الجمـــهوريـــة الجـــزائريـــة الديـمقـراطيـــــــة الشـعبيـــــة
REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
وزارة التعـلـيـم الـعـالـي والـبـحـث الـعـلـمي
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE
جـامـعـة عمّـار ثـلـيـجـي بالأغـواط
UNIVERSITE AMAR TELIDJI LAGHOUAT
كـلـيـــة الـعـلـــوم
FACULTE DES SCIENCES
DEPARTEMENT DE MATHEMATIQUES ET INFORMATIQUE

Mémoire de MASTER

Domain : Mathématiques et Informatique
Filière : Informatique
Option : Systèmes d'Information et de Décision

Par:
Mr. HOUACHE Daoud
Mr. MEHARZI Nassredine

THEME

Vérification et validation des diagrammes UML avec Spin

Soutenu publiquement devant le jury composé de :

Mr. Bendouma Tahar	*M.C.(B)*	*Président*
Mr. Chellama Laâredj	*M.A.(A)*	*Examinateur*
Mr. Guellouma Younes	*M.A.(A)*	*Examinateur*
Mme. Belabbaci Amel	*M.A.(A)*	*Encadreur*

Année Universitaire 2014/2015

Remerciement

En tout premier lieu, nous remercions Allah le tout puissant de nous avoir permis de mener à terme ce mémoire de fin d'étude.

En préambule à ce mémoire, nous souhaitons adresser notre profonde gratitude à notre tutrice M^me^ BELABBACI Amel pour l'aide qu'elle nous a apporté et sa contribution à l'élaboration de ce mémoire. Elle était toujours à l'écoute et s'est montrée très disponible tout au long de la réalisation de notre travail. C'était pour nous un immense honneur de travailler avec elle durant notre projet de fin d'études et de partager sa patience, ses expertises, et surtout ses précieux conseils.

Une immense pensée s'adresse à nos parents, pour leur grand soutien. Nous remercions également notre famille, tous nos amis, et tous ceux qui ont participé de près ou de loin à la réalisation de ce travail.

Enfin nous remercions les membres du jury qui ont bien accepté d'évaluer ce modeste travail, et ce malgré leur lourdes et exaltantes responsabilités.

Résumé

UML est largement utilisé pour la modélisation objet. Mais le manque de processus de la vérification rend difficile d'assurer que ces modèles sont corrects. Dans ce mémoire on va citer quelque méthode de vérification de ces modèles.

L'objective de ce mémoire est de faire une exploitation de l'outil Spin pour la vérification de ces modèles, et plus précisément les diagrammes de séquences et d'activités.

Mots-clés : vérification, diagramme de séquences et d'activités, exploitation de l'outil Spin.

Abstract

UML is widely used for object modeling. But the lack of verification process make it difficult to ensure that these models are correct. In this master's thesis, we will cite some verification method of these models.

The objective of this master's thesis is to make an exploitation of the Spin tool for the verification of these models, specifically the sequence diagrams and activity.

Keywords: verification, sequence diagram and activity, exploitation of the Spin tool.

Table des Matières :

Liste des Figures :

Liste des Tableaux :

Introduction générale :

Présentation

et

but du travail

UML (Unified Modeling Language) est un langage de modélisation développé pour modéliser un système en orienté-objet (OO). C'est la fusion des méthodes objets dominantes (OMT, Booch et OOSE), et qui a été normalisé par l'OMG (Object Management Group) en 1997. En quelques années, UML s'est imposé comme standard à utiliser en tant que langage de modélisation objet [1].
Bien qu'UML soit un langage riche, largement utilisé (il peut couvrir toutes les phases d'un cycle de développement), les modèles UML restent toujours en besoin d'être vérifiés pour assurer que le comportement spécifié dans ces modèles sont correctes. Le manque de processus de vérification et de validation rend impossible la :

1) Vérification des diagrammes.
2) Vérification de la cohérence entre les différents diagrammes [2].

Dans ce travail, on va faire une courte étude bibliographique sur les différentes méthodes et outils proposés pour vérifier et valider les différents diagrammes UML.

Notre objectif est de faire une exploitation de l'outil Spin pour la vérification et la validation des diagrammes UML, plus précisément les diagrammes de séquences et d'activités.

En effet, ces deux diagrammes sont les plus utilisés pour la modélisation d'un système. Ils étaient largement étudiés, et plusieurs méthodes et approches de vérification et validation ont été développées. L'outil Spin a été choisi après avoir remarqué que c'est l'outil le plus utilisé pour la vérification des diagrammes UML.

Ce mémoire est organisé comme suit : Dans le premier chapitre nous présentons quelques concepts sur la V & V des diagrammes UML, et une courte étude bibliographique sur les travaux traitant la V & V.
Le deuxième chapitre sera consacré à la présentation de quelques outils utilisés pour faire la vérification.
Enfin dans le troisième chapitre nous présentons et discutons l'exploitation de l'outil Spin pour la vérification.
On termine par une conclusion générale qui résume ce que nous avons réalisé dans ce travail avec des éventuelles perspectives.
Le mémoire contient deux annexes :
L'annexe A pour présenter la syntaxe du langage PROMELA.
Dans l'annexe B, nous avons mentionné quelques erreurs détectées dans un travail sur la sémantique formelle de diagrammes de séquences [25].

Chapitre 1:
Vérification et validation des diagrammes UML

1. Introduction :

Dans ce chapitre, on va présenter une courte étude bibliographique sur la validation et la vérification des diagrammes UML.

2. Les diagrammes UML :

UML dans sa version 2 (UML 2) propose treize diagrammes, ces diagrammes sont regroupés dans deux classes : des diagrammes structurels et d'autres comportementaux [1].

2.1. Les diagrammes structurels :

Les diagrammes structurels représentent l'aspect statique d'un système et qui sont :

1- **Diagramme de classe :** C'est la représentation de la structure statique en termes de classes et de relations.

2- **Diagramme d'objet :** Il permet la représentation d'instances des classes et des liens entre instances.

3- **Diagramme de composant :** Ce diagramme représente les différents constituants du logiciel au niveau de l'implémentation d'un système.

4- **Diagramme de déploiement :** Ce diagramme décrit l'architecture technique d'un système avec une vue centrée sur la répartition des composants dans la configuration d'exploitation.

5- **Diagramme de paquetage** (nouveau dans UML 2) **:** Ce diagramme donne une vue d'ensemble du système structuré en paquetage. Chaque paquetage représente un ensemble homogène d'éléments du système (classes, composants…).

6- **Diagramme de structure composite** (nouveau dans UML 2) **:** Ce diagramme permet de décrire la structure interne d'un ensemble complexe composé par exemple de classes ou d'objets et de composants techniques. Ce diagramme met aussi l'accent sur les liens entre les sous-ensembles qui collaborent.

2.2. Les diagrammes comportementaux :

Les diagrammes comportementaux représentent l'aspect dynamique d'un système et qui sont :

1- **Diagramme de cas d'utilisation :** Ce diagramme est destiné à représenter les besoins des utilisateurs.

2- **Diagramme d'états-transition** (machine d'état) **:** Ce diagramme montre les différents états des objets en réaction aux événements.

3- **Diagramme d'activités :** Ce diagramme donne une vision des enchaînements des activités propres à une opération.

4- **Diagramme de séquences :** Ce diagramme permet de décrire les scénarios de chaque cas d'utilisation en mettant l'accent sur la chronologie des opérations en interaction avec les objets.

5- **Diagramme de communication** (anciennement appelé collaboration) **:** Ce diagramme est une autre représentation des scénarios des cas d'utilisation qui met plus l'accent sur les objets et les messages échangés.

6- **Diagramme global d'interaction** (nouveau dans UML 2) **:** Ce diagramme fournit une vue générale des interactions décrites dans le diagramme de séquences et des flots de contrôle décrits dans le diagramme d'activités.

7- **Diagramme de temps** (nouveau dans UML 2) **:** Ce diagramme permet de représenter les états et les interactions d'objets dans un contexte où le temps a une forte influence sur le comportement du système à gérer.

3. La vérification et la validation des diagrammes UML :

UML devient le langage de modélisation le plus utilisé, par conséquent nous considérons que le V & V est un sujet d'une grande importance.
La V & V cherche à répondre à la question « construisons-nous le bon modèle UML? », pour répondre à cette question on doit étudier la cohérence des diagrammes qui constituent notre modélisation. Les diagrammes utilisés dans une modélisation doivent être à leur tour vérifiés et validés. Pour répondre à la question « construisons nous un diagramme correcte ? »

Cependant ces deux questions sont très vastes pour y répondre. En effet la vérification et la validation des diagrammes UML était largement étudiée, beaucoup de méthodes et d'approches ont été proposées et chacune traite la V &V d'un diagramme et selon un point de vue bien précis.

4. Les méthodes de V & V des diagrammes UML :

Les méthodes de V &V peuvent être classées en deux classes :

1. **Des méthodes de vérification de cohérence entre diagrammes :** Certaine méthodes traitent cette vérification entre plusieurs diagrammes comme la méthode de **Checklists.** D'autres méthodes traitent la vérification de manière précise telle que la vérification de la cohérence entre diagramme de classe et diagramme de séquences.

2. **Des méthodes de vérification des diagrammes :** Ces méthodes traitent la vérification de chaque diagrammes à part.

4.1. Méthodes de vérification de cohérence entre diagrammes :

Nous présentons deux méthodes pour vérifier la cohérence entre les différents diagrammes UML :

Checklists :

Atsushi Ohnish dans [3] propose une méthode pour la vérification de la cohérence entre les diagrammes UML par des éléments de vérification selon une base de connaissances.
Cette méthode permet de vérifier six diagrammes de modèles UML: le diagramme de classes, diagramme d'état, diagramme de séquences, diagramme de collaboration, diagramme d'activités, et diagramme de cas d'utilisation.

	Diagramme de classe (C)				
Etat de transition (S)	1) Classe dans (S) et classe dans (C) 2) confirmation de classe sans diagrammes d'états transition 3) attribut définissant l'état dans (S) et attribut dans (C) 4) plage de valeurs d'attributs Dans (C) et (S) 5) Actions, activités, dans (S) et méthodes dans (C)	**Etat transition (S)**			
Diagramme de séquences (Q)	1) Objets dans (Q) et classes dans (C) 2) Messages entre objets dans (Q) et associations entre classes correspondant dans (C) Messages dans (Q) et méthodes dans (C)	1) Objet dans (Q) et classe dans (S) 2) Messages dans (Q) et actions, activités dans (S)	**Diagramme de séquences (Q)**		
Diagrammes de collaboration (L)	1) Objets dans (L) et classes dans (C) 2) Messages entre objets dans (L) et association entre classes correspondant dans (C) 3) Messages dans (L) et méthode dans (C)	1) Objets dans (L) et classe de (S) 2) Messages dans (L) et actions, activités dans (S)	1) Objets dans (L) et dans (Q) 2) Messages dans (L) et dans (Q) pour directions, séquences, source, et destination	**Diagramme Collaboration(L)**	
Diagramme de cas d'utilisation (U)	1) Acteurs dans (U) et classes dans (C) 2) cas d'utilisation dans (U) et méthodes dans (C)	1) Acteurs dans (U) et objets de (S) 2) cas d'utilisation dans (U) et actions, activités dans (S)	1) acteurs dans (u) et objets dans (Q) 2) cas d'utilisation dans (U) et messages dans (Q)	1) Acteur dans (U) et objets dans (L) 2) cas d'utilisation dans (U) et messages dans (L)	**Cas d'utilisation (U)**
Diagramme d'activités (A)	1) classes dans (A) et dans (C) 2) action dans (A) et méthodes dans (C) 3) contrôler flot entre classes (A) et associations dans (C)	1) classes dans (A) et classe de (S) 2) Actions dans (A) et actions, activités dans (S)	1) Classes dans (A) et objets dans (Q) 2) Actions dans (A) et messages dans (Q) 3) Control flot entre classes et (A) et message dans (Q)	1) classes dans (A) Et objets dans (Q) 2) Actions dans (A) et messages dans (L) 3) Control flot entre classes dans (A) et message dans (L)	1) classes dans (A) et acteurs dans (U) 2) actions dans (A) et cas d'utilisation dans (U)

Tableau 1 : Tableau de la base de connaissances

Le tableau (1) représente la base de connaissances qui permet de vérifier la cohérence entre les différents diagrammes. On peut par exemple vérifier la cohérence entre diagramme de classes (DC) et diagramme de séquences (DS) par:

1) Un objet dans DS doit exister comme classe dans DC.
2) L'existence de communication entre deux objets dans DS, est une association dans DC entre les classes correspondantes.
3) Un message reçu par objet dans DS est une méthode dans DC correspondante.

Méthode basée sur la cohérence entre D. Séquences et D. Classes :

Les grands modèles de conception contiennent des milliers d'éléments, Les concepteurs trouvent des difficultés pour vérifier la cohérence entre ces modèles.

LI, Xiaoshan et al dans [4], proposent une méthode pour vérifier et valider la cohérence entre diagrammes de classes et diagrammes de séquences.

Pour un diagramme de séquences, on peut construire un arbre de structure ordonnée correspondant dans lequel:

1. **Le nœud racine:** indique l'acteur ou l'objet de départ
2. **Les nœuds internes:** sont des objets
3. **Les arcs:** représentent les messages qui passent entre les objets.

Un message est le tuple : msg = (obi : Ci, obj : Cj , action, order) Où:

- **obi :** est l'objet source du message avec le type de classe Ci.
- **obj :** est l'objet cible du message avec le type de la classe Cj.
- **action :** est l'appel d'une méthode.
- **ordre :** est le numéro d'ordre du message de diagramme de séquences dans l'arborescence qui peut être :
  1. **Nœud racine:** départ.
  2. **Les premières branches:** u=1,2,…n de gauche à droite. Les fils d'un nœud à partir du niveau 2 sont représentés de la manière suivante : u.1, u.2, ….u.v où u c'est le n° du père.

Exemple :

Considérons l'exemple suivant [8] :

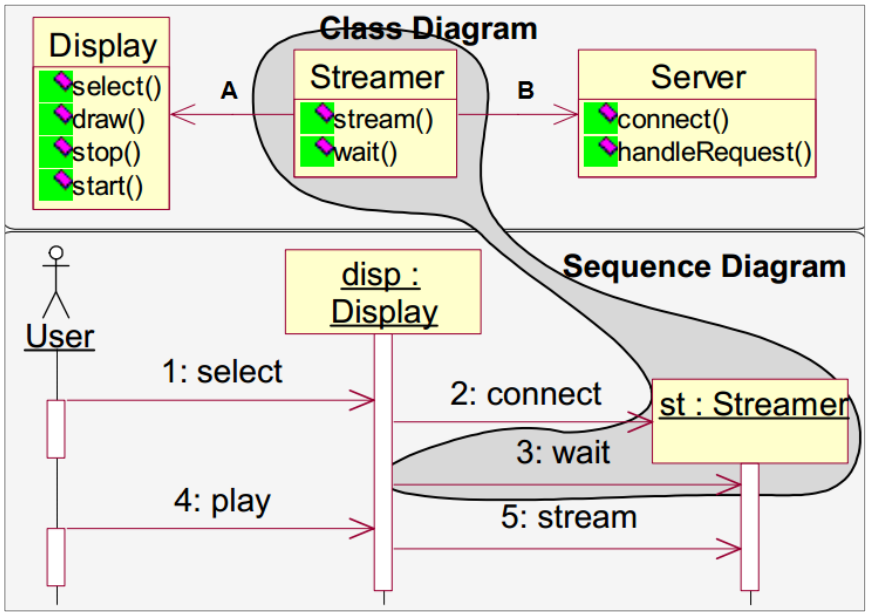

Figure 1 : Exemple de diagramme de séquence et diagramme de classes

Pour le diagramme de séquences, on a l'arbre suivant :

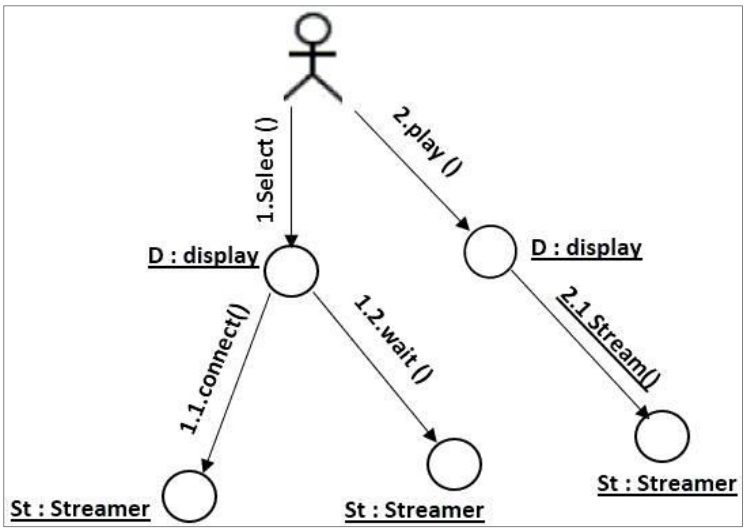

Figure 2 : L'arbre de l'exemple

Pour le diagramme de classes précédent on a :

- **CN** = {Display, Streamer, Server} :Ensembles de classes.
- **AN** = {A, B} : Le nom de chaque relations.
- **Ass** = {<Streamer, A, Display>; <Streamer, B, Server>}: Les associations.
- **Les méthodes :** L'ensemble de méthodes pour chaque classe.
  - **(Display)** = {select(), draw(), stop(), start()}
  - **(Streamer)** = {stream(), wait()}
  - **(Server)** = {connect(), handleRequest()}

Pour vérifier la cohérence entre les deux diagrammes, la règle suivante doit être vérifiée : le nom d'un message doit correspondre à une opération dans la classe du récepteur

1) Si cette règle est appliqué sur le message (**1.2.wait()**) dans l'arborescence (figure2), ça donne :

[wait()] = Streamer ∈ CN ∧ Display ∈ CN ∧ ∃ A ∈ AN.
<Streamer, A, Display> ∈ Ass ∧ wait() ∈ (Streamer).

La règle dans ce cas retourne alors **« vrai »**…… (1)

2) Si cette règle est appliqué sur le message **connect ()** invoqué par l'objet Display, ça donne :

[connect()] = Streamer ∈ CN ∧ Display ∈ CN ∧ ∃ A ∈ AN.
< Streamer, A, Display> ∈ Ass ∧ connect() ∃! (Streamer).

La règle dans ce cas retourne alors **« faux »**……..(2)

À partir de : (1) et (2) nous concluons que ces deux diagrammes ne sont pas cohérents.

4.2. Méthodes de vérification des diagrammes :

Réseaux de Petri :

Les réseaux de Petri peuvent être utilisés pour vérifier les diagrammes UML, Un réseau de Petri (RdP) est un graphe constitué de deux sortes de nœuds :

1- Les places (représentées par des nœuds).
2- les transitions (représentées par des rectangles).

Le graphe est orienté : Les arcs du graphe ne peuvent relier que des places vers des transitions, ou des transitions vers des places (jamais de places à places, ou de transitions à transitions directement).

Pour un diagramme de séquences par exemple (J. Cardoso et al dans [2]), les réseaux de Petri permettent de le vérifier le contrôle local de chaque objet (les émissions et les réceptions des messages qu'il réalise).

1- À chaque objet est associe un réseau de Petri qui se déploie selon sa ligne de vie, et ces réseaux sont connectés par des places qui correspondent aux messages échangés
2- Chaque message *m* envoyé par un objet O1 à un objet O2 donne lieu à un motif qui comporte :
   a. une place correspondant au transfert du message m de l'objet O1 vers l'objet O2.
   b. du côté de l'objet O1 : une transition qui réalise l'action !m, encadrée par une place *initiale* et une place *résultat*.
   c. du côté de l'objet O2 : une transition qui réalise l'action ?m, encadrée par une place *initiale* et une place *résultat.*

Exemple 01 :

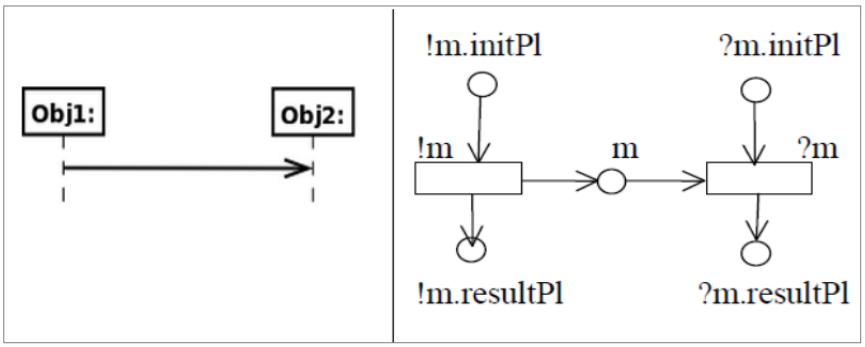

Figure 3 : Représentation d'un message asynchrone

La figure (3) montre la traduction d'un diagramme de séquences avec un message asynchrone en RdP.

Exemple 02 :

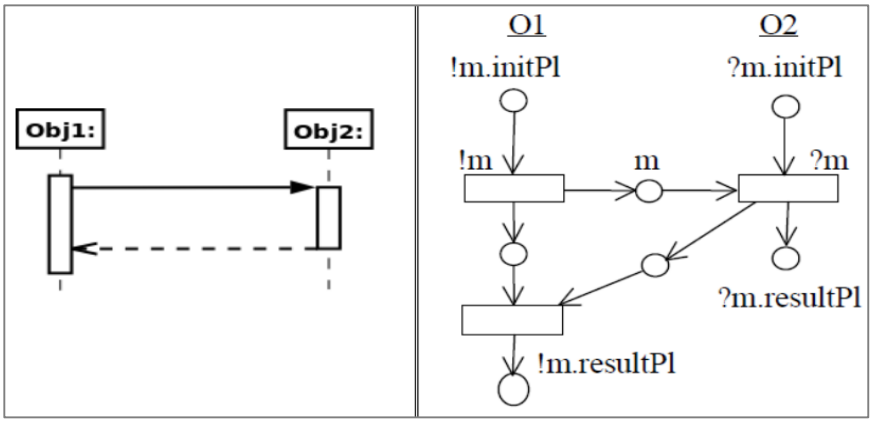

Figure 4 : Représentation d'un message synchrone

La figure (4) montre la traduction d'un diagramme de séquences avec un message synchrone en RdP (l'objet1 doit attendre l'objet2 pour terminer son exécution).

- **L'ordonnancement des actions :**

Pour connecter correctement les motifs associés aux messages, Il faut déterminer, pour chaque action, celles qui doivent être réalisées juste avant elle ainsi que celles qui peuvent l'être juste après.

Les ordres	Définitions
Minimal <	*1. pour tout message m, !m < ?m* *2. si m1 précède m2 alors !m1 < !m2* *3. si m est un message activateur, ?m < m_begin* *4. si m précède m1 et !m est synchrone, m_end < !m1*
Maximal ≪	*si m1* précède *m2*, alors *!m1 << ?m1 << !m2 << ?m2*
Local l<	*a1 l<* a2 ssi a1 <<* a2 et a1 et a2 sont réalisées par le même objet.*

Tableau 2 : La définition des ordres

Exemple 03 :

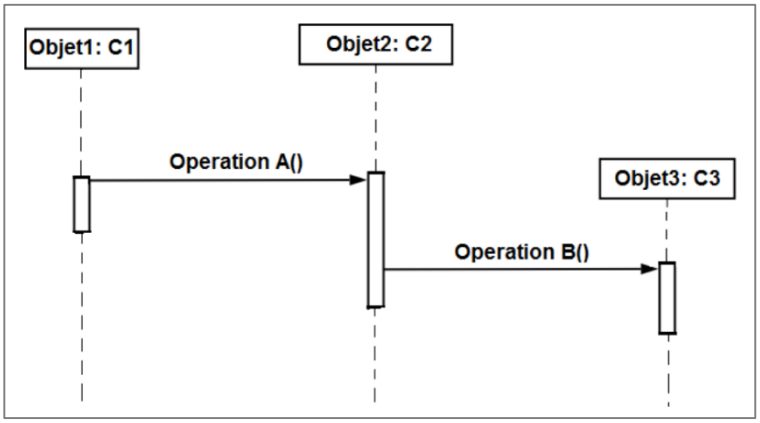

Figure 5 : Exemple d'un diagramme de séquences

Pour le diagramme de séquence représenté dans la figure (5), le RdP correspondent est représenté dans la figure (6).

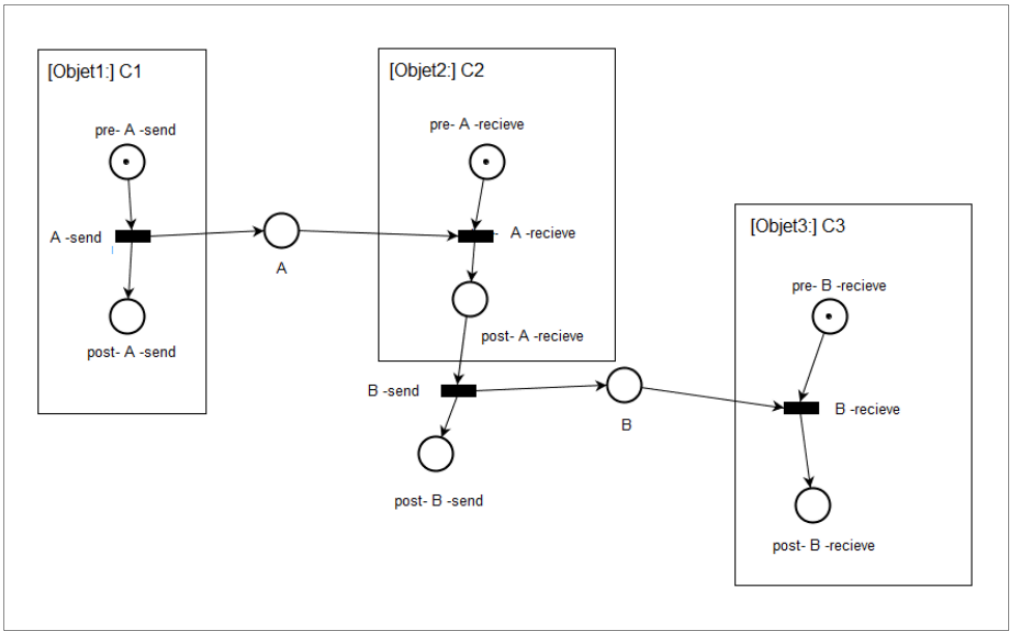

Figure 6 : Transformation de diagramme de séquences vers réseau de Petri

5. Conclusion:

Dans ce chapitre on a représenté la V&V des diagrammes UML avec une courte étude bibliographique. Plusieurs outils ont été développés pour réaliser cette V&V. Dans le prochain chapitre, on va présenter quelques-uns.

Chapitre 2 :
Quelques outils
de
vérification et validation

1. Introduction :

Dans ce chapitre, on va présenter quelques outils pour réaliser la V & V des diagrammes UML, certains outils sont libres et accessibles au public, d'autres ne sont pas accessibles au public comme vUML.

2. vUML :

vUML [5] est un outil développé par Johan Lilius et Iván Porres Paltor en 1999 qui permet de vérifier automatiquement les modèles UML. Les comportements des objets sont décrits avec des diagrammes d'états transitions.

Cet outil utilise l'outil Spin pour la vérification. Mais cette utilisation reste transparente et l'utilisation de l'outil n'a pas à connaître comment l'outil Spin est utilisé.

Si une erreur est rencontrée dans la vérification, l'outil crée un diagramme de séquences qui montre comment reproduire l'erreur dans le modèle.

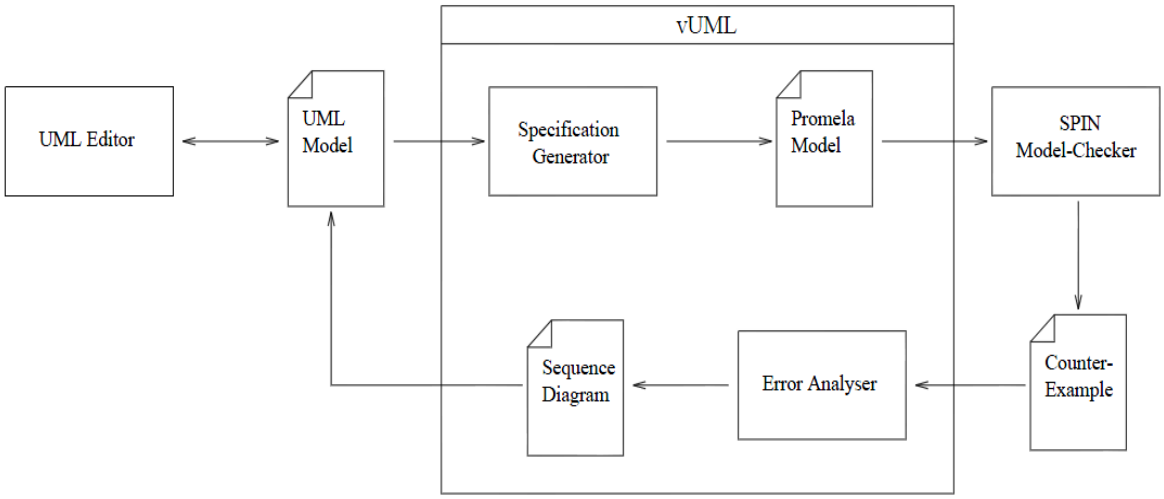

Figure 7 : Structure générale de vUML

3. Hugo/RT :

Hugo/RT est un outil développé par l'institut informatique de l'université München [6]. C'est un modèle traducteur pour la vérification de modèles, de théorèmes et de génération de code.

Le modèle UML à vérifier peut contenir un diagramme de classes, diagramme d'état transition, diagramme d'interactions et des contraintes OCL.

L'outil utilise ensuite Spin pour réaliser la vérification des modèles UML.

4. MINERVA :

MINERVA [7] est un outil qui permet de vérifier les modèles UML (diagramme de classes, diagramme d'états transitions, diagramme de séquences et de collaborations).

MINERVA exécute un graphe orienté sur les diagrammes UML (Figure 8 partie A). Si MINERVA ne trouve pas d'erreurs, il exporte ces diagrammes dans HIL (une représentation textuelle intermédiaire). L'outil Hydra effectue des analyses de compilation (partie A). Si Hydra ne trouve aucune erreur, il traduit la représentation HIL en langage PROMELA (partie B). L'utilisateur exécute Spin dans le mode simulation ou vérification (partie C) afin d'analyser les diagrammes UML via les spécifications de PROMELA.

Si on utilise Spin en mode vérification (model-checking), l'utilisateur peut créer des exigences fondées sur les propriétés LTL (partie D) pour le vérifié.

MINERVA visualise les résultats structurels de vérification de la cohérence ainsi que la simulation de comportement et les traces de contre-exemple en terme des diagrammes UML originaux ou généré des nouveaux diagrammes (partie E) pour faciliter la correction et le raffinement des diagrammes.

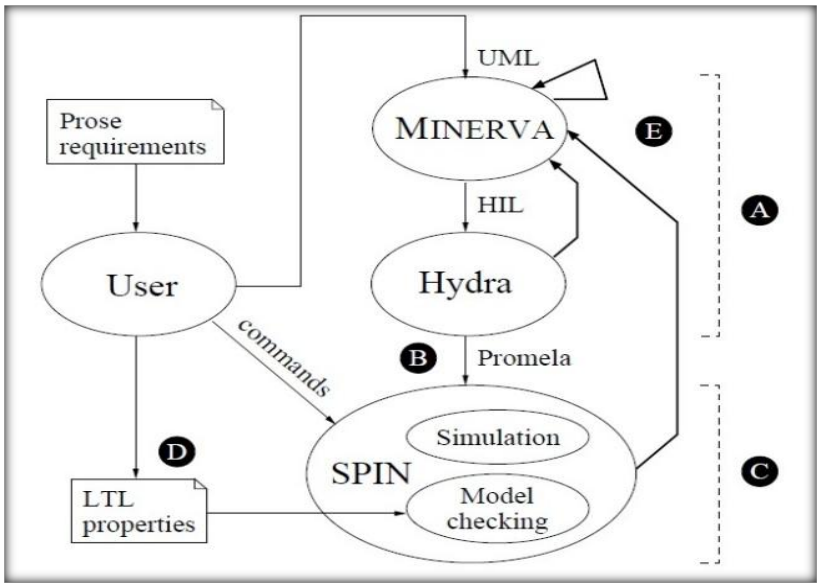

Figure 8 : Structure générale de l'outil MINERVA

5. UML / Analyzer :

UML / Analyzer [8] est un outil libre dédié à la vérification de la cohérence instantanée des modèles UML. L'outil aide les concepteurs dans la détection et le suivi des incohérences. A chaque changement de conception, il conserve la trace de toutes les incohérences au fil du temps. L'outil est entièrement automatique et ne nécessite pas une assistance manuelle.

L'outil UML / Analyzer est intégré dans l'outil de modélisation UML IBM Rational Rose™ pour une large utilisation.

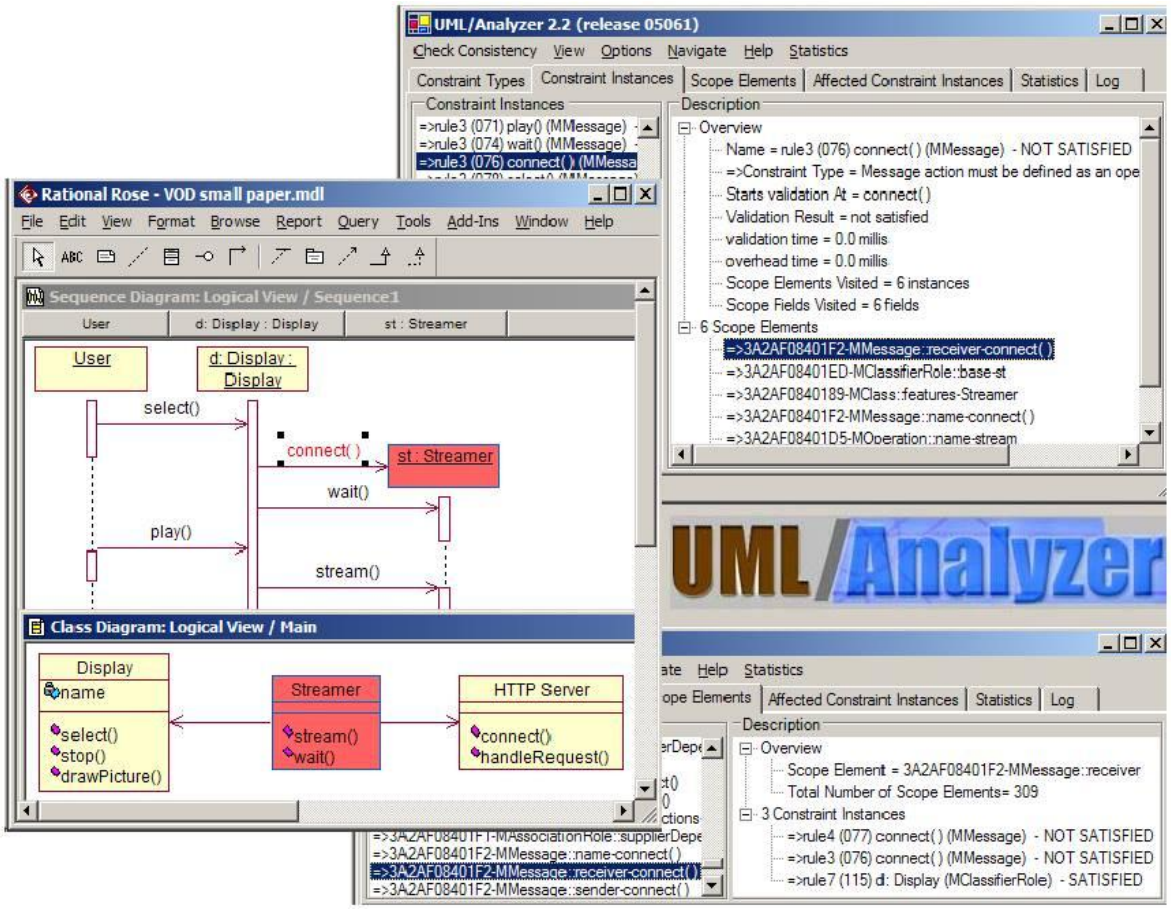

Figure 9 : L'interface de l'outil UML / Analyzer

6. Spin (Simple PROMELA INterpreter) :

Spin (Holzmann, 2003) est l'un des outils pour la vérification et la simulation des systèmes concurrents finis, le plus connu, le plus répandu et le plus utilisé. Il est open-source et multiplateforme [9].
Spin est employé pour la détection des erreurs dans les systèmes distribués, tels que les systèmes d'exploitation, les protocoles de communication de données, les algorithmes concurrents, etc… [10].

Dans Spin, chaque processus est décrit dans le langage de spécification de haut niveau PROMELA (PROcess MEta LAnguage) le langage de modélisation de Spin, et les propriétés sont écrites dans la syntaxe de LTL.

Le modèle PROMELA peut ensuite être analysé avec Spin par :

— **Simulation** : le modèle est exécuté pas à pas, éviter la construction globale du graphe d'exécution comme pré requis de la vérification. Il construit le graphe d'exécution en fonction de ses besoins, c'est-à-dire en fonction de la formule à vérifier [10].

— **Vérification** : les états du modèle sont explorés exhaustivement pour vérifier que le modèle satisfait des propriétés (par exemple, exclusion mutuelle) spécifiés en LTL (Linear Temporal Logic) et donner le cas échéant un contre-exemple [9].

6.1. Le langage de spécification LTL :

LTL *(*Linear Temporal Logic*) :* C'est un langage qui permet de formuler et spécifier les propriétés d'exactitude d'un système [11].
LTL est bien adapté pour exprimer les besoins d'un système [12], par exemple dans un système d'ascenseur le système doit satisfaire la propriété d'invariance (sécurité) suivante: [] p, Où p = ! (Portes==Ouvertes && Ascenseur==Bouge).

Pour écrire une formule LTL, les opérateurs utilisés sont listés dans le tableau suivant : [13]

description		symbole
opérateurs de logique temporelle	toujours	`[ ]`
	éventuellement	`<>`
	jusqu'à	`U`
	opérateur dual de jusqu'à	`V`
opérateur booléens	et	`&&`
	négation	`!`
	ou	`||`
	implication	`->`
	équivalence	`<->`
exemple de propriétés génériques	invariance	`[ ] p`
	réponse	`p -> <> q`
	précédence	`p -> (q U r)`
	but	`p-><>(q || r)`

Tableau 3 : Les opérateurs de LTL

6.2. L'interface graphique jSpin :

L'interface graphique jSpin est développée par M. Ben-Ari [14]. Cette interface contient en particulier une visualisation graphique de l'espace des automates crées (SpinSpider).
Il s'agit d'une alternative à l'interface graphique XSPIN et a été développé principalement à des fins pédagogiques.

Tous les aspects de jSpin sont configurables: certains au moment de la compilation, certains à l'initialisation grâce à un fichier de configuration et certains à l'exécution. [14]

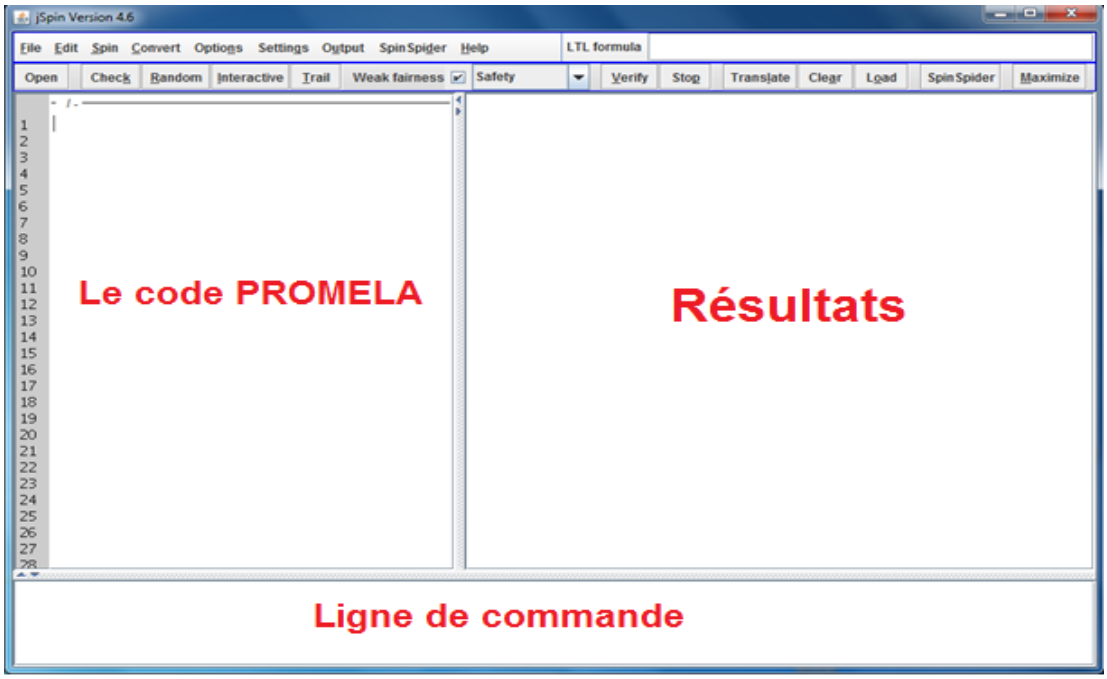

Figure 10 : L'interface jSpin

Dans la barre d'outils de jSpin il y a cinq sélections pour l'exécution. Ils utilisent tous un fichier source PROMELA [15] :

Check : Exécute une vérification de la syntaxe (générer le code C à partir de PROMELA).

Random : Exécute une simulation aléatoire.

Interactive : Exécute une simulation interactive (A chaque point de décision on a le choix de décider lequel de suivre).

Trail : pour voir le résultat d'une trace d'erreur après l'exécution de la vérification qui a signalé les erreurs.

Verify: Exécute une vérification des propriétés LTL.

SpinSpider : jSpin inclut l'outil SpinSpider qui donne les automates pour le système montrant l'interaction entre les différents processus, Ce dernier affiche le source du programme PROMELA comme un ensemble d'automates. Pour chaque **proctype** (qui représente un processus), un automate est créé, dont Le nœud représente le numéro de ligne du code et l'arête indique la condition qui est nécessaire pour satisfaire le passage à l'état suivant [16].

6.3. L'interface graphique iSpin :

iSpin est une interface graphique pour Spin écrit en Tcl/Tk, la dernière version est la version 1.1.4 - 27 Novembre 2014.

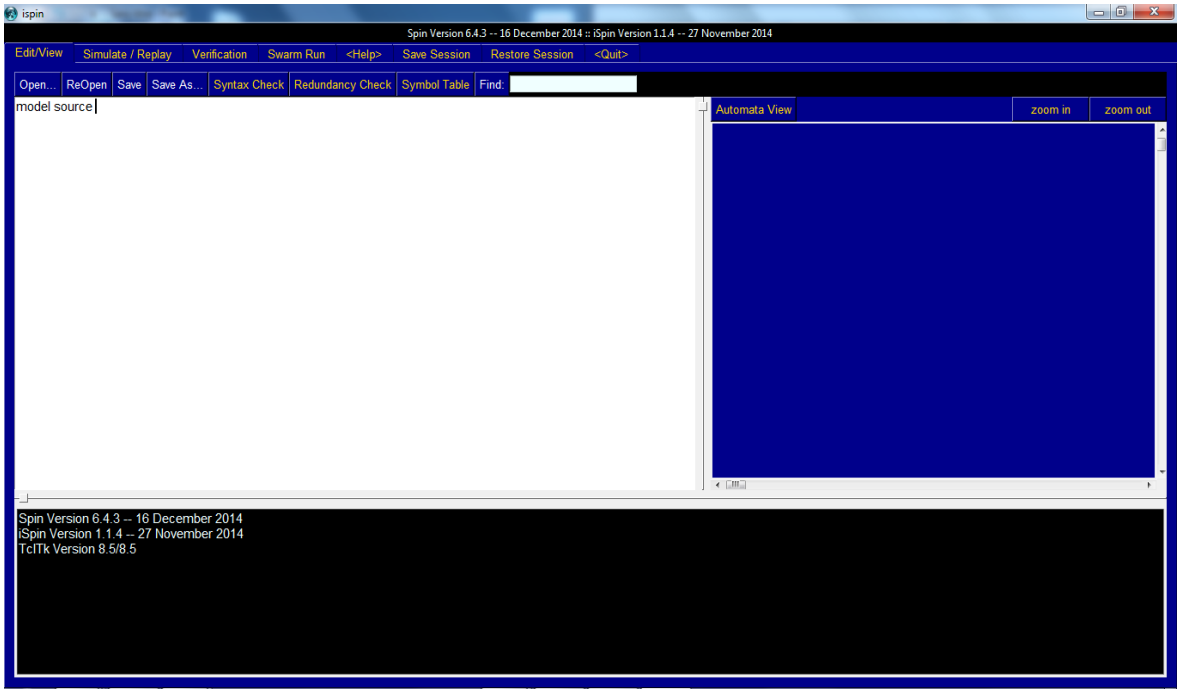

Figure 11 : L'interface graphique iSpin

6.4. L'interface graphique Spin en ligne :

Un site [17] offre une interface en ligne qui a été développée par Bart van Delft en 2013 pour le modèle de vérification Spin, cette interface a été développée pour simplifier son utilisation dans un cours à l'université Chalmers.
On ne trouve pas toutes les fonctionnalités de Spin et jSpin dans cette interface. Certaines opérations telles que l'exécution interactive sont techniquement trop compliqué, mais d'autres fonctionnalités sont laissés de côté pour garder l'interface simple.

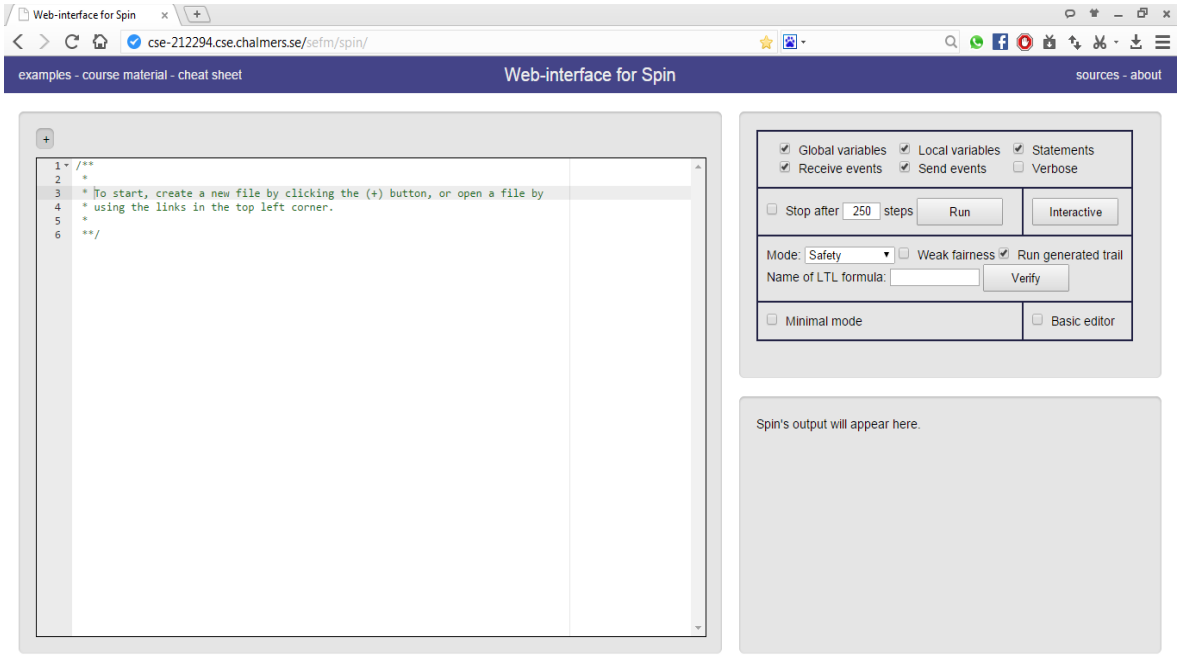

Figure 12 : L'interface web de l'outil Spin

7. Conclusion :

Dans ce chapitre on a présenté quelques outils utilisé pour la vérification et la validation des diagrammes UML.

On a remarqué que l'outil Spin est largement utilisé par les outils vUML, Minerva et Hugo/RT pour réaliser la V&V des diagrammes UML, et ces derniers ne sont que des moyens pour convertir les différents modèles UML en code PROMELA pour utiliser Spin.

Dans le prochain chapitre, on va exploiter Spin pour la V&V des diagrammes UML, et plus précisément le diagramme de séquence et d'activités.

Chapitre 3 :

Exploitation de l'outil Spin pour la V &V des diagrammes UML

1. Introduction :

Dans ce chapitre nous présentons et discutons l'exploitation de l'outil Spin pour la vérification des diagrammes UML. Plus précisément les diagrammes de séquences et d'activités.

Le diagramme de séquences et le diagramme d'activités parmi les cinq des diagrammes UML les plus fréquemment utilisés : activités, classes, état-transition, séquences et cas d'utilisation. C'est pour cette raison qu'on a choisi ces diagrammes pour une étude détaillée.

2. Digramme de séquences :

Plusieurs travaux sont réalisé pour la vérification de diagramme de séquence, parmi ces travaux, M. Ait Oubelli et al dans [17], a proposé une approche basé sur la technique de transformation des graphes (proposer une grammaire) pour générer le code PROMELA du diagramme de séquence, ils utilisent un outil de transformation de graphe ATOM³. Le code PROMELA est utilisé par l'outil d'analyse Spin pour la simulation et la vérification des propriétés écrites en LTL.

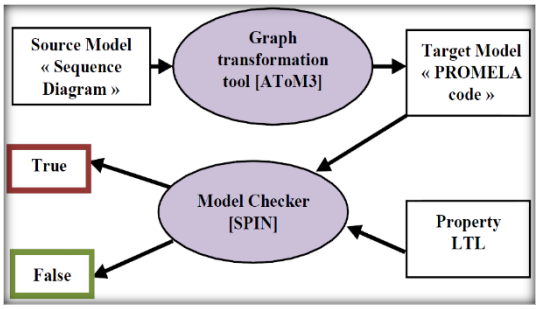

Figure 13 : Aperçu de l'approche proposée [17]

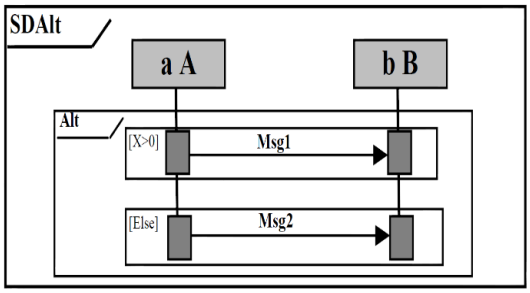

Figure 14: Fragment Alternative [17]

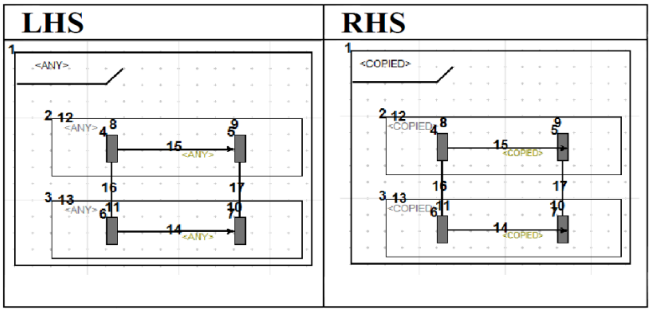

Figure 15: Règle de transformation pour déférent fragment combiné au PROMELA [17]

```
Prototype a() {
 if
   ::(X>0) -> ab_msg2!msg2;
   :: else -> ab_msg1!msg1;
 fi;}
prototype b() {
 if
   ::(X>0) -> ab_msg2?msg2;
   :: else -> ab_msg1?msg1;
 fi;}
init {
   if
     :: (true) -> X>0=true;
     :: (true) -> X>0=false;
fi;}
```

Figure 16: Code PROMELA associé [17]

La figure (13) représente un aperçu sur l'approche proposé, et La figure (14) montre le diagramme de séquence qui va être l'entrée pour l'outil ATOM3, le diagramme va être transformé par l'application des règles de transformation représenté dans la figure (15), la figure (16) c'est le code PROMELA correspondant au diagramme de séquences.

S. Leue et P. B.Ladkin dans [18], propose une traduction des MSCs (Message Sequence Charts) en PROMELA, L'approche traite seulement les composants de base de diagramme de séquences (ne couvre pas les fragments combinés). PROMELA/Spin a été choisi parce qu'il fournit des concepts importants (les primitives d'envoi et de réception des messages, les canaux de communication et les processus concurrents).

MSC décrit la séquence d'échange de messages par la communication entre les processus.

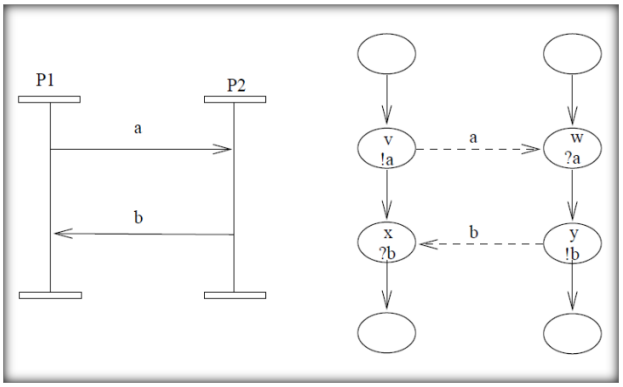

Figure 17 : Simple exemple de MSC

```
mtype {a, b};
chan vw = [1] of { byte };
chan xy = [1] of { byte };
proctype P1(){
atomic { vw!a; printf("!a\n") };
atomic { xy?[b] -> xy?b;
printf("?b\n")}
}
proctype P2(){
atomic { vw?[a] -> vw?a;
printf("?a\n") };
atomic { xy!b; printf("!b\n")}
}
init { atomic { run P1(); run P2() }; }
```

Figure 18 : Code PROMELA associé

Le côté gauche de la figure (17) montre un MSC de base et le côté à droite représente la transformation du MSCs en MFGs (Message Flow Graphs) où Les nœuds sont reliés par des flèches dirigées représentant deux relations :

- *Next-Event (ne)* relation représentant le flux de contrôle dans un processus représenté par flèches avec des lignes continues.
- *Signal (sig)* relation représentant des flux de messages représenté par des flèches avec des lignes discontinu.

La figure (18) montre le code PROMELA du MSC figure (17) par l'application des règles de transformation.

2.1. Traduction diagramme de séquence en PROMELA :

Les instructions suivantes sont utilisées pour représenter les composants de base du diagramme de séquence en PROMELA:

- **proctype :** il est utilisé pour déclarer un nouveau processus.
- **mtype :** il est utilisé pour définir le nom du message qui est utilisé dans le processus de communication.
- **chan :** il est utilisé pour déclarer les canaux de communication pour chaque flèche de message.
- **Les opérateurs !/ ? :** Ces opérateurs sont utilisés respectivement pour l'envoi et la réception des messages vers les (resp à partir) les canaux.

2.2. Des exemples de base (séquence en PROMELA) :

Dans ce qui suit, on présent des exemples de base et leur transformation en PROMELA cités dans [20]. A partir de ces exemples on peut générer le code PROMELA pour des diagrammes de séquences combinés.

Exemple 01 (Simple D. séquence):

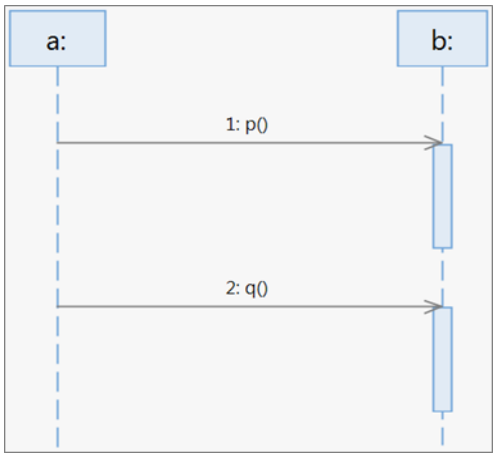

Figure 19 : Simple D. séquence

```
1. mtype  {p,q} ;
2. chan ab_p = [1] of {mtype} ;
3. chan ab_q = [1] of {mtype} ;
4. proctype a(){
5. ab_p!p ; ab_q!q ;
6. };
7. proctype b(){
8. ab_p?p ; ab_q?q ;
9. };
10. init {
11. atomic { run a() ; run b() };}
```

Figure 20 : Code PROMELA associé

Discussion :

Dans le diagramme de séquences -figure (19)- on a deux objets (a:) et (b:).
L'objet (a:) envoie deux message « p » et « q », tels que :
L'envoi du message « q » doit être après la réception du message « p » par l'objet (b:) ou bien juste après l'envoi du message « p » qui sont reçu par l'objet (b:) séquentiellement.

Concernant le code PROMELA associé -figure (20)-:
Dans la 1ère ligne les messages « p » et « q » sont déclarés, la 2ème et la 3ème lignes représentent les canaux sur lesquels les messages « p » et « q » sont envoyés, la 4ème et la 7ème lignes représentent les processus « a » et « b », le corps de chaque processus implémente le comportement de communication, ex : L'instruction (ab_p !p) indique un envoi d'un message « p » sur le canal « ab_p » et L'instruction (ab_p ?p) indique une réception d'un message « p » sur le canal « ab_p ».

La 10ème ligne est l'instanciation du système.

PROMELA nécessite l'utilisation de l'instruction **atomic** -ligne (11)- pour assurer que les opérations à l'intérieur des accolades sont exécutées comme une action atomique, sans entrelacement avec d'autres événements.

Exemple 02 (D. séquence avec l'alternative):

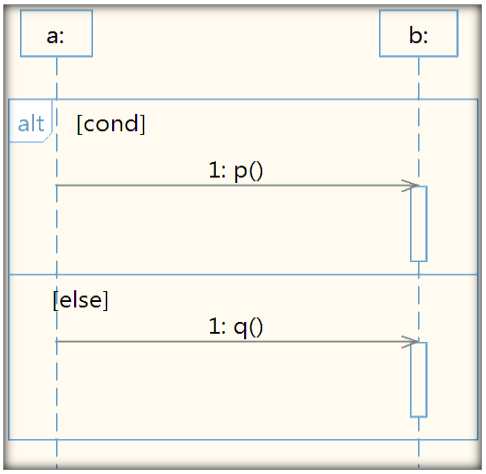

Figure 21 : D. séquences avec l'alternative

```
1. mtype  {p,q} ; bool cond;
2. chan ab_p = [1] of {mtype} ;
3. chan ab_q = [1] of {mtype} ;
4. proctype a( ){
5. if
6.        ::(cond) -> ab_p!p ;
7.        ::else -> ab_q!q ;
8. fi ;}
9.  proctype b( ){
10. if
11.       ::(cond) -> ab_p?p ;
12.       :: else    -> ab_q?q ;
13. fi;}
14. init{
15. if
16.       :: (true) -> cond=true ;
17.       :: (true) -> cond=false ;
18. fi ;
19. atomic{ run a() ; run b() } ; }
```

Figure 22 : Code PROMELA associé

Discussion :

Dans le diagramme de séquences -figure (21)- on a deux objets (a:) et (b:) et l'option **alt** (alternative).
L'objet (a :) envoie le message « p » ou « q » selon la condition, tels que : si la condition (cond) est satisfaite l'objet (a:) envoie le message « p », sinon il envoie le message « q », le message envoyé va être reçu par l'objet (b:).

Concernant le code PROMELA associé -figure (22)-:
Le variable « cond » est déclaré globalement pour l'utiliser par les deux processus et obtenir la même décision au point de choix.
Dans l'instanciation on utilise **if** avec deux conditions exécutables pour affecter à la variable **cond** la valeur **true** ou **false** -lignes (16 et 17)-. Au moment de l'exécution, Spin choisit au hasard une option et continue la simulation.
S'il choisit « true », Spin exécute dans le processus a() la ligne (6) et dans le processus b() la ligne (11), et s'il choisit « false », Spin exécute dans le processus a() la ligne (7) et dans le processus b() la ligne (12).

Exemple 03 (D. séquence avec la boucle):

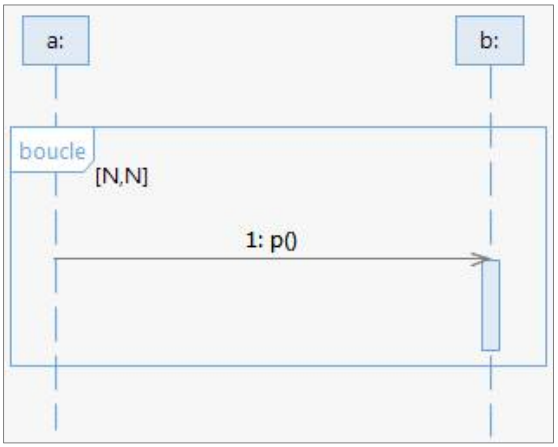

Figure 23 : D. séquences avec la boucle

```
1.int N=5, i=0;
2. mtype {m};
3. chan ab_m =[1] of {mtype} ;
4. proctype a(){
5.   do
6.   :: (i<N) -> atomic {ab_m!m; i++;};
7.   :: else -> break ;
8.   od }
9.  proctype b(){
10.    do
11.    ::(i<N) -> atomic {ab_m?m; i++;};
12.    :: else -> break;
13.    od}
14. init{
15.  atomic {run a(); run b();};}
```

Figure 24 : Code PROMELA associé

Discussion :

Dans le diagramme de séquence -figure (23)- on a deux objets (a:) et (b:) et l'option **boucle** (Loop).
L'objet (a:) envoie le message « p », et répète l'envoi de message selon le nombre (N), et l'objet (b:) reçoit les messages selon le même nombre (N).
Concernant le code PROMELA associé -figure (24)-:
Le variable N est déclaré globalement pour l'utiliser par les deux processus.
Le processus a() -ligne (4)- envoie le message selon le nombre N, et le processus b() -ligne (9)- reçoit le message selon le même nombre N.

2.3. Méthode pour création du code PROMELA:

Dans cette section, on propose notre propre méthode pour créer le code PROMELA d'un diagramme de séquences. On propose de suivre les étapes suivantes:

1- On commence par la déclaration de tous les massages de communication entre les processus par l'instruction **mtype**.
2- Ensuite nous déclarons pour chaque message, le canal où nous l'envoyons par l'instruction **chan** avec la définition de type et la taille de chaque canal.
3- On définit notre processus par l'instruction **proctype** où on définit pour chaque processus les émissions et les réceptions des messages qu'il réalise par les opérateurs (**!** pour l'envoi de message, et **?** pour la réception).
4- Enfin, on définit le point d'entrer de programme par l'instruction **init** où on lance notre processus par l'instruction **run**.

2.4. Exemple (GasPompe) :

Le diagramme de séquence -figure (25)- contient trois objets (Utilisateur, GasPompe et Bank).

L'utilisateur insère sa carte dans l'appareil GasPompe, ensuite l'application demande le code pin, l'utilisateur insère le code pin, l'application accède vers le compte de l'utilisateur dans la banque et elle vérifier son code.

Si le code est valide l'application démarre la pompe, après le remplissage, l'utilisateur raccroche la pompe et éjecte sa carte.

Si le code n'est pas valide l'application demande à l'utilisateur d'éjecter sa carte.

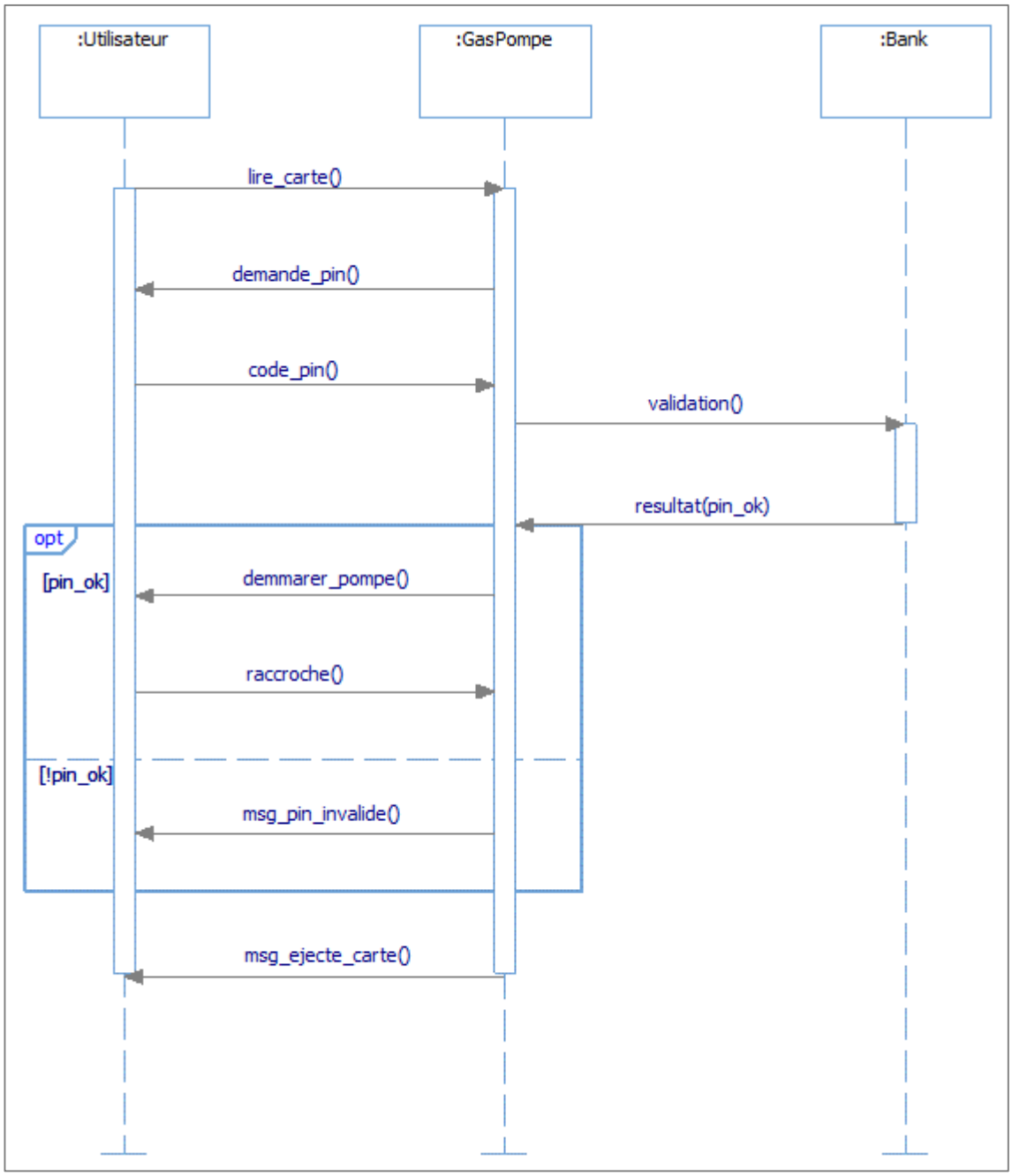

Figure 25 : D. séquences d'une pompe à gaz

```
1.  /* déclaration des messages et des canaux */
2. mtype {readCard, requestPin, pinCode, validate, result, startFuel,hangUp,
3. invalidPin, cardOut};
4. chan ab1 = [1] of {mtype};
5. chan ba2 = [1] of {mtype};
6. chan ab3 = [1] of {mtype};
7. chan bc4 = [1] of {mtype};
8. chan cb5 = [1] of {mtype};
9. chan ba6 = [1] of {mtype};
10. chan ab7 = [1] of {mtype};
11. chan ba8 = [1] of {mtype};
12. chan ba9 = [1] of {mtype};
13. /* déclaration de la variable qui contient le résultat de la validation */
14. bool pinOK;
15. /* déclaration de processus de l'utilisateur */
16. proctype User(){
17. ab1!readCard;
18. ba2?requestPin;
19. ab3!pinCode;
20.         if
21.             :: (pinOK) -> ba6?startFuel; ab7!hangUp;
22.             :: else -> ba8?invalidPin;
23.         fi;
24. ba9?cardOut;
25. }
26. /* déclaration de processus de Gaspump */
27. proctype GasPump(){
28.       ab1?readCard;
29.       ba2!requestPin;
30.       ab3?pinCode;
31.       bc4!validate;
32.       cb5?result;
33.         if
34.             :: (pinOK) -> ba6!startFuel; ab7?hangUp
35.             :: else -> ba8!invalidPin;
36.       fi;
37.       ba9!cardOut;
38. }
39. /* déclaration de processus de la Bank */
40. proctype Bank(){
41.       bc4?validate;
42.       if
43.       :: (true) -> pinOK= true;
44.       :: (true) -> pinOK= false;
45.       fi;
46.       cb5!result;
47. }
48. init { atomic{ run User();run GasPump();run Bank();} }
```

Figure 26 : Code PROMELA associé

Discussion 01 :

Concernant le code PROMELA -figure (26)-:

Le variable « pinOK » est déclaré globalement -ligne (14)- pour l'utiliser par les processus pour obtenir la même décision au point de choix.

Spin va lancer les trois processus (**User()**, **GasPump()**, **Bank()**) utilisant l'instruction **atomic** qui permet d'exécuter les processus de manière atomique et sans intervention des autre processus.

Lorsque le processus **Bank()** reçoit le message validation -ligne (41)-, le processus va faire une simulation du système par production de résultat de **pinOK** -ligne (43, 44)- et l'envoyé au processus **GasPump()**.

La sélection de bouton SpinSpider donne le résultat suivant :

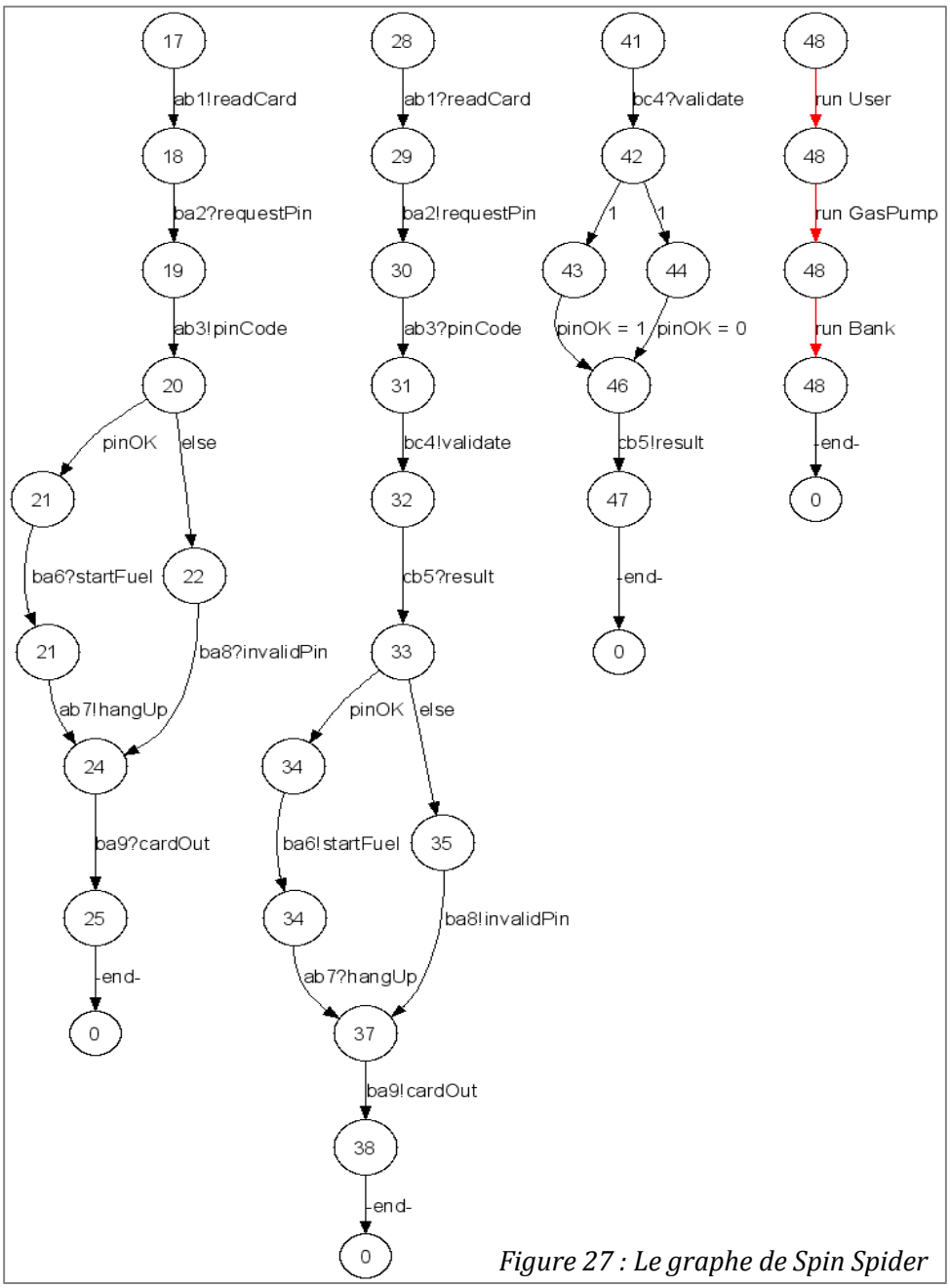

Figure 27 : Le graphe de Spin Spider

Discussion 02 :

Concernant le graphe de SpinSpider -figure (27)-:

Quatre processus (automates) ont été créés : proctype User(), proctype GasPump(), proctype Bank(), et init(). (De gauche à droite).

Chaque nœud dans l'automate représente la ligne de commande dans le code PROMELA, par exemple : le premier nœud contient le numéro (17), et représente la ligne (17) dans le code PROMELA.

Et chaque transition dans l'automate représente une instruction dans le code PROMELA par exemple : la transition de (17) vers (18) qui contient (ab1! readCard) représente l'instruction de la ligne (17) dans le code PROMELA.

Les nœuds qui contiennent (0) représentent les états finaux de chaque automate.

Le graphe –figure (27)- contient des lignes rouge quand on utilise la fonction **atomic,** pour montrer que ces processus sont exécutés de manière atomique, c'est-à-dire ils ne peuvent pas être interrompus par d'autres processus ou événements.

2.5. Vérification des propriétés LTL :

La méthodologie consiste à créer un modèle PROMELA et suivre l'état d'une interaction en utilisant des **flags** (indicateurs) [20]**.**

On place des **flags** dans chaque **émission/réception** pour chaque **source/destination** de messages, ce qui permet aux concepteurs d'écrire propriétés LTL en termes d'expressions booléennes sur les flags.

Ce mécanisme permet de déterminer **qui envoie/reçoit** à tout moment de l'exécution. Cette information est très utile lorsque l'on veut écrire propriétés LTL pour la vérifier.

Les valeurs de flags sont mises à jour avec chaque évènement d'envoi ou de réception. La mise à jour se faite en utilisant l'instruction **atomic** pour exécuter l'évènement et faire effectuer les nouvelles valeurs des flags comme une étape à la durée d'exécution.

Exemple 01 :

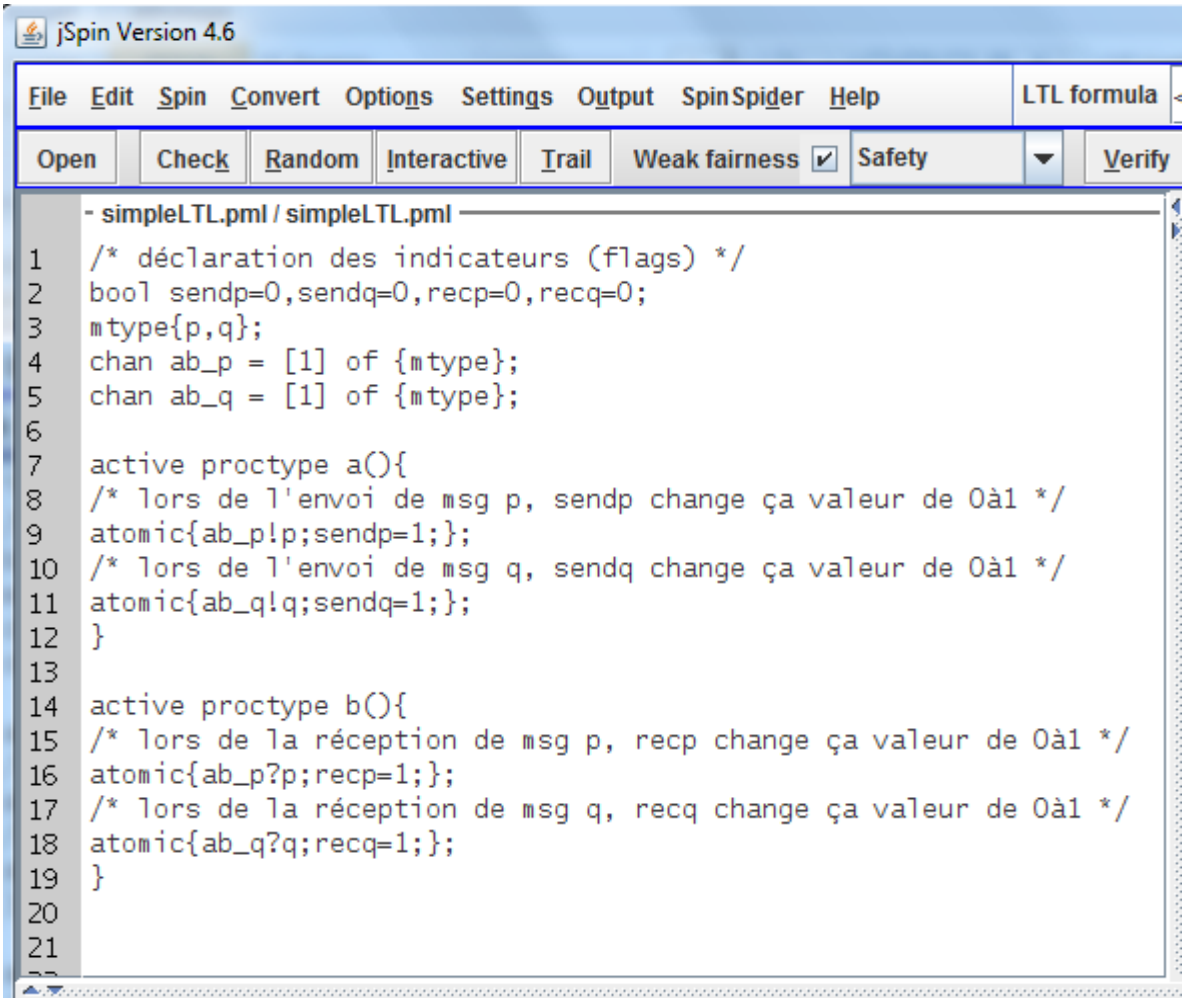

Figure 28 : Code PROMELA d'un simple diagramme de séquences avec des flags

Dans l'exemple on a utilisé (sendp, sendq, recp, et recq) comme des indicateurs (**flags**) pour les utiliser lorsqu'on veut écrire les propriétés LTL.
Les indicateurs sont initialisés à 0, puis quand on envoie les messages p et q les indicateurs (sendp, sendq) changent leurs valeurs de 0 à 1. Et quand on reçoit les messages p et q les indicateurs (recp, recq) changent leurs valeurs de 0 à 1.

Exemple LTL non satisfaite :

On veut vérifier dans l'exemple précédent –figure (29)- que (toujours l'envoi du message **q** devra être après la réception de message **p**. La formule LTL associée à cette propriété est : « [] (! sendq U recp) ».

Figure 29 : Zone de texte de propriété LTL

Après la vérification de la propriété, Spin affiche une erreur ce que signifie que la propriété n'est pas valide.

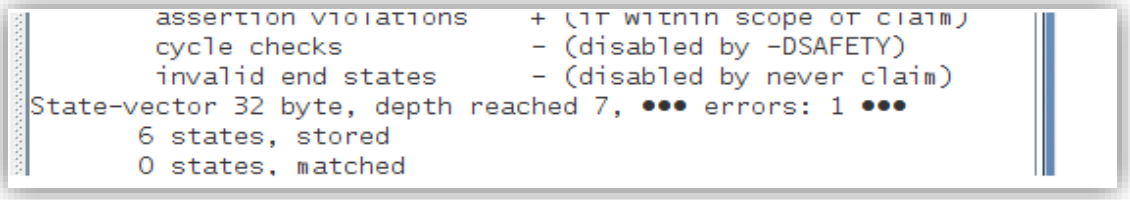

Figure 30 : Résultat de vérification de la propriété LTL

Donc le processus a() peut envoyer le message **q** avant la réception du message **p**.

Exemple LTL satisfaite :

On veut vérifier dans l'exemple précédent que toujours la réception de message **q** sera éventuellement après l'envoi de message **q**.
La formule LTL associe à cette propriété est : « [] (sendq -> <> recq) ».

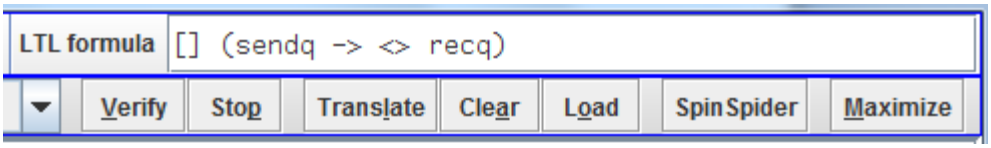

Figure 31 : Zone de texte de propriété LTL

Après la vérification de la propriété, Spin n'affiche pas une erreur ce que signifier que la propriété est valide.

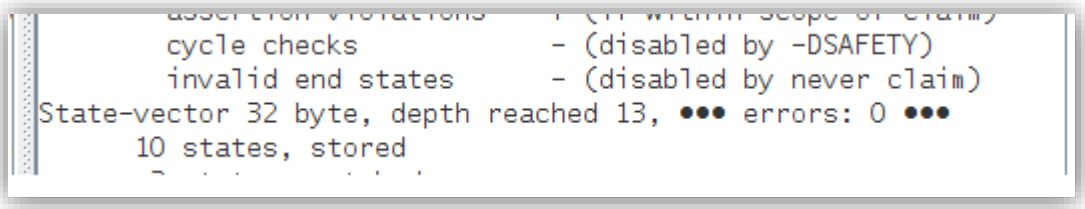

Figure 32 : Résultat de vérification de la propriété LTL

Ce que signifie que le processus b() reçoit le message **q** après son envoi par le processus a().

Exemple 02 :

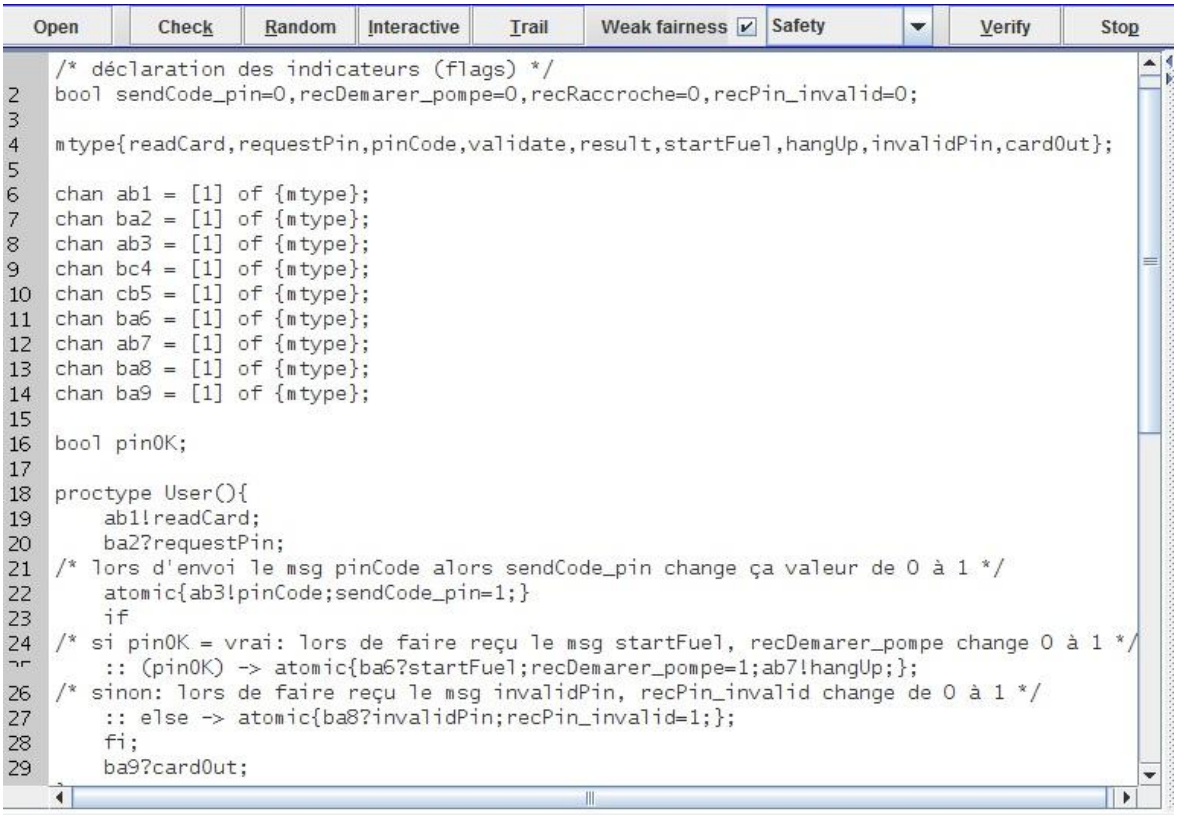

Figure 33 : Le code PROMELA de la (figure 25) avec des flags

Dans l'exemple précédent –figure (25)-on a utilisé :

(sendCode_pin, recDemarer_pompe, recRaccroche et recPin_invalid) comme des indicateurs (**flags**) pour les utiliser lorsqu'on veut écrire les propriétés LTL.

Les indicateurs sont initialisés à 0, puis quand on envoie le message code_pin l'indicateur sendCode_pin change sa valeur de 0 à 1, et quand on reçoit les messages (demarer_pompe, raccroche et pin_invalid) les indicateurs (recDemarer_pompe, recRaccroche et recPin_invalid) changent leurs valeurs de 0 à 1.

Exemple LTL satisfaite :

On veut vérifier dans l'exemple précédent –figure (33)- que toujours la réception de message (**demarer_pompe** et **raccroche**) ou **pin_invalide** seront après l'envoi de message **code_pin**).

La formule LTL associée à cette propriété est :

«[](sendCode_pin-><>((recDemarer_pompe&& recRaccroche)||(recPin_invalid)))»

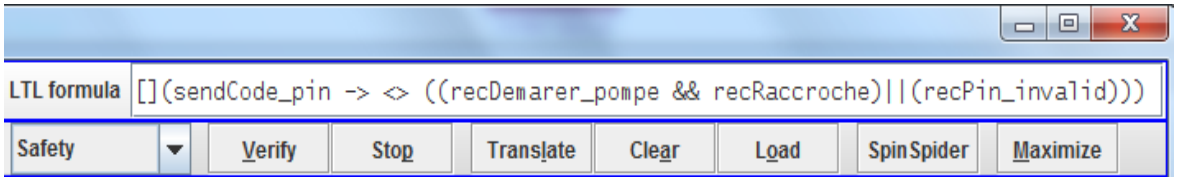

Figure 34 : Zone de texte de propriété LTL

Après la vérification de la propriété, Spin n'affiche pas une erreur ce que signifier que la propriété est valide.

```
        assertion violations    + (if within scope of claim)
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      - (disabled by never claim)
State-vector 72 byte, depth reached 52, ••• errors: 0 •••
     114 states, stored
      96 states, matched
```

Figure 35 : Résultat de vérification de la propriété LTL

Donc la réception des messages (**demarer_pompe** et **raccroche**) ou **pin_invalide** seront après l'envoi de message **code_pin**.

Exemple LTL non satisfaite :

On veut vérifier dans l'exemple précédent que : toujours la réception de message **pin_invalide** (le résultat attendu est négative).
La formule LTL associée à cette propriété est : [](recPin_invalid) .

Figure 36 : Zone de texte de propriété LTL

Après la vérification de la propriété, Spin affiche une erreur ce qui signifie que la propriété n'est pas valide.

```
        assertion violations    + (if within scope of claim)
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      - (disabled by never claim)
State-vector 72 byte, depth reached 5, ••• errors: 1 •••
       2 states, stored
       0 states, matched
```

Figure 37 : Résultat de vérification de la propriété LTL

Ce que signifie que toujours la réception de message (**demarer_pompe** et **raccroche**) ou **pin_invalide** sera après l'envoi de message **code_pin**). C'est la conclusion de l'exemple précédent.
Pour cet exemple la conclusion sera:
Donc on ne reçoit pas toujours le message **pin_invalide.**

3. Diagramme d'activités :

À l'aide de diagrammes d'activités, UML permet de représenter graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation. Le passage d'une activité vers une autre est matérialisé par une transition. Les transitions sont déclenchées par la fin d'une activité et provoquent le début immédiat d'une autre [21].

Parmi les travaux réalisés pour la vérification de diagramme d'activités on trouve la technique proposée par Li jing et al dans [22], pour faire la vérification de diagramme d'activités par l'outil Spin.

Tout d'abord, les diagrammes d'activités UML sont convertis en automates hiérarchique étendu (EHA). Ensuite, les modèles EHA sont transformées dans le modèle PROMELA qui est l'entrée de l'outil Spin.

Yutaka Yamada et al dans [23], propose une méthode automatique pour convertir le diagramme d'activités en code PROMELA.

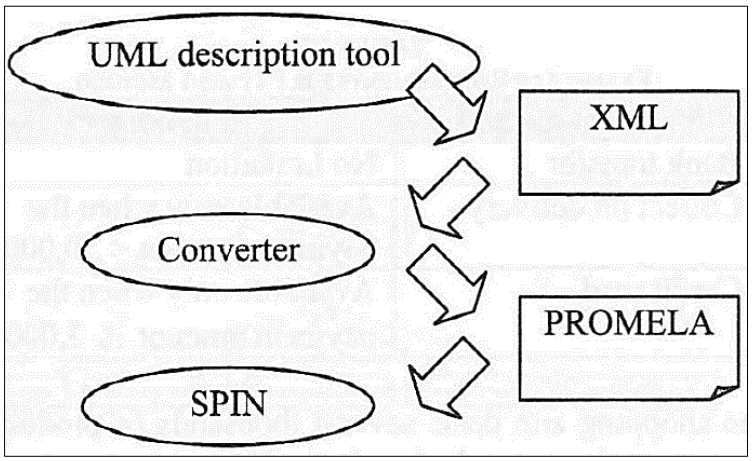

Figure 38 : Flux de conversion automatique

La figure (38) montre le flux de conversion automatique de diagramme d'activités au code PROMELA, le convertisseur développe en utilisant le langage PHP.

La premier étape est d'utiliser un outil de modélisation pour modéliser le diagramme et de l'exporter en XML, le programme de conversion utilise le fichier XML comme entrée pour générer le code PROMELA, ce dernier sera vérifié par l'outil Spin.

On propose dans ce qui suit, notre vision pour la création du code Promela. Cette vision a été faite après avoir étudié le papier Li jing et al dans [22].

On propose de suivre les points suivants :

- Déclarer les flux comme des variables booléennes initialisés à 0, ces variable changent leurs valeurs de 0 à1 pour indiquer la fin d'une action.
- La condition est représentée par l'instruction **if** (comme switch dans C) plus un **goto** pour sauter à l'action choisie.
- Si on a **fork**, on utilise l'instruction **if,** où tous les cas sont acceptés et toutes les actions sont exécutées en parallèle.
- La notation **joint** marque la fin d'exécution de toutes les actions de **fork**.
- On peut représenter le nœud initial et le nœud finale par :
  Init{
  goto fin;/* pour chaque nœuds final on saut vers l'étiquette fin*/
  fin : skip ;}

3.1. Exemple de simulation :

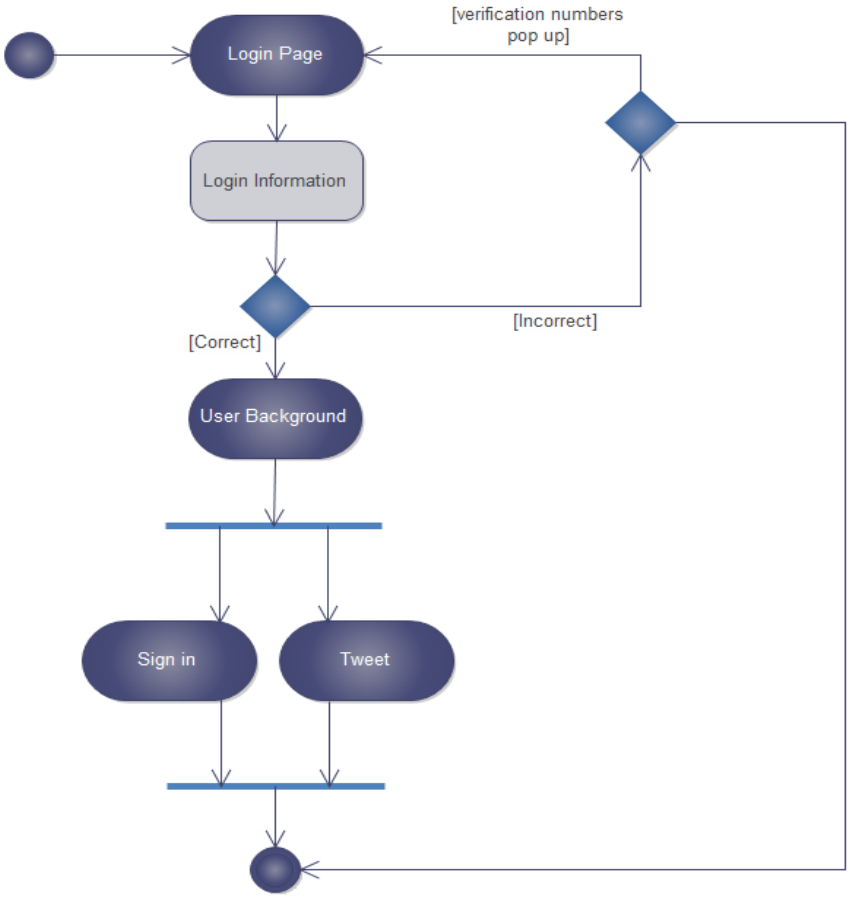

Figure 39 : D. d'activités de l'authentification [26]

Dans l'exemple l'utilisateur va insérer ses informations, si les informations ne sont pas correctes il peut les réinsérer N fois au maximum et si non il va attendre une durée de temps. Si les informations sont correctes il va accéder à son profile et la page lance une tweet.
Le code PROMELA associé à ce diagramme est :

Open | Check | Random | Interactive | Trail | Weak fairness ☑ | Safety

- activity2.pml / activity2.pml

```
1  #define p (nbrVer>=0 && nbrVer <=5)
2  bool loginPage=0,loginInformation=0,userBackground=0,signIn=0,tweet=0;
3  bool correct;
4  int nbrVer=5;
5  init{
6  et2:if
7  ::(true)->loginPage=1;
8  fi;
9  if
10 ::(loginPage)-> loginInformation=1;
11
12     if
13     :: (true)-> correct=true;goto et1;
14     :: (true)-> correct=false;if
15                                ::(nbrVer>0)->nbrVer--;goto et2;
16                                ::else-> goto fin;
17                               fi;
18     fi;
19 fi;
20 et1: if
21         ::(correct)-> userBackground=1;
22      fi;
23 if
24 ::(userBackground)-> if
25                     :: (true)->signIn=1;
26                     :: (true)->tweet=1;
27                     fi;
28 fi;
29 if
30 ::(signIn && tweet)-> goto fin;
31 fi;
32
33 fin: skip;}
```

Figure 40 : Le code PROMELA de diagramme d'activités

On a déclaré le variable « correct » -ligne (3)- pour le générer -ligne (13,14)- et obtenir le résultat de vérification des informations.

Le variable « NbrVer » est de type entier qui utilisé dans la ligne (15) pour vérifier le nombre de réinsertions des informations (boucle).

Les variables « LoginPage, LoginInformation, UserBackground, SignIn, Tweet » sont initialisé à 0, A la fin d'exécution de chaque action, les variables changent leurs valeurs de 0 à 1 ce qui permet de stimuler l'action suivante. Les variables sont aussi utilisées comme des indicateurs pour que nous puissions écrire les propriétés LTL.

La sélection de bouton SpinSpider donne le résultat dans la figure président :

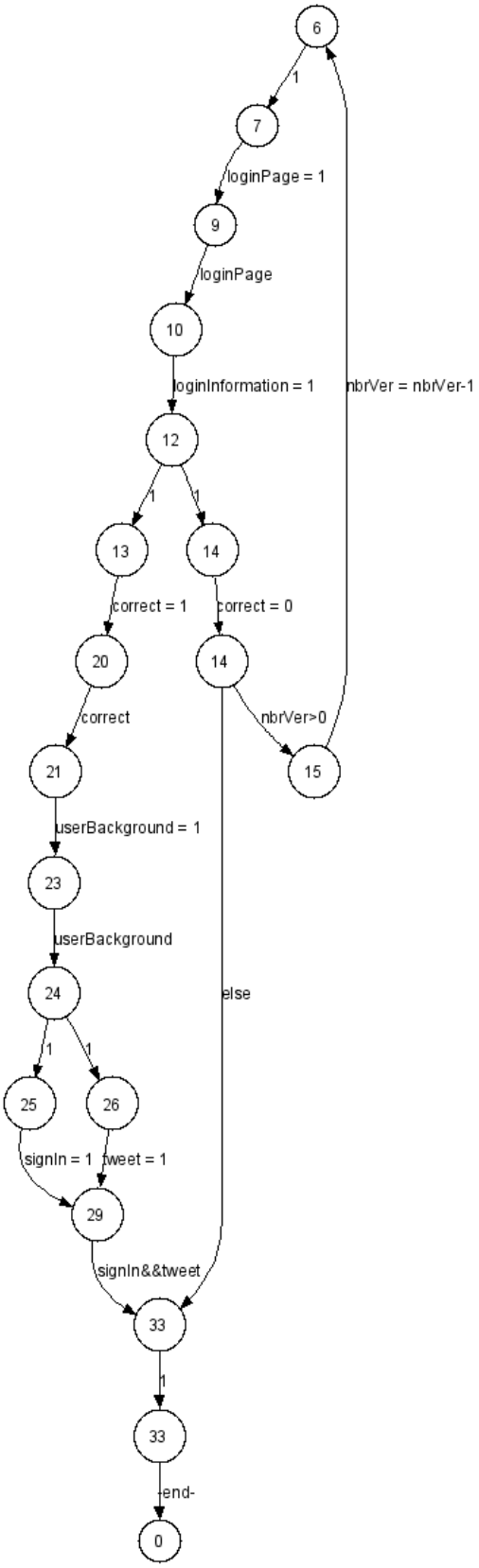

Figure 41 : Le graphe de Spin Spider

Discussion :

L'utilisateur va insérer ces informations -nœud (10)- puis, le nœud (12) représente la décision après la vérification de ces informations. Si ces informations sont correctes alors la branche qui sera choisie est vers le nœud (13) et exécuté successivement les actions jusqu'à la fin, si non la branche qui sera choisie sera vers le nœud (14).

Le nœud (14) représente la décision sur le nombre des essais permis à l'utilisateur. Si ce nombre est supérieur à 0, alors on choisit la branche vers le nœud (15) et on décrémente le nombre des essais pour retourner vers le nœud (6). Et sinon on sortit vers la fin du programme.

3.2. Vérification des propriétés LTL

Exemple LTL non satisfaite :

On veut vérifier dans l'exemple précédent que toujours login d'information éventuellement est correct.

La formule LTL associée est : [] (loginInformation -> <> signIn && tweet)

Figure 42 : Zone de texte de propriété LTL

Après la vérification de la propriété, Spin affiche une erreur ce que signifier que la propriété n'est pas valide.

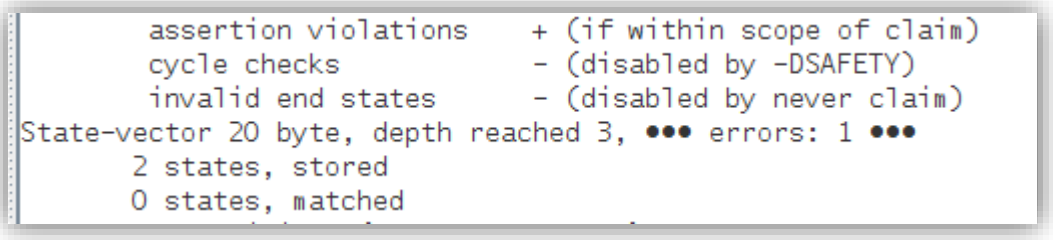

Figure 43 : Résultat de vérification de la propriété LTL

Ce que signifie que ce n'est pas toujours après login d'information on obtient un résultat correct.

Exemple LTL satisfaite :

On veut vérifier que toujours le nombre d'essais est supérieur ou égale à 0 et inférieur ou égale à 5. On définit cette condition dans le code PROMELA par :

#define p (nbrVer>=0 && nbrVer <=5)

La formule LTL associée est : [] p

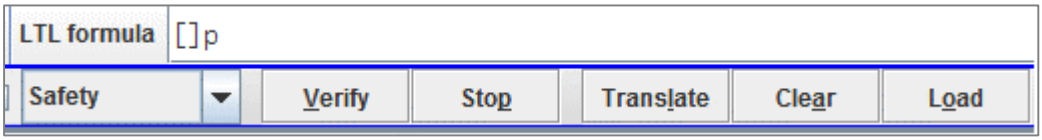

Figure 44 : Zone de texte de propriété LTL

Après la vérification de la propriété, spin n'affiche pas une erreur ce que signifier que la propriété est valide.

Ce que signifie que la boucle est bien générée.

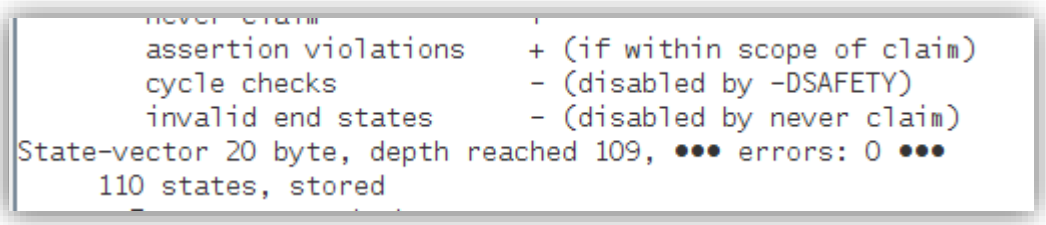

Figure 45 : Résultat de vérification de la propriété LTL

4. Conclusion :

Dans ce chapitre, on a exploité l'outil Spin pour la V&V des diagrammes UML, et plus précisément le diagramme des séquences et d'activités.

Quelques articles traitant l'implémentation de diagramme de séquence en PROMELA, mais ne mentionnent que les notations de base de ce diagramme. Ces notations étaient utiles pour nous, car on a pu avec ces notions de proposer une méthode pour implémenter n'importe quel diagramme de séquences.

Pour le diagramme d'activités la conversion de diagramme en code PROMELA été faite par des outils non disponibles. Alors on a essayé d'étudier quelques articles pour pouvoir proposer notre propre méthode de conversion.

Conclusion générale

Notre objectif via ce travail est de faire une exploitation de l'outil Spin pour la vérification et la validation des diagrammes UML, plus précisément les diagrammes de séquences et d'activités.

Dans le premier chapitre nous avons parlé de la vérification et la validation des diagrammes UML, et on a cité et détaillé quelques méthodes de vérification des diagrammes UML, telle que : les méthodes de vérification de cohérences entre diagrammes et une autre pour vérifier la cohérence entre diagramme de classes et de séquences, ainsi que des méthodes qui traitent la vérification de chaque diagrammes à part comme les réseaux de Petri.

Dans le deuxième chapitre nous avons présenté quelques outils qui sont développés pour réaliser cette V&V, telle que: UML/Analyzer qui vérifie la cohérence entre les diagrammes. Les outils vUML, MINERVA, et Hugo/RT sont des outils pour convertir les différents modèles UML en code PROMELA et les vérifiés par l'outil Spin. Certains de ces outils sont libres et d'autre non comme vUML. On a remarqué que l'outil Spin est largement utilisé par ces outils.

Enfin dans le troisième chapitre on a exploité l'outil Spin pour la V&V des diagrammes de séquences et d'activités. Pour le diagramme de séquences on a proposé une méthode pour traduire n'importe quel diagramme de séquences en PROMELA et expliqué comment vérifier les propriétés LTL. Pour le diagramme d'activités on a cité et exploité notre propre méthode de conversion vers le code PROMELA et le vérifié par des formules LTL.

Comme perspectives on souhaite améliorer notre proposition par l'ajout de la notion de temps, et les couloirs d'activité …pour les diagrammes d'activités. On souhaite également étudier la conversion des autres diagrammes UML en code PROMELA pour les vérifier par l'outil Spin.

Comme l'outil Spin est si important pour réaliser la vérification et la validation des diagrammes UML, on souhaite son utilisation effective par les étudiants pour apprendre à vérifier et à valider leurs diagrammes UML.

Annexe A :

La syntaxe de PROMELA (PROcess MEta LAnguage) :

Le langage PROMELA est spécifique à l'outil Spin, dont la syntaxe est très proche de celle du C mais enrichi de quelques primitives de communication.

PROMELA nous permet de définir un ensemble de processus qui interagissent au travers de variables partagées ou de canaux de communication.
Un programme PROMELA est une liste de déclarations de types constantes, variables et processus, suivie d'un point d'entrée du programme (init) [9].

Les variables :

Les variables peuvent être de cinq types de données **(bit, bool, short, byte et int) globales** ou **locales**. PROMELA fournit aussi la possibilité de créer des structures et des tableaux (d'une seule dimension). La définition ci-dessous déclare un ensemble des variables de type basic, un tableau de type **bit** et une structure contenant un champ entier et un autre byte [24].

Les types de base

bit	turn=1;	[0..1]
bool	flag;	[0..1]
byte	counter;	[0..255]
short	s;	[-216-1.. 216 –1]
int	msg;	[-232-1.. 232 –1]

Les tableaux:

bit flags[4];

Définition de structure:

typedef Record

{ int f1; byte f2; }

Record rr;

rr.f1 = 31; [17]

Les canaux de communication :

Les canaux de communication sont utilisés pour le transfert de messages entre les processus.
La communication via ces canaux est généralement asynchrone, mais on peut également définir un port rendez-vous pour la communication synchrone [24].
Les canaux sont déclarés en utilisant le mot clé **chan** suivi du nom du canal et optionnellement de sa longueur et du type des messages qui circulent [19] :

chan name **= [dim] of {t1,t2, ... tn};**
Par exemple:

chan Ouvre_porte **= [0]** of **{byte, bit},**
chan Transfer = **[2]** of **{bit, short, chan};**

- **Ouvre_porte** est un canal synchrone, car sa longueur est 0, ce qui correspond à un rendez-vous.
- **Transfer** est un canal asynchrone, car il peut stocker au plus de deux messages [13].
- **La communication :** Les messages sont insérés et retirés dans un ordre FIFO.
  1- ! Envoi - mettre un message dans un canal : **ch ! expr ;**
  - La valeur de **expr** doit correspondre avec le type de la déclaration de canal.
  - Une instruction d'envoi est exécutable si le canal ne soit pas plein.

  2- ? Réception - reçois un message sur un canal : **ch ? var ;**
  - le message est récupéré depuis le canal et le message est stocké dans **var**.
  - Une instruction de réception est exécutable si le canal ne soit pas vide.

Les processus :

La forme la plus simple de déclaration de processus est :

proctype nom (paramètres d'entrer) {**Les instructions ;**}

Où les paramètres formels sont déclarés comme en C mais sépares par des points-virgules.
Un processus peut être lancé de deux manières :

- **La manière implicite :** qui peut s'appliquer que si la liste de paramètres formels est vide, consiste à préfixer « proctype » par le mot clé **active**. Il est Même possible de lancer un nombre fixe du processus en utilisant la manière implicite.

  Par exemple :
  active proctype ascenseur () {/* un processus 'ascensseur' lance */ …}
  active [3] proctype porte () {/* trois processus 'porte' lances */ ...}

- **La manière explicite :** consiste à utiliser l'instruction **run** dans le point d'entrée du modèle (voir init) ou dans un autre processus :
  run nom (paramètresactuels)

- **les Instructions :**

Les instructions dans PROMELA s'inspirent peu de la syntaxe du C.
Les points communs avec le C sont :

- L'affectation '**=**',
- le test d'égalité des valeurs '**==**',
- l'utilisation des accolades pour délimiter les blocs d'instructions,
- la syntaxe des opérations sur les entiers (**++**, **--**, etc.),
- l'utilisation des étiquettes sur les lignes et l'instruction **goto**,
- l'utilisation des expressions comme instructions.

Le corps d'un processus se compose d'une séquence d'instructions. Une instruction est soit :

- **Exécutable**: l'instruction peut être exécutée immédiatement, comme l'affectation '**=**'. L'instruction **run** est exécutable seulement si un nouveau processus peut être créé.
- **Bloqué**: l'instruction ne peut être exécutée [19], alors elle est exécutable quand sa valeur devient vraie (autre processus changé la valeur des variables partagées).

Une expression est aussi une instruction; il est exécutable si elle évalue à non-zéro [19] :

2 < 3 toujours exécutable
x < 27 exécutable seulement si : **x** inférieur à **27**
3 + x exécutable si : **x** est différent à **–3**

- **atomic { } :**

PROMELA offre également la possibilité de définir des séquences d'exécution atomiques (sans entrelacement avec d'autres processus).
L'affectation, l'expression et le lancement d'un processus sont des instructions atomiques en PROMELA. Les autres instructions ne le sont pas.
Pour rendre atomique un bloc d'instructions, on doit utiliser l'instruction «**atomic**».

Par exemple : **atomic {** stat1; stat2; … statn **}**

D'autres instructions sont à découvrir sur site de Spin.

Point d'entrée du modèle :

L'exécution du système commence par la section **init** qui est exécutée en parallèle avec les processus actifs. Le corps de la section **init** est le même que celui des processus: une liste de déclarations de variables locales et une liste d'instructions [9].
Par exemple : **init** { run a(); run b(); }.

Annexe B :

Certaines incohérences et erreurs ont été détectées dans le mémoire du master de l'année universitaire (2012/2013) sur le thème de (sémantique formelle des diagrammes de séquence) [25].

Erreur 1 : Dans la page (45) : pour le code PROMELA suivant :

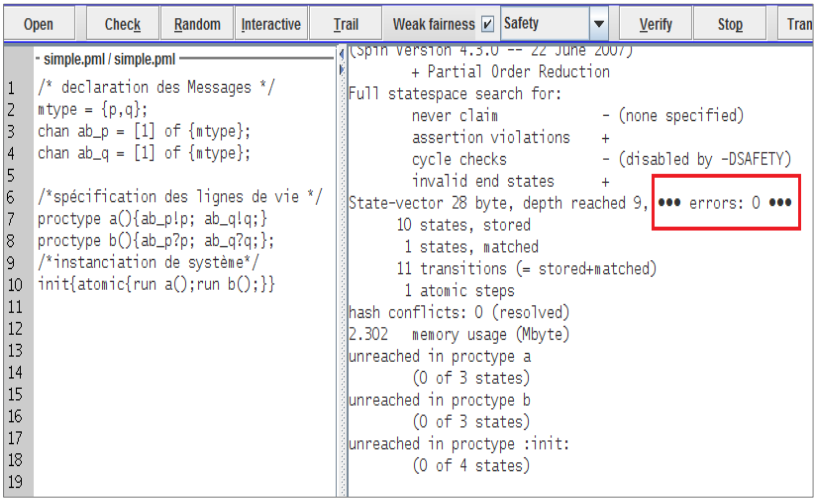

Figure 46 : Code PROMELA d'un diagramme de séquences

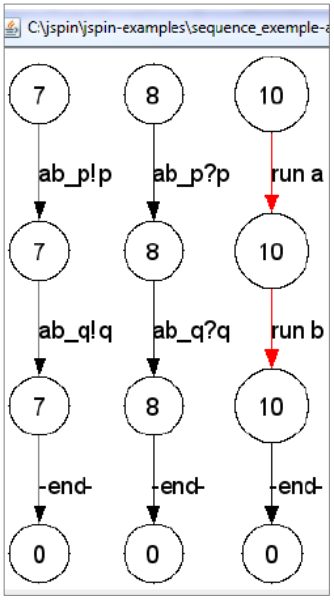

Figure 47 : L'automate de SpinSpider

Les étudiantes ont justifié dans la page suivante que « la vérification avec jSpin donne le résultat (errors=0) et que les flèches rouges dans l'automate signifient qu'il y a une ambiguïté dans ce diagramme alors le diagramme est formellement correct mais sémantiquement non ».

Erreur 2 : Dans la page (49) : pour le code PROMELA suivant :

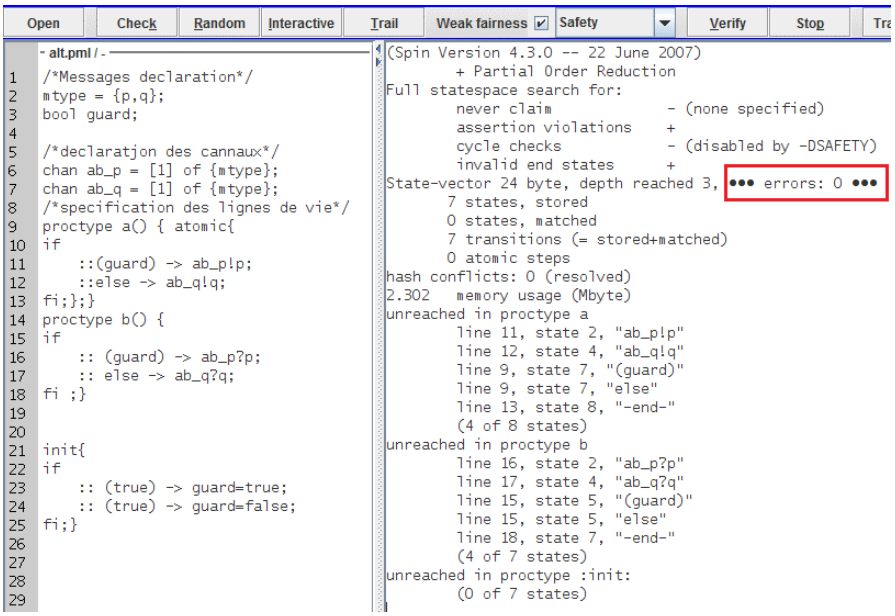

Figure 48 : Code PROMELA d'un diagramme de séquences

L'automate correspond à ce code :

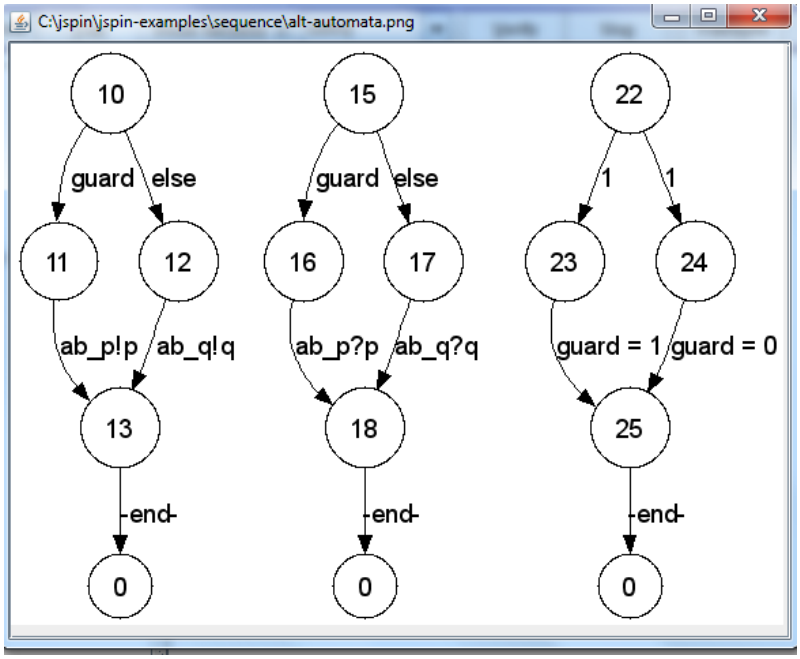

Figure 49 : L'automate SpinSpider

Les étudiantes ont commenté que cet automate ne contient aucune flèche rouge ce qui signifie que le diagramme est correct. Le problème est que les instructions du lancement des processus ne sont pas ajouter dans le code PROMELA et si on ajoute les instructions de lancement des processus [run a() ; run b() ;] le résultat obtenu est représenté dans la figure suivante :

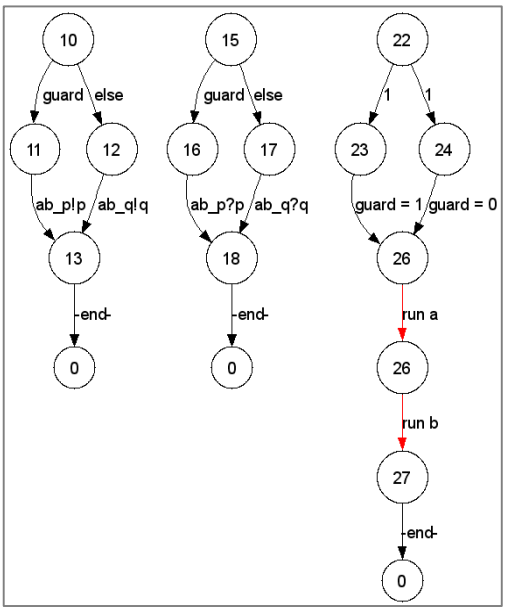

Figure 50 : L'automate SpinSpider

Mais ça n'a rien avoir avec la vérification, parce que Spin ne fait pas la vérification automatique. La vérification ce fait par les formules LTL et la preuve : si on enlève l'instruction de la réception de message p (ab_p?p) la ligne (8) dans l'exemple précédent, et en lance la vérification, le résultat obtenu dans la figure suivante :

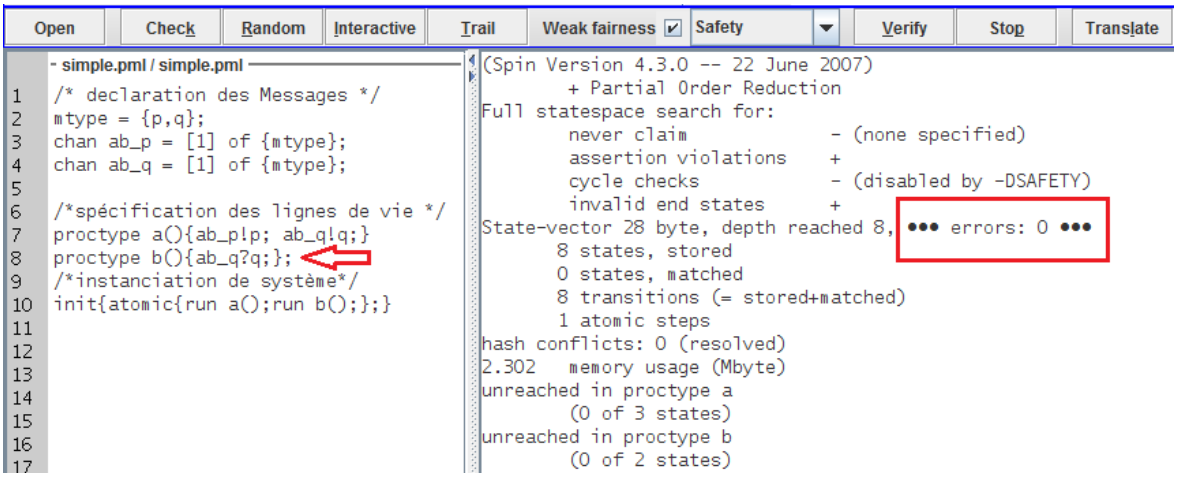

Figure 51 : Le code PROMELA de diagramme de séquences

Donc, on remarque qu'après la vérification Spin n'affiche aucune erreurs (errors=0).

Et l'automate –figure (52)- aussi n'affiche aucune flèche rouge sur le manque de l'instruction de la réception de message p.

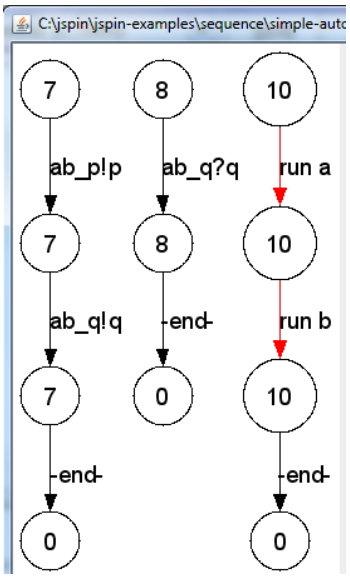

Figure 52 : L'automate SpinSpider

Erreur 3 : La vérification des diagrammes par les formules LTL se fait par l'utilisation des indicateurs dans le code PROMELA. Dans la page (51) les étudiantes ont déclaré la formule LTL suivante :

«!(send && msg_q)U(proc1_b && recieve && msg_p) »

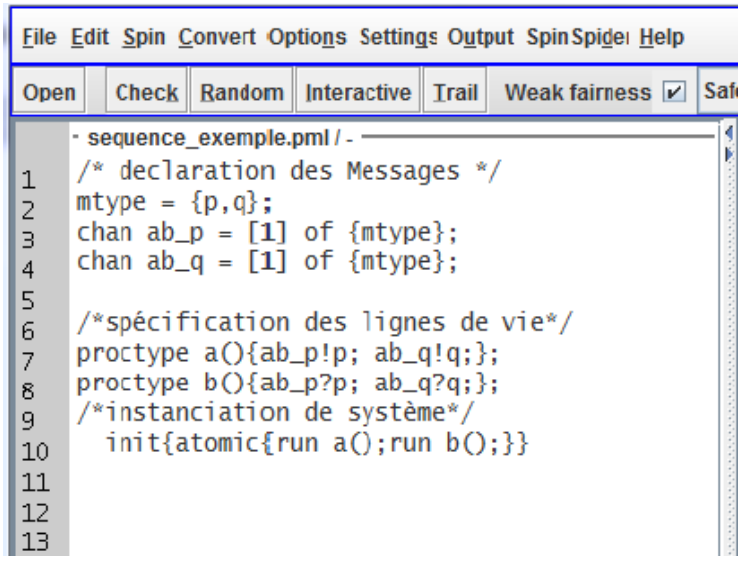

Figure 53 : Code PROMELA de diagramme de séquences

Cette formule a été vérifiée sans l'existence des indicateurs dans le code PROMELA précédent.

Erreur 4 : Dans la page (51) elles ont appliqué la formule LTL précédente sur le deuxième exemple sans ajouter les indicateurs :

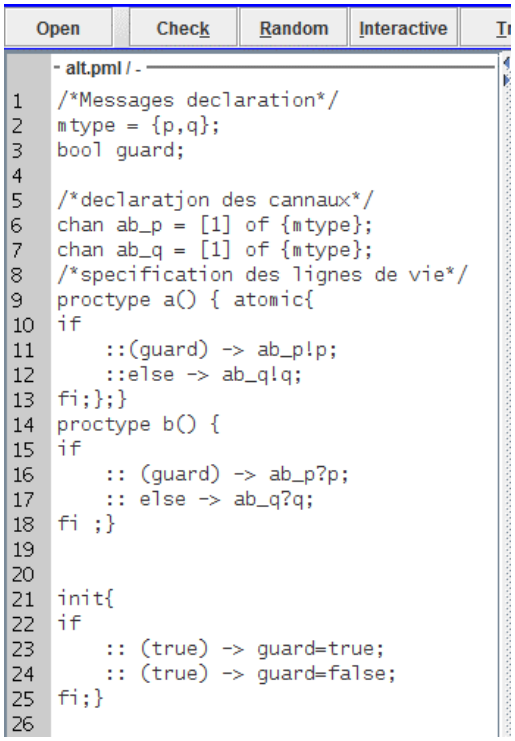

Figure 54 : Code PROMELA de diagramme de

Erreur 5 : Une contradiction obtenue : Les étudiantes ont dit dans la page (51) que l'apparition du contre-exemple signifie que la propriété est satisfaite sinon elle n'est pas satisfaite. Et dans la page (56) elles disent qu'après la vérification du diagramme, Spin montre que cette propriété n'est pas vérifiée alors que le contre-exemple est présenté.

Bibliographie

1- Gabay, Joseph, and David Gabay. UML 2 Analyse et conception-Mise en oeuvre guidée avec études de cas : Mise en œuvre guidée avec études de cas. Dunod, 2008.

2- Cardoso, Janette, Christophe Sibertin-Blanc, and Chantal Soulé-Dupuy. "Une sémantique formelle des diagrammes d'interaction d'UML via les réseaux de Petri." Colloque Francophone sur la Modélisation des Systèmes Réactifs (MSR'01). 2001.

3- Ohnishi, Atsushi. "Management and Verification of the Consistency among UML models." Proc. of Workshop on Knowledge-Based Object-Oriented Software Engineering (KBOOSE). Vol. 20028. 2002.

4- Li, Xiaoshan, Zhiming Liu, and He Jifeng. "A formal semantics of UML sequence diagram." Software Engineering Conference, 2004. Proceedings. 2004 Australian. IEEE, 2004.

5- Lilius, Johan, and I. Porres Paltor. "vUML: A tool for verifying UML models." Automated Software Engineering, 1999. 14th IEEE International Conference on. IEEE, 1999.

6- L'outil Hugo-RT,

URL : http://www.pst.informatik.uni-muenchen.de/projekte/hugo/index.html. (dernière visite 31/05/2015)

7- Campbell, Laura A., et al. "Automatically detecting and visualizing errors in UML diagrams." Requirements Engineering 7.4 (2002): 264-287.

8- Egyed, Alexander. "Uml/analyzer: A tool for the instant consistency checking of uml models." Software Engineering, 2007. ICSE 2007. 29th International Conference on. IEEE, 2007.

9- Laboratoire d'Informatique Algorithmique,

URL : http://www.liafa.jussieu.fr/~cenea/mc/description_Spin.pdf , Université Paris Diderot - Paris 7. (dernière visite 31/05/2015)

10- Oarga, Raveca-Maria. Vérification à la volée de contraintes OCL étendues sur des modèles UML. Diss. École polytechnique, 2006.

11- Computer Science and Engineering Michigan State University,

URL : http://www.cse.msu.edu/~cse470/PromelaManual/ltl.html , (dernière visite 31/05/2015)

12- Laxman, Parne Balu. Validation of UML Models for Interactive Systems with CPN and SPIN. Diss. 2013.

13- Lionel Duroyon. DTest : un Framework de Tests pour Applications Distribuées. Onera Centre de Toulouse Juin 2008.

14- Ben-Ari, Mordechai Moti. "JSPIN-Java GUI for SPIN User's Guide." (2010).

15- Ben-Ari, Mordechai. Principles of the Spin model checker. Springer Science & Business Media, 2008.

16- L'interface de l'outil Spin en ligne,

URL : http://cse-212294.cse.chalmers.se/sefm/spin/, (dernière visite 31/05/2015)

17- Ait Oubelli, Mouna, et al. "From UML 2.0 Sequence Diagrams to PROMELA code by Graph Transformation Using AToM 3." (2011).

18- Leue Stefan, and Peter B. Ladkin. Implementing Message Sequence Charts in Promela - Preliminary Extended Abstract-. Proceedings of the First SPIN Workshop. INRS T'el'ecommunications. 1995.

19- Ruys, Theo C. "Spin beginners' tutorial." SPIN 2002 (2002): 1.

20- Lima, Vitor, et al. "Formal verification and validation of UML 2.0 sequence diagrams using source and destination of messages." Electronic Notes in Theoretical Computer Science 254 (2009): 143-160.

21- Site UML en français URL : http://uml.free.fr/cours/i-p21.html , (dernière visite 31/05/2015).

22- Jing, Li, Li Jinhua, and Zhang Fangning. "Model checking UML activity diagrams with SPIN". Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on. IEEE, 2009.

23- Yamada, Y., and K. Wasaki. "Automatic generation of SPIN model checking code from UML activity diagram and its application to Web application design." Digital Content, Multimedia Technology and its Applications (IDCTA), 2011 7th International Conference on. IEEE, 2011.

24- Chami, Aida. "Vérification de processus BPEL à l'aide de promela-spin." (2008).

25- Ben benlghit fatiha, Rahmouni Naima. "Sémantique formelle des diagrammes de séqunce". Année Universitaire (2012/2013)

26- Edraw, site fournit des solutions de visualisation pour logiciels-,

URL: https://www.edrawsoft.com/template-microblog-uml-activity.php , (dernière visite 31/05/2015)