$, noté $Status(\mathcal{AC}, \sigma)$, désigne l'ensemble[1] des états de $\mathcal{AC}$ accessibles via σ.

Une séquence finie $\sigma \in \Sigma^*(\mathcal{A})$ de longueur $n \geq 0$ est acceptée par un automate classique $\mathcal{AC} = (\mathcal{A}, Q, \delta, Q^0, F)$ si et seulement s'il existe un état final $q \in F$ accessible via σ.

Le langage accepté par un automate classique $\mathcal{AC} = (\mathcal{A}, Q, \delta, Q^0, F)$ est défini par $\mathcal{L}(\mathcal{AC}) = \{\sigma \in \Sigma^*(\mathcal{A}) \mid \sigma \text{ est accepté par } \mathcal{AC}\}$.

Dans l'exemple 2.3.1, les séquences acceptées sont celles qui se terminent par a (comme par exemple $\langle a \rangle$, $\langle b, a \rangle$, $\langle a, b, a, b, b, a \rangle$) ainsi que la séquence vide ϵ.

Les automates finis sur des séquences infinies : les automates de Büchi

Les automates finis sur des séquences infinies les plus simples sont les automates de Büchi, qui ont les mêmes composants que les automates classiques (ce qui permet de les représenter de façon similaire), si ce n'est que les états finaux sont appelés états accepteurs. Un automate de Büchi diffère d'un automate classique par le langage qu'il accepte.

Intuitivement, la différence relève de la condition d'acceptation de la séquence. Alors qu'une séquence finie est acceptée par un automate classique si son dernier état visité est un état final, une séquence infinie est acceptée par

[1] les automates que nous avons définis ci-dessus sont non déterministes : à partir d'un état de l'automate et d'un élément de l'alphabet, on peut potentiellement atteindre plusieurs états

un automate de Büchi si elle visite infiniment souvent un état faisant partie de l'ensemble des états accepteurs.

Formellement, un automate de Büchi est un quintuplet $\mathcal{AB} = (\mathcal{A}, Q, \delta, Q^0, F)$ tel que :
- $\mathcal{A}$ est l'alphabet (fini)
- Q est l'ensemble (fini) d'états
- $\delta \subseteq Q \times \mathcal{A} \times Q$ est la relation de transition
- $Q^0 \subseteq Q$ est l'ensemble des états initiaux
- $F \subseteq Q$ est l'ensemble des états accepteurs

Une séquence infinie $\sigma \in \Sigma^\omega(\mathcal{A})$ est acceptée par un automate de Büchi $\mathcal{AB} = (\mathcal{A}, Q, \delta, Q^0, F)$ si et seulement s'il existe une séquence infinie d'états $\rho \in \Sigma^\omega(Q)$ telle que :
- $\rho(1) \in Q^0$ (le premier état est un état initial)
- $\forall i \geq 1 : (\rho(i), \sigma(i), \rho(i+1)) \in \delta$ (la séquence des états est construite conformément à la relation de transition et à la séquence σ)
- $\forall i \geq 1 : \exists j > i : \rho(j) \in F$ (la séquence d'états contient une infinité d'états appartenant à l'ensemble des états accepteurs)

Le langage accepté par un automate de Büchi $\mathcal{AB} = (\mathcal{A}, Q, \delta, Q^0, F)$ est défini par $\mathcal{L}(\mathcal{AB}) = \{\sigma \in \Sigma^\omega(\mathcal{A}) \mid \sigma \text{ est accepté par } \mathcal{AB}\}$.

Les automates finis sur des séquences infinies : les automates de Büchi généralisés

Les automates de Büchi généralisés sont semblables aux automates de Büchi si ce n'est qu'on y trouve un ensemble d'ensembles d'états accepteurs au lieu d'un ensemble d'états accepteurs. Une séquence infinie est acceptée par un tel automate si elle visite infiniment souvent au moins un état de chaque ensemble d'états accepteurs.

Formellement, un automate de Büchi généralisé est un quintuplet $\mathcal{ABG} = (\mathcal{A}, Q, \delta, Q^0, \mathcal{F})$ tel que :
- $\mathcal{A}$ est l'alphabet (fini)
- Q est l'ensemble (fini) d'états
- $\delta \subseteq Q \times \mathcal{A} \times Q$ est la relation de transition
- $Q^0 \subseteq Q$ est l'ensemble des états initiaux
- $\mathcal{F} \subseteq 2^Q$ est l'ensemble des ensembles d'états accepteurs

Une séquence infinie $\sigma \in \Sigma^\omega(\mathcal{A})$ est acceptée par un automate de Büchi généralisés $\mathcal{ABG} = (\mathcal{A}, Q, \delta, Q^0, \mathcal{F})$ si et seulement s'il existe une séquence infinie d'états $\rho \in \Sigma^\omega(Q)$ telle que :
- $\rho(1) \in Q^0$ (le premier état est un état initial)
- $\forall i \geq 1 : (\rho(i), \sigma(i), \rho(i+1)) \in \delta$ (la séquence des états est construite conformément à la relation de transition et à la séquence σ)
- $\forall i \geq 1 : \forall F \in \mathcal{F} : \exists j > i : \rho(j) \in F$ (la séquence d'états contient une infinité d'états appartenant à chacun des ensembles d'états accepteurs)

Le langage accepté par un automate de Büchi généralisé $\mathcal{ABG} = (\mathcal{A}, Q, \delta, Q^0, \mathcal{F})$ est défini par $\mathcal{L}(\mathcal{ABG}) = \{\sigma \in \Sigma^\omega(\mathcal{A}) \mid \sigma \text{ est accepté par } \mathcal{ABG}\}$.

Il est possible de transformer un automate de Büchi généralisé en automate de Büchi qui accepte le même langage [26].

LTL et les automates de Büchi

Etant donné un ensemble de propositions P, une méthode attractive pour pouvoir évaluer la validité d'une séquence $\sigma \in \Sigma^\omega(2^P)$ par rapport à une expression $\varphi \in \mathcal{LTL}(P)$ est de construire un automate de Büchi $\mathcal{AB} = (\mathcal{A}, Q, \delta, Q^0, F)$ dont l'alphabet $\mathcal{A} = 2^P$ et tel que le langage qu'il accepte corresponde au langage de φ pour la relation $\models_\omega$, soit $\mathcal{L}_{\models_\omega}(\varphi) = \mathcal{L}(\mathcal{AB})$.

Avant d'aborder la transformation proprement dite en automates de Büchi, nous présentons quelques exemples d'expressions LTL, relatives à un ensemble de propositions $P = \{p, q, r\}$ accompagnées chacune d'un automate de Büchi qui accepte le même langage.

Notons que pour simplifier la représentation des automates, les transitions sont étiquetées par une formule propositionnelle, et ce pour représenter l'ensemble des configurations qui vérifie cette formule. Par exemple, l'étiquette $p \wedge \neg q$ pour un ensemble de propositions $P = \{p, q, r\}$ représente les configurations $\{p\}$ et $\{p, r\}$. L'étiquette particulière **true** est satisfaite par n'importe quelle configuration de P.

Les figures 2.2 (p), 2.3 ($\circ p$), 2.4 ($\diamond p$), 2.5 ($\Box p$), 2.6 ($p \mathbb{U} q$) et 2.7 ($p \mathbb{R} q$) montrent les automates pour les expressions de base correspondant à chacun des opérateurs temporels.

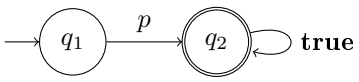


FIG. 2.2 – Un automate de Büchi correspondant à l'expression p

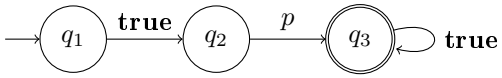


FIG. 2.3 – Un automate de Büchi correspondant à l'expression $\circ p$

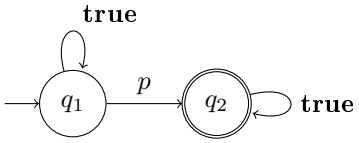


FIG. 2.4 – Un automate de Büchi correspondant à l'expression $\diamond p$

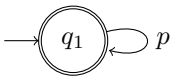


FIG. 2.5 – Un automate de Büchi correspondant à l'expression $\Box p$

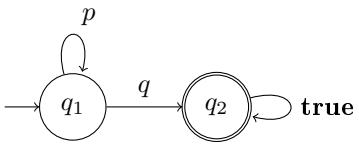


FIG. 2.6 – Un automate de Büchi correspondant à l'expression $p \mathbb{U} q$

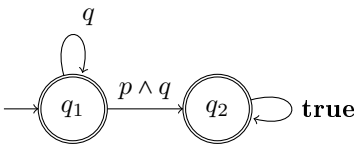


FIG. 2.7 – Un automate de Büchi correspondant à l'expression $p \mathbb{R} q$

Les figures 2.8 ($\diamond(\Box p)$), 2.9 ($\Box(\diamond p)$), 2.10 ($\diamond(q \vee \Box p)$), 2.11 ($(p \mathbb{U} q) \vee (\Box r)$) et 2.12 ($\diamond p \wedge \diamond q$) montrent des automates de Büchi relatifs à des expressions LTL plus complexes.

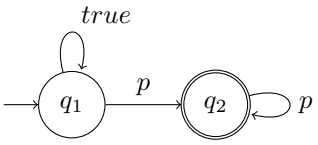


FIG. 2.8 – Un automate de Büchi correspondant à l'expression $\diamond(\Box p)$

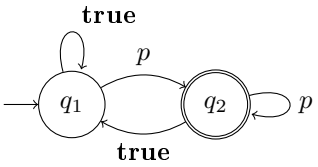


FIG. 2.9 – Un automate de Büchi correspondant à l'expression $\Box(\diamond p)$

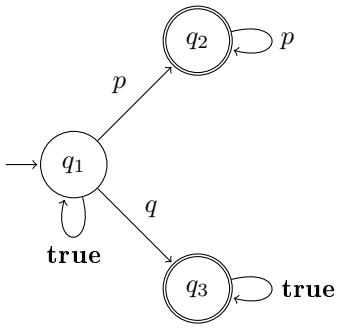


FIG. 2.10 – Un automate de Büchi correspondant à l'expression $\diamond(q \vee \Box p)$

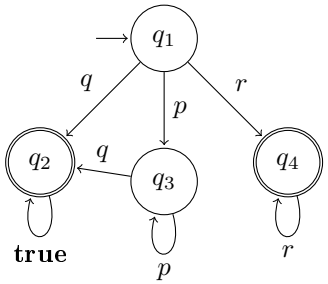


FIG. 2.11 – Un automate de Büchi correspondant à l'expression $(p \mathbb{U} q) \vee (\Box r)$

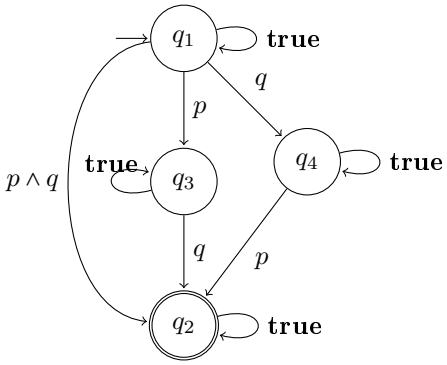


FIG. 2.12 – Un automate de Büchi correspondant à l'expression $\diamond p \wedge \diamond q$

Un domaine de recherche florissant

La construction d'automates à partir d'expressions de logique temporelle suscite depuis ces vingt dernières années, et de manière croissante, un intérêt particulièrement important, notamment en raison de son utilisation en *model checking* [20, 14, 21, 22]. La base des différents travaux relatifs à ce domaine repose sur les résultats du travail de Büchi sur la décidabilité de la théorie monadique du premier et du second ordre à un successeur [27], qui ont été obtenus grâce à une transformation en automates définis sur des mots infinis.

Il est assez clair que la logique temporelle linéaire peut être traduite en logique monadique du premier ordre à un successeur, et donc en automates définis sur des mots infinis.

Comme le problème est PSPACE-complet [28], les algorithmes permettant de générer des automates à partir d'expressions LTL sont de complexité au pire des cas exponentielle en fonction de la taille de l'expression LTL considérée (voir la définition 2.3.1) [29, 26, 18].

Définition 2.3.1 *La taille* $|\varphi|$ *d'une expression* LTL $\varphi \in \mathcal{LTL}(P)$ *est définie par induction comme suit :*

- $|\boldsymbol{true}| = |\boldsymbol{false}| = |p| = 1,\ \forall p \in P$
- $|\neg\varphi| = |\varphi|,\ |\varphi_1 \vee \varphi_2| = |\varphi_1| + |\varphi_2|,\ |\circ\varphi| = 1 + |\varphi|$ *et* $|\varphi_1\ \mathbb{U}\ \varphi_2| = 1 + |\varphi_1| + |\varphi_2|$

La première méthode de construction apparue [30, 31] (et à nouveau présentée dans [26]), utilisant la méthode des tableaux déclaratifs [32, 33] est directement issue des résultats théoriques et réalise en pratique un grand nombre de calculs inutiles, notamment en construisant d'office tous les états candidats sans tenir compte de la structure de l'expression LTL. En fait, cet algorithme est exponentiel dans le pire comme dans le meilleur des cas. Il est clair que cette construction peut être réalisée d'une façon telle que, en pratique, notamment pour les expressions LTL utilisées le plus souvent, les performances en temps et en taille soient meilleures. C'est ainsi que de nombreux travaux ont été réalisés pour proposer des algorithmes basés sur la méthode des tableaux incrémentaux ([32, 33]) qui cherche à pallier au trop grand nombre de calculs effectués, en ne générant que des états accessibles, les ajoutant à l'automate au fur et à mesure. On peut citer [34] dont un algorithme inspiré de ce travail sera présenté ci-après. De nombreuses améliorations, dont [35, 36, 37, 38], ont été proposées depuis et fournissent des résultats forts satisfaisants, tandis que des voies quittant la démarche habituelle, basée sur la méthode des tableaux, sont explorées avec succès ([39]). On notera qu'une implémentation très performante a été réalisée tout récemment dans l'entreprise *Mission Critical*. Celle-ci améliore de manière significative la méthode connue la plus efficace ([39]) en y intégrant notamment une optimisation présentée au chapitre 4.

Bien qu'on puisse se poser la question sur la façon dont on peut mesurer ces améliorations puisque la complexité dans le pire des cas reste toujours la même, l'expérience a montré que pour un grand nombre d'expressions LTL utilisées en pratique dans le cadre du *model checking*, les automates restent assez petits. On peut tenter de comparer pragmatiquement les algorithmes en utilisant l'outil [40]. Malgré son processus de construction intrinsèquement exponentiel, construire efficacement un automate à partir d'une expression LTL semble donc être possible en pratique.

Présentation d'un algorithme en 3 phases

Nous présentons ci-dessous un algorithme, que nous appellerons algorithme en 3 phases, permettant de générer un automate de Büchi à partir d'une expression LTL $\varphi \in \mathcal{LTL}(P)$ pour un ensemble de propositions P. Cet algorithme est équivalent à celui présenté dans [34, 14] auquel nous renvoyons le lecteur pour

la preuve complète et détaillée de sa correction. Il est notamment utilisé par l'outil Spin [24]. Nous avons choisi de le présenter car il est assez intuitif tout en étant à la base de nombreuses méthodes très efficaces.

Comme nous l'avons évoqué précédemment, la génération d'automates à partir d'une expression LTL est un problème de complexité exponentielle. Néanmoins, cet algorithme, par sa méthode incrémentale, évite un grand nombre de calculs inutiles.

Phase 1 : Normalisation de l'expression LTL

Avant d'appliquer l'algorithme proprement dit, il convient de transformer l'expression LTL dans sa forme normale négative (voir définition 2.3.2), en repoussant les négations jusqu'aux propositions ainsi qu'en remplaçant toutes les sous-expressions (on fait référence à toutes les expressions LTL qui composent l'expression considérée, en ce compris elle-même) de la forme $\diamond\varphi$ par **true** $\mathbb{U}\ \varphi$ et les sous-expressions de la forme $\Box\varphi$ par **false** $\mathbb{R}\ \varphi$. Par la suite, nous allons supposer que les expressions LTL sont écrites dans leur forme normale négative.

Définition 2.3.2 *Soit P un ensemble de propositions,*

- ***true***, ***false*** *sont des expressions* LTL *relatives à P en forme normale négative*
- *$\forall p \in P : p$ et $\neg p$ sont des expressions* LTL *relatives à P en forme normale négative*
- *Si φ_1 et φ_2 sont des expressions* LTL *relatives à P en forme normale négative, alors $\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2$, $\circ\varphi_1$, $\varphi_1\ \mathbb{U}\ \varphi_2$ et $\varphi_1\ \mathbb{R}\ \varphi_2$ sont des expressions* LTL *relatives à P en forme normale négative.*

On peut également procéder à une simplification de l'expression LTL. Il existe de nombreuses simplifications possibles [18]. Par exemple : $\varphi \wedge (\mu\ \mathbb{U}\ \varphi) \equiv \varphi$.

Phase 2 : Génération d'un graphe à partir d'une expression LTL normalisée

La structure de base utilisée par cet algorithme est le noeud (*node*) :

record *node* = [*ID* : NodeID,
Incoming : NodeID set,
Old : Formula set,
New : Formula set,
Next : Formula set] ;

Pour un noeud q :

- *ID* est l'identifiant du noeud q
- *Incoming* est l'ensemble des identifiants des noeuds prédécesseurs de q. Chaque identifiant i de la liste représente un lien du noeud i vers q
- *Old*, *New* et *Next* sont des ensembles destinés à contenir des sous-expressions de φ

L'idée est de constuire un ensemble *Nodes* de noeuds traités qui représenteront les états de l'automate de Buchi. Intuitivement, le traitement d'un noeud est le suivant : initialement, on a une série de sous-expressions de φ dans le

champ New, qu'on peut voir comme une conjonction d'expressions LTL que doit satisfaire une séquence de configurations de P, soit $\sigma = \langle conf_0, conf_1, conf_2, \ldots \rangle$. Au début du processus, New contiendra l'expression φ. L'idée du traitement est d'isoler, en analysant en profondeur les expressions de New, d'une part ce que doit partiellement satisfaire la séquence $\langle conf_0 \rangle$, et d'autre part ce que doit satisfaire le reste de la séquence, soit $\langle conf_1, conf_2, \ldots \rangle$. Les sous-expressions de φ qui doivent être partiellement satisfaites par la séquence $\langle conf_0 \rangle$ sont stockées dans Old et les sous-expressions qui doivent être satisfaites par le reste de la séquence sont stockées dans $Next$. Ceci est illustré par le schéma ci-dessous :

$$\underbrace{\langle \underbrace{conf_0}_{Old}, \underbrace{conf_1, conf_2, \ldots}_{Next} \rangle}_{New}$$

Par exemple, si une séquence de configuration de $\{p, r\}$ doit satisfaire une expression $r \wedge \circ p$ ($New = \{r, \circ p\}$), il faut que la séquence constituée uniquement du premier élément de la configuration satisfasse partiellement r ($Old = \{r\}$) et que le reste de la séquence satisfasse p ($Next = \{p\}$). Notons que le noeud est divisé en deux noeuds s'il existe deux manières différentes de satisfaire les expressions de New. Par exemple, si une séquence de configuration de $\{p, r\}$ doit satisfaire une expression $p \mathbb{U} r$ ($New = \{p \mathbb{U} r\}$), il y a deux possibilités : soit la séquence constituée uniquement du premier élément de la configuration satisfait partiellement p ($Old = \{p\}$) et le reste de la séquence doit satisfaire $p \mathbb{U} r$ ($Next = \{p \mathbb{U} r\}$), soit la séquence constituée uniquement du premier élément de la configuration satisfait partiellement r ($Old = \{r\}$) et on n'exige rien de particulier pour le reste de la séquence ($Next = \varnothing$).

Intuitivement, un noeud est traité lorsque toutes les sous-expressions de New ont été "converties" dans les champs Old et $Next$, ce champ New étant donc vide. Lorsqu'un noeud est traité, on ajoute ce noeud à l'ensemble $Nodes$ de tous les noeuds traités et on crée un noeud "fils" (le lien de parenté étant retenu par le champ $Incoming$) qui réitère l'opération pour le reste de la séquence (on copie le champ $Next$ du noeud parent dans le champ New du noeud fils). Notons que si un noeud équivalent, c'est-à-dire dont les champs $Next$ et Old sont les mêmes, fait déjà partie de l'ensemble des noeuds traités, on n'ajoute pas ce noeud (et on ne crée pas de noeud fils) mais on fusionne les champs $Incoming$ des noeuds équivalents. C'est ainsi que l'on peut garantir que l'algorithme se termine puisque les champs $Next$ et Old contiennent uniquement des sous-expressions de φ et sont par conséquent bornés.

Nous utiliserons le symbôle $\Leftarrow$ pour assigner une valeur à un champ. Par exemple, $New \Leftarrow \{\varphi\}$ assigne un ensemble contenant l'expression φ au champ New. $< X > (q)$ désignera le champ $< X >$ d'un noeud q. Par exemple, $Old(q)$ désigne le champ Old du noeud q. $init$ est une valeur particulière de type $NodeID$. Nous utiliserons également la fonction $new\_ID()$, qui génère une nouvelle valeur de type $NodeID$ (différente de $init$) à chaque fois qu'elle est appelée, et la fonction Neg qui est définie comme ceci pour $p \in P$: $Neg(p) = \neg p$, $Neg(\neg p) = p$, $Neg(\mathbf{true}) = \mathbf{false}$ et $Neg(\mathbf{false}) = \mathbf{true}$.

La fonction $create\_graph(\varphi)$ retourne un ensemble de noeuds qui permettra de déduire un automate de Büchi pour l'expression φ. Le noeud de base est le noeud *init* à partir duquel l'expression φ doit être satisfaite (le champ $Next$ égal $\{\varphi\}$).

---

fonction $create\_graph(\varphi)$

$expand($ [$ID \Leftarrow init$,
$Incoming \Leftarrow \{\}$,
$Old \Leftarrow \varnothing$,
$New \Leftarrow \varnothing$,
$Next \Leftarrow \{\varphi\}$] , $\varnothing$);

La fonction récursive *expand* accepte deux paramètres : le noeud courant et l'ensemble des noeuds précédemment construits et retourne un ensemble de noeuds.

---

fonction $expand(q, Nodes)$

---

if $New(q)$ est vide **then**
  if Il existe un noeud r dans $Nodes$ tel que
  $Old(r) = Old(q)$ et $Next(r) = Next(q)$ **then**
    $Incoming(r) \Leftarrow Incoming(r) \cup Incoming(q)$;
    return $Nodes$;
  else
    $ID(q) \Leftarrow new\_ID()$
    $expand($ [$ID \Leftarrow ID(q)$,
      $Incoming \Leftarrow \{ID(q)\}$,
      $Old \Leftarrow \varnothing$,
      $New \Leftarrow Next(q)$,
      $Next \Leftarrow \varnothing$] , $Nodes \cup \{q\}$);
  end if ;
else
  Soit $\eta \in New(q)$;
  $New(q) \Leftarrow New(q) - \{\eta\}$;
  if $\eta \in Old(q)$ **then**
    return $expand(q, Nodes)$;
  end if ;
  if $\eta \in \{p \mid p \in P\} \cup \{\neg p \mid p \in P\} \cup \{\mathbf{true}, \mathbf{false}\}$ **then**
    if $\eta =$ **false** ou $Neg(\eta) \in Old(q)$ **then**
      return $Nodes$;
    end if ;
    if $\eta =$ **true** ou $Neg(\eta) \notin Old(q)$ **then**
      $q' :=$ [$ID \Leftarrow ID(q)$,
        $Incoming \Leftarrow Incoming(q)$,
        $Old \Leftarrow Old(q) \cup \{\eta\}$,
        $New \Leftarrow New(q)$,
        $Next \Leftarrow Next(q)$] ;
      return $expand(q', Nodes)$;
    end if ;

end if ;
if $\eta = \mu \mathbb{U} \psi$ **then**
  $q_1 := [\ ID \Leftarrow ID(q),$
  $Incoming \Leftarrow Incoming(q),$
  $Old \Leftarrow Old(q) \cup \{\eta\},$
  $New \Leftarrow New(q) \cup \{\mu\},$
  $Next \Leftarrow Next(q) \cup \{\mu \mathbb{U} \psi\}\ ]$;
  $q_2 := [\ ID \Leftarrow ID(q),$
  $Incoming \Leftarrow Incoming(q),$
  $Old \Leftarrow Old(q) \cup \{\eta\},$
  $New \Leftarrow New(q) \cup \{\psi\},$
  $Next \Leftarrow Next(q)\ ]$;
  return $expand(q_2, expand(q_1, Nodes))$;
end if ;
if $\eta = \mu \mathbb{R} \psi$ **then**
  $q_1 := [\ ID \Leftarrow ID(q),$
  $Incoming \Leftarrow Incoming(q),$
  $Old \Leftarrow Old(q) \cup \{\eta\},$
  $New \Leftarrow New(q) \cup \{\mu \wedge \psi\},$
  $Next \Leftarrow Next(q)\ ]$;
  $q_2 := [\ ID \Leftarrow ID(q),$
  $Incoming \Leftarrow Incoming(q),$
  $Old \Leftarrow Old(q) \cup \{\eta\},$
  $New \Leftarrow New(q) \cup \{\psi\},$
  $Next \Leftarrow Next(q) \cup \{\mu \mathbb{R} \psi\}\ ]$;
  return $expand(q_2, expand(q_1, Nodes))$;
end if ;
if $\eta = \mu \vee \psi$ **then**
  $q_1 := [\ ID \Leftarrow ID(q),$
  $Incoming \Leftarrow Incoming(q),$
  $Old \Leftarrow Old(q) \cup \{\eta\},$
  $New \Leftarrow New(q) \cup \{\mu\},$
  $Next \Leftarrow Next(q)\ ]$;
  $q_2 := [\ ID \Leftarrow ID(q),$
  $Incoming \Leftarrow Incoming(q),$
  $Old \Leftarrow Old(q) \cup \{\eta\},$
  $New \Leftarrow New(q) \cup \{\psi\},$
  $Next \Leftarrow Next(q)\ ]$;
  return $expand(q_2, expand(q_1, Nodes))$;
end if ;
if $\eta = \mu \wedge \psi$ **then**
  $q' := [\ ID \Leftarrow ID(q),$
  $Incoming \Leftarrow Incoming(q),$
  $Old \Leftarrow Old(q) \cup \{\eta\},$
  $New \Leftarrow New(q) \cup \{\mu, \psi\},$
  $Next \Leftarrow Next(q)\ ]$;
  return $expand(q', Nodes)$;
end if ;
if $\eta = \circ\mu$ **then**
  $q' := [\ ID \Leftarrow ID(q),$

$Incoming \Leftarrow Incoming(q)$,
$Old \Leftarrow Old(q) \cup \{\eta\}$,
$New \Leftarrow New(q)$,
$Next \Leftarrow Next(q) \cup \{\mu\}$] ;
return $expand(q', Nodes)$;
end if ;
end if ;

Pour le noeud courant q, on vérifie que le champ New de q est vide. Si c'est le cas, on regarde s'il existe un noeud r dans $Nodes$ dont les champs Old et $Next$ sont identiques, dans quel cas la liste des prédécesseurs de q est ajoutée à la liste des prédécesseurs de r. S'il n'existe pas de tel noeud, on génère un nouvel identifiant à q qu'on ajoute ensuite à $Nodes$ et la fonction $expand$ est appelée à nouveau à partir d'un nouveau noeud courant q' formé comme suit :

- on crée un lien entre q et q' en initialisant le champ $Incoming$ de q' avec l'identifiant de q
- le champ New de q' est initialisé à $Next(q)$
- les champs Old et $Next$ sont vides

Si le champ New de q n'est pas vide, une expression η est sélectionnée et retirée de cet ensemble. Si η appartient déjà à Old, on applique à nouveau la fonction $expand$ sur le noeud q.

Supposons maintenant que η n'est pas dans $Old(q)$. Selon la structure de l'expression η, le noeud q est scindé en deux parties q_1 et q_2 ou remplacé par une nouvelle version q'. Ces nouveaux noeuds sont formés en copiant les champs $Incoming$, Old, New et $Next$ à partir du noeud q et en ajoutant η au champ Old. En fonction de la structure de l'expression, des expressions LTL peuvent être ajoutées au champ New et $Next$ selon les cas suivants :

η **est une proposition** p**, la négation d'une proposition** $\neg p$ **ou une constante booléenne true ou false.** Si η est **false** ou que $\neg\eta$ est dans Old, le noeud courant q contient une contradiction, il n'est donc pas sélectionné et on retourne simplement la liste $Nodes$ sans y ajouter q. Sinon, la fonction $expand$ est à nouveau appelée pour une copie q' de q.

η **est du type** $\mu \mathbb{U} \psi$**.** Comme $\mu \mathbb{U} \psi$ est une expression équivalente[1] à $\psi \vee (\mu \wedge \circ(\mu \mathbb{U} \psi)$, le noeud q est scindé en deux noeuds q_1 ou q_2. μ est ajoutée à $New(q_1)$ et $\mu \mathbb{U} \psi$ à $Next(q_1)$. ψ est ajoutée à $New(q_2)$.

η **est du type** $\mu \mathbb{R} \psi$**.** Comme $\mu \mathbb{R} \psi$ est équivalente[2] à $(\psi \wedge \mu) \vee (\psi \wedge \circ(\mu \mathbb{R} \psi)$, le noeud q est scindé, de façon similaire, en deux noeuds q_1 ou q_2. $\mu \wedge \psi$ est ajoutée à $New(q_1)$. ψ est ajoutée à $New(q_2)$ et $\mu \mathbb{R} \psi$ à $Next(q_2)$.

η **est du type** $\mu \vee \psi$**.** Le noeud q est scindé à nouveau en deux noeuds q_1 et q_2. Dans ce cas, μ est ajoutée au champ New de q_1 et ψ au champ New de q_2.

[1] En effet, pour que $\mu \mathbb{U} \psi$ soit vraie à un moment donné, il faut que soit ψ soit vraie à ce moment, soit que ce soit μ qui soit vraie à ce moment et que $\mu \mathbb{U} \psi$ soit vraie à partir de l'instant suivant

[2] En effet, pour que $\mu \mathbb{R} \psi$ soit vraie à un moment donné, il faut que soit ψ et μ soient vraies à ce moment, soit que ψ soit vraie à ce moment et que $\mu \mathbb{R} \psi$ soit vraie à partir de l'instant suivant

η **est du type** $\mu \wedge \psi$**.** Le noeud q est remplacé par q'. μ et ψ sont ajoutées au champ New de q' vu que ces expressions doivent être vraies toutes les deux pour que η soit vraie.

η **est du type** $\circ\mu$**.** Le noeud q est remplacé par q'. μ est ajoutée au champ $Next$ de q'.

La correction de l'algorithme, dont la preuve complète est donnée dans [34], est la conséquence de deux invariants. Soit $\bigwedge S$ la conjonction de tous les éléments de l'ensemble S :

1. Quand un noeud q est scindé en deux noeuds q_1 et q_2, l'invariant suivant est maintenu :

$$\begin{aligned} &\bigwedge Old(q) \wedge \bigwedge New(q) \wedge \bigwedge Next(q) \longleftrightarrow \\ &(\bigwedge Old(q_1) \wedge \bigwedge New(q_1) \wedge \bigwedge Next(q_1)) \vee \\ &(\bigwedge Old(q_2) \wedge \bigwedge New(q_2) \wedge \bigwedge Next(q_2)) \end{aligned}$$

2. Quand un noeud q est remplacé par un noeud q', l'invariant suivant est maintenu :

$$\begin{aligned} &\bigwedge Old(q) \wedge \bigwedge New(q) \wedge \bigwedge Next(q) \longleftrightarrow \\ &\bigwedge Old(q') \wedge \bigwedge New(q') \wedge \bigwedge Next(q') \end{aligned}$$

Il faut noter qu'en pratique, nous avons procédé à une optimisation de l'algorithme présenté précédemment, dans une démarche qui s'est avérée par la suite similaire à [38]. L'objectif de cette optimisation a été d'améliorer la détection de noeuds équivalents de manière à fusionner un plus grand nombre de noeuds au cours de la génération, évitant ainsi de nombreux calculs inutiles. La structure du noeud est dans ce cas plus complexe et c'est pour cela que nous avons préféré présenter, dans un souci de clarté, l'algorithme de base.

Phase 3 : Conversion en automate

Pour un ensemble de propositions P, la liste de noeuds $Nodes$ construite par l'algorithme décrit ci-dessus pour l'expression $\varphi \in \mathcal{LTL}(P)$ peut maintenant être convertie en un automate de Büchi généralisé $\mathcal{ABG} = (\mathcal{A}, Q, \delta, Q^0, \mathcal{F})$:

- L'alphabet $\mathcal{A}$ est égal à l'ensemble 2^P
- L'ensemble des états Q est constitué des noeuds de $Nodes$
- La relation de transition δ est constituée comme ceci : $(r, \mu, r') \in \delta$ si et seulement si $r \in Incoming(r')$ et μ satisfait la conjonction des propositions (sous formes positives et négatives) de $Old(r')$
- L'ensemble des états initiaux Q^0 est constitué du noeud dont l'identifiant est *init*
- L'ensemble des ensembles d'états accepteurs $\mathcal{F}$ est constitué comme ceci : pour chaque sous-expression de φ de type $\mu \,\mathbb{U}\, \psi$, on construit un ensemble $F = \{q \in Nodes \mid \mu \,\mathbb{U}\, \psi \notin Old(q) \text{ ou } \psi \in Old(q)\}$. $\mathcal{F}$ est constitué de tous ces ensembles.

Cette conversion en automate de Büchi généralisé est assez intuitive. Notons que les expressions de type $\mu \mathbb{U} \psi$ sont les seules (dans une expression LTL normalisée) qui imposent l'existence future d'une configuration particulière (en l'occurrence, qui satisfait l'expression ψ appelée "partie droite" de l'expression). C'est pour cette raison que l'on impose des conditions d'acceptations pour chacune de ces expressions de type $\mathbb{U}$. L'ensemble d'états accepteurs pour une expression de type $\mathbb{U}$ est constitué des états qui, s'ils satisfont l'expression considérée, satisfont également sa partie droite.

Pour rappel, la preuve formelle de la correction de cette conversion est donnée dans [34]. On rappelle également qu'il est possible de transformer l'automate de Büchi généralisé ci-dessus en un automate de Büchi classique [26].

Exemple illustratif

Soit un ensemble de propositions $P = \{p\}$, nous allons construire un automate de Büchi qui accepte le même langage que le langage de l'expression $\neg(\Box p)$ pour la relation $\models_\omega$.

Phase 1 : Normalisation de $\neg(\Box p)$ Il convient d'abord de normaliser cette expression.

$$\begin{aligned} \neg(\Box p) &\equiv \Diamond \neg p \\ &\equiv true \ \mathbb{U} \ \neg p \end{aligned}$$

Phase 2 : Génération d'un graphe à partir de $true \ \mathbb{U} \ \neg p$ La figure 2.13 représente les noeuds construits au cours de l'exécution de la fonction $create\_graph(\mathbf{true} \ \mathbb{U} \ \neg p)$. Les noeuds mis en évidence sont ceux qui font partie de l'ensemble de noeuds renvoyés par la fonction.

Phase 3 : Conversion en automate

La figure 2.14 montre l'automate de Büchi qui est déduit à partir de la liste de noeuds renvoyée par la fonction $create\_graph(\mathbf{true} \ \mathbb{U} \ \neg p)$. L'ensemble des ensembles d'états accepteurs est constitué d'un seul ensemble d'états accepteurs, mis en évidence en vert.

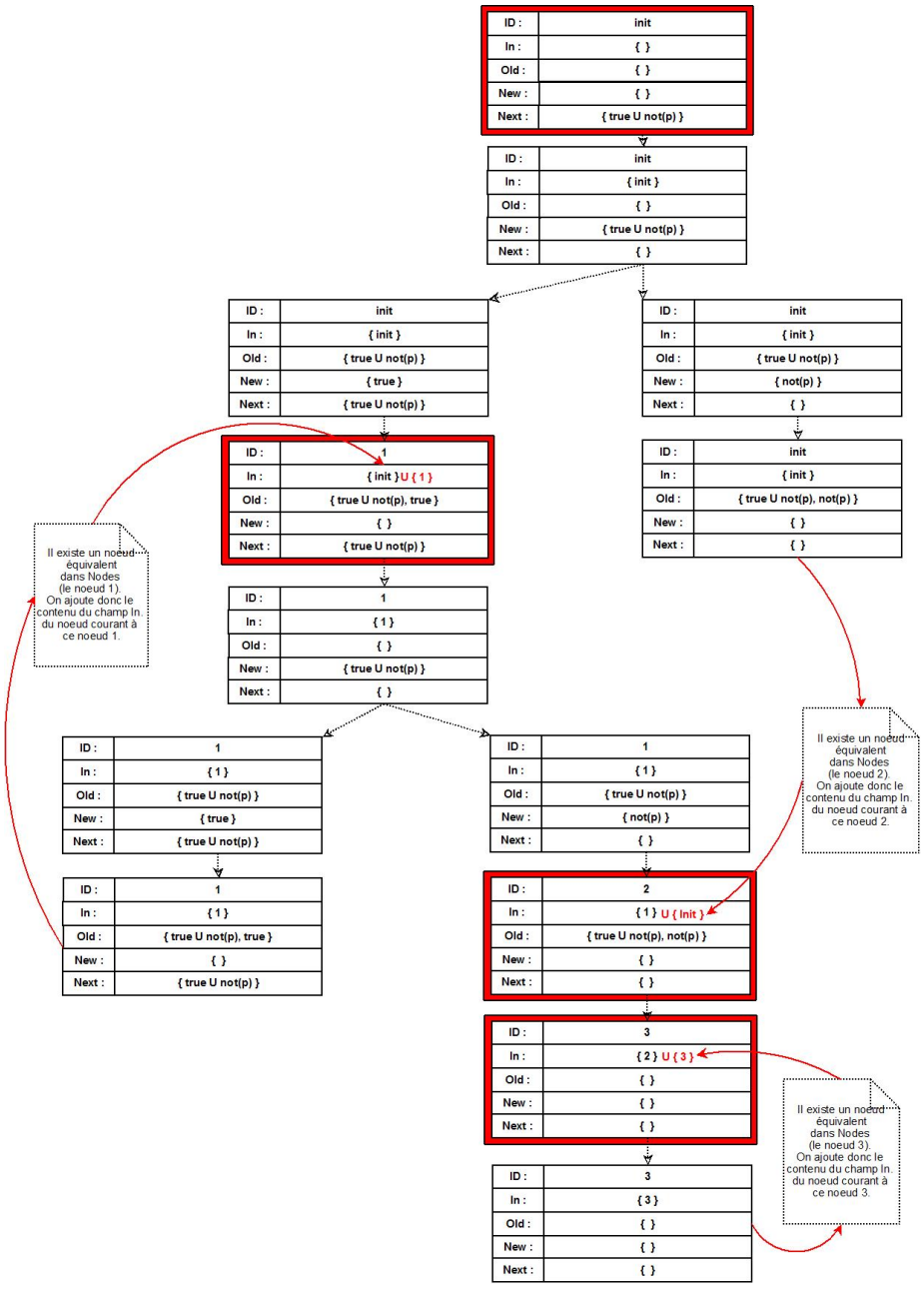


FIG. 2.13 – Aperçu des noeuds construits par la fonction *create_graph*(**true** $\mathbb{U}$ $\neg p$)

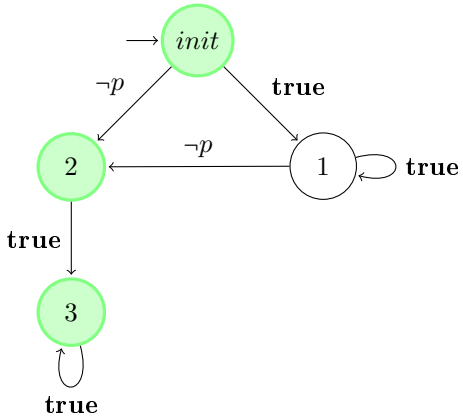


FIG. 2.14 – L'automate de Büchi généralisé déduit de la liste de noeuds renvoyée par la fonction $create\_graph(\mathbf{true}\ \mathbb{U}\ \neg p)$

Rappelons que les automates de Büchi ne concernent que des séquences infinies, il n'est donc pas erroné de considérer un état qui n'est visité qu'au début d'une séquence, comme l'état initial, comme faisant partie de l'ensemble d'états accepteurs.

2.4 LTL et les séquences finies

La logique temporelle linéaire a été jusqu'ici abordée en relation avec des séquences infinies. Les automates que génère l'algorithme en 3 phases précédemment décrit sont par conséquent définis sur des séquences infinies. Dans le cadre de ce mémoire, nous allons utiliser LTL pour spécifier le comportement d'un système composé d'un ensemble de tâches à réaliser. Nous développerons cette idée au cours du chapitre 3. L'idée principale est de s'assurer qu'une séquence finie de tâches, appelée exécution, est conforme à un certain nombre de contraintes et d'exprimer ces contraintes à l'aide des opérateurs LTL. Dans ce contexte, nous allons adapter la sémantique de LTL à des séquences finies, et modifier l'algorithme de transformation d'une expression LTL en conséquence, en produisant un automate classique plutôt qu'un automate de Büchi. Cette démarche a été également abordée par [41], [42] et [43].

Définition de LTL par rapport à des séquences finies

La sémantique de la logique temporelle linéaire LTL est définie par rapport à des séquences finies de configurations via la relation $\vDash_*$:

Définition 2.4.1 *Soit P un ensemble de propositions, $\sigma \in \Sigma^*(2^P)$ une séquence finie et $\varphi \in \mathcal{LTL}(P)$ une expression* LTL, *σ satisfait φ si la relation $\sigma \vDash_* \varphi$, qui est définie ci dessous, est respectée :*

- $\sigma \vDash_* \boldsymbol{true}$
- $\sigma \nvDash_* \neg \boldsymbol{true}$
- $\sigma \vDash_* p$ *pour* $p \in P$ *ssi* $|\sigma| \geq 1 \wedge p \in \sigma(1)$
- $\sigma \vDash_* \neg p$ *pour* $p \in P$ *ssi* $|\sigma| \geq 1 \wedge p \notin \sigma(1)$
- $\sigma \vDash_* \varphi_1 \vee \varphi_2$ *ssi* $\sigma \vDash_* \varphi_1 \vee \sigma \vDash_* \varphi_2$
- $\sigma \vDash_* \neg(\varphi_1 \vee \varphi_2)$ *ssi* $\sigma \vDash_* \neg\varphi_1 \wedge \sigma \vDash_* \neg\varphi_2$
- $\sigma \vDash_* \circ\varphi$ *ssi* $|\sigma| \geq 1 \wedge \sigma^1 \vDash_* \varphi$
- $\sigma \vDash_* \neg(\circ\varphi)$ *ssi* $\sigma \vDash_* \circ\neg\varphi$
- $\sigma \vDash_* \varphi_1 \mathbb{U} \varphi_2$ *ssi* $\exists i : 0 \leq i < |\sigma| : \sigma^i \vDash_* \varphi_2 \wedge \forall j : 0 \leq j < i : \sigma^j \vDash_* \varphi_1$
- $\sigma \vDash_* \neg(\varphi_1 \mathbb{U} \varphi_2)$ *ssi* $\forall i : 0 \leq i < |\sigma| : \sigma^i \vDash_* \neg\varphi_2 \vee \exists j : 0 \leq j < i : \sigma^j \vDash_* \neg\varphi_1$
- $\sigma \vDash_* \neg(\neg\varphi)$ *ssi* $\sigma \vDash_* \varphi$

Une séquence finie σ satisfait partiellement une expression LTL φ si et seulement s'il existe une séquence finie σ' telle que $\sigma\sigma'$ satisfait l'expression φ. Si une séquence σ ne satisfait pas partiellement une expression φ, on dit qu'elle viole l'expression φ.

Les opérateurs dérivés sont définis de manière semblable dans le cadre de la relation $\vDash_*$ et $\vDash_\omega$:

$$
\begin{aligned}
\textbf{false} &\stackrel{def}{=} \neg\textbf{true} \\
\varphi_1 \wedge \varphi_2 &\stackrel{def}{=} \neg(\neg\varphi_1 \vee \neg\varphi_2) \\
\varphi_1 \Rightarrow \varphi_2 &\stackrel{def}{=} \neg\varphi_1 \vee \varphi_2 \\
\varphi_1 \ \mathbb{R}\ \varphi_2 &\stackrel{def}{=} \neg(\neg\varphi_1 \ \mathbb{U}\ \neg\varphi_2) \\
\diamond\varphi &\stackrel{def}{=} \textbf{true}\ \mathbb{U}\ \varphi \\
\Box\varphi &\stackrel{def}{=} \textbf{false}\ \mathbb{R}\ \varphi
\end{aligned}
$$

On peut facilement vérifier que :

$$
\begin{aligned}
\sigma &\not\models_* \textbf{false} \\
\sigma &\models_* \varphi_1 \wedge \varphi_2 \text{ ssi } \sigma \models_* \varphi_1 \wedge \sigma \models_* \varphi_2 \\
\sigma &\models_* \diamond\varphi \text{ ssi } \exists i : 0 \leq i < |\sigma| : \sigma^i \models_* \varphi \\
\sigma &\models_* \Box\varphi \text{ ssi } \forall i : 0 \leq i < |\sigma| : \sigma^i \models_* \varphi \\
\sigma &\models_* \varphi_1 \ \mathbb{R}\ \varphi_2 \text{ ssi } \forall i : 0 \leq i < |\sigma| : \sigma^i \models_* \varphi_2 \vee \exists j : 0 \leq j < i : \sigma^j \models_* \varphi_1
\end{aligned}
$$

Les équivalences suivantes sont toujours vérifiées :
- $\neg(\varphi_1 \ \mathbb{U}\ \varphi_2) \equiv \neg\varphi_1 \ \mathbb{R}\ \neg\varphi_2$
- $\neg(\varphi_1 \ \mathbb{R}\ \varphi_2) \equiv \neg\varphi_1 \ \mathbb{U}\ \neg\varphi_2$
- $\neg(\circ\varphi) \equiv \circ\neg\varphi$
- $\neg(\diamond\varphi) \equiv \Box\neg\varphi$
- $\neg(\Box\varphi) \equiv \diamond\neg\varphi$

Notons que la définition de LTL par rapport à des séquences finies fait l'objet d'un débat très animé, en raison notamment de la difficulté à trouver une sémantique adéquate pour l'opérateur $\circ$, et de nombreuses approches différentes ont été explorées : [44, 45, 46, 47, 48, 42, 43]. La définition de la relation $\models_*$ que nous présentons ci-dessus, si elle n'est pas la plus élégante[1], est le fruit d'une longue réflexion personnelle pour donner l'expressivité la plus adéquate possible à LTL dans le cas des séquences finies tout en maintenant les équivalences ci-dessus, ce qui nous permettra d'adapter très simplement l'algorithme en 3 phases pour convertir les expressions LTL en automates classiques sans changer la structure des noeuds utilisés.

On donne également la définition du langage relatif à une expression LTL pour la relation $\models_*$, qui correspond à l'ensemble des séquences finies qui satisfont l'expression considérée, :

Définition 2.4.2 *Le langage d'une expression $\varphi \in \mathcal{LTL}(P)$ pour la relation $\models_*$ est défini par $\mathcal{L}_{\models_*}(\varphi) = \{\sigma \in \Sigma^*(2^P) \mid \sigma \models_* \varphi\}$*

LTL et les automates classiques

De manière similaire au lien qui existe entre les automates de Büchi et la logique temporelle linéaire définie sur des séquences infinies, on peut associer les

[1] Nous invitons le lecteur intéressé à se référer à [48]

automates classiques à la logique temporelle linéaire définie sur des séquences finies.

Formellement, étant donné un ensemble de propositions P, on peut évaluer la validité d'une séquence $\sigma \in \Sigma^*(2^P)$ par rapport à une expression LTL $\varphi \in \mathcal{LTL}(P)$ en construisant un automate classique $\mathcal{AC} = (\mathcal{A}, Q, \delta, Q^0, F)$ dont l'alphabet $\mathcal{A} = 2^P$ et tel que le langage qu'il accepte corresponde au langage de l'expression LTL φ pour la relation $\vDash_*$, soit $\mathcal{L}_{\vDash_*}(\varphi) = \mathcal{L}(\mathcal{AC})$.

Pour ce faire, on peut adapter l'algorithme en 3 phases décrit dans la section 2.3. Seule la phase 3 de cet algorithme doit être modifiée. Pour un ensemble de propositions P, la liste $Nodes$ de noeuds produits pour l'expression $\varphi \in \mathcal{LTL}(P)$ est utilisée pour construire un automate classique[1] $\mathcal{AC} = (\mathcal{A}, Q, \delta, Q^0, F)$:

- L'alphabet $\mathcal{A}$ est égal à l'ensemble 2^P
- L'ensemble des états Q est constitué des noeuds de $Nodes$
- La relation de transition δ est constitué comme ceci : $(r, \mu, r') \in \delta$ si et seulement si $r \in Incoming(r')$ et μ satisfait la conjonction des propositions (sous formes positives et négatives) de $Old(r')$
- L'ensemble des états initiaux Q^0 est constitué du noeud dont l'identifiant est *init*
- L'ensemble des états finaux F est constitué de tous les noeuds $q \in Nodes$ tels que $\forall \mu \in Next(q) : \epsilon \vDash_* \mu$ (la séquence vide satisfait μ).

Le seul aspect de l'algorithme qui est modifié est donc au niveau des conditions d'acceptation d'une séquence. Comme nous l'avons vu, le champ $Next$ d'un noeud représente un ensemble d'expressions LTL qui doivent toutes être satisfaites par le reste de la séquence encore à traiter. Pour qu'un état soit final, il faut donc qu'une séquence vide satisfasse, selon la relation $\vDash_*$, chacune de ces expressions LTL.

On peut facilement vérifier, par la relation $\vDash_*$, que l'ensemble des expressions LTL en forme normale négative qui sont satisfaites par la séquence vide pour un ensemble de propositions P est défini comme ceci :

- **true** satisfait la séquence vide
- $\forall \nu, \psi \in \mathcal{LTL}(P) : \nu \ \mathbb{R} \ \psi$ satisfait la séquence vide
- $\forall \nu, \psi \in \mathcal{LTL}(P) : \nu \vee \psi$ satisfait la séquence vide si et seulement si ν ou ψ satisfait la séquence vide
- $\forall \nu, \psi \in \mathcal{LTL}(P) : \nu \wedge \psi$ satisfait la séquence vide si et seulement si ν et ψ satisfait la séquence vide

La preuve formelle de cette extension de l'algorithme en 3 phases, est introduite dans [41]. Il faut noter que la définition de la relation $\vDash_*$ est légèrement différente, en ce sens que celle-ci impose une longueur de séquence minimale pour satisfaire certaines expressions, mais la preuve reste similaire. Seule la sélection des états finaux est légèrement différente : il y a ici moins d'expressions LTL qui satisfont la séquence vide (selon [41], seules les expressions de type $\mathbb{U}$ ne satisfont pas la séquence vide.)

[1] au lieu d'un automate de Büchi généralisé

La figure 2.15 montre l'automate sur séquence finie qui est déduit à partir de la liste de noeuds renvoyée par la fonction $create\_graph(\mathbf{true}\ \mathbb{U}\ \neg p)$.

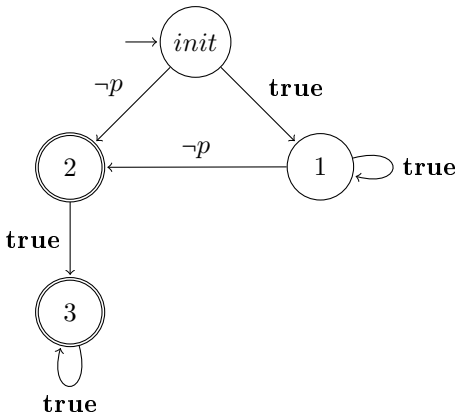


FIG. 2.15 – L'automate classique déduit de la liste de noeuds renvoyée par la fonction $create\_graph(\mathbf{true}\ \mathbb{U}\ \neg p)$

Notons qu'il est possible que les automates générés contiennent des états non finaux qui n'offrent aucune possibilité d'atteindre (via la relation de transition) un état final. Ces états sont appelés états de type rouge. Les états non finaux qui offrent la possibilité d'atteindre un état final sont appelés états de type orange et les états finaux états de type vert. Les états de type rouge ainsi que l'ensemble des transitions faisant intervenir ces états peuvent être supprimés de l'automate sans que le langage accepté par celui-c ne soit modifié. Pour l'illustrer, on constate que l'automate généré (figure 2.16) pour l'expression LTL qui impose l'existence d'une et une seule configuration contenant p :

$$\diamond p \wedge \Box(p \Rightarrow \circ(\Box\neg p))$$

accepte le même langage que l'automate obtenu en supprimant l'état 2 (figure 2.17). Par la suite, on considérera toujours des automates dépourvus des états de type rouge.

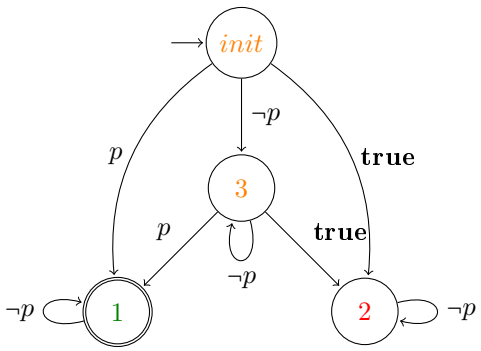


FIG. 2.16 – Un automate sur séquence finie pour l'expression $\diamond p \wedge \Box(p \Rightarrow \circ(\Box\neg p))$

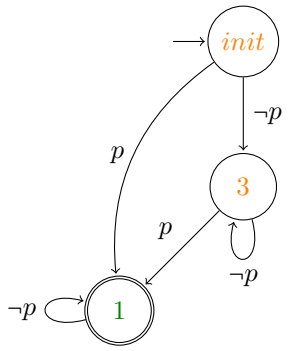
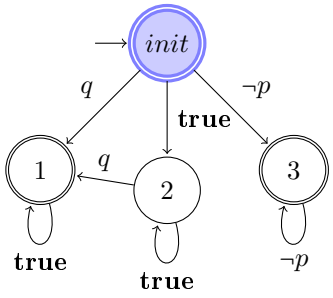


FIG. 2.17 – Un automate sur séquence finie pour l'expression $\diamond p \wedge \Box(p \Rightarrow \circ(\Box \neg p))$

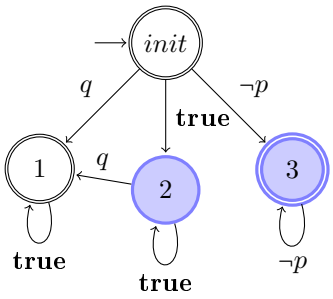
Illustration de l'utilisation des automates

Illustrons intuitivement, étant donné l'ensemble de propositions $P = \{p, q\}$, comment l'on peut utiliser les automates pour vérifier qu'une séquence, en l'occurrence $\langle\{\}, \{p\}, \{p, q\}\rangle \in \Sigma^*(P)$, satisfait l'expression LTL $(\diamond p) \Rightarrow (\diamond q) \in \mathcal{LTL}(P)$ (qui impose l'existence d'une configuration contenant q si une configuration contenant p existe).

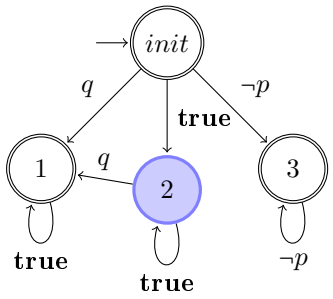
L'automate correspondant à cette expression est présenté ci-dessous. Les états du statut courant de l'automate seront représentés en bleu. Initialement, le statut est constitué d'un seul état, l'état initial. Notons que cet état est final, ce qui signifie que la séquence vide satisfait l'expression considérée.



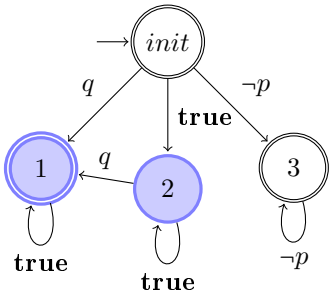
Le premier élément de la séquence est $\{\}$, qui signifie que les propositions p et q sont fausses. Le statut de l'automate pour la séquence $\langle\{\}\rangle$ est constitué des états 2 et 3 car cette première configuration satisfait les expressions **true** et $\neg p$. Notons que la séquence $\langle\{\}\rangle$ satisfait l'expression φ, ce qui se manifeste par l'existence d'un état final dans le statut de l'automate.



Le second élément de la séquence est $\{p\}$. Cette configuration respecte bien sûr l'expression **true** mais pas $\neg p$ ni q. Le statut de l'automate pour la séquence $\langle\{\},\{p\}\rangle$ est donc constitué du seul état 2. Notons que la séquence $\langle\{\},\{p\}\rangle$ ne satisfait que partiellement l'expression φ, ce qui se manifeste par le fait qu'il n'existe pas d'état final dans le statut de l'automate.



Le dernier élément de la séquence est $\{p,q\}$. Cette configuration respecte l'expression **true** et q. Le statut de l'automate pour la séquence $\langle\{\},\{p\},\{p,q\}\rangle$ est donc constitué des états 1 et 2. Comme il existe un état final dans le statut de l'automate, cela signifie que la séquence $\langle\{\},\{p\},\{p,q\}\rangle$ satisfait l'expression φ.



Chapitre 3

Les *workflow Saturn*

Dans ce chapitre, nous allons présenter un formalisme, appelé *Saturn*, qui va permettre de spécifier des *workflow* de manière déclarative en utilisant des contraintes (c.f. chapitre 1). La logique temporelle linéaire (LTL) servira de base pour l'expression de ces contraintes. Cependant, comme les expressions LTL sont assez complexes et difficiles à créer pour des non-experts, nous allons utiliser des modèles de contraintes qui permettent d'exprimer plus facilement des contraintes typiquement utilisées dans le cadre des *workflow*. Ces modèles constitueront le langage *Saturn* qui sera présenté dans la section 3.1.. Un *workflow Saturn* est un processus métier modélisé à l'aide de ce langage. Nous verrons formellement, tout au long de la section 3.2, comment on peut interpréter ces *workflow* en fonction de différents types de processus métiers de complexité croissante. Cette section est particulièrement importante car elle définit pour la première fois comment un langage de spécification de *workflow* déclaratif peut être utilisé de manière adéquate dans le cadre de processus constitués de tâches asynchrones, de processus sensibles à un environnement et de processus hiérarchiques.

3.1 Présentation du langage déclaratif *Saturn*

Le langage *Saturn*, à l'instar du langage *ConDec* [12] dont il est fortement inspiré, est un langage déclaratif qui peut être utilisé pour spécifier des *workflow* représentant une grande variété de processus métiers, des plus "stricts", qui fixent en détail l'ordre de réalisation des différentes tâches, aux plus "relâchés", qui indiquent simplement quelles tâches peuvent être réalisées au cours du processus métier sans spécifier comment elles doivent s'organiser entre elles.

Un *workflow* exprimé dans le langage *Saturn*, appelé *workflow Saturn*, consiste en un ensemble de tâches (les tâches du processus métier que le *workflow* modélise), un ensemble de conditions (représentant l'environnement dans lequel s'exécute le processus métier) et un ensemble de contraintes *Saturn* relatives à ces tâches et à ces conditions. Les *workflow Saturn* et la manière dont on peut les interpréter seront introduits formellement dans la section 3.2. Notons qu'on peut voir le langage *Saturn* comme l'extension du langage ConDec qui permet de prendre en compte l'environnement des processus métiers, un *workflow* ConDec étant constitué uniquement de tâches et de contraintes sur ces tâches [12].

Le *workflow Saturn* le plus simple n'est constitué d'aucune contrainte *Saturn*. Dans ce cas, les utilisateurs du *workflow* peuvent réaliser les tâches qu'ils désirent, autant de fois qu'ils le souhaitent et dans n'importe quel ordre. Cependant, les processus métiers doivent souvent s'exécuter conformément à certaines règles. Par exemple, dans un processus de vente, on peut imaginer les règles suivantes : l'envoi de la facture doit se faire une et une seule fois, l'envoi de la facture doit se faire après la réception de la commande et l'envoi de la facture ne peut se faire que si la poste est ouverte (condition). Ces règles peuvent être prises en compte par le *workflow Saturn* en spécifiant notamment des exigences concernant la cardinalité des tâches et les relations entre les tâches.

La notion de relation entre tâches des *workflow Saturn* est très différente de celle des *workflow* traditionnels. En effet, ces relations, dans le cadre des *workflow* impératifs, décrivent l'ordre de réalisation des tâches, en spécifiant comment les tâches s'enchaînent. Les relations entre tâches d'un *workflow Saturn* sont vues comme des contraintes. Une contrainte représente une politique, une règle propre au processus métier. A tout moment durant l'exécution du processus, chaque contrainte a une valeur vraie (la contrainte est vérifiée par l'exécution en cours) ou fausse (la contrainte n'est pas vérifiée par l'exécution en cours). Cette valeur peut changer pendant l'exécution. Notons qu'une contrainte peut être fausse à un moment donné sans que ce soit considéré comme une erreur. Imaginons par exemple une contrainte qui spécifie que chaque réalisation de la tâche A doit être suivie par une réalisation de la tâche B. Au début, la contrainte est évaluée à vraie. Lorsque A est réalisée, la contrainte devient fausse, puis lorsque B est à son tour réalisée, la contrainte redevient vraie. Le but ultime d'un *workflow Saturn* est qu'à la fin de l'exécution du processus métier qu'il modélise, toutes ses contraintes soient satisfaites. [12]

A l'instar du langage ConDec [12], on utilise la logique temporelle linéaire (LTL), définie dans le chapitre 2, pour décrire les contraintes *Saturn*.

Définition 3.1.1 (contrainte *Saturn*) *Une contrainte Saturn définie sur un ensemble de tâches T et un ensemble de conditions $Cond$ est défini par une expression* LTL $\varphi \in \mathcal{LTL}(T \cup Cond)$ *relatives à* $T \cup Cond$.

Par exemple, $\Box(\neg c \Rightarrow \neg A) \wedge \Diamond A \Rightarrow (\neg B \,\mathbb{U}\, A)$ est une contrainte *Saturn* définie sur un ensemble de tâches $\{A, B\}$ et un ensemble de conditions $\{c\}$.

Soit $SatConstr = \{c_1, c_2, \ldots, c_n\}$ un ensemble de contraintes *Saturn* définies sur T et $Cond$, $\bigwedge SatConstr$ désigne l'expression LTL $c_1 \wedge c_2 \wedge \ldots \wedge c_n \in \mathcal{LTL}(T \cup Cond)$. Une contrainte *Saturn* définie sur un ensemble de tâches T est une contrainte *Saturn* définie sur un ensemble de tâches T et un ensemble de conditions $\varnothing$.

Les modèles de contraintes *Saturn* permettent d'exprimer facilement des contraintes typiquement utilisées dans le cadre des *workflow*. Ces modèles constituent le langage *Saturn*. Chaque modèle de contraintes *Saturn* est associé à une expression LTL et possède un nom ainsi qu'une représentation graphique. Prenons par exemple le modèle de contraintes *Saturn response* qui prend en argument deux tâches et dont la représentation graphique est donnée à la figure 3.1. La sémantique de cette contrainte est donnée par son expression LTL $response(A, B) \equiv \Box(A \Rightarrow \Diamond B)$ qui signifie que toute exécution de A est suivie par (au moins) une exécution de B. $response(A, B)$ représente donc une contrainte *Saturn* définie sur un ensemble de tâches $\{A, B\}$. Les modèles de contraintes permettent d'exprimer différentes exigences et dépendances relatives aux tâches et aux conditions à un niveau d'abstraction plus élevé et de représenter celles-ci graphiquement, ce qui facilite la compréhension et la spécification des *workflow Saturn*. De plus, une fois défini, un modèle de contraintes peut être utilisé pour spécifier de nouvelles contraintes ou de nouveaux modèles de contraintes puisqu'il est directement traduisible en une expression LTL. Il est donc facile d'ajouter et de modifier des modèles de contraintes, ce qui fait que le langage *Saturn* peut évoluer et être étendu en fonction du domaine et du contexte dans lequel il est utilisé.

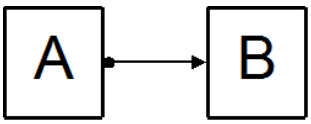


FIG. 3.1 – Représentation graphique du modèle de contraintes $response(A, B)$

Avant de présenter une série de modèles de contraintes *Saturn*, nous donnons un petit exemple illustrant l'intérêt de l'utilisation de ces modèles. La figure 3.2 illustre un *workflow Saturn* [1] constitué de 3 tâches : A, B et C. Une tâche est représentée par un rectangle comprenant le nom de la tâche. Les tâches A et B sont annotées chacune avec une contrainte d'existence qui spécifie le nombre de fois que la tâche doit être réalisée. Notons que puisqu'une tâche, en toute généralité, peut être réalisée plusieurs fois, on fera référence à une réalisation

[1] ou ConDec [12] vu que la représentation graphique est la même pour les modèles présents dans cet exemple

particulière d'une tâche en la désignant comme une instance de la tâche en question. La contrainte qui spécifie que la tâche A doit être réalisée au plus une fois (0..1) est appelée $absence_2(A)$. La tâche B doit être réalisée exactement deux fois (2), ce qui est exprimé par la contrainte $exactly_2(B)$. S'il n'y a pas de contrainte d'existence sur la tâche, cela signifie qu'elle peut être réalisée un nombre arbitraire de fois (0..*), comme c'est le cas de la tâche C. L'arc entre A et B correspond à la contrainte $response(A, B) \equiv \Box(A \Rightarrow \diamond B)$. Cette contrainte est satisfaite si, à chaque fois que la tâche A est réalisée, la tâche B est réalisée par après. Notons que cette contrainte n'exige pas que B soit réalisée "juste après" A : la réalisation d'autres tâches peut survenir entre la réalisation de A et de B. L'arc entre C et A dénote la contrainte $coexistence(A, C) \equiv \diamond A \Leftrightarrow \diamond C$. Cette contrainte exige que si C est réalisée au cours du processus, A est également réalisée, et vice-versa.

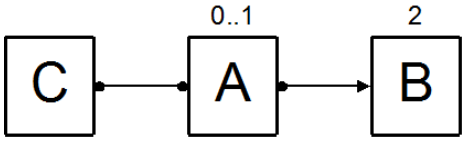


FIG. 3.2 – Un exemple de *workflow Saturn*

Notons que la construction d'un *workflow* avec un langage impératif autorisant tous les comportements modélisés par le *workflow Saturn* représenté à la figure 3.2 serait très difficile pour les raisons que nous avons introduites au cours du chapitre 1 et que nous avons illustrées (figure 1.2) pour un exemple similaire. Remarquons enfin qu'un *workflow Saturn* peut être développé comme un *workflow* impératif en utilisant les bonnes contraintes. Il est en effet possible d'exprimer en LTL les successions directes entre tâches. [12]

Pour terminer, nous allons présenter quelques modèles de contraintes utiles dans le contexte des *workflow* et nous verrons comment l'on peut représenter des *workflow Saturn* hiérarchiques. On distingue actuellement trois groupes de modèles de contraintes : les modèles de type existence, relation et conditionnel. Notons que les deux premiers groupes de modèles sont directement repris de [12] et ne prennent que des tâches en considération. Les modèles de type existence portent sur le nombre d'occurrences d'une tâche au cours du processus métier. Les modèles de type relation portent sur les dépendances entre différentes tâches. Le troisième type, propre au langage *Saturn*, porte sur les dépendances entre les tâches et les conditions. Notons que nous nous sommes limités, dans le cadre de ce mémoire, à un nombre restreint de modèles de contraintes assez simples. Il va de soi qu'il existe potentiellement d'autres modèles de contraintes qui peuvent être utiles dans le cadre de la modélisation de *workflow*. Les articles [13, 11] présentent d'ailleurs d'autres modèles de contraintes pour les deux premiers groupes.

Les modèles de contraintes de type existence

Le tableau 3.1 présente une liste de modèles de contraintes de type existence portant sur une tâche quelconque A. La première colonne donne le nom des

modèles de contraintes avec les arguments entre parenthèses, la seconde colonne exprime la sémantique de la contrainte en fournissant l'expression LTL qui la définit et la troisième colonne montre comment on peut représenter graphiquement une contrainte de ce modèle.

Nom	Expression LTL	Représentation
$existence(A)$	$\diamond A$	
$existence_2(A)$	$\diamond(A \wedge \circ(\diamond A))$	
$absence(A)$	$\Box\neg A$	
$absence_2(A)$	$\neg existence_2(A)$	
$exactly(A)$	$\diamond A \wedge \Box(A \Rightarrow \circ(\Box\neg A))$	
$exactly_2(A)$	$\neg A \,\mathbb{U}\, (A \wedge \circ(exactly(A)))$	

TAB. 3.1 – Les modèles de contraintes de type existence

Nous allons commenter intuitivement les significations des différents modèles en considérant un ensemble de tâches $\{A, B, C, D\}$ et nous utiliserons des séquences de cet ensemble pour représenter l'ordre dans lequel les tâches sont réalisées. La contrainte $existence(A)$ impose que la tâche A soit réalisée au moins une fois. Par exemple, $\langle B, C, A, D\rangle$ vérifie cette contrainte. La contrainte $existence_2(A)$ impose que la tâche A soit réalisée au moins deux fois ($\langle A, B, C, A, D, A\rangle$ vérifie cette contrainte). La contrainte $absence(A)$ interdit la réalisation de la tâche A ($\langle B, C, D\rangle$ vérifie cette contrainte). La contrainte $absence_2(A)$ impose que la tâche A soit réalisée au plus une fois ($\langle B, C, D\rangle$ et $\langle B, C, A, D\rangle$ vérifie cette contrainte). La contrainte $exactly(A)$ impose que la tâche A soit réalisée exactement une fois ($\langle B, A, C, D\rangle$ vérifie cette contrainte). La contrainte $exactly_2(A)$ impose que la tâche A soit réalisée exactement deux fois ($\langle B, A, C, A, D\rangle$ vérifie cette contrainte).

Nous avons présenté ces modèles de contraintes en considérant qu'ils prenaient une tâche en argument. Cependant, ils peuvent être facilement étendus pour pouvoir prendre plus d'une tâche en argument. On utilisera notamment la disjonction de tâches comme argument de ces modèles. Par exemple, $exactly(A \vee B) \equiv \diamond(A \vee B) \wedge \Box((A \vee B) \Rightarrow \circ(\Box\neg(A \vee B)))$ impose qu'une seule des tâches parmi A et B soit réalisée et ce, pas plus d'une seule fois. On représentera graphiquement ce type de contraintes comme l'illustre la figure 3.3

Les modèles de contraintes de type relation

Le tableau 3.2 montre un ensemble de modèles de contraintes de type relation portant sur deux tâches quelconques A et B. Alors que les contraintes de type existence décrivent la cardinalité d'une tâche, les contraintes de type

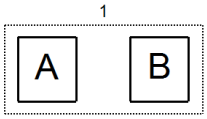


FIG. 3.3 – $exactly(A \vee B)$

relation décrivent les dépendances entre deux tâches. Elles acceptent donc deux arguments en paramètres. La dépendance entre deux tâches est représentée graphiquement par un lien entre ces deux tâches. Ce lien est particulier pour chaque modèle de contraintes et reflète la sémantique de ce modèle. Remarquons que la (simple ou double) flèche représente une relation d'ordre tandis que le point est associé à la tâche à laquelle l'exigence est liée (intuitivement, c'est "lorsque" cette tâche est effectuée, ou "si" cette tâche est effectuée, qu'une exigence est imposée). Le lien barré indique intuitivement la négation d'une autre contrainte.

Intuitivement, en considérant l'ensemble de tâches $\{A, B, C, D\}$, la contrainte $existence\_resp(A, B)$ impose que si la tâche A est réalisée au moins une fois au cours du processus, la tâche B soit également réalisée au moins une fois. Que la tâche B soit réalisée avant et/ou après la tâche A n'a pas d'importance pour cette contrainte. Par exemple, $\langle B, C, A, D, A\rangle$, $\langle B, C\rangle$ et $\langle C, C\rangle$ vérifient $existence\_resp(A, B)$. La contrainte $coexistence(A, B)$ impose que si une des tâche A ou B est réalisée, l'autre doit l'être également.

Contrairement aux modèles présentés jusqu'ici, les modèles de contraintes *response*, *precedence* et *succession* sont sensibles à l'ordre dans lequel les tâches sont réalisées. La contrainte $response(A, B)$ impose qu'à chaque fois que A est réalisée, B doit être réalisée par après. Remarquons que c'est une contrainte de réponse très flexible car B ne doit pas nécessairement être réalisée juste après A et d'autres instances de A peuvent apparaître avant que B ne soit réalisées. Par exemple, $\langle B, C, A, D, A, D, B\rangle$ vérifie cette contrainte. La contrainte $precedence(A, B)$ impose que toute réalisation de la tâche B soit précédée d'(au moins) une réalisation de la tâche A. Autrement dit, B ne peut pas être réalisée tant que A n'a pas été réalisée. Selon cette contrainte, $\langle A, C, B, B, A\rangle$ est correct. La combinaison de $response(A, B)$ et $precedence(A, B)$ forme la contrainte $succession(A, B)$. Elle spécifie que chaque réalisation de la tâche A doit être suivie d'une réalisation de la tâche B et que chaque réalisation de la tâche B doit être précédé d'une réalisation de la tâche A. Par exemple, $\langle A, C, A, B, D\rangle$ vérifie cette contrainte.

Intuitivement, on peut voir les modèles de contraintes *alternate_response*, *alternate_precedence* et *alternate_succession* comme des versions respectivement plus rigides de *response*, *precedence* et *succession*. Ainsi, la contrainte $alternate\_response(A, B)$ impose qu'à chaque fois que A est réalisée, B soit réalisée par après et qu'entre-temps (tant que B n'a pas été réalisée), A ne soit plus réalisée. Par exemple, $\langle B, A, C, B, B\rangle$ vérifie cette contrainte au contraire de $\langle B, A, A, B\rangle$. De façon similaire, la contrainte $alternate\_precedence(A, B)$ impose que chaque instance de B soit précédée d'une instance de A et qu'entre deux instances de B, il y ait au moins une instance de A. Par exemple, $\langle A, C, B, A, A, B\rangle$ vérifie cette contrainte au contraire de $\langle A, C, C, B, C, B, C, D\rangle$. La contrainte

Nom	Expression LTL	Représentation
existence_resp(A, B)	$\diamond A \Rightarrow \diamond B$	
coexistence(A, B)	$\diamond A \Leftrightarrow \diamond B$	
response(A, B)	$\Box(A \Rightarrow \diamond B)$	
precedence(A, B)	$\diamond A \Rightarrow (\neg B \mathbb{U} A)$	
succession(A, B)	*response*$(A, B) \wedge$ *precedence*(A, B)	
alternate_response(A, B)	*response*$(A, B) \wedge$ *always_between*(B, A)	
alternate_precedence(A, B)	*precedence*$(A, B) \wedge$ *always_between*(A, B)	
alternate_succession(A, B)	*alternate_response*$(A, B) \wedge$ *alternate_precedence*(A, B)	
not_coexistence(A, B)	$\diamond A \Rightarrow \Box\neg B$	
not_succession(A, B)	$\Box(A \Rightarrow \Box\neg B)$	
always_between(A, B)	$\Box(B \Rightarrow \circ($*precedence*$(A, B)))$	

TAB. 3.2 – Les modèles de contraintes de type relation

$alternate\_succession(A, B)$ est une combinaison de $alternate\_response(A, B)$ et $alternate\_precedence(A, B)$. Cette contrainte serait vérifiée, notamment, par la séquence $\langle A, C, B, A, B\rangle$.

Les modèles de contraintes $not\_coexistence$ et $not\_succession$ peuvent être vues comme les négations respectives des contraintes *coexistence* et *succession*. La contrainte $not\_coexistence(A, B)$ impose que la tâche B ne soit pas réalisée si la tâche A est réalisée au cours du processus. Notons que logiquement, cela implique que la tâche A ne soit pas réalisée si la tâche B est réalisée, Les séquences $\langle A, C, C, A, D\rangle$, $\langle B, C, C, B, D\rangle$ et $\langle C, C, D\rangle$ vérifient cette contrainte au contraire de $\langle A, C, C, B, D\rangle$. La contrainte $not\_succession(A, B)$ impose qu'aucune tâche B ne soit réalisée après que la tâche A ait été réalisée. Notons que cela impose logiquement qu'aucune tâche A ne soit réalisée avant que la tâche B ait été réalisée. Les séquences $\langle B, B, C, A, C, C, A, D\rangle$, $\langle B, C, C, B, D\rangle$ et $\langle C, C, D, A\rangle$ vérifient cette contrainte au contraire de $\langle A, C, C, B, D\rangle$.

Dans une démarche similaire à celle appliquée pour les modèles de type existence, on peut étendre ces modèles afin qu'ils puissent prendre en argument une disjonction de tâches. Par exemple, $response(A, B \vee C) \equiv \Box(A \Rightarrow \Diamond(B \vee C))$ impose que lorsqu'une tâche A est réalisée, au moins l'une des tâches B ou C soit réalisée par après. On représentera ce type de contraintes de la manière illustrée à la figure 3.4.

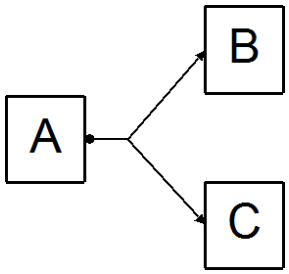


FIG. 3.4 – $response(A, B \vee C)$

Les modèles de contraintes de type conditionnel

Les modèles introduits précédemment ont été très fortement inspirés du langage ConDec. Le tableau 3.3 présente un dernier modèle de contraintes particulièrement utile et propre à *Saturn* qui établit une dépendance entre une tâche et une condition (une valeur booléenne quelconque appartenant à l'environnement du processus métier). La contrainte $condition\_to\_exist(A, c)$ où A est une tâche quelconque et c une condition, requiert que la condition c soit vraie au moment où la tâche A est réalisée. Autrement dit, la tâche A ne peut être réalisée si, au même moment, la condition c n'est pas satisfaite. On verra au cours de la section suivante quel peut être l'apport des contraintes faisant intervenir des conditions.

Notons qu'on peut également étendre ce modèle afin qu'il prenne en argument des formules de logique propositionnelle définies sur des tâches (premier argument) et des conditions (second argument).

Nom	Expression LTL	Représentation
$condition\_to\_exist(A, c)$	$\Box(\neg c \Rightarrow \neg A)$	

TAB. 3.3 – Contraintes conditionnelles

Représentation des *workflow* *Saturn* hiérarchiques

Nous allons présenter dans la section suivante comment on peut exprimer un processus métier, dit hiérarchique, dont certaines tâches peuvent elles-mêmes être représentées par un autre processus métier à l'aide d'une structure constituée de plusieurs *workflow* *Saturn*, formant un *workflow* *Saturn* hiérarchique. A notre connaissance, *Saturn* est le premier formalisme déclaratif à proposer la possibilité de définir des *workflow* hiérarchiques. La figure 3.5 montre comment l'on peut représenter un *workflow* hiérarchique constitué de deux tâches A et B dont la tâche A se réalise selon le *workflow* constitué de la tâche C et D.

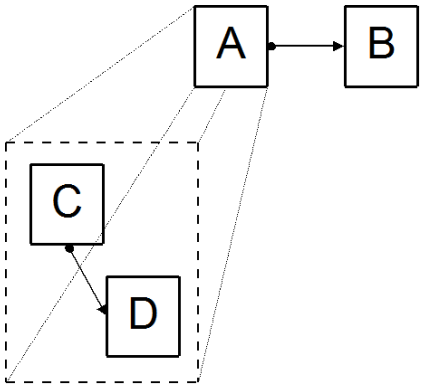


FIG. 3.5 – La représentation d'un *workflow* *Saturn* hiérarchique

3.2 Interprétation formelle des *workflow Saturn*

Dans cette section, nous allons définir formellement les concepts liés à l'interprétation des *workflow* exprimés à l'aide du langage *Saturn*, présenté dans la section précédente, en fonction de la manière dont s'exécutent les processus métiers qui sont modélisés par ces *workflow*.

Pour faciliter la compréhension, nous allons tout d'abord introduire les concepts dans le contexte le plus simple, c'est-à-dire lorsque les tâches du processus métier (PM) associé au *workflow* sont synchrones. On entend par là que, durant le déroulement du processus métier, il n'y a jamais plus d'une tâche qui est en train d'être effectuée et que chaque fois qu'une tâche est effectuée, elle est effectuée complètement et sans interruption.

Nous adapterons ensuite ces concepts pour des processus métiers au cours desquels les tâches peuvent s'exécuter en parallèle, de manière asynchrone. Pour ce faire, on distinguera le début (démarrage d'une tâche) et la fin (complétion de la tâche) de la réalisation d'une tâche. Comme la réalisation d'une tâche par un acteur du processus métier peut échouer, on introduira également la possibilité d'annuler la réalisation d'une tâche préalablement démarrée. Vu que la réalisation proprement dite d'une tâche est une action qui n'est pas sous la responsabilité du *workflow*, on ne peut pas prédire si une tâche démarrée sera complétée ou annulée ni à quel moment. On s'assurera donc que le *workflow Saturn* n'autorise le démarrage d'une tâche que s'il peut garantir que cette tâche puisse être complétée ou annulée à tout moment. Il faut noter que cette garantie constitue une des contributions majeures de ce mémoire.

Certains processus métiers peuvent s'exécuter de différentes manières selon l'évolution de leur environnement. Par exemple, l'envoi d'une information à un client peut être réalisé via l'envoi d'un courrier électronique si le serveur mail (environnement du processus métier) est disponible ou via l'envoi d'un courrier postal s'il est en panne. Nous augmenterons donc encore l'expressivité des *workflow Saturn* en y ajoutant un environnement, constitué d'un ensemble de conditions dont les différentes valeurs successives influenceront le cours de l'exécution du processus métier. Une telle gestion de l'environnement par un *workflow* de type déclaratif n'avait, à notre connaissance, jamais été abordée précédemment.

Enfin, nous introduirons les *workflow Saturn* dit "hiérarchiques" qui représentent des processus métiers dont certaines tâches sont constituées d'autres processus métiers qu'on peut représenter à leur tour par autant de *workflow Saturn* hiérarchiques, dit "sous-*workflow*". Cette possibilité de définir des *workflow* déclaratifs de manière hiérarchique est introduite et définie pour la première fois dans le cadre de ce mémoire.

3.2.1 *Workflow Saturn* et PM constitué de tâches synchrones

Nous allons définir les *workflow Saturn* dans le contexte de processus métiers très simples, constitués d'un ensemble de tâches réalisées à tour de rôle et sans annulation. Du point de vue d'un *workflow* relatif à un processus de ce type, on peut voir la réalisation de ces tâches comme étant instantanée.

Dans ce contexte, on définit l'exécution d'un ensemble de tâches comme une simple séquence de tâches (définition 3.2.1).

Définition 3.2.1 (exécution) *Une exécution d'un ensemble de tâches T est une séquence finie $\sigma \in \Sigma^*(T)$.*

Par exemple, $\langle sendbill, getorder, getorder, sendbill \rangle$ est une exécution de l'ensemble de tâches $T = \{getorder, sendbook, sendbill, bepaid, update\}$. Notons qu'une même tâche peut être effectuée plusieurs fois au cours d'une même exécution.

Dans ce contexte, les *workflow* Saturn sont constitués d'un ensemble de tâches et d'un ensemble de contraintes sur ces tâches. Notons que dans ce contexte, on ne prend en compte que les contraintes Saturn qui ne font intervenir que des tâches.

Définition 3.2.2 (*workflow* Saturn) *Un workflow Saturn w est un couple $(T, SatConstr)$ tel que :*
- *T est un ensemble de tâches*
- *$SatConstr$ est un ensemble de contraintes Saturn définies sur l'ensemble de tâches T.*

Exemple 3.2.1 *Soit le workflow $w = (T, SatConstr)$ tel que :*
- *$T = \{sellproducts, getpaid, update, supplystock\}$ et*
- *$SatConstr = \{coexistence(sellproducts, getpaid), response(sellproducts, update)\}$*

Celui-ci est représenté graphiquement à la figure 3.6.

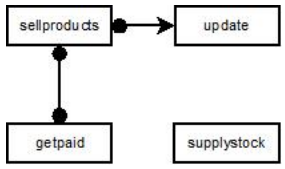


FIG. 3.6 – Représentation graphique du *workflow* w

Le *workflow* de l'exemple 3.2.1 modélise le comportement d'un processus métier particulièrement flexible correspondant à la vente de produits. Celui-ci admet les tâches suivantes :
- *sellproducts* : correspond à la vente d'un ou plusieurs produits
- *getpaid* : correspond à la réception d'un paiement
- *update* : correspond à la mise à jour des fichiers relatifs aux stocks
- *supplystock* : correspond au réapprovisionnement du stock.

Le processus métier est caractérisé par les 5 règles suivantes :
-R1- un client peut payer avant ou après avoir acheté le produit
-R2- il peut payer un produit en plusieurs fois ou peut payer plusieurs produits en même temps
-R3- si aucun produit n'est vendu, aucun paiement n'est perçu et vice versa.
-R4- le stock peut être ré-approvisionné à tout moment
-R5- après (mais pas nécessairement juste après) qu'un produit soit vendu, il faut que la mise à jour du stock soit effectuée.

Notons que les règles R1, R2 et R3 sont exprimées grâce à la contrainte *Saturn coexistence*(*sellproduct*), la règle R4 par le fait qu'il n'y ait aucune contrainte faisant intervenir la tâche *supplystock* et la règle R5 par la contrainte $response(sellproducts, update)$.

Nous allons maintenant définir comment une exécution est validée par un *workflow Saturn*. Intuitivement, une exécution est validée par un *workflow* si elle respecte ses différentes contraintes, autrement dit si la séquence associée à cette exécution satisfait la conjonction des expressions LTL qui constituent les contraintes du *workflow*, ce qui est exprimé formellement dans la définition suivante.

Définition 3.2.3 (validité d'une exécution) *Une exécution σ d'un ensemble de tâches T est validée par un workflow Saturn $w = (T, SatConstr)$ si et seulement si $\exists \sigma' \in \Sigma^*(2^T) : \forall i : \sigma'(i) = \{\sigma(i)\} \wedge \sigma' \vDash_* \bigwedge SatConstr$.*

Une exécution σ d'un ensemble de tâches T est partiellement validée par un *workflow Saturn* $w = (T, SatConstr)$ s'il existe une séquence $\sigma' \in \Sigma^*(T)$ telle que $\sigma\sigma'$ est une exécution de T validée par w. Une exécution qui n'est pas partiellement valide est dite invalide.

En guise d'illustration, voici quelques exécutions[1] validées par le *workflow* de l'exemple 3.2.1 :

$$sellproducts\ update\ getpaid \tag{3.1}$$
$$getpaid\ getpaid\ sellproducts\ update \tag{3.2}$$
$$sellproducts\ sellproducts\ getpaid\ update \tag{3.3}$$
$$sellproducts\ supplystock\ update\ supplystock\ getpaid \tag{3.4}$$
$$update\ update \tag{3.5}$$
$$supplystock \tag{3.6}$$
$$\epsilon \tag{3.7}$$

et quelques exécutions partiellement validées avec entre parenthèses la plus petite séquence dont la concaténation permet d'obtenir une exécution valide.

$$sellproducts\ getpaid\ getpaid\ (update) \tag{3.8}$$
$$getpaid\ getpaid\ (sellproducts\ update) \tag{3.9}$$
$$sellproducts\ update\ (getpaid) \tag{3.10}$$

Notons que dans l'exemple ci-dessus, toutes les exécutions sont partiellement validées par le *workflow* w. Il est important de signaler qu'il ne s'agit là que d'un cas particulier. Des exécutions peuvent bien sûr être invalidées par certains *workflow* : prenons par exemple le *workflow* très restrictif $y = (\{A, B\}, \{exactly(A), exactly(B)\})$ qui représente un processus métier constitué de deux tâches A et B qui doivent chacune être effectuée une et une seule fois. $\langle A, B\rangle$ et $\langle B, A\rangle$ sont des exécutions valides tandis que ϵ, $\langle A\rangle$ et $\langle B\rangle$ sont partiellement valides. Toutes les autres exécutions sont invalides.

[1] Lorsque cela ne risque pas de porter à confusion (comme dans le cas de cet exemple), nous noterons parfois $e_1 e_2 \ldots e_n$ des exécutions de la forme $\langle e_1, e_2, \ldots, e_n\rangle$.

3.2.2 *Workflow Saturn* et PM constitué de tâches asynchrones

Nous allons maintenant étendre ces différents concepts dans le contexte de processus métiers au cours desquels plusieurs tâches peuvent s'effectuer en même temps et peuvent échouer. Dans ce contexte, il est utile de distinguer[1] une tâche et une instance de tâche. Une instance d'une tâche représente une seule réalisation de cette tâche. Contrairement au cas des PM constitués de tâches synchrones où la réalisation d'une tâche était perçue comme instantanée, la réalisation d'une tâche se fait dans ce contexte en deux temps du point de vue du *workflow*. Elle est tout d'abord démarrée puis elle se clôture, ce qui peut se faire de deux manières distinctes : soit la réalisation de la tâche s'effectue avec succès et on dit que l'instance de la tâche se complète, soit pour une raison quelconque la réalisation de la tâche échoue (elle n'a donc pas été effectuée) et on dit que l'instance est annulée. Entre le démarrage et la clôture (complétion ou annulation) d'une instance, d'autres actions liées à d'autres instances (de la même ou d'une autre tâche) peuvent être effectuées.

Le démarrage, la complétion et l'annulation sont trois actions, notées respectivement ▶, ■ et ⊗, que nous regroupons dans un ensemble noté *Actions* qui sera utilisé tout au long de ce chapitre pour représenter l'ensemble des opérations que l'on peut effectuer sur une instance de tâche dans le cadre des *workflow Saturn*.

Définition 3.2.4 (action) *Soit un ensemble de tâches T, l'action a de t, notée t_a, est un couple (a,t) où $a \in Actions$ et $t \in T$.*

Nous allons devoir redéfinir le concept d'exécution d'un ensemble de tâches pour l'adapter à un contexte asynchrone. Pour ce faire, nous allons utiliser la définition d'une sous-séquence et celle d'une restriction d'une séquence.

> Soit une séquence finie de n éléments $\sigma = \langle e_1, e_2, \ldots, e_n \rangle$, une séquence finie de $k \leq n$ éléments $\sigma' = \langle e'_1, e'_2, \ldots, e'_k \rangle$ est une sous-séquence de σ s'il existe un ensemble d'indices $\{i_1, i_2, \ldots, i_k\}$, noté $_{\mathcal{INDEX}}(\sigma, \sigma')$, tel que $1 \leq i_1 < i_2 < \ldots < i_k \leq n$ et $\sigma(i_j) = \sigma'(j)$ pour tout $j : 1 \leq j \leq k$.
>
> Intuitivement, une sous-séquence d'une séquence donnée est constituée de la séquence de départ à laquelle on a retiré certains éléments. Par exemple, soit la séquence $\sigma = \langle c, a, a, b, d, c \rangle$, la séquence $\sigma' = \langle c, a, d \rangle$ est une sous-séquence de σ et $_{\mathcal{INDEX}}(\sigma, \sigma') = \{1, 2, 5\}$ ou $\{1, 3, 5\}$.
>
> Soit une séquence finie $\sigma \in \Sigma^*(E)$ et un ensemble $E' \subseteq E$, la restriction $\chi(\sigma, E')$ de σ pour l'ensemble E' désigne la plus longue sous-séquence de σ telle que chacun de ses éléments fait partie de l'ensemble E'.
>
> Par exemple, soit $\sigma = \langle c, a, a, b, d, c \rangle$, la restriction $\chi(\sigma, \{a, d\}) = \langle a, a, d \rangle$

[1] Par abus de langage, quand la situation ne prête pas à confusion, on pourra néanmoins confondre les deux concepts

Pour redéfinir le concept d'exécution d'un ensemble de tâches, nous donnons tout d'abord la définition de l'exécution d'une tâche.

Définition 3.2.5 (exécution d'une tâche) *Soit un ensemble de tâches T, une exécution de $t \in T$ est une séquence $\sigma \in \Sigma^*(Actions \times T)$ qui satisfait la grammaire*

$$\begin{array}{rcl} S & ::= & \epsilon \\ & | & S\ S \\ & | & t_{\blacktriangleright}\ S\ t_{\blacksquare} \\ & | & t_{\blacktriangleright}\ S\ t_{\otimes} \end{array}$$

Par exemple, $\langle A_{\blacktriangleright}, A_{\blacktriangleright}, A_{\otimes}, A_{\blacktriangleright}, A_{\blacksquare}, A_{\blacksquare}\rangle$ est une exécution de la tâche A.

L'exécution d'un ensemble de tâches est redéfini dans le cadre des PM constitués de tâches asynchrones selon la définition ci-dessous. Intuitivement, une exécution d'un ensemble de tâches correspond à un entrelacement d'exécutions de tâches de cet ensemble.

Définition 3.2.6 (exécution) *Une exécution d'un ensemble de tâches T est une séquence $\sigma \in \Sigma^*(Actions \times T)$ telle que $\forall t \in T : \chi(\sigma, \{t_{\blacktriangleright}, t_{\otimes}, t_{\blacksquare}\})$ est une exécution de t.*

Une exécution partielle d'un ensemble de tâches T est une séquence $\sigma \in \Sigma^*(Actions \times T)$ telle que $\exists \sigma' \in \Sigma^*(Actions \times T) : \sigma\sigma'$ est une exécution de T.

Par exemple, $\langle A_{\blacktriangleright}, B_{\blacktriangleright}, A_{\otimes}, C_{\blacktriangleright}, C_{\blacksquare}, B_{\blacksquare}, B_{\blacktriangleright}, B_{\blacksquare}\rangle$ est une exécution de l'ensemble de tâches $\{A, B, C\}$.

Une question cruciale est de savoir, dans ce contexte asynchrone, comment représenter le *workflow* et sous quels critères une exécution est validée par ce *workflow*. Nous allons tout d'abord ouvrir une parenthèse dans cette section pour présenter l'approche la plus évidente, dont nous avons pu constater qu'elle était appliquée dans le cadre du projet Declare ([10, 11, 12, 13]). Nous allons pointer une faiblesse dans cette approche, qui n'est selon nous pas acceptable.

Une approche évidente est d'étendre la définition 3.2.2 du *workflow Saturn* présenté précédemment de façon telle que les contraintes ne portent plus sur l'ensemble des tâches T mais sur l'ensemble $Actions \times T$. Une exécution serait donc valide si elle respectait ces contraintes. Par exemple, on pourrait redéfinir la contrainte *exactly*, qui signifie qu'une tâche doit être effectuée une et une seule fois de la manière suivante : $exactly(t) \equiv \diamond t_{\blacksquare} \wedge \Box(t_{\blacksquare} \Rightarrow \circ(\Box(\neg t_{\blacktriangleright} \wedge \neg t_{\blacksquare})))$. Intuitivement, elle signifie qu'une instance de tâche t doit être effectuée complètement et que lorsqu'elle est effectuée, on ne peut plus ni démarrer ni compléter d'instance de cette tâche.[1] A première vue, cela semble correspondre au comportement désiré. Par exemple, les exécutions $t_{\blacktriangleright}t_{\blacksquare}$ et $t_{\blacktriangleright}t_{\otimes}t_{\blacktriangleright}t_{\blacksquare}$ satisfont cette contrainte tandis que ϵ, $t_{\blacktriangleright}t_{\otimes}$ ou $t_{\blacktriangleright}t_{\blacksquare}t_{\blacktriangleright}t_{\blacksquare}$ ne la satisfont pas. Néanmoins, en y regardant de plus près, on se rend compte que l'exécution partielle $t_{\blacktriangleright}t_{\blacktriangleright}t_{\blacksquare}$ satisfait également la contrainte. Néanmoins, on ne peut rajouter $t_{\blacksquare}$ pour compléter

[1] On remarquera que cette approche implique que les expressions LTL sont plus complexes que dans le cadre des PM constitués de tâches synchrones

l'exécution sous peine de violer la contrainte. On est donc contraint d'annuler la réalisation de t en ajoutant $t_{\otimes}$ à cette exécution partielle. Autrement dit, cette approche autorise le démarrage de tâche sans garantir que cette tâche a réellement le droit d'être exécutée.

Nous pensons que cette caractéristique est inacceptable. Rappelons que l'un des buts de la modélisation de *workflow* pour représenter un processus métier est de pouvoir contrôler dynamiquement son déroulement, grâce à un moteur de *workflow*, en s'assurant que les utilisateurs du *workflow* effectuent les tâches dans un ordre conforme à ce *workflow*. Ce système peut autoriser ou non les utilisateurs de *workflow* à effectuer une tâche mais n'est en aucun cas responsable de la réalisation de cette tâche. Autrement dit, du point de vue "*workflow*", une fois que le démarrage d'une tâche a été autorisé, on n'a plus aucune emprise sur le fait que la réalisation de la tâche va échouer ou non, et on ne sait pas non plus à quel moment de l'exécution (à quel "endroit" de la séquence) celle-ci va se clôturer. Potentiellement, une tâche démarrée peut se terminer à tout moment que ce soit par un succès (complétion) ou un échec (annulation). Pour garantir que toute tâche en cours de réalisation puisse toujours être complétée et annulée, nous allons présenter une approche adéquate fondamentalement différente de l'approche présentée ci-dessus.

Pour les PM constitués de tâches asynchrones, la définition 3.2.2 du *workflow Saturn* reste la même que pour les PM constitués de tâches synchrones[1]. Nous modifions par contre la manière dont une exécution est validée par ce même *workflow*. Pour ce faire, nous allons définir quelques concepts utiles.

Intuitivement, lorsque l'on considère une exécution partielle, il subsiste potentiellement un ensemble d'instances de tâches qui ont été démarrées mais qui n'ont pas encore été conclues, que ce soit par un succès ou un échec. Intuitivement, on peut représenter ces instances à l'aide d'un ensemble pouvant contenir plusieurs fois les mêmes éléments ou multi-ensemble. Les définitions qui suivent permettent d'introduire formellement les multi-ensembles ainsi que les opérations qui vont nous être utiles par la suite.

> Un multi-ensemble est un couple (E, m) où E est un ensemble appelé support et $m : E \to \mathbb{N}$ est une fonction appelée multiplicité. La multiplicité renvoie pour chaque élément du support le nombre d'occurrences de celui-ci dans le multi-ensemble. L'ensemble de tous les multi-ensembles dont le support est E est noté $\mathcal{M}(E)$. Un multi-ensemble peut être vu comme un ensemble pouvant contenir plusieurs fois les mêmes éléments.
>
> Par exemple, le multi-ensemble contenant 2 occurrences de a et 3 de b, que nous pouvons écrire $\{a, a, b, b, b\}_{\mathcal{M}}$, est un couple $(\{a, b\}, m)$ où $m(a) = 2$ et $m(b) = 3$.
>
> Définissons quelques opérateurs pour les multi-ensembles : soit un ensemble E, un élément $e \in E$ et les multi-ensembles $M = (E, m)$ et $N = (E, n)$:

[1] Les expressions LTL traitées ne sont donc pas plus complexes que dans le cas des PM constitués de tâches synchrones

- $M \cup_{\mathcal{M}} N = (E, f)$ où $\forall x \in E : f(x) = m(x) + n(x)$
- $M \setminus_{\mathcal{M}} N = (E, f)$ où $\forall x \in E : f(x) = max(0, m(x) - n(x))$
- $M \subseteq_{\mathcal{M}} N$ signifie que $\forall x \in E : m(x) \leq n(x)$
- $M \subseteq_{\mathcal{M}} N \wedge N \subseteq_{\mathcal{M}} M \equiv M = N$
- $e \in M$ si et seulement si $m(e) > 0$

La cardinalité d'un multi-ensemble $M = (E, m)$, noté $\#M$, correspond à la somme des multiplicités : $\#M = \sum_{e \in E} m(e)$. Elle indique le nombre total d'éléments du multi-ensemble.

On peut ainsi désigner les tâches en cours de réalisation par un multi-ensemble qui contient autant de tâches t qu'il existe d'instances de cette tâche t en cours de réalisation.

Définition 3.2.7 (tâches en cours de réalisation) *Soit une exécution partielle σ d'un ensemble de tâches T, les tâches en cours de réalisation pour σ, noté $running\_tasks(\sigma)$, désigne le multi-ensemble (S, m) tel que :*

- $S = \{\{t\} \mid t \in T\}$
- $\forall t \in T : m(\{t\}) = |\chi(\sigma, \{t_{\blacktriangleright}\})| - |\chi(\sigma, \{t_{\blacksquare}, t_{\otimes}\})|$

Par exemple, $running\_tasks(A_{\blacktriangleright} A_{\blacktriangleright} A_{\otimes} B_{\blacktriangleright} A_{\blacktriangleright}) = \{\{A\}, \{B\}, \{A\}\}_{\mathcal{M}}$.

Toujours en considérant une exécution partielle, il est également utile de pouvoir faire référence à la séquence des tâches qui ont déjà été effectuées (c-à-d la sous-séquence constituée des tâches pour lesquelles l'action ■ a été effectuée).

Définition 3.2.8 (séquence effectuée) *Soit une exécution partielle σ d'un ensemble de tâches T, la séquence de tâches effectuées par σ, notée $exec\_order(\sigma)$, désigne une séquence $\sigma' \in \Sigma^*(2^T)$ telle que $\forall i : \sigma''(i) = t_{\blacksquare} \Rightarrow \sigma'(i) = \{t\}$ où $\sigma'' = \chi(\sigma, \{t'_{\blacksquare} \mid t' \in T\})$.*

Par exemple, $exec\_order(A_{\blacktriangleright} B_{\blacktriangleright} A_{\otimes} C_{\blacktriangleright} C_{\blacksquare} B_{\blacksquare} B_{\blacktriangleright} B_{\blacksquare} A_{\blacktriangleright}) = \langle C, B, B \rangle$

Intuitivement, pour garantir que la complétion et l'annulation d'une tâche en cours de réalisation sont toujours autorisées, notre approche consiste à s'assurer que, tout au long de l'exécution, les tâches en cours de réalisation peuvent se clôturer (complétion ou annulation) dans n'importe quel ordre sans violer les contraintes. Ceci peut se faire en s'assurant que chaque permutation des tâches en cours de réalisation complète la séquence de tâches déjà effectuées sans violer de contraintes. Comme les tâches en cours de réalisation sont représentées par un multi-ensemble, il convient de définir formellement la permutation d'un multi-ensemble.

Une permutation d'un multi-ensemble $M = (E, m)$ est une séquence $\sigma \in \Sigma^*(E)$ de longueur $\#M$ telle que, $\forall e \in E : m(e) = |\chi(\sigma, \{e\})|$. L'ensemble de toutes les permutations pour le multi-ensemble M est noté $Perm(M)$.

Par exemple, $Perm(\{a, a, b\}_{\mathcal{M}}) = \{\langle a, a, b \rangle, \langle a, b, a \rangle, \langle b, a, a \rangle\}$.

Le démarrage d'une tâche n'est autorisé que si elle est disponible, c'est-à-dire si, après son démarrage, aucune permutation des tâches en cours de réalisation n'est susceptible de violer une contrainte. On peut ainsi définir un ensemble de tâches disponibles pour une exécution partielle et une expression LTL représentant les contraintes.

Définition 3.2.9 (tâches disponibles) *Soit une exécution partielle σ d'un ensemble de tâches T et une expression* LTL *$\varphi \in \mathcal{LTL}(T)$, l'ensemble des tâches disponibles selon φ après σ, noté $available(\sigma, \varphi)$, est l'ensemble*

$$\{t \in T \mid \forall p \in Perm(running\_tasks(\sigma) \cup_{\mathcal{M}} \{\{t\}\}_{\mathcal{M}}) : exec\_order(\sigma)p \text{ satisfait partiellement } \varphi\}.$$

Pour l'illustrer, considérons un ensemble de tâches $T = \{A\}$ et l'expression LTL $\varphi \equiv \diamond A \wedge \Box(A \Rightarrow \circ(\Box(\neg A)))$ qui correspond à la contrainte *Saturn* $exactly(A)$,

$$available(\epsilon, \varphi) = \{A\} \quad (3.11)$$
$$available(A_{\blacktriangleright}, \varphi) = \{\} \quad (3.12)$$
$$available(A_{\blacktriangleright}A_{\otimes}, \varphi) = \{A\} \quad (3.13)$$
$$available(A_{\blacktriangleright}A_{\otimes}A_{\blacktriangleright}, \varphi) = \{\} \quad (3.14)$$
$$available(A_{\blacktriangleright}A_{\otimes}A_{\blacktriangleright}A_{\blacksquare}, \varphi) = \{\} \quad (3.15)$$

En effet, $\langle\{A\}\rangle$ satisfait l'expression φ tandis que $\langle\{A\}, \{A\}\rangle$ la viole.

On peut ainsi re-définir la validité d'une exécution.

Définition 3.2.10 (validité d'une exécution) *Une exécution σ d'un ensemble de tâches T est validée par un workflow* Saturn *$w = (T, SatConstr)$ si et seulement si $exec\_order(\sigma) \vDash_* \bigwedge SatConstr$ et*

$$\forall \sigma' : \sigma = \sigma' t_{\blacktriangleright} \sigma'' : t \in available(\sigma', \bigwedge SatConstr).$$

Une exécution partielle σ d'un ensemble de tâches T est partiellement validée[1] par un *workflow* *Saturn* $w = (T, SatConstr)$ s'il existe une séquence $\sigma' \in \Sigma^*(Actions \times T)$ telle que $\sigma\sigma'$ est une exécution de T validée par w.

Cette définition garantit que, pour toute exécution partiellement validée par un *workflow*, n'importe quelle tâche en cours de réalisation peut être conclue que ce soit en étant complétée ou annulée, comme c'est exprimé formellement par la proposition suivante, conséquence directe des définitions ci-dessus.

Proposition 3.2.1 *Soit une exécution partielle $\sigma \in \Sigma^*(Actions \times T)$ partiellement validée par un workflow* Saturn *$w = (T, SatConstr)$,*

$$\forall t \in running\_tasks(\sigma) : \sigma\langle t_{\blacksquare}\rangle \text{ et } \sigma\langle t_{\otimes}\rangle \text{ sont partiellement validées par } w.$$

Exemple 3.2.2 *Soit le workflow $w = (T, SatConstr)$ tel que :*
- *$T = \{write, add, send\}$ et*
- *$SatConstr = \{exactly(send), precedence(add, send), not\_succession(send, add), not\_succession(send, write)\}$*

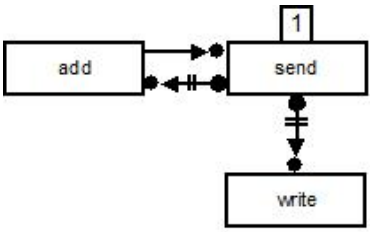


FIG. 3.7 – Représentation graphique du *workflow* w

Celui-ci est représenté graphiquement à la figure 3.7.

Le *workflow* de l'exemple 3.2.2 modélise le comportement d'un processus métier correspondant à la rédaction et l'envoi d'un e-mail :
- la tâche *write* correspond à la rédaction de l'e-mail
- la tâche *add* correspond à l'ajout d'un destinataire à l'e-mail
- la tâche *send* correspond à l'envoi de l'e-mail

La tâche *send* doit être effectuée une et une seule fois. Avant que cette tâche ne soit effectuée, au moins un destinataire doit avoir été spécifié. Il est possible d'envoyer un mail vide. Une fois l'e-mail envoyé, on ne peut logiquement plus l'écrire ni y ajouter des destinataires.

Les exécutions

$$add_{\blacktriangleright} add_{\blacksquare} send_{\blacktriangleright} send_{\blacksquare}$$

et

$$write_{\blacktriangleright} add_{\blacktriangleright} add_{\blacksquare} add_{\blacktriangleright} add_{\blacksquare} write_{\blacksquare} send_{\blacktriangleright} send_{\otimes} add_{\blacktriangleright} add_{\blacksquare} send_{\blacktriangleright} send_{\blacksquare}$$

sont validées par le *workflow* w. Par contre, les exécutions

$$add_{\blacktriangleright} add_{\blacksquare} send_{\blacktriangleright} send_{\blacktriangleright} send_{\blacksquare} send_{\otimes}$$

et

$$add_{\blacktriangleright} add_{\blacksquare} send_{\blacktriangleright} send_{\blacksquare} write_{\blacktriangleright} write_{\blacksquare}$$

sont invalidées par le *workflow* w.

3.2.3 *Workflow Saturn* et PM sensible à un environnement

Il arrive fréquemment que des processus métiers ne soient pas caractérisés uniquement par un ensemble de tâches et les relations entre celles-ci mais qu'un certain nombre d'éléments extérieurs, qu'on regroupe sous le terme d'environnement du *workflow*, influencent son déroulement. Cet environnement peut évoluer au cours de la réalisation d'un processus métier. Nous allons exprimer cet environnement en intégrant aux différents concepts présentés précédemment un ensemble de conditions. Intuitivement, puisque les contraintes *Saturn* sont susceptibles de faire intervenir ces conditions, les tâches disponibles durant l'exécution du processus métier peuvent varier suivant la configuration de l'environnement.

[1] Lorsqu'on ne se préoccupe pas de savoir si l'exécution partielle partiellement validée est une exécution, on l'appellera par facilité exécution partiellement validée

Dans ce contexte, une tâche s'effectue dans un environnement représenté par l'ensemble des conditions satisfaites au moment du démarrage de la tâche. On redéfinit donc les actions et les exécutions de tâches pour les adapter aux processus métiers sensibles à un environnement constitué de conditions.

Soit un ensemble de tâches T et un ensemble de conditions $Cond$, l'action a de t sous conditions c, notée $\overset{c}{t_a}$, est un triplet (a,t,c) où $a \in Actions$, $t \in T$ et $c \subseteq Cond$.

Définition 3.2.11 (exécution d'une tâche) *Soit un ensemble de tâches T et un ensemble de conditions $Cond$, une exécution de $t \in T$ sous conditions $c \subseteq Cond$ est une séquence $\sigma \in \Sigma^*(Actions \times T \times 2^{Cond})$ qui satisfait la grammaire*

$$\begin{array}{rcl} S & ::= & \epsilon \\ & | & S\ S \\ & | & \overset{c}{t_{\blacktriangleright}}\ S\ \overset{c}{t_{\blacksquare}} \\ & | & \overset{c}{t_{\blacktriangleright}}\ S\ \overset{c}{t_{\otimes}} \end{array}$$

Définition 3.2.12 (exécution) *Une exécution d'un ensemble de tâches T pour un ensemble de conditions $Cond$ est une séquence $\sigma \in \Sigma^*(Actions \times T \times 2^{Cond})$ telle que $\forall t \in T, c \subseteq Cond : \chi(\sigma, \{\overset{c}{t_{\blacktriangleright}}, \overset{c}{t_{\blacksquare}}, \overset{c}{t_{\otimes}}\})$ est une exécution de t sous conditions c.*

Une exécution partielle d'un ensemble de tâches T pour un ensemble de conditions $Cond$ est une séquence $\sigma \in \Sigma^*(Actions \times T \times 2^{Cond})$ telle que $\exists \sigma' \in \Sigma^*(Actions \times T \times 2^{Cond}) : \sigma\sigma'$ est une exécution de T pour $Cond$.

Par exemple, $\langle \overset{\{c_1,c_2\}}{A_{\blacktriangleright}}, \overset{\{c_1\}}{B_{\blacktriangleright}}, \overset{\{c_1,c_2\}}{A_{\blacksquare}}, \overset{\{c_1\}}{B_{\otimes}} \rangle$ est une exécution de l'ensemble de tâches $\{A,B\}$ pour un ensemble de conditions $\{c_1,c_2,c_3\}$.

La définition du worklow *Saturn* est également modifiée pour intégrer les conditions et prendre en compte des contraintes *Saturn* définies sur un ensemble de tâches et un ensemble de conditions.

Définition 3.2.13 (*workflow* Saturn) *Un workflow Saturn w est un triplet $(T, Cond, SatConstr)$ tel que :*
- *T est un ensemble de tâches*
- *$Cond$ est un ensemble de conditions*
- *$SatConstr$ est un ensemble de contraintes Saturn définies sur T et $Cond$.*

La validité d'une exécution et les définitions relatives sont finalement adaptées pour prendre en compte cet ensemble de conditions.

Définition 3.2.14 (tâches en cours de réalisation) *Soit une exécution partielle σ d'un ensemble de tâches T pour un ensemble de conditions $Cond$, les tâches en cours de réalisation pour σ, noté $running\_tasks(\sigma)$, désigne le multi-ensemble (S,m) tel que :*
- $S = \{\{t\} \cup c \mid t \in T \wedge c \subseteq Cond\}$
- $\forall t \in T, c \subseteq Cond : m(\{t\} \cup c) = |\chi(\sigma, \{t_{\blacktriangleright}\})| - |\chi(\sigma, \{t_{\blacksquare}, t_{\otimes}\})|$

Par exemple, pour un ensemble de tâches $\{A, B\}$ et un ensemble de conditions $\{c_1, c_2, c_3\}$, $running\_tasks(\langle \overset{\{c_1,c_2\}}{A_\blacktriangleright}, \overset{\{c_1\}}{B_\blacktriangleright}\rangle) = \{\{A, c_1, c_2\}, \{B, c_1\}\}_{\mathcal{M}}$

Définition 3.2.15 (séquence effectuée) *Soit une exécution partielle σ d'un ensemble de tâches T pour un ensemble de conditions $Cond$, la séquence de tâches exécutées par σ, notée $exec\_order(\sigma)$, désigne une séquence $\sigma' \in \Sigma^*(2^{T \cup Cond})$ telle que $\forall i : \sigma''(i) = \overset{c}{t_\blacksquare} \Rightarrow \sigma'(i) = \{t\} \cup c$ où $\sigma'' = \chi(\sigma, \{\overset{c'}{t'_\blacksquare} \mid t' \in T \wedge c' \subseteq Cond\})$.*

Par exemple, pour un ensemble de tâches $\{A, B\}$ et un ensemble de conditions $\{c_1, c_2, c_3\}$, $exec\_order(\langle \overset{\{c_1,c_2\}}{A_\blacktriangleright}, \overset{\{c_1\}}{B_\blacktriangleright}, \overset{\{c_1,c_2\}}{A_\blacksquare}, \overset{\{c_1\}}{B_\blacksquare}\rangle = \langle\{A, c_1, c_2\}, \{B, c_1\}\rangle$.

Définition 3.2.16 (tâches disponibles) *Soit une exécution partielle σ d'un ensemble de tâches T pour un ensemble de conditions $Cond$ et une expression* LTL *$\varphi \in \mathcal{LTL}(T \cup Cond)$, l'ensemble des tâches disponibles selon φ après σ, noté $available(\sigma, \varphi)$, est l'ensemble*

$$\{(t, c) \in (T \times 2^{Cond}) \mid \forall p \in Perm(running\_tasks(\sigma) \cup_{\mathcal{M}} \{\{t\} \cup c\}_{\mathcal{M}}) : exec\_order(\sigma)p \text{ satisfait partiellement } \varphi\}.$$

Considérons par exemple un ensemble de tâches $\{A\}$ et un ensemble de conditions $\{c_1\}$ et l'expression LTL $\varphi \equiv \Box(A \Rightarrow c_1)$ qui correspond à la contrainte *Saturn* $condition\_to\_exist(A, c_1)$, pour une exécution partielle $\sigma \in \Sigma^*(Actions \times \{A\} \times 2^{\{c_1\}})$, $available(\sigma, \varphi) = \{\{A, c_1\}\}$. Autrement dit, A est une tâche disponible si et seulement si la condition c_1 est satisfaite.

Définition 3.2.17 (validité d'une exécution) *Une exécution σ d'un ensemble de tâches T pour un ensemble de conditions $Cond$ est validée par un workflow Saturn $w = (T, Cond, SatConstr)$ si et seulement si $exec\_order(\sigma) \vDash_* \bigwedge SatConstr$ et*

$$\forall \sigma' : \sigma = \sigma' \overset{c}{t_\blacktriangleright} \sigma'' : (t, c) \in available(\sigma', \bigwedge SatConstr).$$

Une exécution partielle σ d'un ensemble de tâches T pour un ensemble de conditions $Cond$ est partiellement validée par un *workflow* *Saturn* $w = (T, Cond, SatConstr)$ s'il existe une séquence $\sigma' \in \Sigma^*(Actions \times T \times 2^{Cond})$ telle que $\sigma\sigma'$ est une exécution de T pour $Cond$ validée par w.

Exemple 3.2.3 *Soit le workflow $w = (T, Cond, SatConstr)$ tel que :*
- $T = \{write, sendbypost, sendbymail\}$
- $Cond = \{workingserver\}$
- $SatConstr = \{exactly(write), exactly(sendbypost \vee sendbymail), precedence(write, sendbypost), precedence(write, sendbymail), condition\_to\_exist(sendbymail, workingserver)\}$

Celui-ci est représenté graphiquement à la figure 3.8.

Le *workflow* de l'exemple 3.2.3 modélise le comportement d'un processus métier correspondant à la rédaction et l'envoi d'un message :
- la tâche *write* correspond à la rédaction d'un message

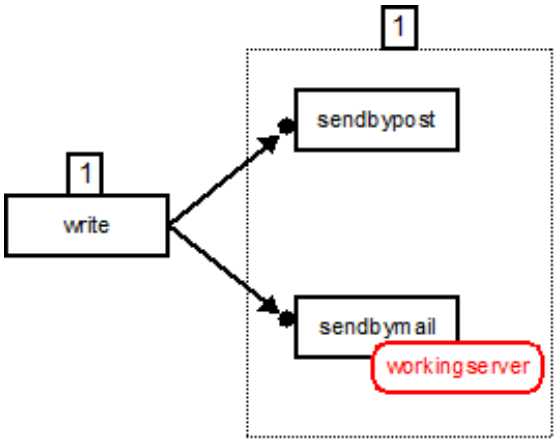


FIG. 3.8 – Représentation graphique du *workflow* w

- la tâche *sendbypost* correspond à l'envoi du message par la poste
- la tâche *sendbymail* correspond à l'envoi du message par e-mail
- la condition *workingserver* est satisfaite si et seulement si le serveur e-mail est opérationnel

La tâche *write* doit être effectuée une et une seule fois. Le message doit être envoyé une et une seule fois et bien sûr après la rédaction du message. Le message ne peut être envoyé par mail que si le serveur e-mail est opérationnel.

Par exemple, les exécutions

- $\overset{\{\}}{write_{\blacktriangleright}}\ \overset{\{\}}{write_{\blacksquare}}\ \overset{\{workingserver\}}{sendbymail_{\blacktriangleright}}\ \overset{\{workingserver\}}{sendbymail_{\blacksquare}}$,
- $write_{\blacktriangleright}\, write_{\blacksquare}\, sendbymail_{\blacktriangleright}\, sendbymail_{\otimes}\, sendbypost_{\blacktriangleright}\, sendbypost_{\blacksquare}$ et
- $write_{\blacktriangleright}\, write_{\blacksquare}\, sendbypost_{\blacktriangleright}\, sendbypost_{\blacksquare}$

sont validées par le *workflow* w.

3.2.4 *Workflow Saturn* et PM hiérarchique

Nous allons à présent considérer des processus métiers hiérarchiques, c'est-à-dire que certaines tâches du *workflow Saturn* associé à ce PM constituent elles-mêmes des processus métiers (éventuellement hiérarchiques à leur tour) qui peuvent être eux aussi exprimés par des *workflow Saturn*, qu'on appelle dans ce cas sous-*workflow*. On dit que ces tâches sont composites par opposition aux tâches atomiques considérées jusqu'ici.

Intuitivement, la réalisation d'une tâche atomique incombe aux utilisateurs du *workflow*, elle n'est pas de la responsabilité du *workflow* proprement dit. C'est pour cela que ce type de tâche est susceptible d'échouer et qu'on ne sait prévoir à quel moment elle échouera ou se complétera. Par contre, la réalisation d'une tâche composite est contrôlée par le sous-*workflow* associé à cette tâche (bien que la réalisation effective des tâches atomiques qui composent ce sous-*workflow* ne lui incombe pas). L'annulation d'une tâche hiérarchique posant un grand nombre de questions qui dépassent le cadre de ce mémoire, on considérera

que ce type de tâche ne peut pas échouer. On redéfinit donc l'exécution d'une tâche en fonction de cette remarque. Les autres concepts (exécution, tâches en cours de réalisation, séquence effectuée, tâches disponibles) restant identiques étant donné cette nouvelle définition.

Définition 3.2.18 (exécution d'une tâche) *Soit un ensemble de tâches T et un ensemble de conditions $Cond$, une exécution de $t \in T$ sous conditions $c \subseteq Cond$ est une séquence $\sigma \in \Sigma^*(Actions \times T \times 2^{Cond})$ qui satisfait la grammaire*

$$\begin{array}{rcl} S & ::= & \epsilon \\ & | & S\ S \\ & | & t^{c}_{\blacktriangleright}\ S\ t^{c}_{\blacksquare} \\ & | & t^{c}_{\blacktriangleright}\ S\ t^{c}_{\otimes} \end{array}$$

si t est une tâche atomique, ou qui satisfait la grammaire

$$\begin{array}{rcl} S & ::= & \epsilon \\ & | & S\ S \\ & | & t^{c}_{\blacktriangleright}\ S\ t^{c}_{\blacksquare} \end{array}$$

si t est une tâche composite.

On adapte la définition du *workflow* Saturn afin de différencier les tâches atomiques des tâches composites. On adapte également la définition de la validité d'une exécution en conséquence.

Définition 3.2.19 (*workflow* Saturn) *Un workflow Saturn w est un quadruplet $(T_A, T_H, Cond, SatConstr)$ tel que :*
- *T_A est un ensemble de tâches atomiques*
- *T_H est un ensemble de tâches composites*
- *$Cond$ est un ensemble de conditions*
- *$SatConstr$ est un ensemble de contraintes Saturn définies sur $T_A \cup T_H$ et $Cond$.*

Pour un *workflow* $w = (T_A, T_H, Cond, SatConstr)$, on notera T l'ensemble de toutes les tâches $T_A \cup T_H$.

Lorsque nous introduirons un *workflow* w, $T_A(w)$ fera référence à l'ensemble de tâches atomiques de w, $T_H(w)$ à l'ensemble de tâches composites, $T(w)$ à l'ensemble de toutes les tâches, $Cond(w)$ à l'ensemble des conditions et $SatConstr(w)$ à l'ensemble de contraintes.

Définition 3.2.20 (validité d'une exécution) *Une exécution σ d'un ensemble de tâches $T = T_A \cup T_H$ pour un ensemble de conditions $Cond$ est validée par un workflow Saturn $w = (T_A, T_H, Cond, SatConstr)$ si et seulement si $exec\_order(\sigma) \vDash_* \bigwedge SatConstr$ et*

$$\forall \sigma' : \sigma = \sigma'\ t^{c}_{\blacktriangleright}\ \sigma'' : (t, c) \in available(\sigma', \bigwedge SatConstr).$$

Définissons maintenant une structure, appelée *workflow* Saturn hiérarchique, qui permet d'exprimer des processus métiers hiérarchiques à l'aide de *workflow* Saturn. Intuitivement, un *workflow* Saturn hiérarchique est composé d'un

workflow Saturn de base. A chaque tâche composite de ce *workflow* est associé un sous-*workflow* composé d'un ensemble de tâches différents. Ce dernier peut également contenir des tâches composites qui définissent de nouveaux sous-*workflow*, et ainsi de suite. Le *workflow* qui contient une tâche composite est appelé le *workflow* parent du *workflow* associé à cette tâche. La définition suivante présente formellement un worklow Saturn hiérarchique.

Définition 3.2.21 (*workflow* Saturn hiérarchique) *Un worflow Saturn hiérarchique h est un sextuplet $(W, w_0, T_A^?, T_H^?, Cond, map)$ tel que :*

- *W est un ensemble de workflow Saturn*
- *$w_0 \in W$ est le workflow de base. Les autres workflow de W sont appelés sous-workflow*
- *$T_A^? = \bigcup_{w \in W} T_A(w)$ est l'ensemble de toutes les tâches atomiques des workflow de h*
- *$T_H^? = \bigcup_{w \in W} T_H(w)$ est l'ensemble de toutes les tâches composites des workflow de h*
- *$\forall w_1, w_2 \in W : w_1 \neq w_2 \Rightarrow T(w_1) \cap T(w_2) = \varnothing$ (les tâches des différents workflow sont distinctes)*
- *$\forall w \in W : Cond(w) = Cond$ (l'environnement est le même pour tous les workflow de h)*
- *$\#T_H^? = \#(W \setminus \{w_0\})$ (le nombre de tâches composites correspond au nombre de sous-workflow)*
- *$map : T_H^? \rightarrow W \setminus \{w_0\}$ est une fonction bijective qui fait correspondre un sous-workflow à chaque tâches composites*

Pour un *workflow* $h = (W, w_0, T_A^?, T_H^?, Env, map)$, on notera $T^?$ l'ensemble de tâches $T_A^? \cup T_H^?$.

Exemple 3.2.4 *Imaginons un processus métier relatif à un demandeur d'emploi (fortement simplifié pour cette illustration). Celui-ci peut suivre des formations (qu'on représentera par la tâche $follow\_form$) et chercher des emplois (tâche $find\_job$). Suivre une formation consiste à s'inscrire à une formation (tâche subscribe), puis suivre et réussir les cours (tâche pass). L'inscription doit être payée (tâche pay). Cela peut se faire en plusieurs paiements et à n'importe quel moment, y compris avant l'inscription ou après avoir suivi et réussi les cours. Chercher un emploi consiste à envoyer des C.V. (tâche $send\_cv$) jusqu'à l'obtention d'un emploi (tâche hired). Un demandeur d'emploi peut suivre autant de formations qu'il le désire, y compris s'il a déjà obtenu un emploi. Il peut également chercher un nouvel emploi même s'il en a déjà obtenu un. On impose, que pour se mettre à chercher un emploi, une personne ait suivi au moins une formation.*

Ce processus peut être représenté par le workflow Saturn hiérarchique $h = (\{w_0, w_1, w_2\}, w_0, T_A^?, T_H^?, \varnothing, map)$ tel que :

- *$w_0 = (\varnothing, T_H^0, \varnothing, SatConstr^0)$ tel que :*
  - *$T_H^0 = \{follow\_form, find\_job\}$*
  - *$SatConstr^0 = \{existence(find\_job), precedence(follow\_form, find\_job)\}$*
- *$w_1 = (T_A^1, \varnothing, \varnothing, SatConstr^1)$ tel que :*
  - *$T_A^1 = \{subscribe, pass, pay\}$*

- $SatConstr^1 = \{exactly(subscribe), exactly(pass), precedence(subscribe, pass), coexistence(subscribe, pay)\}$

– $w_2 = (T_A^2, \varnothing, \varnothing, SatConstr^2)$ *tel que :*
- $T_A^2 = \{send\_cv, hired\}$
- $SatConstr^2 = \{exactly(hired), precedence(send\_cv, hired), not\_succession(hired, send\_cv)\}$

– $T_A^1 = \{subscribe, pass, pay, send\_cv, hired\}$
– $T_H^1 = \{follow\_form, find\_job\}$
– $map(follow\_form) = w_1$ *et* $map(find\_job) = w_2$

Celui-ci est représenté graphiquement à la figure 3.9.

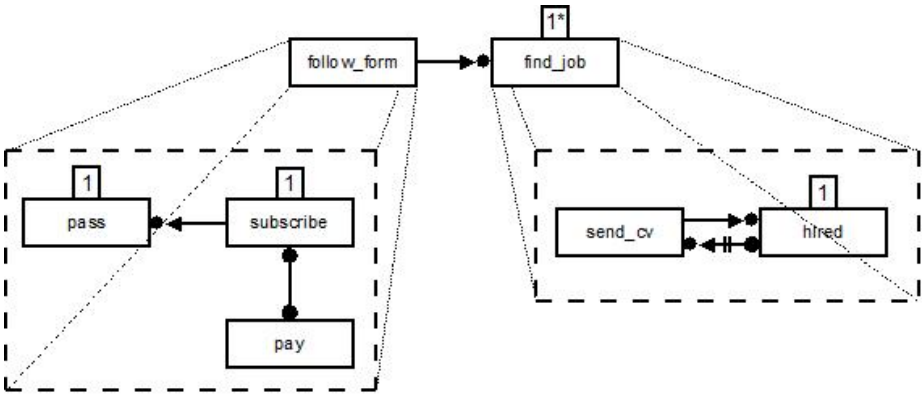


FIG. 3.9 – Représentation graphique du *workflow* hiérarchique h

Avant de définir la validation d'une exécution dans le cas d'un *workflow* *Saturn* hiérarchique, il convient de définir le concept de partition d'une séquence que nous allons utiliser.

> Soit une séquence finie $\sigma \in \Sigma^*(E)$, un ensemble de séquences finies $\{{}_1\sigma, {}_2\sigma, \ldots, {}_k\sigma\}$ forme une partition de σ si
> – $\forall l : 1 \leq l \leq k : {}_l\sigma$ est une sous-séquence de σ
> – $(\bigcap_{1=l}^{k} \mathcal{INS}(\sigma, {}_l\sigma)) = \varnothing$
> – $(\bigcup_{1=l}^{k} \mathcal{INS}(\sigma, {}_l\sigma)) = \{1, \ldots, |\sigma|\}$
>
> Par exemple, l'ensemble constitué des séquences $\langle c, b, c\rangle$, $\langle a, d\rangle$ et $\langle a\rangle$ forme une partition de $\langle c, a, a, b, d, c\rangle$

Présentons maintenant la manière dont une exécution est validée par un *workflow* *Saturn* hiérarchique. Intuitivement, l'exécution débute en réalisant des tâches conformément au *workflow* de base. La partie de l'exécution qui est relative à ce *workflow* de base doit être une exécution validée par celui-ci. A chaque fois qu'une instance de tâche composite est démarrée, des actions relatives au sous-*workflow* associé à cette tâche peuvent être effectuées conformément à celui-ci. Cette instance de tâche composite ne peut être complétée par

le *workflow* parent que si l'exécution relative à cette instance est validée par le sous-*workflow* de cette tâche. Notons que vu que l'on considère toujours des tâches asynchrones, les exécutions relatives à des instances de tâches composites de différents niveaux de *workflow* peuvent être entrelacées. Formellement, la validité d'une exécution par rapport à un *workflow* hiérarchique est définie ci-dessous.

Définition 3.2.22 (validité d'une exécution - *workflow* Saturn hiérarchique) *Une exécution σ d'un ensemble de tâches $T_A^{\lambda} \cup T_H^{\lambda}$ pour un ensemble de conditions Cond est* validée *par un workflow* Saturn *hiérarchique $h = (W, w_0, T_A^{\lambda}, T_H^{\lambda}, Cond, map)$ si et seulement si*

- $\sigma' = \chi(\sigma, \{\overset{c}{t_a} \mid a \in Actions, t \in T(w_0), c \subseteq Cond\})$ *est validée par le workflow w_0*
- *soit $\{i_1, \ldots, i_k\}$ le plus grand ensemble d'indices tel que $\forall l : 1 \leq l \leq k$: $\sigma(i_l) = \overset{c}{t_{\blacktriangleright}}$ où $t \in T_H(w_o)$ et $c \subseteq Cond$, il existe un ensemble d'indice $\{j_1, \ldots, j_k\}$ et un ensemble $\{{}_1\sigma, \ldots, {}_k\sigma\}$ de sous-séquences de σ tels que $\{\sigma', {}_1\sigma, \ldots, {}_k\sigma\}$ est une partition de σ et $\forall l : 1 \leq l \leq k$:*
  - *$i_l < j_l \wedge \sigma(i_l) = \overset{c}{t_{\blacktriangleright}} \Rightarrow \sigma(j_l) = \overset{c}{t_{\blacksquare}}$ (j_l correspond à la complétion de l'instance de tâche composite démarrée en i_l)*
  - *$min(_{\mathcal{IND}}(\sigma, {}_l\sigma)) > i_l \wedge max(_{\mathcal{IND}}(\sigma, {}_l\sigma)) < j_l$ (${}_l\sigma$ se trouve entre le démarrage et la complétion de l'instance de tâche composite correspondante)*
  - *$\sigma(i_l) = \overset{c}{t_{\blacktriangleright}} \Rightarrow {}_l\sigma$ est une exécution validée par le workflow* Saturn *hiérarchique $(W, map(t), T_A^{\lambda}, T_H^{\lambda}, Cond, map)$*

Une exécution partielle σ est partiellement validée par un *workflow* Saturn hiérarchique h s'il existe une séquence σ' telle que $\sigma\sigma'$ est validée par w.

Illustrons cette définition en présentant l'exécution $\sigma = \langle \overset{\varnothing}{follow\_form}_{\blacktriangleright}, \overset{\varnothing}{subscribe}_{\blacktriangleright}, \overset{\varnothing}{subscribe}_{\blacksquare}, \overset{\varnothing}{pass}_{\blacktriangleright}, \overset{\varnothing}{pass}_{\otimes}, \overset{\varnothing}{pass}_{\blacktriangleright}, \overset{\varnothing}{follow\_form}_{\blacktriangleright}, \overset{\varnothing}{pass}_{\blacksquare}, \overset{\varnothing}{pay}_{\blacktriangleright}, \overset{\varnothing}{pay}_{\blacksquare}, \overset{\varnothing}{pay}_{\blacktriangleright}, \overset{\varnothing}{subscribe}_{\blacktriangleright}, \overset{\varnothing}{pay}_{\blacksquare}, \overset{\varnothing}{follow\_form}_{\blacksquare}, \overset{\varnothing}{subscribe}_{\blacksquare}, \overset{\varnothing}{find\_job}_{\blacktriangleright}, \overset{\varnothing}{send\_cv}_{\blacktriangleright}, \overset{\varnothing}{send\_cv}_{\blacksquare}, \overset{\varnothing}{pass}_{\blacktriangleright}, \overset{\varnothing}{pass}_{\blacksquare}, \overset{\varnothing}{follow\_form}_{\blacksquare}, \overset{\varnothing}{hired}_{\blacktriangleright}, \overset{\varnothing}{hired}_{\blacksquare}, \overset{\varnothing}{find\_job}_{\blacksquare}\rangle$ qui est validée par le *workflow* Saturn hiérarchique h de l'exemple 3.2.4.

Intuitivement, comme le suggère le tableau 3.4, la partie σ' de l'exécution relative au *workflow* w_0 est validée par celui-ci, et, entre le démarrage et la complétion des 3 instances (deux instances de $follow\_form$ et une de $find\_job$) de tâches composites qui apparaissent dans cette exécution, on peut trouver autant de sous-séquences propres (${}_1\sigma$, ${}_2\sigma$ et ${}_3\sigma$) qui sont des exécutions validées par le *workflow* associé à cette tâche (ou plus exactement par le *workflow* hiérarchique qui prend comme *workflow* de base le *workflow* associé à cette tâche). En effet, $\{i_1 = 1, i_2 = 7, i_3 = 16\}$ est l'ensemble des indices correspondant au démarrage des instances de tâches composites. On peut trouver un ensemble d'indices $\{j_1 = 14, j_2 = 21, j_3 = 24\}$ correspondant respectivement à la complétion des instances de tâches composites, ainsi qu'un ensemble de sous-séquences $\{{}_1\sigma, {}_2\sigma, {}_3\sigma\}$ formant avec σ' une partition de σ et se trouvant respectivement "entre" le démarrage et la complétion des instances de tâches composites et tel

que chaque sous-séquence est une exécution validée par le sous-*workflow* de la tâche associée, à savoir w_1 pour ${}_1\sigma$ et ${}_2\sigma$, w_2 pour ${}_3\sigma$.

	σ'	${}_1\sigma$	${}_2\sigma$	${}_3\sigma$
$1 = i_1$	$\overset{\varnothing}{follow\_form}_{\blacktriangleright}$	-	-	-
2		$\overset{\varnothing}{subscribe}_{\blacktriangleright}$	-	-
3		$\overset{\varnothing}{subscribe}_{\blacksquare}$	-	-
4		$\overset{\varnothing}{pass}_{\blacktriangleright}$	-	-
5		$\overset{\varnothing}{pass}_{\otimes}$	-	-
6		$\overset{\varnothing}{pass}_{\blacktriangleright}$	-	-
$7 = i_2$	$\overset{\varnothing}{follow\_form}_{\blacktriangleright}$		-	-
8		$\overset{\varnothing}{pass}_{\blacksquare}$		-
9		$\overset{\varnothing}{pay}_{\blacktriangleright}$		-
10		$\overset{\varnothing}{pay}_{\blacksquare}$		-
11			$\overset{\varnothing}{pay}_{\blacktriangleright}$	-
12			$\overset{\varnothing}{subscribe}_{\blacktriangleright}$	-
13			$\overset{\varnothing}{pay}_{\blacksquare}$	-
$14 = j_1$	$\overset{\varnothing}{follow\_form}_{\blacksquare}$	-		-
15		-	$\overset{\varnothing}{subscribe}_{\blacksquare}$	-
$16 = i_3$	$\overset{\varnothing}{find\_job}_{\blacktriangleright}$	-		-
17		-		$\overset{\varnothing}{send\_cv}_{\blacktriangleright}$
18		-		$\overset{\varnothing}{send\_cv}_{\blacksquare}$
19		-	$\overset{\varnothing}{pass}_{\blacktriangleright}$	
20		-	$\overset{\varnothing}{pass}_{\blacksquare}$	
$21 = j_2$	$\overset{\varnothing}{follow\_form}_{\blacksquare}$	-	-	
22		-	-	$\overset{\varnothing}{hired}_{\blacktriangleright}$
23		-	-	$\overset{\varnothing}{hired}_{\blacksquare}$
$24 = j_3$	$\overset{\varnothing}{find\_job}_{\blacksquare}$	-	-	-

TAB. 3.4 – Illustration de la validité de l'exécution σ pour le *workflow* h de l'exemple 3.2.4

Chapitre 4

Moteur de *workflow Saturn*

Comme nous l'avons décrit au chapitre 1, une des motivations principales de la spécification d'un *workflow* dans un formalisme bien défini est de pouvoir implémenter un système, appelé moteur de *workflow*, qui peut contrôler dynamiquement l'exécution d'un processus métier (PM) en fonction du *workflow* qui le représente. Au cours d'un stage dans l'entreprise Mission Critical[1], nous avons développé un outil, l'outil *Saturn*, qui supporte le développement et l'exécution de PM spécifiés par des *workflow Saturn*.

Notre outil permet de contrôler l'exécution de processus métiers conformément aux formalismes décrits à la section 3.2. Il gère donc, grâce au moteur *Saturn*, l'exécution de PM constitués de tâches asynchrones sans jamais interdire la complétion ou l'annulation d'une tâche en cours de réalisation. Ce moteur permet également de gérer l'exécution de PM sensibles à un environnement. De plus, le moteur *Saturn* hiérarchique permet l'exécution de PM hiérarchiques. Notons que l'outil Declare [10], qui permet l'exécution de PM modélisés à l'aide du langage ConDec, est moins puissant que l'outil *Saturn* puisqu'il gère l'exécution de PM constitués de tâches asynchrones de manière inadéquate (sans garantir qu'une tâche en cours de réalisation puisse être complétée ou annulée à tout moment) et ne permet pas d'exprimer des contraintes intégrant des conditions qui représentent l'environnement des PM ni de représenter des PM hiérarchiques.

Nous décrirons, au cours de la section 4.1, la structure de l'outil *Saturn* et nous illustrerons son fonctionnement. Nous aborderons ensuite, lors de la section 4.2, les problèmes de performances rencontrés par ce système et nous présenterons une série d'optimisation qui permet d'améliorer ces performances.

[1] le rapport de ce stage est donné en annexe

4.1 Structure

On peut représenter l'outil *Saturn* par un modèle en couches, comme l'illustre la figure 4.1. Nous avons vu au cours du chapitre 2 qu'il est possible de supporter algorithmiquement (grâce à l'utilisation d'automates finis) la création d'une séquence, élément par élément, qui satisfait une expression LTL (page 41). C'est sur cette base que fonctionne actuellement l'outil *Saturn*. La couche LTL fournit et implémente les opérations nécessaires à la construction de séquences vérifiant une expression LTL donnée. Les opérations fournies par la couche LTL sont utilisées par la couche *Saturn* pour implémenter un système, le moteur *Saturn*, qui fournit des opérations qui permettent de contrôler l'exécution de processus métiers (PM) conformément à un *workflow Saturn*. Ces opérations sont utilisées par la couche supérieure qui permet de contrôler l'exécution de PM hiérarchiques représentés par des *workflow Saturn* hiérarchiques. Notons que les deux couches supérieures utilisent les opérations de sa couche inférieure au travers d'une interface bien délimitée et assez générique pour assurer l'évolutivité de cet outil.

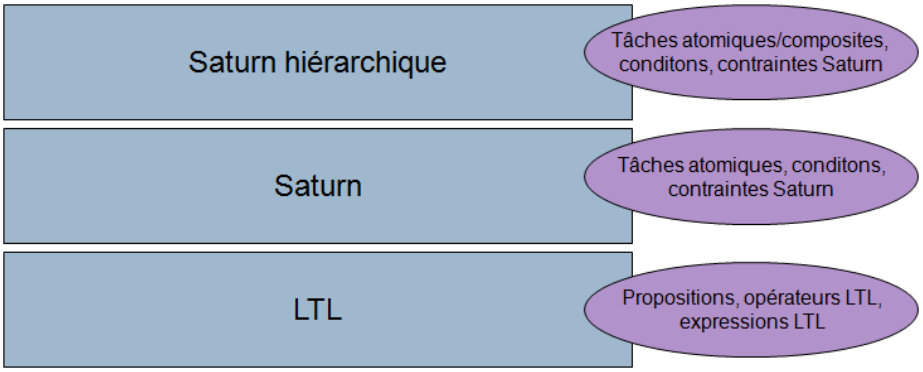


FIG. 4.1 – Un modèle en couche illustrant la structure de l'outil *Saturn*

La structure de l'outil *Saturn* est détaillée à la figure 4.2. On y distingue la phase d'initialisation (*start-up time*) et la phase d'exécution (*run time*). Lors de la phase d'initialisation, les contraintes *Saturn* du *workflow* fourni en entrée sont traduites en expressions LTL qui sont à leur tour transformées en automates acceptant le même langage, ce qui constitue un des points critiques au niveau de la performance (c.f. section 4.2). Ces automates sont utilisés pour représenter le processus métier lors de la phase d'exécution et constituent ainsi la définition du PM.

Lors de cette phase d'exécution, un moteur *Saturn* hiérarchique est mis à contribution. Il offre la possibilité d'exécuter le processus métier conformément à la définition qu'on lui fournit et ce, grâce à 5 types d'opérations qui constituent l'interface de ce moteur. Pour réaliser une opération au cours d'une exécution, on doit fournir au moteur, en plus de la définition, l'état de l'instance de *workflow* (qui évolue en fonction des actions effectuées) et la valeur des conditions qui représentent l'environnement du PM. La valeur des conditions peut bien sûr changer au cours d'une exécution.

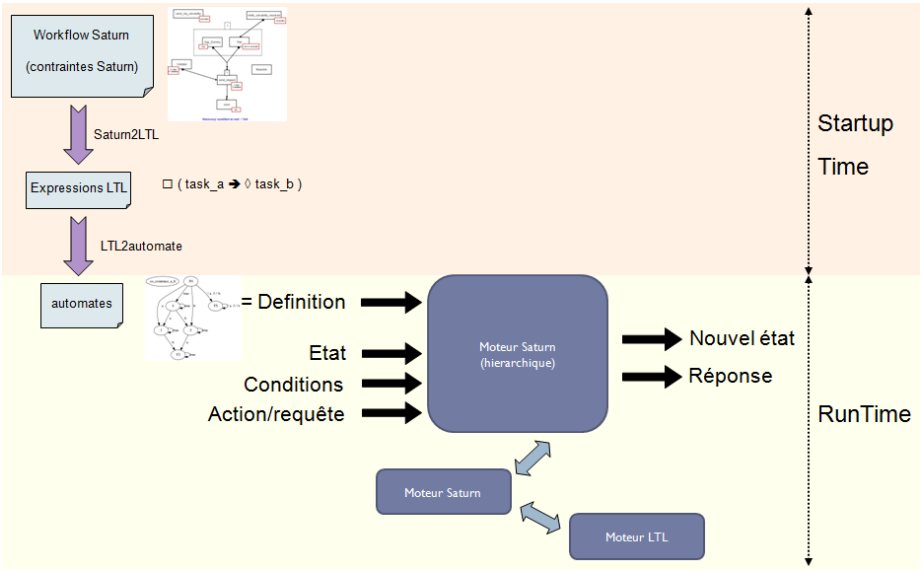


FIG. 4.2 – Structure de l'outil *Saturn*

Les opérations que l'on peut effectuer sont alors :

available_tasks : demander la liste des tâches que l'on peut démarrer.

checkout_task : démarrer une tâche.

complete_task : compléter la réalisation d'une tâche préalablement démarrée.

cancel_task : annuler la réalisation d'une tâche préalablement démarrée.

end : terminer le PM (autrement dit, fermer l'instance de *workflow*).

L'opération **checkout_task** peut échouer si la tâche que l'on veut démarrer ne fait pas partie de la liste des tâches autorisées. Les opérations **complete_task** et **cancel_task** ne peuvent échouer lorsqu'elles sont appliquées à une instance de tâche atomique. L'opération **cancel_task** n'est pas autorisée pour une tâche composite. L'opération **end** échoue si l'état de l'instance indique que les actions effectuées jusqu'ici ne satisfont pas (encore) le *workflow*.

Le moteur de *workflow* peut être vu comme un système qui s'assure qu'une exécution reste toujours partiellement validée par le *workflow* qu'on lui fournit. Il peut en outre indiquer à tout moment si l'exécution est validée ou partiellement validée par ce *workflow*. Le moteur de *workflow* *Saturn* hiérarchique interagit bien sûr avec le moteur de *workflow* *Saturn* qui, lui-même, utilise les opérations qui permettent de construire des séquences satisfaisant des expressions LTL (représentées à la figure 4.2 par le moteur LTL). Notons que si le *workflow* de base n'est pas hiérarchique on peut utiliser directement l'interface du moteur *Saturn*.

Remarquons enfin que l'interface décrit ci-dessus coïncide avec l'interface d'un moteur développé chez *Mission Critical* permettant d'exécuter des processus métiers exprimés à l'aide du langage spécification de *workflow* de nature impérative YAWL, et cela pour faciliter une future intégration des deux formalismes.

Exemple

Nous allons illustrer le fonctionnement de l'outil *Saturn* au moyen de l'exemple qui suit. Soit un processus métier (non hiérarchique dans cet exemple) composé de deux tâches A et B et de la condition c. A doit être réalisée une et une seule fois. B peut être réalisée un nombre arbitraire de fois. La tâche A ne peut être réalisée que lorsque la condition c est vraie. Le *workflow* *Saturn* $w = (T, Cond, SatConstr)$ tel que

- $T = \{A, B\}$,
- $Cond = \{c\}$ et
- $SatConstr = \{exactly(A), condition\_to\_exist(A, c)\}$

permet de représenter ce processus métier.

La première étape, l'étape d'initialisation, consiste à transformer les contraintes en expressions LTL qui sont alors converties en automates classiques, comme c'est illustré à la figure 4.3.

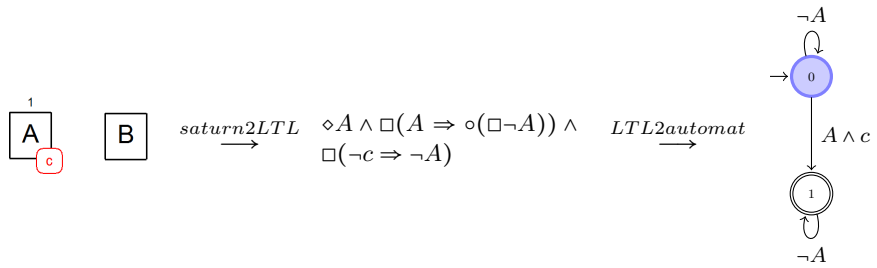
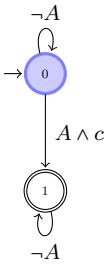


FIG. 4.3 – Evolution de la représentation du processus métier dans *Saturn*

Le tableau qui suit illustre le fonctionnement du moteur *Saturn*. La colonne de gauche représente la trace obtenue en exécutant une instance du *workflow* à l'aide d'un outil en ligne de commande développé dans le cadre de l'outil *Saturn*. Cet outil, qui gère aussi l'exécution de processus métiers hiérarchiques[1], permet d'effectuer les opérations offertes par le moteur *Saturn*. Entre chaque opération, l'utilisateur peut visualiser les tâches disponibles, les tâches en cours d'exécution, la valeur actuelle des conditions (qu'il peut changer à tout moment) et peut savoir s'il est possible de cloturer le processus (si les contraintes sont toutes satisfaites). La colonne de droite met en évidence le statut de l'automate entre chaque opération. La conjonction de ce statut et des tâches en cours d'exécution (pour pouvoir calculer les permutations, voir section 3.2) constitue l'état de l'instance du *workflow*. Cette trace représente l'exécution $\langle \overset{\{\}}{B_{\blacktriangleright}}, \overset{\{c\}}{A_{\blacktriangleright}}, \overset{\{c\}}{A_{\blacksquare}}, \overset{\{\}}{B_{\otimes}} \rangle$.

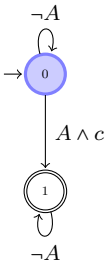
[1] d'où l'entier qui précède les tâches en indiquant son niveau dans la hiérarchie

```
---------------------------------
   Available tasks : 1.B
   Running tasks   :
   End authorized  : No
   Conditions      : c : false
---------------------------------
Select an action [1-5] :

1 - Checkout a task
2 - Complete a running task
3 - Cancel a running task
4 - End
5 - Change conditions
```



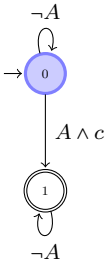

```
 Make a choice > 1

Select the task to checkout :

1 - 1.B

  Make a choice > 1
```

```
---------------------------------
   Available tasks : B
   Running tasks   : B
   End authorized  : No
   Conditions      : c : false
---------------------------------
Select an action [1-5] :

1 - Checkout a task
2 - Complete a running task
3 - Cancel a running task
4 - End
5 - Change conditions
```



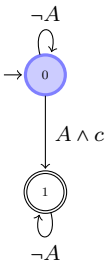

```
 Make a choice > 5

      Give me the true conditions around : ["c"] : c
```

```
 ---------------------------------
   Available tasks : 1.A - 1.B
   Running tasks   : B
   End authorized  : No
   Conditions      : c : true
---------------------------------
Select an action [1-5] :

1 - Checkout a task
2 - Complete a running task
3 - Cancel a running task
4 - End
5 - Change conditions
```



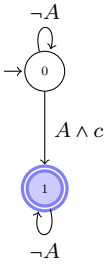
```
 Make a choice > 1

Select the task to checkout :

1 - 1.A
2 - 1.B

  Make a choice > 1
```

```
---------------------------------
   Available tasks : 1.B
   Running tasks   : A - B
   End authorized  : No
   Conditions      : c : true
---------------------------------
Select an action [1-5] :

1 - Checkout a task
2 - Complete a running task
3 - Cancel a running task
4 - End
5 - Change conditions
```



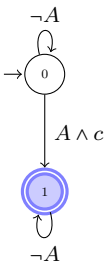

```
 Make a choice > 2

Select the task to complete :

1 - A
2 - B

 Make a choice > 1
```

```
---------------------------------
   Available tasks : 1.B
   Running tasks   : B
   End authorized  : No
   Conditions      : c : true
---------------------------------
Select an action [1-5] :

1 - Checkout a task
2 - Complete a running task
3 - Cancel a running task
4 - End
5 - Change conditions
```



```
 Make a choice > 1

Select the atomic task to cancel :

1 - B

 Make a choice > 1
```

```
---------------------------------
   Available tasks : 1.B
   Running tasks   :
   End authorized  : Yes
   Conditions      : c : true
---------------------------------
Select an action [1-5] :

1 - Checkout a task
2 - Complete a running task
3 - Cancel a running task
4 - End
5 - Change conditions
```




```
 Make a choice > 4
```

Outil additionnel pour faciliter la compréhension des *workflow Saturn*

Bien que les *workflow Saturn* peuvent être représentés par des notations graphiques adéquates, nous nous sommes rendu compte que ces dernières ne sont pas toujours évidentes à comprendre rapidement à moins d'être particulièrement familiarisé avec ces notations. Donner une représentation d'une contrainte est en effet intuitivement moins évident que de donner une représentation d'une relation directe de cause à effet, par exemple. Dans le but de simplifier la lecture et la compréhension directe de *workflow Saturn*, nous avons réalisé un outil (l'outil *saturn2english*) qui permet d'exprimer automatiquement la signification d'un *workflow Saturn* par un texte écrit en langage naturel, comme l'illustre la figure 4.4. Ce texte décrit les différentes règles qui sont représentées par le *workflow*.

Implémentation de l'outil *Saturn*

Nous tenons enfin à faire remarquer que l'outil *Saturn* a été développé à l'aide du langage de programmation déclaratif *Mercury* [2, 3]. L'intérêt de l'utilisation de *Mercury* est abordé dans [4]. Sans revenir en détail sur tous les avantages de la programmation déclarative[1], nous tenons à attirer l'attention sur la concision des programmes écrits en *Mercury* comparativement à des programmes équivalents écrits dans d'autres langages plus traditionnels. Nous avons en effet pu comparer l'outil open-source Declare, écrit en *Java*, avec l'outil *Saturn*. Le tableau 4.1 donne un aperçu du nombre de fichiers et de lignes de code pour chacun des outils. Pour plus de clarté, on a séparé la partie spécifique à LTL et la partie qui constitue le coeur de leur système respectif. Ces chiffres sont

[1] Voir à ce sujet le témoignage de l'utilisation de *Mercury* à l'annexe A.2

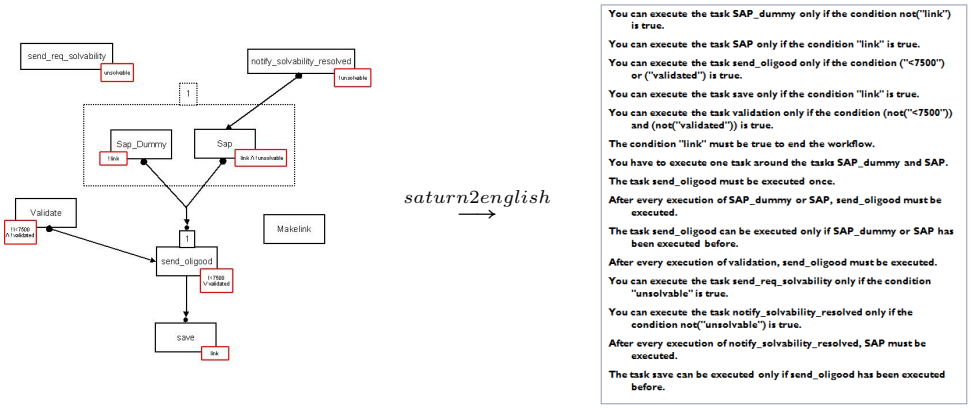


FIG. 4.4 – L'outil *saturn2english*

bien sûr à prendre avec précaution puisque les deux outils ne sont pas identiques, mais ils sont tout de même très pertinents. Cette concision est bien sûr bénéfique puisqu'elle facilite le développement, l'évolution et la maintenance de l'outil *Saturn*.

		Fichiers	Lignes	
Implém. *Java*				
	Declare Core	122	11 564	
	LTL part	33	6 371	
	Total	155	17 935	
Implém. *Mercury*				% fichiers (lignes) *Java*
	Saturn Core	3	815	2 (7) %
	LTL part	2	879	6 (14) %
	Total	5	1 694	3 (9) %

TAB. 4.1 – Comparaison des implémentations de l'outil Declare et de l'outil *Saturn*

4.2 Performances

Comme nous l'avons vu, une des principales raisons motivant l'utilisation d'un langage de spécification de *workflow* particulier est de pouvoir contrôler dynamiquement l'exécution de processus métiers (PM) exprimés dans ce langage, grâce à un système appelé moteur de *workflow*. Ce système doit fonctionner de manière performante, c'est-à-dire qu'il doit être utilisable en pratique pour des *workflow* de taille raisonnable. Intuitivement, les problèmes de performances dans le cadre de *workflow* impératifs sont peu fréquents car la description du *workflow* constitue déjà en soit une description de la manière dont doit s'exécuter le PM. Comme nous l'avons déjà évoqué, les *workflow* déclaratifs comme *Saturn* expriment le PM grâce à un ensemble de contraintes, permettant une flexibilité maximale dans l'ordre de réalisation des tâches tant que ces contraintes sont respectées. Le prix à payer de cette grande flexibilité offerte par les *workflow Saturn* est que le moteur de *workflow* doit lui-même déterminer comment exécuter le PM en fonction du *workflow Saturn* donné, ce qui peut donner lieu à des problèmes de performances pour des *workflow* de taille conséquente. Obtenir des performances satisfaisantes est un des enjeux les plus importants dans le cadre des *workflow* déclaratifs.

Lors du développement de l'outil *Saturn*, nous avons identifié deux problèmes de performances pour lesquels nous allons présenter des solutions pragmatiques qui nous permettent d'affirmer qu'en pratique, les *workflow Saturn* peuvent être utilisés pour supporter l'exécution de PM. Le premier problème survient lors de la phase d'initialisation (voir figure 4.2) et a déjà été identifié lors du chapitre 2 puisqu'il s'agit de la génération d'automates à partir d'expressions LTL (les contraintes *Saturn* qui constituent le *workflow*) qui se réalisent en un temps exponentiel. Le second problème, de complexité factorielle, survient lors de la phase d'exécution lorsque l'on doit calculer toutes les permutations des tâches en cours de réalisation pour s'assurer qu'une nouvelle tâche peut être démarrée (chapitre 3, page 58).

4.2.1 Performances à l'initialisation

Nous avons vu que la conversion d'expressions LTL en automates était nécessaire pour vérifier dynamiquement la validité d'une exécution par rapport à un *workflow Saturn*. Comme nous l'avons vu au chapitre 2, c'est un problème de complexité exponentielle. Néanmoins, elle est utilisée avec succès dans le cadre du Model Checking [20, 14, 21, 22] et nous bénéficions des nombreux travaux effectués dans ce cadre pour améliorer les performances de cette conversion ([35, 36, 37, 38, 39]).

Dans le cadre du développement de l'outil *Saturn*, nous avons découvert qu'il était encore possible d'optimiser la génération d'automates. Nous allons présenter 3 optimisations de types différents et nous verrons l'impact que celles-ci peuvent avoir sur les performances.

Optimisation 1 : automates parallèles

Comme nous l'avons vu (définition 3.2.16), la validation d'une exécution pour un *workflow Saturn* passe par l'utilisation d'une expression LTL qui est

constituée de la conjonction de l'ensemble des contraintes du *workflow*. Plus il y a des contraintes, plus la taille de l'expression augmente, plus les risques d'explosion combinatoire sont présents lors de la génération de l'automate.

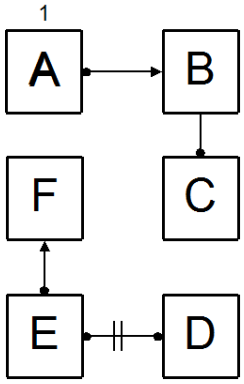


FIG. 4.5 – Un *workflow* saturn

Par exemple, pour le *workflow* Saturn représenté à la figure 4.5, on réalise la conjonction de l'ensemble des contraintes $\{exactly(A), response(A,B), existence\_resp(C,B), not\_coexistence(E,D), response(E,F)\}$ et on construit un automate correspondant à cette expression LTL, représenté à la figure 4.6. On constate que cet automate est conséquent vu la taille de l'expression LTL.

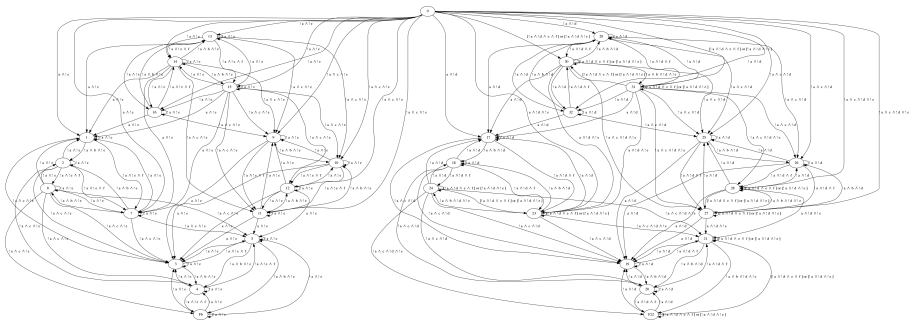


FIG. 4.6 – Représentation du *workflow* de la figure 4.5 en un seul automate

Une heuristique qui permet de réduire la taille des automates et le temps nécessaire à leur création consiste à partitionner l'ensemble des contraintes Saturn en un nombre maximum de classes de telle manière que les ensembles constitués des tâches et conditions qui apparaissent dans chacune des classes sont disjoints.[1] On génère alors un automate pour chaque classe de contraintes. L'exécution du processus métier est réalisée en faisant évoluer le statut de chaque automate en parallèle et la validation d'une exécution est déterminée par la satisfaction de chaque expression LTL relative aux différentes classes de contraintes.[2]

[1] C'est un problème de type "Union-find" de complexité $\mathcal{O}(n^2)$.
[2] Cette optimisation peut être généralisée au niveau LTL.

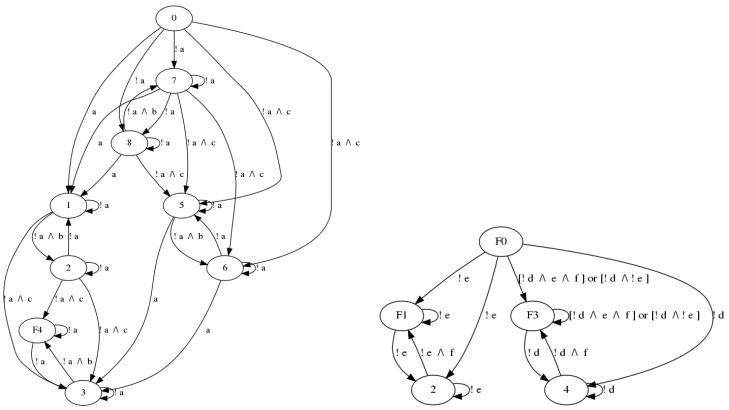


FIG. 4.7 – Représentation du *workflow* de la figure 4.5 en deux automates

Pour le *workflow* Saturn représenté à la figure 4.5, on peut partitionner l'ensemble des contraintes en deux sous-ensembles, $\{exactly(A), response(A,B), existence\_resp(C,B)\}$ et $\{not\_coexistence(E,D), response(E,F)\ \}$, et générer ainsi un automate pour chacune d'entre elles. Ces automates (figure 4.7) sont beaucoup plus petits et rapides à construire.

Optimisation 2 : détection d'états rouges à la volée

Nous avons vu au cours du chapitre 2 que les automates générés par l'algorithme en 3 phases (page 27) peuvent contenir des états qui ne peuvent mener vers aucun état final, appelés états rouges. Ces états non-désirés peuvent être facilement supprimés après génération de l'automate. Ils ralentissent néanmoins le processus de conversion d'expressions LTL en automates. Une méthode permettant d'augmenter les performances de l'algorithme est de détecter à la volée, au cours de la génération du graphe (phase 2, page 28), des noeuds assurés de devenir des états rouges, dits "noeuds rouges".

En analysant les noeuds rouges apparaissant régulièrement, nous avons constaté que le champ $Next$ de beaucoup d'entre eux comprenait notamment une expression **true** $\mathbb{U}$ A (soit $\diamond(A)$) et une expression **false** $\mathbb{R}$ $\neg A$ (soit $\Box(\neg A)$), où A est une expression LTL quelconque. Cela traduit une exigence impossible puisque cela signifie que le reste de la séquence à traiter doit contenir au moins une configuration qui satisfait A mais également que toutes les configurations doivent satisfaire $\neg A$. En détectant ces phénomènes durant la génération, on évite donc un grand nombre de calculs inutiles puisqu'on on ne doit pas traiter ces noeuds ainsi que tous leurs descendants. La figure 4.8 montre à quel point cette optimisation peut simplifier les automates.

Optimisation 3 : heuristique *Saturn*

Les optimisations présentées jusqu'ici sont applicables au niveau de la couche LTL (voir figure 4.1) et s'inscrivent donc dans la continuité des recherches effectuées pour améliorer les performances de la conversion d'expressions LTL en automates. La solution qui suit s'inscrit quant à elle dans une démarche qui

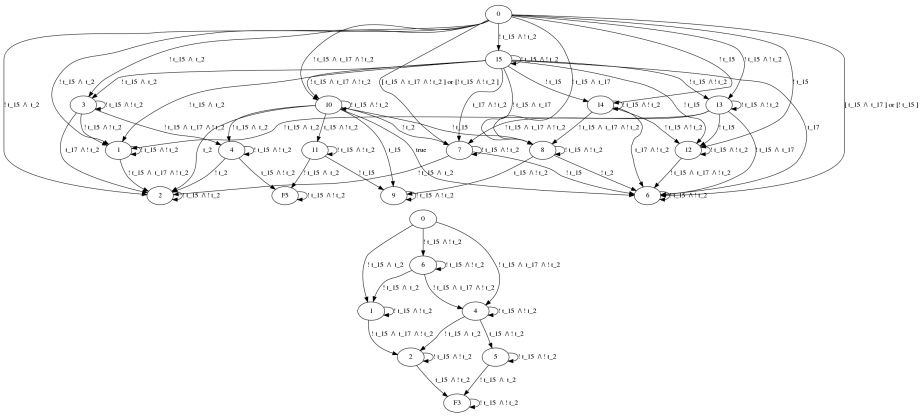


FIG. 4.8 – Automate généré pour le même ensemble de contraintes avec (en haut) et sans états rouges (en bas)

consiste, lorsque cela est possible, à optimiser la génération des automates en fonction du contexte dans lequel LTL est utilisé.

Dans le cas de *Saturn*, certains noeuds sont générés alors qu'ils ne sont pas pertinents pour la raison suivante : comme l'illustre la définition 3.2.15, les séquences qui sont soumises aux expressions LTL sont constituées de configurations qui contiennent une et une seule tâche. Intuitivement, cela provient du fait que, lors d'une exécution, on n'effectue qu'une seule action sur une tâche "à la fois". Par conséquent, les transitions qui exigent que plusieurs propositions représentant des tâches différentes soient satisfaites ne sont jamais utilisées et les états qui ne peuvent être atteints que par ce type de transition sont inutiles.

Par exemple, étant donné un ensemble de propositions $\{A, B, C\}$ qui représentent des tâches dans le cadre d'un *workflow Saturn*, la figure 4.9 montre deux transitions : la première est susceptible d'être utilisée dans le cadre d'une exécution au contraire de la seconde.

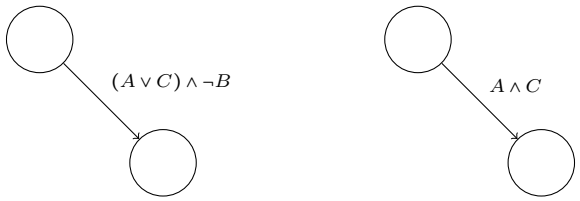


FIG. 4.9 – Une transition pertinente (à gauche) et non pertinente (à droite) dans le cadre de *workflow Saturn*

On peut donc détecter, durant la génération du graphe (à la phase 2), les noeuds dont le champ *Old* (duquel les transitions sont déduites à la phase 3) comprend plus d'une proposition atomique en forme positive correspondant à une tâche. Ces noeuds ne sont pas traités et de nombreux calculs sont alors évités. Les figures 4.10 et 4.11 illustrent les gains obtenus par cette optimisation.

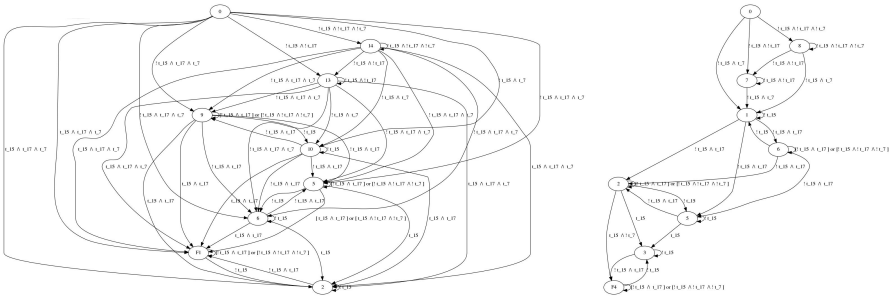


FIG. 4.10 – Automate généré pour le même ensemble de contraintes avec (à droite) et sans (à gauche) heuristique

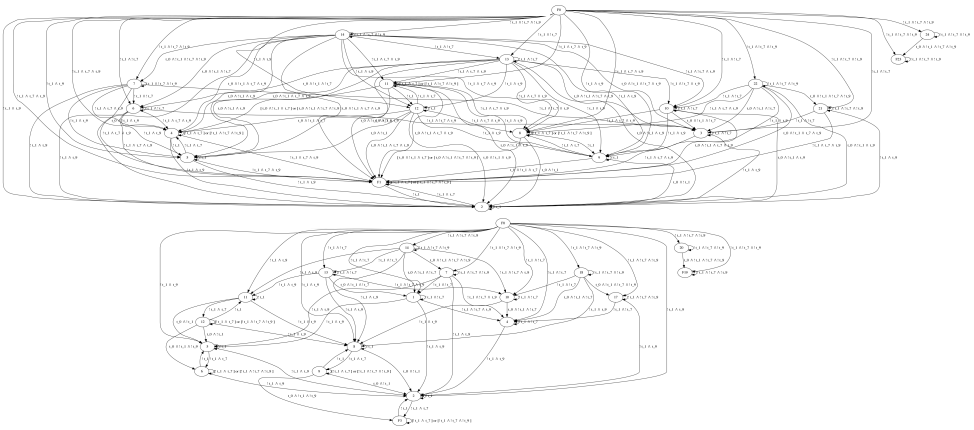


FIG. 4.11 – Automate généré pour le même ensemble de contraintes avec (en bas) et sans (en haut) heuristique

Notons que cette heuristique ne modifie pas la correction (seules les exécutions qui sont validées par le *workflow* sont autorisées) ni la complétude (toutes les exécutions validées par le *workflow* sont autorisées) de l'outil *Saturn* par rapport au formalisme qui décrit la validation des exécutions par les *workflow Saturn*.

Résultats et perspectives

Les tableaux des figures 4.12 et 4.13 présentent les résultats de tests de performances effectués après l'ajout successif des optimisations 1 à 3. Les tests sont effectués par rapport à une taille LTL donnée. Un test consiste à générer un échantillon de *workflow Saturn* dont le nombre de tâches et le nombre de conditions sont donnés et dont les contraintes sont générées aléatoirement de telle manière que leur conjonction atteint la taille LTL spécifiée. On calcule alors le temps nécessaire pour exécuter la phase d'initialisation de tous les *workflow* générés, et ce en ne considérant aucune optimisation, puis en ajoutant successivement les optimisations 1 (automates parallèles), 2 (détection à la volée d'états rouges) et 3 (heuristiques *Saturn*). Lorsque la génération pour l'ensemble de l'échantillon prend plus de 5 minutes, on l'indique par le symbole ∞.

Valeurs fixes du test :
– Nombre de tâches pour chaque *workflow* généré : 20
– Nombre de conditions pour chaque *workflow* généré : 5
– Nombre de *workflow* générés pour chaque taille LTL : 30

Taille LTL	Sans Opt.	Opt. 1	Opt. 1-2	Opt. 1-2-3
10	0.07	0.04	0.02	0.03
20	0.21	0.07	0.02	0.02
30	1.59	0.05	0.05	0.03
40	10.86	0.05	0.05	0.04
50	102.72	0.14	0.10	0.09
60	∞	0.65	0.30	0.25
70	∞	14.49	2.28	1.16
80	∞	9.68	1.60	0.99
90	∞	106.09	15.03	7.60
100	∞	∞	7.20	4.81
110	∞	∞	∞	98.18

Les résultats portent sur l'intégralité de l'échantillon et sont exprimés en secondes. Le timeout (∞) est fixé à 600 secondes.
(Processeur : Intel(R) Core(TM)2 Duo TM 7300 @2.00 Ghz. RAM : 4096 Kb)

FIG. 4.12 – Performances pour des *workflow* de 20 tâches

Valeurs fixes du test :
– Nombre de tâches pour chaque *workflow* généré : 25
– Nombre de conditions pour chaque *workflow* généré : 5
– Nombre de *workflow* générés (échantillon) pour chaque taille LTL : 30

Taille LTL	Sans Opt.	Opt. 1	Opt. 1-2	Opt. 1-2-3
10	0.02	0.01	0.01	0.00
20	0.23	0.02	0.02	0.01
30	3.17	0.03	0.02	0.03
40	39.69	0.05	0.05	0.07
50	172.68	0.08	0.07	0.10
60	∞	0.36	0.17	0.18
70	∞	0.78	0.46	0.48
80	∞	47.18	9.04	4.55
90	∞	24.40	4.25	3.39
100	∞	∞	80.00	39.31
110	∞	∞	∞	226.01

Les résultats portent sur l'intégralité de l'échantillon et sont exprimés en secondes. Le timeout (∞) est fixé à 600 secondes.

(Processeur : Intel(R) Core(TM)2 Duo TM 7300 @2.00 Ghz. RAM : 4096 Kb)

FIG. 4.13 – Performances pour des *workflow* de 25 tâches

Comme on peut le constater, l'impact de chacune des optimisations est important. Pour donner un ordre d'idées, la taille LTL moyenne des modèles de contraintes présentées dans la section 3.1 est d'un peu plus de $7,5$. Notre outil peut donc actuellement gérer des *workflow* constitués d'environ une quinzaine de contraintes, bien que ce nombre puisse varier en fonction des cas.

Notons que les tests que nous avons effectués pour comparer les performances de l'outil Declare [10] avec l'outil *Saturn* ont révélé que ce dernier était plus rapide. Pour certains *workflow*, la différence est même très nette : nous avons en effet constaté que des *workflow* étaient initialisés par l'outil *Saturn* en quelques secondes alors que plusieurs minutes étaient nécessaires pour que l'outil Declare réalise la même opération.

Grâce aux différents travaux de recherche relatifs à la génération d'automates à partir d'expressions LTL et aux optimisations que nous avons présentées ci-dessus, nous avons pu développer un outil dont les performances sont satisfaisantes pour des *workflow* de taille réelles (des *workflow* *Saturn* constitués d'une dizaine, voir d'une quinzaine de contraintes peuvent exprimer des processus métiers complexes).

Néanmoins, dans l'optique de repousser encore les limites pratiques de cet outil, nous allons présenter quelques perspectives et pistes qu'il peut être intéressant d'explorer.

Remarquons tout d'abord que notre outil profitera des nombreuses recherches qui continuent à être effectuées dans le cadre de LTL. En analysant les automates générés par notre moteur LTL, nous avons pu constater certaines redondances et nous sommes donc persuadés que des optimisations intéressantes vont encore être découvertes.

La découverte de nouvelles heuristiques, propres à *Saturn* (à l'image de l'optimisation 3), est prometteuse, surtout dans les cas où on tolère des heuristiques correctes mais incomplètes (l'outil n'admettrait qu'une partie des exécutions validées par le *workflow*). Le gain de performances pourrait ainsi se faire en échange d'une légère perte de flexibilité, ce qui peut être acceptable pour certains processus métiers.

Nous avons également constaté que les groupes de contraintes qui avaient tendance à donner lieu à une explosion combinatoire, étaient souvent ceux qui, intuitivement, simulaient sous forme de contraintes une exécution qu'il serait plus naturel d'exprimer à l'aide d'un formalisme impératif. Ceci motive encore plus un travail d'intégration entre le formalisme *Saturn* et un formalisme impératif comme YAWL. Grâce à la gestion de processus métiers hiérarchiques par chacun de ces langages, on peut imaginer un *workflow* dont certaines tâches sont constituées de sous-*workflow* *Saturn* et d'autres de sous-*workflow* YAWL. Cela permettrait au concepteur de choisir le formalisme le plus approprié en fonction des cas, et d'améliorer du même coup les performances de manière significative.

Enfin, on encouragera le concepteur du *workflow* *Saturn* à utiliser la possibilité de décrire les processus métiers de manière hiérarchique. En effet, cela permet de répartir la complexité du processus métier en plusieurs *workflow*, et minimise donc les chances d'explosion combinatoire.

4.2.2 Performances à l'exécution

Comme nous l'avons vu au cours de la section 3.2 (page 58), il est nécessaire de calculer au cours d'une exécution toutes les permutations du multi-ensemble constitué des tâches en cours de réalisation pour vérifier qu'aucune de ces permutations ne viole de contraintes.

Pour illustrer concrètement ce problème, considérons le *workflow* $w = (\{A, B, C, D\}, \varnothing, \{absence_2(B), precedence(A, B)\})$ de la figure 4.14. L'automate correspondant à ce *workflow* est présenté à la figure 4.15. L'exécution partielle $\overset{\varnothing}{C}_{\blacktriangleright}\overset{\varnothing}{C}_{\blacksquare}\overset{\varnothing}{A}_{\blacktriangleright}\overset{\varnothing}{C}_{\blacktriangleright}$ est partiellement valide car il y a au moins un état de l'automate accessible pour chacune des séquences suivantes : $\langle\{C\}\rangle$ $\langle\{A\},\{C\}\rangle$ et $\langle\{C\}\rangle$ $\langle\{C\},\{A\}\rangle$. Par contre, l'exécution $\overset{\varnothing}{A}_{\blacktriangleright}\overset{\varnothing}{B}_{\blacktriangleright}$ n'est pas partiellement valide car il n'y pas d'état accessible pour la séquence $\langle\{B\},\{A\}\rangle$.

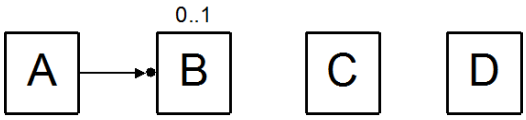


FIG. 4.14 – Le *workflow* $(\{A, B, C, D\}, \varnothing, \{absence_2(B), precedence(A, B)\})$

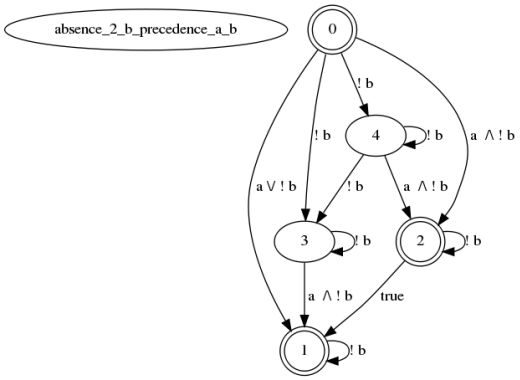


FIG. 4.15 – Automate correspondant au *workflow* $(\{A, B, C, D\}, \varnothing, \{absence_2(B), precedence(A, B)\})$

La génération de toutes les permutations est bien entendu un problème de complexité factorielle. Heureusement, il est possibler d'éviter le calcul d'un grand nombre de permutations grâce à la structure des automates générés. Nous tenons à signaler que notre objectif n'est pas de réaliser une analyse théorique approfondie à ce sujet mais bien d'obtenir des performances satisfaisantes grâce à des solutions pragmatiques et simples à implémenter. C'est dans ce contexte que nous présentons brièvement trois optimisations que nous avons utilisées dans le cadre du développement de l'outil *Saturn* et une propriété intéressante pour l'utilisation d'une mémoire cache. Leur utilisation conjointe permet en pratique de résoudre ce problème complexe de manière très rapide.

Formalisons le problème à résoudre. Soit un automate $\mathcal{AC} = (\mathcal{A}, Q, \delta, Q^0, F)$, une séquence $\sigma \in \Sigma^*(\mathcal{A})$ et un multi-ensemble $M \in \mathcal{M}(\mathcal{A})$, il s'agit de déterminer si $\forall p \in Perm(M) : \exists q \in Q : q$ est accessible via σp (ou autrement dit, le statut de l'automate pour la séquence σp est non-vide), dans quel cas on dira que le multi-ensemble M est valide par rapport à σ.

Première optimisation On constate que les automates générés à partir d'expressions LTL comportent un grand nombre de boucles (définies ci-dessous). On peut exploiter cette caractéristique pour éviter de calculer toutes les permutations dans certains cas. Pour s'assurer que le multi-ensemble M est valide par rapport à σ, il suffit en effet de trouver, parmi le statut de l'automate $\mathcal{AC}$ pour la séquence σ, un état qui admet une boucle satisfaite par tous les éléments de M. En effet, quelle que soit la permutation p de M, l'état en question fera logiquement toujours partie du statut de l'automate pour la séquence σp.

Définition 4.2.1 *Soit un automate $\mathcal{AC} = (\mathcal{A}, Q, \delta, Q^0, F)$, on dit qu'un état $q \in Q$ admet une boucle si et seulement s'il existe un ensemble non vide de transitions $(q, x, q) \in \delta$ où $x \in \mathcal{A}$. Pour un tel état, on dit que sa boucle est satisfaite par un élément de $t \in \mathcal{A}$ s'il existe une transition $(q, t, q) \in \delta$.*

Par exemple, après la réalisation de la tâche C, le statut de l'automate de la figure 4.15 est composé des états 1, 3 et 4 qui admettent chacun une boucle. Par conséquent, sans "tester" chaque permutation de l'ensemble des tâches en cours de réalisation, on peut déterminer que l'exécution partielle $\overset{\varnothing}{C}_{\blacktriangleright}\overset{\varnothing}{C}_{\blacksquare}\overset{\varnothing}{A}_{\blacktriangleright}\overset{\varnothing}{C}_{\blacktriangleright}\overset{\varnothing}{C}_{\blacktriangleright}\overset{\varnothing}{D}_{\blacktriangleright}\overset{\varnothing}{A}_{\blacktriangleright}$ est partiellement valide car tous les éléments du multi-ensemble $\{\{A\},\{C\},\{C\},\{D\},\{A\}\}_{\mathcal{M}}$ satisfont la boucle $\neg B$.

Seconde optimisation Les automates générés à partir d'expressions LTL sont également souvent constitués de structures que nous appelons ponts, semblables à celles présentées à la figure 4.16 et dont on peut se servir pour éviter de calculer toutes les permutations d'un multi-ensemble dans certains cas. La définition formelle d'un pont pour un multi-ensemble est donnée ci-dessous. Pour s'assurer que M est valide par rapport à σ, il suffit de trouver un pont pour M tel que son état source fait partie du statut de l'automate $\mathcal{AC}$ pour la séquence σ. On est ainsi certain que, quelle que soit la permutation p de M, l'état destination du pont en question fera partie du statut de l'automate pour la séquence σp.

Définition 4.2.2 *Un automate $\mathcal{AC} = (\mathcal{A}, Q, \delta, Q^0, F)$ admet un pont pour le multi-ensemble $M = (\mathcal{A}, m)$ s'il existe une transition $(q, x, q') \in \delta$ telle que :*
- *$m(x) = 1$*
- *q (l'état source) et q' (l'état destination) admettent chacun une boucle*
- *$\forall e \in M : e \neq x : e$ satisfait les boucles de q et de q'*

Par exemple, après la réalisation de la tâche A, le statut de l'automate de la figure 4.15 est composé des états 1, 2, 3 et 4. Sans tester chaque permutation de l'ensemble des tâches en cours de réalisation, on peut déterminer que l'exécution partielle $\overset{\varnothing}{A}_{\blacktriangleright}\overset{\varnothing}{A}_{\blacksquare}\overset{\varnothing}{B}_{\blacktriangleright}\overset{\varnothing}{C}_{\blacktriangleright}\overset{\varnothing}{C}_{\blacktriangleright}\overset{\varnothing}{D}_{\blacktriangleright}\overset{\varnothing}{A}_{\blacktriangleright}$ est partiellement valide car il existe un pont

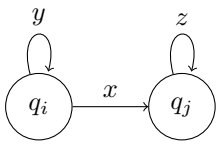


FIG. 4.16 – Un pont

pour $\{\{B\},\{C\},\{C\},\{D\},\{A\}\}_{\mathcal{M}}$ dont l'état source est 2 et l'état destination est 1. En effet, $\{B\}$ satisfait **true** et tous les autres éléments du multi-ensemble satisfont les boucles de l'état 1 et 2 à savoir $\neg B$.

Troisième optimisation Cette troisième optimisation très simple permet de détecter des multi-ensembles qui ne sont pas valides sans devoir effectuer de permutations. En effet, pour s'assurer qu'un multi ensemble M n'est pas valide par rapport à σ, il suffit que $\exists e \in M : \forall (q,e,q') \in \delta : q \notin Status(\mathcal{AC},\sigma)$. Intuitivement, si un des éléments de M ne satisfait aucune transition à partir des états atteints par σ, il existe donc au moins une permutation p de M (celle(s) qui commence(nt) par cet élément) telle que σp n'accédera à aucun état.

Par exemple, après la réalisation de la tâche A, puis de la tâche B, le statut de l'automate de la figure 4.15 est composé de l'état 1. Sans tester chaque permutation de l'ensemble des tâches en cours de réalisation, on peut déterminer que l'exécution partielle $\overset{\varnothing}{A}_{\blacktriangleright}\overset{\varnothing}{A}_{\blacksquare}\overset{\varnothing}{B}_{\blacktriangleright}\overset{\varnothing}{B}_{\blacksquare}\overset{\varnothing}{C}_{\blacktriangleright}\overset{\varnothing}{C}_{\blacktriangleright}\overset{\varnothing}{C}_{\blacktriangleright}\overset{\varnothing}{D}_{\blacktriangleright}\overset{\varnothing}{B}_{\blacktriangleright}$ est invalide car un élément de $\{\{C\},\{C\},\{C\},\{D\},\{B\}\}_{\mathcal{M}}$, à savoir $\{B\}$ ne satisfait pas la seule transition qui part de l'état 1 à savoir $\neg B$.

Mémoire cache Enfin, il est également utile de garder en mémoire les résultats obtenus pour chaque combinaison multi-ensemble - statut rencontré et ce dans le but d'éviter des (re)calculs inutiles. On tiendra compte de l'intéressante propriété qui suit.

Propriété 4.2.1 *Soit*
- *un automate* $\mathcal{AC} = (\mathcal{A}, Q, \delta, Q^0, F)$
- $M, M' \in \mathcal{M}(\mathcal{A}) : M' \subseteq_{\mathcal{M}} M$
- $\sigma, \sigma' \in \Sigma^*(\mathcal{A}) : Status(\mathcal{AC},\sigma) \subseteq Status(\mathcal{AC},\sigma')$,

M est valide par rapport à $\sigma \Rightarrow M'$ est valide par rapport à σ'.

En effet, soit un multi-ensemble M qui est valide par rapport à une séquence σ, cela signifie intuitivement qu'à partir du statut de l'automate pour σ, toutes les permutations de M permettent d'atteindre au moins un état. Si on considère un ensemble $M' \subseteq_{\mathcal{M}} M$, chaque permutation de M' est nécessairement un préfixe d'une permutation de M et permet donc également d'atteindre au moins un état à partir du statut de l'automate pour σ. De plus, cette propriété reste logiquement vraie pour n'importe quel ensemble d'états qui est un sous-ensemble de ce statut.

Grâce aux optimisations présentées ci-dessus et à la mémorisation des résultats, et bien qu'il existe certainement d'autres solutions plus complètes, les

performances affichées en pratique par l'outil *Saturn* sont excellentes puisque, pour tous les tests que nous avons effectués, nous n'avons jamais constaté de temps de réponse supérieur à la seconde, même lorsqu'il fallait gérer un très grand nombre de tâches en cours de réalisation.

Chapitre 5

Conclusion et perspectives

Nous avons introduit, au cours du chapitre 1, la problématique liée à la modélisation de processus métiers par des *workflow*. Nous avons mis en évidence les limites des langages de spécification de *workflow* actuellement utilisés et nous avons pu constater que ces limites étaient inhérentes à la nature impérative de ces langages. Dans cette optique, nous nous sommes fixé comme objectif d'explorer et développer une approche originale, de nature déclarative, pour la modélisation de processus métiers, l'idée principale étant de décrire ceux-ci au moyen de contraintes.

Dans ce contexte, nous nous sommes intéressés à la logique temporelle linéaire (LTL) qui fournit une base formelle pour l'expression de ces contraintes. Nous avons présenté en détail la logique temporelle linéaire au cours du chapitre 2. En outre, nous avons montré comment l'on pouvait convertir des expressions LTL en automates et nous avons adapté la sémantique de ces expressions aux séquences finies.

Nous avons ensuite défini, au cours du chaptire 3 un nouveau formalisme permettant de modéliser des processus métiers de manière déclarative. Pour ce faire, nous avons présenté le langage *Saturn*, basé sur LTL, et nous avons défini comment les *workflow* exprimés à l'aide de ce langage, les *workflow Saturn*, pouvaient être interprétés.

Enfin, nous avons présenté, au cours du chapitre 4, un système appelé moteur de *workflow Saturn* qui permet de contrôler dynamiquement l'exécution de processus métiers exprimés sous forme de *workflow Saturn*. Nous avons également pu identifier les problèmes de performances que rencontrait un tel système et donner des solutions pragmatiques permettant d'obtenir des performances satisfaites, rendant ce système pertinent dans la pratique.

Pour conclure ce mémoire, nous allons rappeler les contributions que nous avons pu apporter à ce domaine de recherche aussi jeune que prometteur qu'est la modélisation déclarative de processus métiers. Nous analyserons ensuite les pistes qui pourront être explorées par la suite et les perspectives que l'on peut envisager.

Contributions

Les résultats de ce mémoire s'inscrivent dans la continuité de la recherche entamée dans le cadre du projet Declare ([10, 11, 12, 13]). Dans ce contexte, l'apport du formalisme que nous avons décrit est très important. En effet, nous avons décrit une méthode déclarative permettant de modéliser des processus métiers...

... constitués de tâches asynchrones Nous avons défini pour la première fois un formalisme qui gère des processus métiers dont les tâches peuvent se réaliser en parallèle tout en garantissant que la réalisation d'une tâche puisse réussir ou échouer, et ce à n'importe quel moment.

... sensibles à un environnement Grâce à l'intégration de conditions, ce formalisme gère des processus métiers dont le déroulement peut être affecté par l'état de son environnement, augmentant par la même occasion l'expressivité du langage *Saturn*.

... hiérarchiques Il arrive fréquemment que certaines tâches d'un processus métier puissent être exprimées sous la forme d'un autre processus métier. Nous avons donc décrit pour la première fois un formalisme déclaratif capable de modéliser de manière hiérarchique des processus métiers.

La description de l'implémentation de notre outil *Saturn*[1] prouve qu'il est possible de gérer l'exécution de processus métiers conformément à ce nouveau formalisme. De plus, nous avons apporté des solutions originales et pragmatiques qui permettent à cet outil de fonctionner de manière performante pour des *workflow* de taille concrète, en faisant par la même occasion évoluer la recherche autour de la génération d'automates à partir d'expressions LTL. De manière générale, ce mémoire prouve qu'il est possible d'utiliser une approche déclarative pour modéliser des processus métiers complexes.

Perspectives

Il est important de remarquer que le formalisme déclaratif que nous avons présenté ne remplace pas les formalismes impératifs existants. En effet, le choix du type de formalisme utilisé pour modéliser un processus métier dépend de la nature de ce processus et de la flexibilité qu'on souhaite lui donner. Certaines relations ou dépendances sont plus faciles à décrire de manière impérative quand d'autres sont plus simples à exprimer de manière déclarative. Comme l'illustre la figure 1.3 du chapitre 1, ces approches sont donc plus complémentaires que concurrentes. Ainsi, une des pistes les plus intéressantes à exploiter est l'intégration des deux approches au sein d'un même système. Notons que l'outil *Saturn* a été développé dans ce sens puisque son moteur partage la même interface que celle développée pour le langage YAWL. Il est ainsi déjà possible d'intégrer les deux approches dans le cadre de processus métiers hiérarchiques (les sous-*workflow* pouvant être exprimés dans un formalisme de type différent de celui de leur *workflow* parent).

Le langage de spécifications de *workflow* *Saturn* n'a pas encore fait l'objet d'une analyse pointue pour déterminer son expressivité. Ainsi, il reste à créer de

[1] utilisé actuellement par l'entreprise Mission Critical

nombreux modèles de contraintes utiles, notamment des modèles de type conditionnel. De plus, même si l'on est confiant concernant l'expressivité du langage, on a peu de références par rapport à des exemples[1] d'utilisation dans des cas concrets. Il sera également intéressant de produire une méthodologie de développement de *workflow* *Saturn*. Enfin, nous pensons qu'un outil de support qui permettrait d'une manière simple d'exprimer les contraintes *Saturn* en langage naturel[2] serait bénéfique pour faciliter le développement de *workflow* qui n'est pas toujours très intuitif en utilisant la méthode graphique.

Les problèmes de performances constituaient le principal risque lié au développement d'une approche déclarative pour la spécification et l'exécution de processus métiers. Si les performances affichées pour l'outil que nous avons développé permettent d'ores et déjà de traiter des processus métiers de taille raisonnable, elles restent toujours un souci dans le cas de processus métiers de très grande taille. Heureusement, nous sommes encore loin d'avoir exploité toutes nos ressources et de nombreuses pistes de réflexion apparaissent. Nous n'avons pas encore utilisé de simplifications des *workflow* *Saturn* (il est clair que la conjonction de certaines contraintes provoquent des redondances) ni des expressions LTL. Comme nous l'avons déjà signalé, l'utilisation d'heuristiques (correctes, mais éventuellement incomplètes) peut réduire fortement le temps de calcul nécessaire à la génération des automates à partir des expressions LTL. Nous sommes d'ailleurs convaincus qu'il est encore possible d'optimiser cette dernière. En ce qui concerne les performances liées à la phase d'exécution, il serait intéressant d'aborder le problème de manière plus fondamentale bien que les solutions pragmatiques que nous avons présentées permettent déjà d'obtenir des résultats satisfaisants.

[1] Un processus métier réel a été modélisé en *Saturn* lors du stage chez Mission Critical et a donné des résultats satisfaisants

[2] En quelque sorte l'outil *saturn2english* inversé

Bibliographie

[1] www.missioncriticalit.com/.

[2] Z. Somogyi, F. J. Henderson, and T. C. Conway. Mercury, an efficient purely declarative logic programming language. In *In Proceedings of the Australian Computer Science Conference*, pages 499–512, 1995.

[3] www.cs.mu.oz.au/research/mercury/.

[4] Michel Vanden Bossche. High-quality and predictable mission-critical software. Mission Critical IT : 2001.

[5] Van Der Aalst, L. Aldred, M. Dumas, and A. H. M. Ter Hofstede. Design and implementation of the yawl system. In *In : Proc. of CAiSE*, 2004.

[6] Diimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management : From process modeling to workflow automation infrastructure, 1995.

[7] Wil Van Der Aalst, Kees Van Hee, Prof. Dr. Kees, Max Hee, Remmert Remmerts De Vries, Jaap Rigter, Eric Verbeek, and Marc Voorhoeve. Workflow management : Models, methods, and systems, 2002.

[8] Van Der Aalst. The application of petri nets to workflow management, 1998.

[9] Version May Authors, Francisco Curbera Ibm, Hitesh Dholakia, Yaron Goland Bea, Johannes Klein Microsoft, Frank Leymann Ibm, Kevin Liu Sap, Dieter Roller Ibm, Doug Smith, Siebel Systems, Siebel Systems, Satish Thatte, Ivana Trickovic Sap, and Sanjiva Weerawarana Ibm. Business process execution language for web services, 2003.

[10] M. Pesic, W.M.P. van der Aalst, M.H. Schonenberg, and N. Sidorova. Constraint-based workflow models : Change made easy.

[11] M. Pesic. Decserflow : Towards a truly declarative service flow language. pages 1–23, 2006.

[12] M. Pesic and W. van der Aalst. A declarative approach for flexible business processes management. *Business Process Management Workshops*, pages 169–180, 2006.

[13] W. M. P. van der Aalst and M. Pesic. Specifying, discovering, and monitoring service flows : Making web services process-aware. *BPM Center Report BPM-06-09, BPM Center*, 2006.

[14] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts and London, UK, 1999.

[15] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13 :45–60.

[16] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In *POPL '81 : Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 164–176, New York, NY, USA, 1981. ACM.

[17] Edmund Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logics of Programs*, 1981.

[18] Denis Oddoux. *Utilisation des automates alternants pour un model-checking efficace des logiques temporelles linéaires.* Thèse de doctorat, Lab. Informatique Algorithmique : Fondements et Applications, Université Paris 7, France, December 2003.

[19] Edmund M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 428–437, London, UK, 1989. Springer-Verlag.

[20] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8 :244–263, 1986.

[21] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[22] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[23] Ekkart Kindler, Humboldt universitat Zu Berlin, and Unter Den Linden. Safety and liveness properties : A survey.

[24] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5) :279–295, 1997.

[25] Orna Bernholtz, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. In *Computer Aided Verification, Proc. 6th Int. Workshop*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155, Stanford, California, June 1994. Springer-Verlag.

[26] Pierre Wolper. Constructing automata from temporal logic formulas : A tutorial. In *Lectures on Formal Methods in Performance Analysis (First EEF/Euro Summer School on Trends in Computer Science)*, volume 2090 of *Lecture Notes in Computer Science*, pages 261–277. Springer-Verlag, July 2001.

[27] J. R. Buchi. On a decision method in restricted second-order arithmetic. In *Proc. 1960 Int. Congr. for Logic, Methodology, and Philosophy of Science*, pages 1–12. Stanford Univ. Press, 1962.

[28] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3) :733–749, 1985.

[29] Pierre Wolper. Constructing automata from temporal logic formulas : A tutorial.

[30] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *Proc. 24th IEEE Symposium on Foundations of Computer Science*, pages 185–194, Tucson, 1983.

[31] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1) :1–37, November 1994.

[32] Franz Baader and Ulrike Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69 :2001, 2000.

[33] Pierre Wolper. The tableau method for temporal logic : An overview. *Logique et Analyse*, (110–111) :119–136, 1985.

[34] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th Work. Protocol Specification, Testing, and Verification*, Warsaw, June 1995. North-Holland.

[35] Marco Daniele, Fausto Giunchiglia, and Moshe Y. Vardi. Improved automata generation for linear temporal logic. In *Computer Aided Verification*, pages 249–260, 1999.

[36] Kousha Etessami and Gerard J. Holzmann. Optimizing büchi automata. pages 153–167. Springer, 2000.

[37] Fabio Somenzi and Roderick Bloem. Efficient büchi automata from ltl formulae. In *CAV 2000, LNCS 1855 :247Ũ263*. Springer-Verlag, 2000.

[38] Dimitra Giannakopoulou and Flavio Lerda. From states to transitions : Improving translation of ltl formulae to buchi automata. In *In Proc. FORTEŠ02., volume 2529 of LNCS*, pages 308–326. Springer, 2002.

[39] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, July 2001. Springer.

[40] Heikki Tauriainen. A randomized testbench for algorithms translating linear temporal logic formulae into Büchi automata. In H.-D. Burkhard, L. Czaja, H.-S. Nguyen, and P. Starke, editors, *Proceedings of the Concurrency, Specification and Programming Workshop (CS&P'99), Warsaw, Poland, September 1999*, pages 251–262, 1999.

[41] Dimitra Giannakopoulou and Klaus Havelund. Automata-based verification of temporal properties on running programs. In *ASE '01 : Proceedings of the 16th IEEE international conference on Automated software engineering*, page 412, Washington, DC, USA, 2001. IEEE Computer Society.

[42] Klaus Havelund and Grigore Rosu. Testing linear temporal logic formulae on finite execution traces. Technical report, 2001.

[43] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Eagle does space efficient ltl monitoring. Technical report, 2003.

[44] Qss Recom Technologies, Grigore Rosu, Grigore Rosu Riacs, Klaus Havelund, Klaus Havelund, and Klaus Havelund. Monitoring java programs with java pathexplorer. In *In Proceedings of Runtime Verification (RVŠ01*, pages 97–114. Elsevier, 2001.

[45] Klaus Havelund Kestrel, Klaus Havelund, Kestrel Technology, and Grigore Rosu. Monitoring programs using rewriting. In *In Proceedings, International Conference on Automated Software Engineering (ASES01*, pages 135–143. IEEE, 2001.

[46] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 342–356, 2002.

[47] Volker Stolz and Eric Bodden. Temporal assertions using aspectj, 2005.

[48] A. Bauer, M. Leucker, and Schallhart C. The good, the bad, and the uglyŮbut how ugly is ugly ? In *In Proceedings of the 7th International Workshop on Runtime Verification*, Vancouver, Canada, 2007. Springer.

Annexe A

Annexe

A.1 Rappel des notions et notations principales

Voici un récapitulatif de notions et notations abordées au cours de ce mémoire. La signification, parfois intuitive, de chacune d'entre elles est donnée ainsi que la référence vers la définition formelle et complète.

Séquences

Nom	Notation usuelle	Signification	Référence	Page(s)
séquence	$\sigma = \langle e_1, e_2, \ldots \rangle$	Liste ordonnée d'éléments dans laquelle un même élément peut apparaître plusieurs fois. Une séquence peut être finie ou infinie.		16
longueur d'une séquence	$\lvert\sigma\rvert$	longueur de la séquence σ.		16
i-ème élément d'une séquence	$\sigma(i)$	$\sigma(i)$ est le i-ème élément d'une séquence σ et i est son indice.		16
suffixe d'une séquence	σ^i	σ^i représente le suffixe de σ obtenu en supprimant ses i premiers éléments.		16
concaténation	$\sigma\sigma'$	séquence obtenue en mettant bout à bout les séquences σ et σ'		16
	$\sigma \in \Sigma^*(E)$	σ est une séquence finie composée d'éléments de l'ensemble E.		17
	$\sigma \in \Sigma^\omega(E)$	σ est une séquence infinie composée d'éléments de l'ensemble E.		17
sous-séquence	σ'	On obtient la sous-séquence d'une séquence σ en retirant certains éléments de cette séquence (sans modifier l'ordre des éléments)		55

indices d'une sous-séquence	$\mathcal{IND}(\sigma, \sigma')$	Soit σ' une sous-séquence de σ, $\mathcal{IND}(\sigma, \sigma')$ désigne l'ensemble des indices des éléments de σ qui forment la sous-séquence σ'. Par exemple, si $\sigma = \langle c, a, b, d \rangle$ et $\sigma' = \langle c, d \rangle$, $\mathcal{IND}(\sigma, \sigma') = \{1,4\}$.		55
restriction	$\chi(\sigma, E)$	La restriction d'une séquence σ pour l'ensemble E est la plus longue sous-séquence de σ telle que chacun de ses éléments fait partie de E. Par exemple, $\chi(\langle c, a, b \rangle, \{a, b\}) = \langle a, b \rangle$.		55
partition d'une séquence		Un ensemble de séquences forme une partition de σ si un entrelacement de ces séquences permet de former la séquence σ.		66

Logique temporelle

Nom	**Notation usuelle**	**Signification**	**Référence**	**Page(s)**
configuration		Une configuration d'un système composé d'un ensemble de propositions P est un sous-ensemble P' de propositions de P. Une configuration permet de représenter l'état du système à un moment donné.		11
ensemble des configurations possibles	2^P	Ensemble de tous les sous-ensembles de P.		11
logique temporelle		La logique temporelle est un formalisme qui permet d'exprimer l'evolution de l'état d'un système.		12
logique temporelle linéaire	LTL	LTL modélise le temps de façon linéaire et permet d'exprimer des propriétés sur des séquences de configurations.		13

logique en temps arborescent	CTL	CTL modélise le temps commme une structure en arbre.		12
opérateurs temporels	$\diamond$, $\circ$, $\Box$, $\mathbb{U}$ et $\mathbb{R}$	Les opérateurs temporels permettent d'étendre la logique propositionnelle pour former une logique temporelle.		12
expression LTL	φ	Expressions formées à l'aide de propositions, d'opérateurs classiques de logique propositionnelle ($\neg$, $\wedge$, $\vee$) et d'opérateurs temporels ($\diamond$, $\circ$, $\Box$, $\mathbb{U}$ et $\mathbb{R}$). Exemple : $\Box(p \vee q)$	Déf. 2.1.1	17
ensemble d'expressions LTL	$\mathcal{LTL}(P)$	Ensemble de toutes les expressions LTL faisant intervenir des propositions de P.	Déf. 2.1.1	17
opérateur Suivant	$\circ\varphi$	φ doit être vraie à l'instant suivant.	Tab. 2.1	16
opérateur Finalement	$\diamond\varphi$	φ doit être vraie à (au moins) un instant de la séquence.	Tab. 2.1	16
opérateur Toujours	$\Box\varphi$	φ doit être vraie tout au long de la séquence.	Tab. 2.1	16
opérateur Jusqu'à	$\varphi_1 \ \mathbb{U} \ \varphi_2$	φ_2 doit être vraie à (au moins) un instant de la séquence. φ_1 doit être vraie jusqu'à cet instant (non compris).	Tab. 2.1	16
opérateur Libération	$\varphi_1 \ \mathbb{R} \ \varphi_2$	φ_2 doit être vraie jusqu'au premier instant où φ_1 est vraie, celui-ci y compris. Si un tel instant n'existe pas, φ_2 doit être vraie indéfiniment.	Tab. 2.1	16
relation $\vDash_\omega$	$\sigma \vDash_\omega \varphi$	La séquence infinie σ satisfait l'expression LTL φ.	Déf. 2.1.2	17
relation $\vDash_*$	$\sigma \vDash_* \varphi$	La séquence finie σ satisfait l'expression LTL φ.	Déf. 2.4.1	37

langage pour $\vDash_\omega$	$\mathcal{L}_{\vDash_\omega}(\varphi)$	Ensemble des séquences infinies qui satisfont l'expression LTL φ.	Déf. 2.1.3	18
langage pour $\vDash_*$	$\mathcal{L}_{\vDash_*}(\varphi)$	Ensemble des séquences finies qui satisfont l'expression LTL φ.	Déf. 2.4.2	38
taille LTL	$\lvert\varphi\rvert$	Représente la taille d'une expression LTL φ.	Déf. 2.3.1	27
forme normale négative		Une expression LTL est en forme normale négative si elle est exprimée uniquement à l'aide des opérateurs temporels $\circ$, $\mathbb{R}$ et $\mathbb{U}$ et si la négation (opérateur $\neg$) n'est appliquée qu'à des propositions.	Déf. 2.3.2	28

Automates

Nom	**Notation usuelle**	**Signification**	**Référence**	**Page(s)**
automate fini		Machine abstraite constituée d'un nombre fini d'états et de transitions entre ces états. Son comportement est dirigé par une séquence fournie en entrée : l'automate passe d'état en état, suivant les transitions, à la lecture successive de chaque élément de la séquence fournie en entrée.		21
automate classique	$\mathcal{AC} = (\mathcal{A}, Q, \delta, Q^0, F)$	Automate fini défini pour des séquences finies tel que : – $\mathcal{A}$ est l'alphabet (fini) – Q est l'ensemble (fini) d'états – $\delta \subseteq Q \times \mathcal{A} \times Q$ est la relation de transition – $Q^0 \subseteq Q$ est l'ensemble des états initiaux – $F \subseteq Q$ est l'ensemble des états finaux		21

automate de Büchi	$\mathcal{AB} = (\mathcal{A}, Q, \delta, Q^0, F)$	Automate fini défini pour des séquences infinies tel que : – $\mathcal{A}$ est l'alphabet (fini) – Q est l'ensemble (fini) d'états – $\delta \subseteq Q \times \mathcal{A} \times Q$ est la relation de transition – $Q^0 \subseteq Q$ est l'ensemble des états initiaux – $F \subseteq Q$ est l'ensemble des états accepteurs		23
automate de Büchi généralisé	$\mathcal{ABG} = (\mathcal{A}, Q, \delta, Q^0, \mathcal{F})$	Automate fini défini pour des séquences infinies tel que : – $\mathcal{A}$ est l'alphabet (fini) – Q est l'ensemble (fini) d'états – $\delta \subseteq Q \times \mathcal{A} \times Q$ est la relation de transition – $Q^0 \subseteq Q$ est l'ensemble des états initiaux – $\mathcal{F} \subseteq 2^Q$ est l'ensemble des ensembles d'états accepteurs		23
statut d'un automate	$Status(\mathcal{AC}, \sigma)$	Le statut de l'automate $\mathcal{AC}$ pour une séquence finie σ désigne l'ensemble des états de l'automate qui sont accessibles (en utilisant la relation de transition à partir des états initiaux) via σ.		22
séquence acceptée par un automate		Une séquence finie est acceptée par un automate classique si elle peut conduire à un état final. Une séquence infinie est acceptée par un automate de Büchi si elle visite infiniment souvent un état accepteur. Une séquence infinie est acceptée par un automate de Büchi généralisé si elle visite infiniment souvent un état de chaque ensemble d'états accepteurs.		22, 23 et 23

langage accepté par un automate		Ensemble des séquences acceptées par un automate		22, 23 et 24
algorithme en 3 phases		Algorithme permettant de construire un automate qui reconnaît le même langage qu'une expression LTL donnée.		27
état de type rouge		Etat non final n'offrant aucune possibilité d'atteindre un état final (via la relation de transition).		40
état de type orange		Etat non final offrant la possibilité d'atteindre un état final (via la relation de transition).		40
état de type vert		Synonyme d'état final.		40

Saturn

Nom	**Notation usuelle**	**Signification**	**Référence**	**Page(s)**
processus métier	PM	Un processus métier désigne un ensemble de tâches qui s'enchaînent de manière chronologique pour atteindre un objectif qui, dans le contexte d'une organisation de travail, est généralement la fourniture d'un service ou la production d'un bien.		5
workflow Saturn		Un *workflow Saturn* est constitué d'un ensemble de tâches, d'un ensemble de conditions et d'un ensemble de contraintes *Saturn*. Une contrainte *Saturn* est défini par une expression LTL.	déf. 3.2.2	53

langage *Saturn*		Le langage *Saturn* est constitué d'un ensemble de modèles de contraintres *Saturn* qui permettent d'exprimer facilement des contraintes *Saturn*.	sec. 3.1	44
PM constitué de tâches synchrones		Processus métier constitué d'un ensemble de tâches réalisées à tour de rôle et sans annulation		52
PM constitué de tâches asynchrones		Processus métier constitué d'un ensemble de tâches qui peuvent se réaliser en parallèle et être annulées.		55
PM sensible à un environnement		Processus métier dont l'exécution est influencée par son environnement.		60
PM hiérarchique		Certaines tâches du processus métier sont représentées par un autre processus métier.		63
exécution d'un processus métier	σ	Séquence représentant l'ordre dans lequel les tâches d'un processus métier sont réalisées. La forme de l'exécution peut varier selon le type de processus métier considéré.	Déf. 3.2.1, 3.2.6 et 3.2.12	53, 56 et 61
validité d'une exécution	σ	Une exécution est validée par un *workflow Saturn* si elle respecte ses contraintes.	Déf. 3.2.3, 3.2.10, 3.2.17 et 3.2.22	54, 59, 62 et 67

tâches en cours de réalisation	$running\_tasks(\sigma)$	Désigne les tâches en cours de réalisation pour σ.	Déf. 3.2.7 et 3.2.14	58 et 61
séquence de tâches effectuées	$exec\_order(\sigma)$	Désigne la séquence constituée des tâches complètement réalisées durant l'exécution σ.	Déf. 3.2.8 et 3.2.15	58 et 62
tâches disponibles	$available(\sigma, \varphi)$	Ensemble de tâches que l'on peut réaliser après l'exécution σ sans violer l'expression φ.	Déf. 3.2.9 et 3.2.16	59 et 62

Multi-ensembles

Nom	Notation usuelle	Signification	Référence	Page(s)
multi-ensemble	$M = (E, m) = \{e_i, e_i, e_j, \ldots\}_{\mathcal{M}}$	Un multi-ensemble peut être vu comme un ensemble pouvant contenir plusieurs fois les mêmes éléments. Formellement, c'est un couple (E, m) où E est un ensemble appelé support et $m : E \to \mathbb{N}$ est une fonction appelée multiplicité. La multiplicité renvoie pour chaque élément du support le nombre d'occurrences de celui-ci dans le multi-ensemble.		57
opérateurs	$M \cup_{\mathcal{M}} N$	Généralise l'opérateur $\cup$ pour les multi-ensembles.		57
	$M \setminus_{\mathcal{M}} N$	Généralise l'opérateur $\setminus$ pour les multi-ensembles.		57
	$M \subseteq_{\mathcal{M}} N$	Généralise l'opérateur $\subseteq$ pour les multi-ensembles.		57
	$e \in M$	Généralise l'opérateur $\in$ pour les multi-ensembles.		57
cardinalité	$\#M$	Nombre total d'éléments du multi-ensemble.		58
permutation	$p \in Perm(M)$	Généralise le concept de permutation pour les multi-ensembles. $Perm(M)$ représente l'ensemble de toutes les permutations d'un multi-ensemble M. Par exemple, $Perm(\{a, a, b\}_{\mathcal{M}}) = \{\langle a, a, b\rangle, \langle a, b, a\rangle, \langle b, a, a\rangle\}$		58

A.2 Témoignage d'expérience du langage *Mercury*

Voici un petit texte qui témoigne de mon expérience du langage destiné à la mailing-list des utilisateurs de *Mercury* C'est dans le cadre de mon stage de fin d'étude (2ème Maitrise) dans la faculté universitaire d'Informatique de Namur (FUNDP) que j'ai eu l'occasion de travailler chez Mission Critical. Ma formation universitaire couvre un apprentissage poussé dans les langages orientés-objets (Java), procéduraux impératifs (C - Pascal) et de base de données (SQL), des connaissances couvrant le domaine des réseaux et des systèmes d'exploitation, quelques cours de sciences économiques et une solide formation mathématique. Ma formation comprenait également un cours de 30 heures de programmation fonctionnelle et logique donnant un aperçu des bases de ce type de langage. Les langages utilisés pour introduire les concepts étaient Haskell et Prolog. C'est le seul cours où ce type de langage est introduit et aucun projet de taille importante n'a été réalisé dans ces langages. Mon expérience en programmation fonctionnelle et logique était donc limitée à un aperçu théorique et à quelques exercices concrets de petite taille.

En arrivant chez Mission Critical, étant donné que mon stage couvre une partie d'implémentation, j'ai dû apprendre le langage *Mercury*. Il est important de signaler qu'il ne s'agissait pas de l'objectif principal : la recherche d'un sujet (Définition de *workflow* par une approche déclarative), la documentation (lecture d'articles scientifiques, thèses liées au sujet,...), l'apprentissage des concepts propres au sujet (*workflow*, logique temporelle,...), la conception et la recherche théorique occupaient la majorité de mon temps et c'est parallèlement à ces activités que j'ai appris à utiliser *Mercury*.

La principale difficulté quand on commence à utiliser un langage de ce type vient du fait que l'on est habitué à raisonner en terme impératif et orienté-objet. Prendre l'habitude de raisonner en terme de "que veut-on obtenir" plutôt que de "comment va-t-on faire" demande une petite période d'adaptation. Comprendre les mécanismes de base de *Mercury* demande également un certain effort. Mon apprentissage s'est fait naturellement au fur et à mesure du développement et des nécessités que celui-ci demandait. Je n'ai pas du tout trouvé contraignant de devoir apprendre ce langage parallèlement aux autres activités du stage, grâce notamment au support des employés de Mission Critical qui étaient toujours disponibles pour m'aider dans mon apprentissage.

Dès les premières semaines, j'ai été capable de développer des algorithmes déjà complexes sans être retardé par l'apprentissage de ce nouveau langage. Au bout de 12 semaines, l'écriture en *Mercury* est devenue complètement naturelle, instinctive et rapide. Sans avoir exploité toutes les possibilités du langage, je pense que mon niveau dans ce langage a rattrapé celui que j'avais dans d'autres langages (Java).

Le projet que j'ai réalisé chez Mission Critical étant déjà un projet d'une taille relativement importante et comprenant certains algorithmes et structures assez complexes, je pense pouvoir donner les premières conclusions sur le langage *Mercury*. Sa grande force est bien sûr le fait que ce soit un langage déclaratif.

Mercury permet d'écrire des programmes de façon concise et sûre. Le débogage est plus facile, l'écriture est plus rapide et la relecture est particulièrement plus aisée que dans des langages comme Java dont la structure, sur de gros projets, ressemble rapidement à un labyrinthe. Cette particularité s'exprime bien par le comparatif de mon programme avec un programme équivalent réalisé en Java. Le coeur de mon projet est composé de 5 fichiers et d'environ 1700 lignes. Une estimation de la partie correspondante en Java est de 150 fichiers pour 18 000 lignes. Je pense que les programmes réalisés en *Mercury* sont également plus facilement rendus robustes. Si je devais choisir un langage parmi ceux que je connais pour réaliser un programme demandant une certaine part de complexité "logique", c'est *Mercury* que je choisirais sans hésiter. En fait, les difficultés que j'ai éprouvées lors du développement sont apparues exclusivement quand je devais programmer des modules de nature plus "impérative" comme un petit outil d'exécution en ligne de commande. C'est uniquement dans ce genre de cas que le développement en *Mercury* m'a semblé moins intuitif, moins adapté. Les rares points faibles qui pourraient encore être cités sont le manque de documentation et de librairies comparativement à des langages plus célèbres ainsi que la réalisation d'interface utilisateur. Mais mon impression générale est très positive puisque si je devais recommencer ce projet, je ne voudrais absolument pas utiliser un autre langage.