UFR SCIENCES ET TECHNIQUES

Rapport de stage
Intégration de propriétés PSL dans des spécifications hétérogènes
Research Project - X2II050

Réalisation : Mattis LE FALHUN
Ossama MOUSTAFA

Période : 19/01/2018 - 23/05/2018

Équipe : AeLoS

2017 - 2018

Remerciements

Nous adressons nos remerciements à **J. Christian ATTIOGBE**, notre maître de stage, qui nous a guidé et fait profiter de ses connaissances et de ses conseils avisés tout au long de ce stage.

Nous remercions aussi **Gerson SUNYE** pour ses cours d'analyse, de conception et de mise en oeuvre de logiciels, qui nous ont aidés à appréhender le sujet de notre stage plus sereinement.

Nous remercions aussi **Hala SKAF MOLLI** pour sa gestion de l'organisation des stages.

Enfin, nous remercions **l'équipe AeLoS** pour son accueil et son ambiance de travail agréable.

Mattis LE FALHUN
Ossama MOUSTAFA

RÉSUMÉ

Notre stage de recherche a eu pour objectif de chercher une solution pour appliquer des propriétés PSL à des systèmes hétérogènes. Cette mission a fait appel à un grand nombre de connaissances que nous avons acquis tout au long de notre licence et de cette première année de master. Pour illustrer les sytèmes hétérogènes, nous avons utilisé des diagrammes SysML, une extension d'UML. Côté langage de propriété, c'est PSL qui a été utilisé, un langage formel qui permet de réaliser une spécification matérielle à l'aide de propriétés et d'assertions. Tout cela dans le but de définir un ensemble de pratiques pour appliquer du PSL à des systèmes, des composants logiciel, qui ne parlent pas la même langue, le même paradigme. Nous retiendrons de ce stage nos compétences forgées dans le domaine de la recherche et en particulier de l'architecture et de la qualité logicielle ainsi que le travail en équipe au sein d'une équipe de chercheurs.

ABSTRACT

Our research internship aimed to find a solution to apply PSL properties to heterogeneous systems. This mission used a lot of knowledge that we acquired throughout our license and this first year of master. To illustrate the heterogeneous systems, we used SysML diagrams, an extension of UML. On the property language side, PSL has been used, a formal language that allows for a hardware specification using properties and assertions. All this in order to define a set of practice to apply PSL to systems, software components, which does not speak the same language, the same paradigm. We will retain from this internship our skills forged in the field of research and in particular architecture and software quality as well as teamwork within a team of researchers.

Mots-clés : Systèmes hétérogènes, PSL, SysML, Propriétés

Table des matières

Introduction

C'est dans le cadre de notre première année de Master Architecture Logiciel (ALMA) à l'université de Nantes, que nous devions réaliser un stage de recherche de 3 mois, au cours de notre second semestre. Au sein de l'équipe de recherche universitaire Architectures et Logiciels Sûrs (AeLoS), nous avons choisi le sujet suivant : Intégration des propriétés Property Specification Language (PSL) dans des systèmes hétérogènes. L'intérêt de cette problématique porte sur la gestion de l'hétérogénéité de différents composants, en utilisant PSL : le langage de spécifications de propriétés.

Ce sujet avait été proposé par l'équipe AeLoS, pour un groupe de deux étudiants master. L'encadrant du sujet étant le responsable de l'équipe de recherche, il a confectionné ce sujet afin qu'il s'applique au mieux à notre cursus universitaire. Il nous a permis de mettre en pratique nos connaissances, mais aussi d'apprendre à faire de la recherche.

Pour donner un exemple concret, prenons le cas d'une entreprise qui conçoit son système : au fur et à mesure que ce système évolue, l'entreprise voudra ajouter divers composants, pour bénéficier des différentes fonctionnalités. Elle ne peut pas se permettre de se restreindre à un seul langage de programmation pour son système, si elle veut avoir un choix plus important. Le système peut alors devenir très complexe, et comporter d'importants risques pour l'entreprise, si elle ne le gère pas de la bonne manière. Pour minimiser ces risques, une bonne solution est donc d'utiliser des contrats entre les composants. C'est pourquoi chaque composant se verra appliquer certaines propriétés, sachant qu'il faut aussi des propriétés globales, qui s'appliquent au système entier. Notre but fut donc de lier les propriétés PSL qui servent de contrats entre ces différents composants hétérogènes.

Afin de représenter au mieux divers composants hétérogènes, nous avons utilisé des diagrammes Systems Modeling Language (SysML), une version améliorée d'UML. Effectivement, un composant représenté dans un diagramme peut être programmé dans n'importe quel langage. Pour manipuler ces diagrammes, nous avons donc dû choisir un logiciel de modélisation qui comprend le SysML. Cela nous a permis de nous intéresser uniquement aux propriétés PSL, et de faire des liaisons à travers celles-ci.

Afin d'approfondir ce sujet, nous allons commencer par nous intéresser un peu plus au coeur du sujet. Par la suite, nous expliquerons les différentes technologiques utilisées, ainsi que notre solution au problème. Cela nous permettra donc de commencer par des propriétés locales au sein des composants des diagrammes. Puis nous globaliserons ces propriétés à l'ensemble de diagrammes SysML. Pour finir, il y aura la façon dont nous avons travaillé, ainsi qu'une conclusion qui développera nos résultats.

Première partie

Contexte et objectif du stage

1 Concepts, outils, et méthodes utilisés

1.1 PSL

PSL signifie "Langage de Spécifications par Propriétés". Comme l'indique son nom, il nous permet de décrire les propriétés d'un système. Il nous a été indiqué d'utiliser ce langage dans le sujet qui nous a été donné. Il utilise un système d'horloges, et peut donc appliquer des conditions temporelles, ce qui peut s'avérer très utile. Il a l'avantage d'être très souple : il peut être utilisé dans différents langages (par exemple : en Hardware Description Language (VHDL) et en Verilog). C'est de ce fait que nous nous intéresserons à des systèmes hétérogènes : composés de parties qui seraient programmées ou modélisées dans différents langages.

1.2 SysML

Afin de représenter ces systèmes hétérogènes, nous avons dû utiliser SysML. Cela nous permettra de définir des propriétés PSL sur nos différentes parties, sans savoir dans quels langages ont été codées ces parties. SysML est basé sur UML 2, et permet de manipuler différents types de diagrammes. Nous avons travaillé sur des types de diagrammes assez simples : les diagrammes comportementaux. Cela correspond aux diagrammes d'état-machine, de séquence, et d'activité. Les diagrammes de cas d'utilisation ne seront pas étudiés ici, car ils ne sont pas optimales pour l'application de contrats PSL entre les différents composants.

1.3 Modelio

Pour manipuler nos diagrammes SysML, nous avons choisi d'utiliser le logiciel Modelio, qui est open source. Il nous a suffit d'appliquer un module d'extension SysML dans Modelio. Ce logiciel est relativement simple d'utilisation, et nous permet d'utiliser tous les différents types de diagrammes inclus dans SysML. D'autres alternatives étaient possibles, mais celui-ci étant d'avantage reconnu, nous avons préféré le choisir. Nous l'avons utilisé, de la même manière, à la fois sur des systèmes d'exploitation Windows, et GNU's Not UNIX (GNU)-Linux.

2 Proposition

Nous nous baserons donc sur des diagrammes SysML, et devrons leur appliquer des propriétés PSL. En fonction du type du diagramme, différentes manipulations vont être proposées pour effectuer ces applications. À partir de ce diagramme SysML, nous aurons certaines conditions à respecter durant l'exécution du programme représenté. Notre but est donc de représenter ces conditions en PSL, puis de les faire apparaître dans notre diagramme. Ce travail portera sur des diagrammes comportementaux : activités, états-machines, et séquences. En effet, il fut préférable pour nous de nous attarder plutôt sur ces types de diagrammes, étant donné leur simplicité, à noter que le travail pourrait être poursuit sur les autres types de diagrammes.

2.1 Déductions de propriétés du diagramme

Pour commencer, nous nous intéresserons d'abord aux propriétés qui sont implicites vis à vis du diagramme. Ces propriétés seront explicitées grâce au langage PSL, d'abord à un niveau local aux composants du diagramme, en nous basant sur les traductions que nous avions établi auparavant. Puis, nous globaliserons ces propriétés, afin d'obtenir les spécificités du système entier, comme les pré-conditions et post-conditions.

Par exemple, dans le cas d'une modélisation d'une voiture avec ses composants, nous expliciterons en PSL le fait que la vitesse doit augmenter lors de l'accélération, et qu'elle doit décroître lors du freinage. Puis, lors de la globalisation des propriétés, nous expliquerons que la voiture doit avoir une vitesse nulle avant le démarrage, et à l'arrêt.

Ici, aucune modification ne sera apportée au diagramme ; en effet, nous ne ferons que déduire des propriétés PSL d'après les spécificités déjà existantes de ce système. La globalisation de ces propriétés permettra notamment de distinguer les pré-conditions et post-conditions du système, ainsi que les contraintes intérieures du système.

2.2 Ajout de nouvelles propriétés au diagramme

Ce cas est utile, surtout dans le cadre de la spécification du système, si certaines nouvelles propriétés doivent être appliquées. Cela permet d'ajouter de nouvelles propriétés à tout moment, même si elles n'étaient pas prévues auparavant.

Par exemple, toujours dans le cas de la voiture, on peut ajouter de nouvelles fonctionnalité au système, comme le fait que la voiture devrait s'éteindre toute seule si elle reste à l'arrêt trop longtemps. Cette nouvelle propriété de *timeout* sera alors traduite en PSL, puis appliquée au diagramme. Cela pourra alors impliquer une modification (légère ou importante) au niveau de la transition qui passe de l'état du moteur allumé à éteint.

Dans cette seconde partie, les nouvelles propriétés apporteront des modifications visibles au diagramme. Ces modifications pourront être légères, comme de simples notes ajoutées au diagramme, jusqu'à des modifications importantes, comme des ajouts/suppressions d'entités/transitions.

Deuxième partie

Expérimentations

3 Explications des traductions SysML - PSL

3.1 Cadre de nos travaux

Afin de tester notre solution au problème, il nous faut tout d'abord l'expérimenter. Pour commencer ceci, nous allons tout d'abord analyser des cas simples (composants/sous-systèmes), afin de bien nous apercevoir des liens entre SysML et PSL.

Comme annoncé auparavant, nous allons nous intéresser à des diagrammes comportementaux : diagramme d'activité, diagramme d'état-machine, et diagramme de séquence. Par la suite, une fois des traductions SysML-PSL faîtes (au chapitre suivant), nous les utiliserons dans des cas plus globaux, pour mieux expérimenter celles-ci.

3.2 Définitions de variables PSL

Afin de traduire les propriétés des diagrammes SysML en PSL, nous créerons des variables booléennes (PSL). Afin de représenter l'activation des modules, nous faisons le choix d'introduire des variables de la forme "`STATE_X`", avec "`X`" correspondant au nom du module en majuscules.

De plus, d'autres variables seront ajoutées, afin de représenter certaines valeurs des sous-systèmes. Elles seront internes à certains modules. Pour ne pas confondre entre elles les variables internes aux modules, et pour facilement comprendre ce qu'elles représentent, nous faisons parfois le choix de les exprimer de la manière suivante : `A#B` correspondra à la variable `B` interne au module `A`.

Ces variables seront nécessaires étant données qu'elles seront utilisées par plusieurs composants, comme nous pourrons le remarquer dans les diagrammes SysML. Nous expliquerons le choix de ces variables et leurs correspondances vis-à-vis des diagrammes.

3.3 Pré-conditions et post-conditions en PSL

Pour certaines entités, il nous sera nécessaire de définir des pré-conditions et post-conditions. Une façon simple d'exprimer ceci en PSL, est d'utiliser les fonctions `before`, et `next()`.

Pour définir les pré-conditions, il existe différentes variantes de `before`, ici, nous nous intéresserons plus particulièrement à deux d'entre elles : `before!`, et `before!_`. "`A before! B`" signifie que `A` doit être activé strictement avant `B`, tandis que "`A before!_ B`" signifie que `A` doit être activé avant, ou en même temps que `B`. Voici un exemple générique de pré-conditions pour un module :
`assert always ((pré-condition1 & pré-condition2) before!_ STATE_MODULE );`.

Pour les post-conditions, `assert always(STATE_A[*] → next(B & !STATE_A));` signifie que si A est actif, alors, quand il ne le sera plus, `B` devra être vrai.

3.4 Structures des applications

Pour représenter au mieux les traductions que nous établirons au chapitre suivant, nous ferons correspondre des parties SysML avec des propriétés PSL.

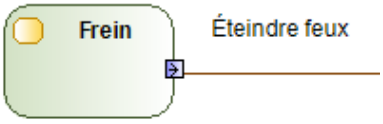
Pour chaque type de diagramme, nous analyserons plusieurs cas différents. Ces cas sont alors d'abord expliqués par un exemple concret, puis rendus génériques afin qu'ils puissent plus facilement être réutilisés dans d'autres systèmes.

4 Applications des traductions

4.1 Diagrammes d'activité

4.1.1 Output - Exemple concret

Diagramme SysML



Booléens PSL

- `STATE_FREIN` : Booléen représentant l'activation du module "`Frein`".
- `ÉteindreFeux` : Booléen représentant la sortie "`Éteindre feux`".

Propriété du module

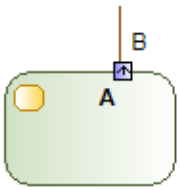
Après avoir freiné, les feux de freinage doivent être ré-éteints.
Cela correspond alors à la post-condition du module "`Frein`".

Traductions en PSL

```
assert always ( STATE_FREIN[*] → next(ÉteindreFeux & !STATE_FREIN) );
```

4.1.2 Output - Générique

Diagramme SysML



Booléens PSL

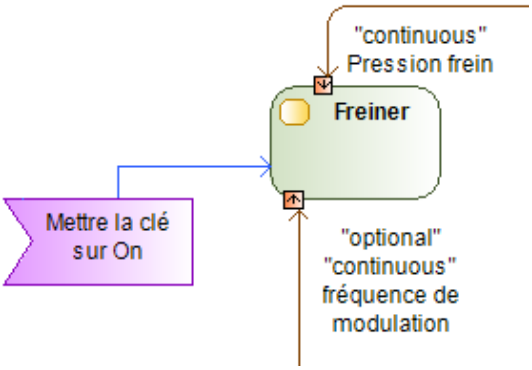
- `STATE_A` : Booléen représentant l'activation du module "`A`".
- `B` : Booléen représentant la sortie "`B`".

Propriété en PSL

```
assert always ( STATE_A[*] → next(B & !STATE_A) );
```

4.1.3 Input - Exemple concret

Diagramme SysML



Booléens PSL

Pour traduire les propriétés, nous aurons besoin de 3 booléens :

- `CleSurOn` : Booléen représentant le signal "`Mettre la clé sur On`"
- `PressionFrein` : Booléen représentant l'entrée "`Pression frein`".
- `STATE_FREINER` : Booléen représentant l'activation du module "`Freiner`".

Propriété du module

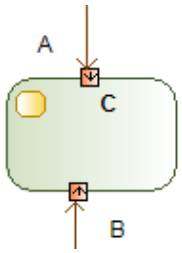
Pour que le frein soit activé, il faut que la clé soit sur On, et que le conducteur exerce une pression au niveau de la pédale de frein. Cela correspond alors à la pré-condition du module "`Freiner`". On remarque que l'entrée "`fréquence de modulation`" est "`optional`", donc non obligatoire pour démarrer le module "`Freiner`".

Traduction en PSL

```
assert always ((CleSurOn & PressionFrein) before!_ STATE_FREINER ) );
```

4.1.4 Input - Générique

Diagramme SysML



Booléens PSL

- `A` : Booléen représentant l'entrée "`A`".
- `B` : Booléen représentant l'entrée "`B`".
- `C` : Booléen représentant l'activation du module "`C`".

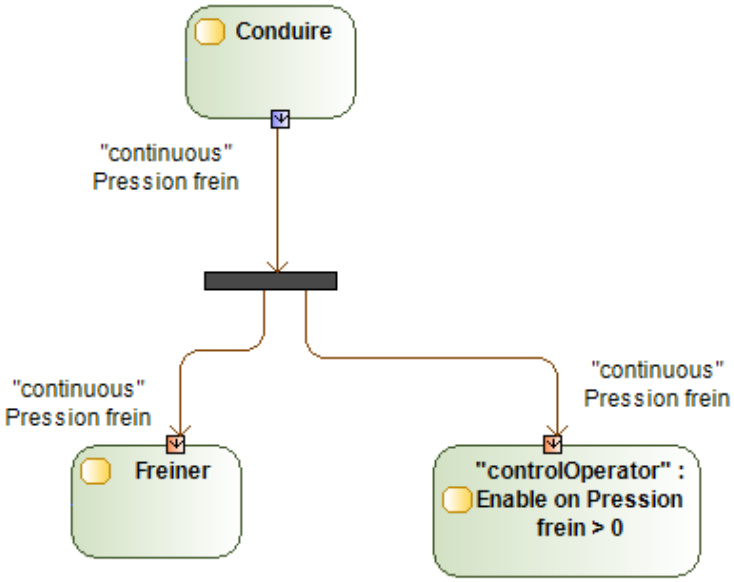
Propriété en PSL

Pré-condition du module `C` :

```
assert always ( (A & B) before!_ C );
```

4.1.5 Fork - Exemple concret

Diagramme SysML



Booléens PSL

Pour traduire ce diagramme d'activité, nous avons besoin de six booléens :

- `CONDUIRE#PressionFrein` :
  Booléen correspondant à la sortie "`Pression frein`" de l'entité "`Conduire`".
- `FREINER#PressionFrein` :
  Booléen correspondant à l'entrée "`Pression frein`" de l'entité "`Freiner`".
- `CONTROLOPERATOR#PressionFrein` :
  Booléen correspondant à l'entrée "`Pression frein`" de l'entité "`controlOperator`".
- `STATE_CONDUIRE` :
  Booléen correspondant à l'activation du module "`Conduire`".
- `STATE_FREINER` :
  Booléen correspondant à l'action du module "`Freiner`".
- `STATE_CONTROLOPERATOR` :
  Booléen correspondant à l'action du module "`controlOperator`".

Propriétés de ces modules

Après que le conducteur ait effectué une pression au niveau du frein, le module de freinage et l'opérateur de contrôle doivent recevoir ce signal de pression. Ces actions représentent une post-condition de l'interface du conducteur, et les pré-conditions des modules de freinage et de "`controlOperator`". On remarque que le fork implique que la sortie de l'interface du conducteur interagit avec plusieurs modules.

Une autre remarque importante est le fait que les "`Pression frein`" soient "`continuous`" : en effet, cela signifie que la transition peut rester active en continue pendant un certain temps. C'est pourquoi, dans la traduction en PSL, la sortie "`CONDUIRE#PRESSIONFREIN`" n'impliquera pas sa négation : elle continuera à impliquer les entrées "`Pression frein`" tant qu'elle sera active.

Traduction en PSL :

Post-condition de `CONDUIRE` :

```
assert always ( STATE_CONDUIRE[*]
→ next( CONDUIRE#PressionFrein & !STATE_CONDUIRE ) );
```

Pré-condition de `FREINER` et `CONTROLOPERATOR` :

```
assert always ((FREINER#PressionFrein & CONTROLOPERATOR#PressionFrein)
              before!_ (STATE_FREINER & STATE_CONTROLOPERATOR));
```

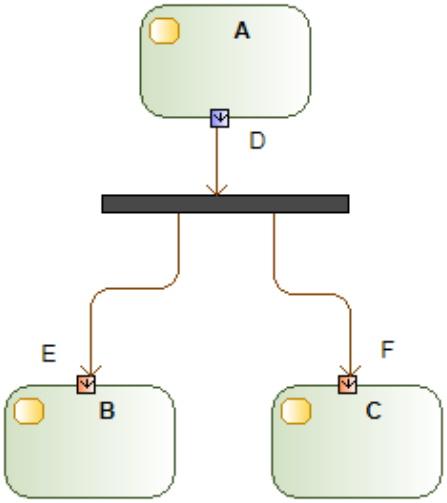
Attention, ici, étant donné que c'est une pré-condition sur deux modules à la fois, on remarquera que les modules n'ont pas de protection spécifique à eux-même. En effet, un seul des deux modules pourrait être activé sans condition n'importe quand. Dans un cas plus globale, il serait important de porter attention à cela.

Liaison entre la post-condition et les pré-conditions :

```
assert always ( CONDUIRE#PressionFrein
→ next ( FREINER#PressionFrein & CONTROLOPERATOR#PressionFrein ) );
```

4.1.6 Fork - Générique

Diagramme SysML



Booléens PSL

Comme pour l'exemple concret, ici aussi nous avons besoin de 6 booléens :

- `A#D` : Booléen correspondant à la sortie "`D`" du module "`A`".
- `B#E` : Booléen correspondant à l'entrée "`E`" du module "`B`".
- `C#F` : Booléen correspondant à l'entrée "`F`" du module "`C`".
- `STATE_A` : Booléen correspondant à l'activation du module "`A`".
- `STATE_B` : Booléen correspondant à l'activation du module "`B`".
- `STATE_C` : Booléen correspondant à l'activation du module "`C`".

Propriétés en PSL

Post-condition du module `A` :

```
assert always ( STATE_A[*] → next( A#D & !STATE_A ));
```

Pré-conditions des modules `B` et `C` :

```
assert always ( B#E before! STATE_B ) ; assert always (C#F before! STATE_C);
```

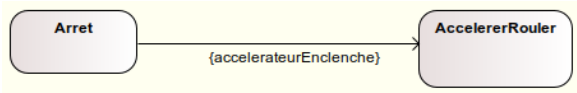
Liaison entre la post-condition et les pré-conditions :

```
assert always (A#D → next(B#E & C#F));
```

4.2 Diagrammes d'état-machine

4.2.1 Transition conditionnelle - Exemple concret

Diagramme SysML



Booléens PSL

Pour ce diagramme, nous avons besoin de 2 variables :

- `accelerateurEnclenche` : Booléen représentant la transition "`accelerateurEnclenche`".
- `STATE_ACCELERERROULER` : Booléen représentant l'activation de l'état "`AccelererRouler`".

Propriété de ce module

Pour que la voiture puisse rouler (grâce au module `AccelererRouler`), il faut que l'accélérateur soit enclenché.

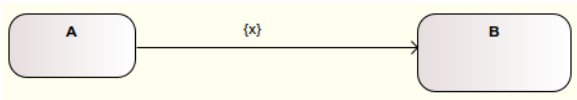
Traduction en PSL

Pré-condition du module `AccelererRouler` :

```
assert always (accelerateurEnclenche before! STATE_ACCELERERROULER);
```

4.2.2 Transition conditionnelle - Générique

Diagramme SysML



Booléens PSL

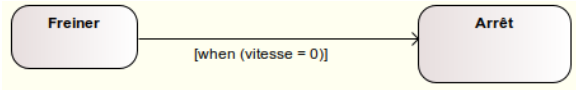
- `x` : Booléen représentant la transition "`x`".
- `STATE_B` : Booléen représentant l'activation du module "`B`".

Propriété en PSL

```
assert always (x before! STATE_B);
```

4.2.3 Transition avec protection - Exemple concret

Diagramme SysML



Booléens PSL

- `STATE_FREINER` : Booléen représentant l'activation du module "`Freiner`".
- `stop` : Booléen représentant l'expression logique "`vitesse = 0`". Cela représente la transition du diagramme. "`vitesse`" étant une variable de type réel interne au système, en PSL on utilisera plutôt le booléen "`stop`".

Propriété de ces modules

Après avoir totalement fini de freiner, la vitesse devient nulle. Une fois cette vitesse nulle, nous n'avons plus besoin de freiner, car la voiture passe alors à l'arrêt. Nous ne pouvons pas appliquer de pré-condition à l'entité "`Arrêt`", étant donné que ce n'est pas une fonctionnalité, mais au contraire une représentation de l'inactivité.

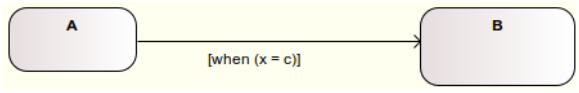
Traduction en PSL

Post-condition du module "`Freiner`" :

```
assert always ( STATE_FREINER[*] → next(stop & !STATE_FREINER) );
```

4.2.4 Transition avec protection - Générique

Diagramme SysML



Booléens PSL

- `STATE_A` : Booléen représentant l'activation du module `A`.
- `condition` : Booléen représentant l'expression logique " `x = c` ".
- `STATE_B` : Booléen représentant l'activation du module `B`.

Propriétés en PSL

Post-condition du module `A` :

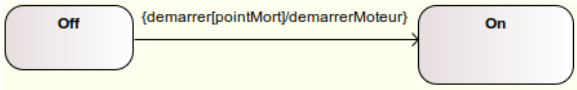
```
assert always (STATE_A[*] → next( condition & !STATE_A ) );
```

Pré-condition du module `B` :

```
assert always (condition before! STATE_B);
```

4.2.5 Transition conditionnelle avec protection - Exemple concret

Diagramme SysML



Booléens PSL

- `demarrer` : Booléen représentant le "trigger", la condition de la transition "`demarrerMoteur`", entre les entités "`Off`" et "`On`".
- `pointMort` : Booléen représentant la protection du trigger "`demarrer`".
- `STATE_ON` : Booléen représentant l'activation du module "`On`".

Propriétés des modules

Afin que la voiture soit dans l'état "`On`", il est nécessaire de la démarrer. Et afin qu'elle soit démarrée, il est nécessaire d'être au point mort.

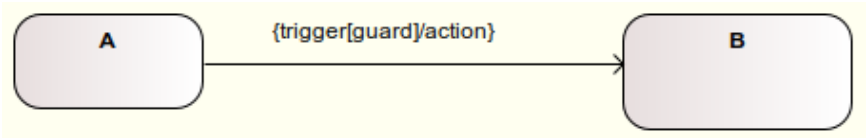
Traduction en PSL

Pré-condition du module "`On`" :

```
assert always(demarrer & pointMort before! STATE_ON);
```

4.2.6 Transition conditionnelle avec protection - Générique

Diagramme SysML



Booléens PSL

- `trigger` : Booléen représentant la condition du la transition "`action`".
- `guard` : Booléen représentant la protection du "`trigger`".
- `STATE_B` : Booléen représentant l'activation du module "`B`".

Propriété en PSL

```
assert always((trigger & guard) before! STATE_B);
```

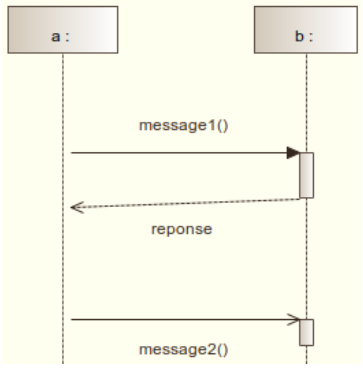
4.3 Diagrammes de séquence

On définit une convention, les pré-post-conditions sur une ligne de vie (`A`) définit un booléen PSL du même nom avec le préfix `STATE_`. Pour cet exemple : `STATE_A`.

Une autre convention que nous définissons : les booléens de transition par des messages. (`message1()`) est traduit en PSL avec le préfix `CALL_`. Pour cet exemple : `CALL_message1`.

4.3.1 Ordre chronologique des diagrammes de séquences

Dans un diagramme de séquence, les messages échangés sont présentés du haut vers le bas le long des lignes de vie, dans un ordre chronologique. Donc chaque message s'effectue AVANT celui d'en dessous. Exemple :



Propriétés PSL :

```
    assert always (SATE_a & CALL_message1 →next (reponse & !CALL_message1));
assert always (SATE_b & CALL_message1 →next (reponse & !CALL_message1));

    assert always ((STATE_a & CALL_message1) before!_ reponse);
assert always ((STATE_b & CALL_message1) before!_ reponse);

    assert always (SATE_a & reponse →next (CALL_message2 & !reponse));
assert always (SATE_b & reponse →next (CALL_message2 & !reponse));

    assert always ((STATE_a & reponse) before!_ CALL_message2);
assert always ((STATE_b & reponse) before!_ CALL_message2);
```

STATE : Les STATE_ doivent être présents dans chaque pré/post-condition, et on répète les pré/post-conditions pour les transitions qui concernent deux lignes de vie.

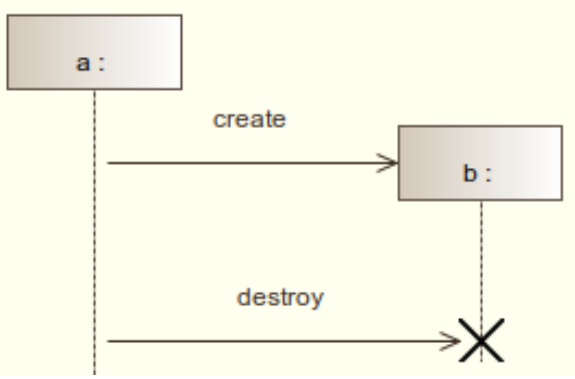
Pré-condition : Chaque transition doit être vraie juste avant la suivante.

Post-condition : Après chaque transition, la suivante est vraie, et la transition courante devient fausse.

NB : Ici quand on parle de transition, c'est à la fois la transition dans le diagramme de séquence et le booléen correspondant dans le PSL.

4.3.2 Create et destroy

Diagramme SysML



Booléens PSL

- `create` : Booléen représentant la transition "`create`".
- `destroy` : Booléen représentant la transition "`destroy`".
- `STATE_B` : Booléen représentant l'activation du module "`b`".

Propriétés de ces modules

Pour que le module "`b`" soit activé, il faut d'abord qu'il soit crée. Puis, pour le désactiver, il devra être détruit.

Traduction en PSL

Pré-condition du module `b` :

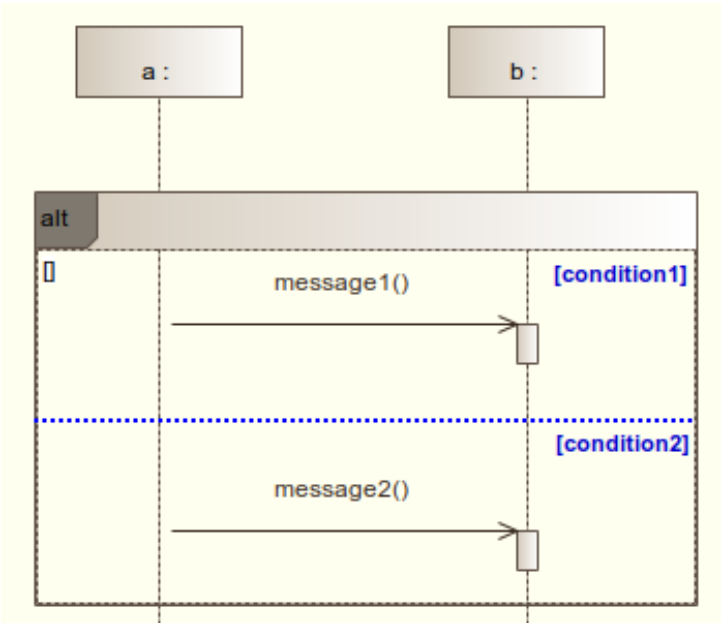
```
assert always (create before! STATE_B);
```

Post-condition du module `b` :

```
assert always (STATE_B[*] → next(!STATE_B & destroy) );
```

4.3.3 Opérateur conditionnel

Diagramme SysML



Booléens PSL

- `STATE_A` : Booléen représentant l'activation du module "`a`".
- `STATE_B` : Booléen représentant l'activation du module "`b`".
- `CALL_message1` : Booléen représentant l'activation de la transition "`message1()`".
- `CALL_message2` : Booléen représentant l'activation de la transition "`message2()`".
- `condition1` : Booléen représentant la condition de la transition "`message1()`".
- `condition2` : Booléen représentant la condition de la transition "`message2()`".
- `DONE_MESSAGE1` : Booléen représentant le fait que le "`message1`" ait déjà été envoyé. Il est initialisé à "faux".
- `DONE_MESSAGE2` : Booléen représentant le fait que le "`message2`" ait déjà été envoyé. Il est initialisé à "faux".

Propriétés de ces modules

Si la `condition1` qui est respectée, alors le "`message1()`" doit sortir du module "`a`". De même, si la `condition2` est respectée, alors le "`message2()`" doit sortir du module "`a`". Dans certains cas, il se peut que la "`condition2`" corresponde à la négation de "`condition1`", soit "`!condition1`" : à ce moment, un et un seul des deux messages doit être envoyé.

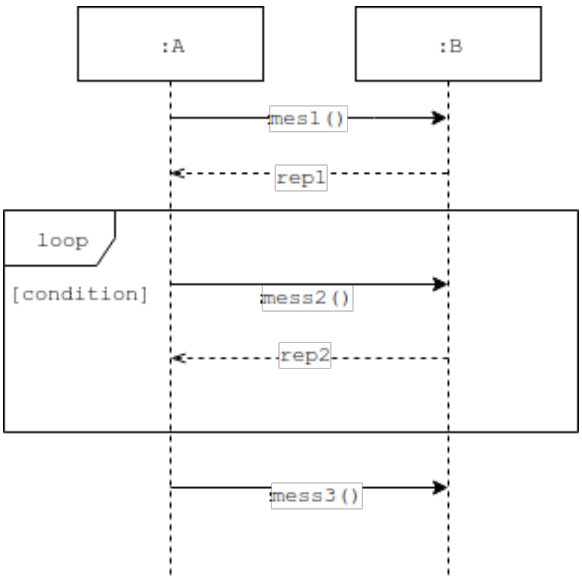
Traduction en PSL

Propriétés des modules "a" et "b" :

```
assert always ((condition1 & !DONE_MESSAGE1) →next(message1 & DONE_MESSAGE1));
assert always ((condition2 & !DONE_MESSAGE2) →next(message2 & DONE_MESSAGE2));
```

4.3.4 Opérateur d'itération

Diagramme SysML



Booléens PSL

- `condition` : Booléen représentant la valeur de la variable "`condition`".
- `CALL_mess1` : Booléen représentant l'envoie du message "`mess1()`".
- `CALL_mess2` : Booléen représentant l'envoie du message "`mess2()`".
- `CALL_mess3` : Booléen représentant l'envoie du message "`mess3()`".
- `rep1` : Booléen représentant l'envoie de la réponse "`rep1`".
- `rep2` : Booléen représentant l'envoie de la réponse "`rep2`".

Propriété de ces modules

La première transition de la boucle doit avec une précondition spéciale. En effet, la transition précédente est : celle d'avant OU la dernière transition de la boucle.

La dernière transition de la boucle doit avec deux post-condition spéciales.

- Si la condition de boucle est encore vrai, on retourne dans la boucle, on revient à la première transition de la boucle.
- Si la condition de boucle est fausse en fin de boucle, on sort de la boucle. On passe à la transition suivante.

Traduction en PSL

Propriété à la fois dans les modules "a" et "b" :

Entrée de boucle :

```
assert always( STATE_A & condition & (rep1 | rep2) before!_( CALL_mess2 ) );
```

Fin de boucle - rebouclage

```
assert always ((STATE_A & condition & rep2) -> next (CALL_mess2 & !rep2));
```

Fin de boucle - sortie de boucle

```
assert always ((STATE_A & !condition & rep2) -> next (CALL_mess3 & !rep2));
```

5 Application à un système

5.1 État-machine

Dans cette partie, nous allons appliquer des propriétés PSL, de manière concrète, à l'ensemble d'un système (ici, un diagramme SysML).

Pour illustrer cette application, nous avons choisi un diagramme d'états-machines de l'Object Management Group (OMG) Systems Modeling Language tutorial :

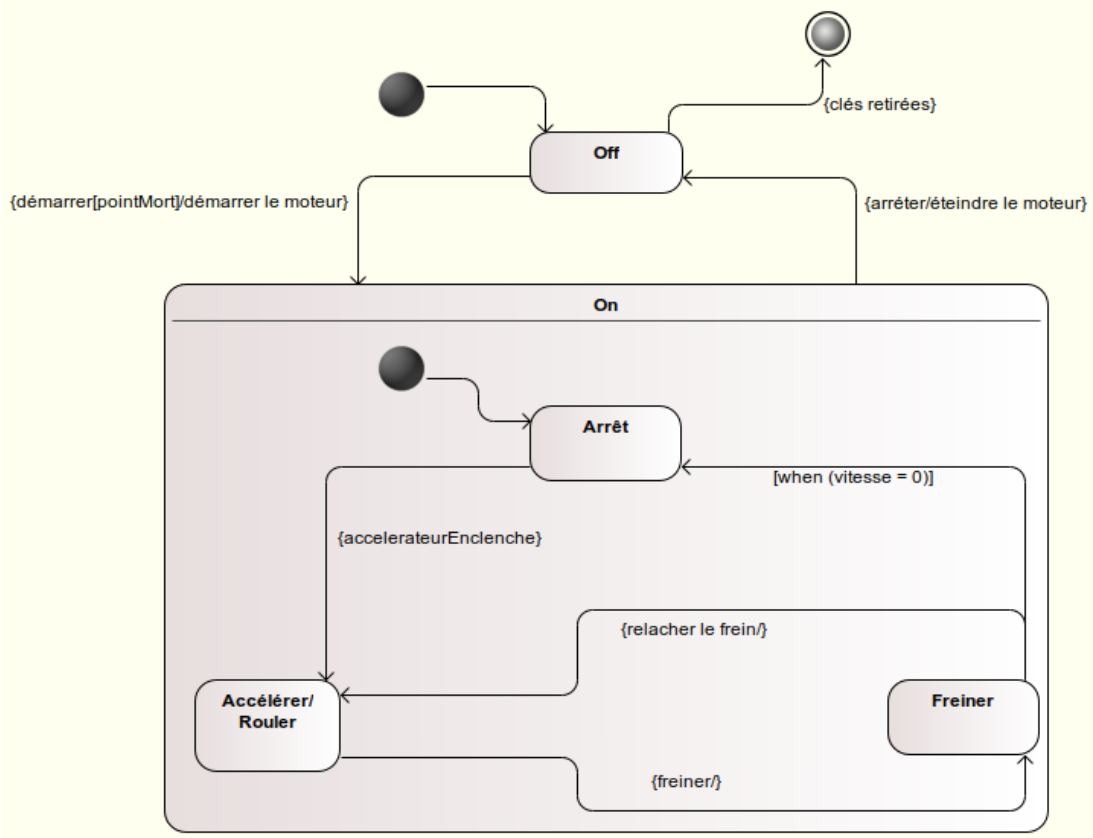


FIGURE 5.1 – Exemple de diagramme états-machine

N'oublions pas que notre but est d'intégrer des propriétés (PSL) à des systèmes hétérogènes. Ici on va donc s'intéresser aux sous-systèmes de notre diagrammes, qui correspondent aux fonctionnalités du système.

Dans notre exemple, on distingue trois grande fonctionnalités (on se permet de simplifier le fonctionnement d'une voiture) :

- Démarrage du moteur
- Accélération
- Freinage

Imaginons que ces fonctionnalités soit des composants logiciels, et qu'elles soient toutes développées avec des langages différents. Même si ces composants sont hétérogènes, ils ont besoin de se comprendre afin que le système (la voiture) fonctionne correctement.

C'est pourquoi nous allons appliquer à chaque composant des propriétés et cette fois, en appliquant toujours le même langage (ici PSL) pour qu'ils puissent se comprendre.

5.1.1 Variables communes au système

Nous avons tout d'abord besoin de variables utilisées par certains composants, et surtout partagées entre eux. Ces variables vont permettre de partager des informations entre les composants tel que la vitesse enclenchée, la pédale enclenchée, etc... Ces variables sont utilisées par les composants mais pas seulement en local, elles vont surtout servir dans les pré/post-conditions.

Booléens de contrôle :

- pointMort : vrai si le levier de vitesse est sur le point mort
- moteurDemarre : vrai si le moteur tourne
- accelerateurEnclenche : vrai si la pédale d'accélération est enclenchée
- freinEnclenche : vrai si la pédale de frein est enclenchée

N.B. pour simplifier les écritures, nous noterons `maVariable` pour un booléen *vrai* et `!maVariable` pour un booléen *faux*.

Grâce à ces variables, on peut maintenant définir les pré/post-conditions de chacune de nos fonctionnalités/composants.

5.1.2 Démarrage du moteur

Pré-conditions

- `pointMort` → Le levier de vitesse doit être au point mort pour que le moteur démarre

Post-conditions

- `moteurDemarre` → après le démarrage du moteur, le moteur tourne. Cette description est triviale mais elle servira à d'autres composants du système

5.1.3 Accélération

Pré-conditions

- `moteurDemarre` → Pour accélérer, le moteur doit être démarré
- `accelerateurEnclenche` → La pédale d'accélération doit être enclenchée

Post-conditions

- `!accelerateurEnclenche` → Après avoir accéléré, la pédale d'accélération est relâchée

5.1.4 Freinage

Pré-conditions

- `freinEnclenche` → Pour freiner, la pédale de frein doit être enclenchée

Post-conditions

- `!freinEnclenche` → Après avoir freiné, la pédale de frein est relâchée

5.1.5 Système résultant

On interprète donc le SysML d'état machine étudié comme un ensemble de composants hétérogènes possédant chacun des pré et post-conditions. Pour illustrer l'hétérogénéité on imagine que le système de démarrage est codé en *Python*, le système d'accélération en *Java* et le système de freinage en *C++*.

Les flèches dans le schéma expliquent que la fonctionnalité d'accélération a besoin que la fonction de démarrage soit terminée.

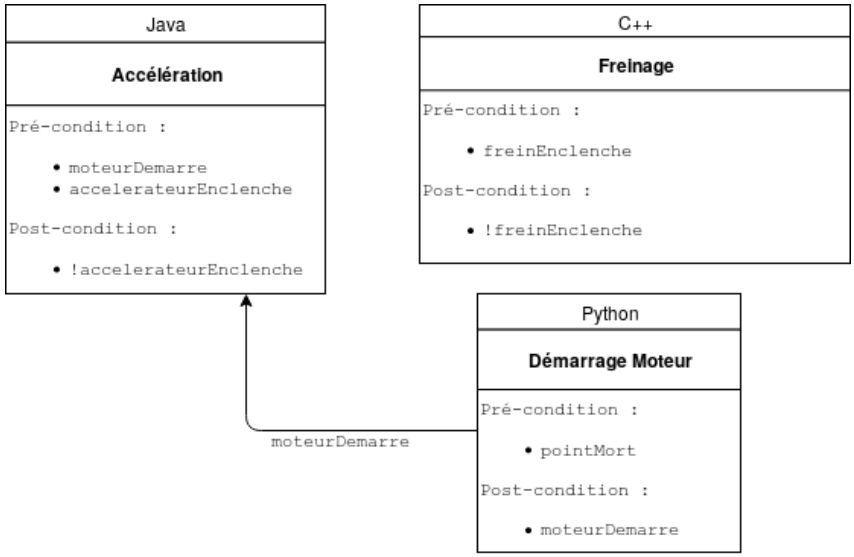


FIGURE 5.2 – Système résultant du state-machine

5.1.6 Traduction PSL

Il nous faut maintenant traduire nos propriétés en un seul langage pour que chaque composant puisse comprendre les propriétés des autres. Vous l'aurez compris, nous utilisons le langage PSL.

Ajout de variable d'état

Rappelons que PSL est un langage, à la base crée pour définir des propriétés sur du *hardware*, et qu'il utilise la notion de temps.

Nous allons avoir besoin, pour la plupart de nos diagrammes comportementaux (et ici le state-machine ne fait pas exception), de variables d'état. En effet, pour spécifier qu'on est dans un état un ou un autre, on a besoin, pour chacun de ces états, d'une variable qui leurs est propre.

Par exemple, si la variable `STATE_ACCELERATION` est vrai, cela signifie qu'on est dans une phase d'acceleration.

Ces variables d'état vont nous servir ici à différencier nos pré-conditions et nos post-conditions. Nos pré-conditions doivent être vérifiées avant d'entrer dans la fonctionnalité (spécifié par le mot-clé `before` en PSL) et nos post-conditions doivent être vérifiées à la fin de la fonctionnalité (spécifié par le mot-clé `next` en PSL).

On définit une norme pour ces variables : elles ont le même nom que l'état auquel elles correspondent, sont écrites en majuscule et possèdent le préfixe : `STATE_`

Traduction

Comme expliqué précédemment, on utilise le mot clé `before` de PSL pour nos pré-conditions de cette manière :

```
assert always ((precondition1 & precondition2) before STATE_ETAT)
```

En différenciant `before!` pour une pré-condition qui doit être vérifiée strictement avant le début de la fonctionnalité dans le temps de `before!_` pour une pré-condition qui doit être vérifiée avant ou en même temps que la fonctionnalité.

Pour les post-conditions, on utilise la fonction `next()` de cette manière :

```
assert always (STATE_ETAT[*] → next(postcondition1 & postcondition2 & !STATE_ETAT))
```

Sans oublié le `[*]` qui spécifie que les postcondition doivent arriver au bout d'un certain temps, et pas strictement juste après le début de la fonctionnalité. On oublie pas non plus le `!STATE_ETAT` qui dit que les post conditions sont vraies quand la fonctionnalitée est terminée.

La traduction en propriétés PSL est illustrée dans la figure 5.3

Bien sur ici, chacune des transaction, des échanges de données et des utilisations de composants par d'autres ne sont pas représentés. Mais à partir de maintenant, grâce à l'intégration des propriétés PSL sur chaque composant, on sera conscient des propriétés d'une fonctionnalité si l'ont veut l'utiliser.

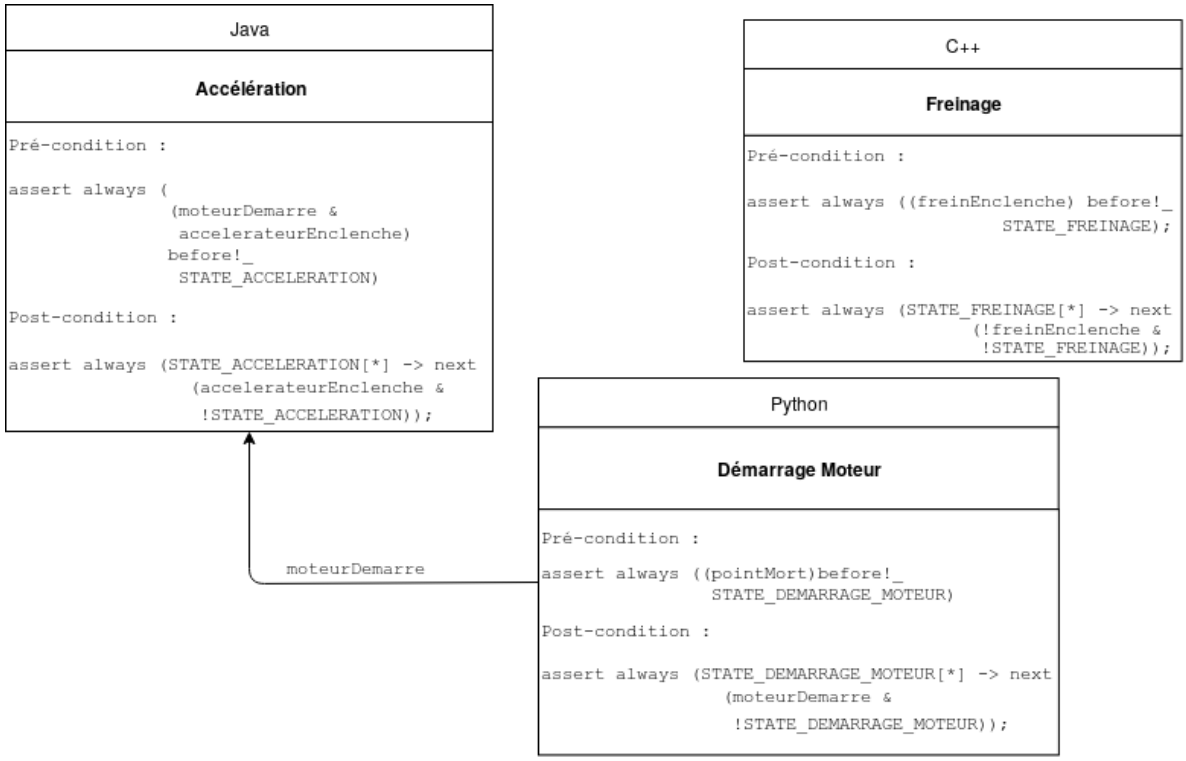


FIGURE 5.3 – Système résultant de la traduction des pré-post-conditions en PSL

5.1.7 Ajout de propriétés

Comme expliqué précédemment, on peut ajouter de nouvelles propriétés oubliées ou ajoutées à la première spécification.

Ici on ajoute une fonctionnalité : Le passage de vitesse. Et on ajoute de nouvelle variables de contrôle pour créer de nouvelles propriétés.

Booléens de contrôle :

- contact : vrai si la clé de la voiture est sur le contact
- embrayageEnclenche : vrai si la pédale d'embrayage est enclenchée
- feuxFrein : vrai si les feux de frein sont allumés

Autres variables de contrôle :

- essence : (Entier) nombre de litres d'essence dans le réservoir
- batterie : (Entier) énergie restante dans la batterie en Watt-heure (Wh)
- vitesse : (Entier) la vitesse de la voiture en km/h
- Freinage#vitesseDebut : (Entier) la vitesse juste avant d'actionner la pédale de frein en km/h
- Freinage#vitesseFin : (Entier) la vitesse juste après avoir relaché la pédale de frein en km/h

On crée de nouvelles pré/post-conditions :

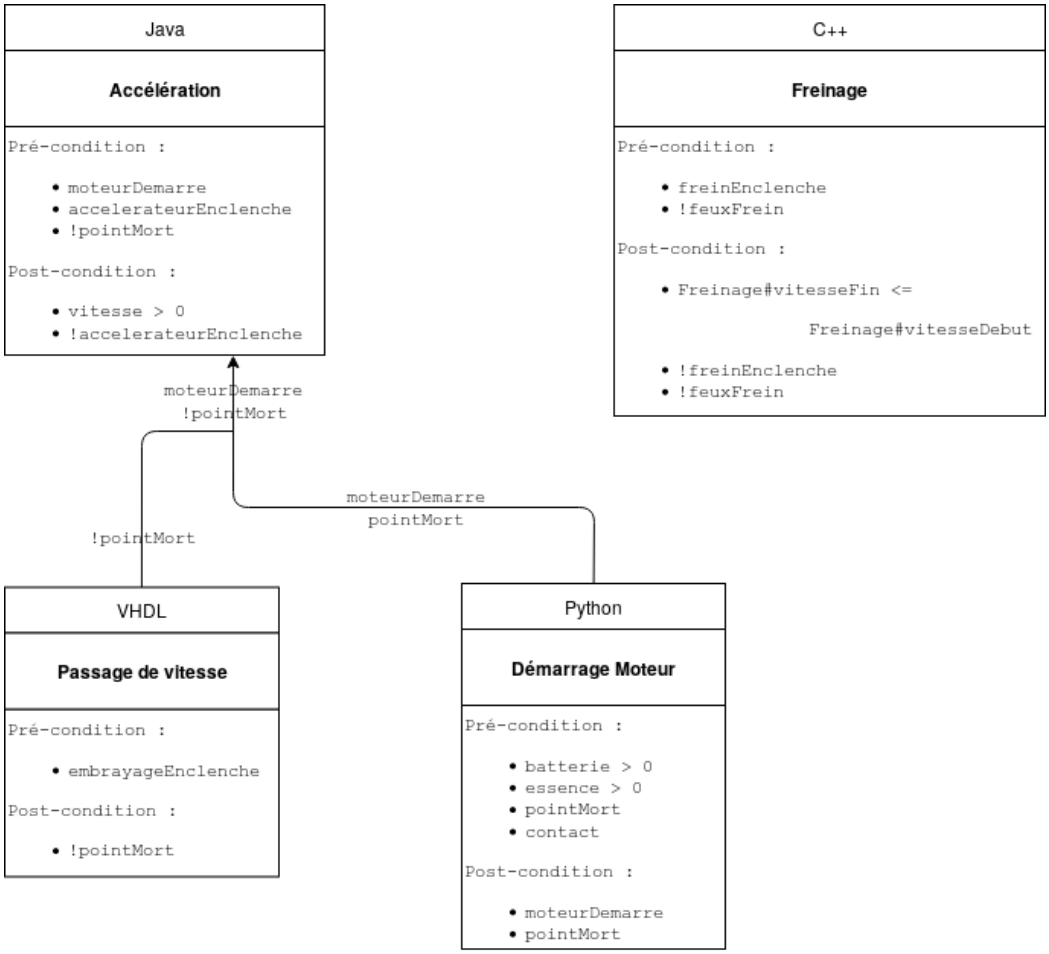


FIGURE 5.4 – Système résultant

On traduit en PSL :

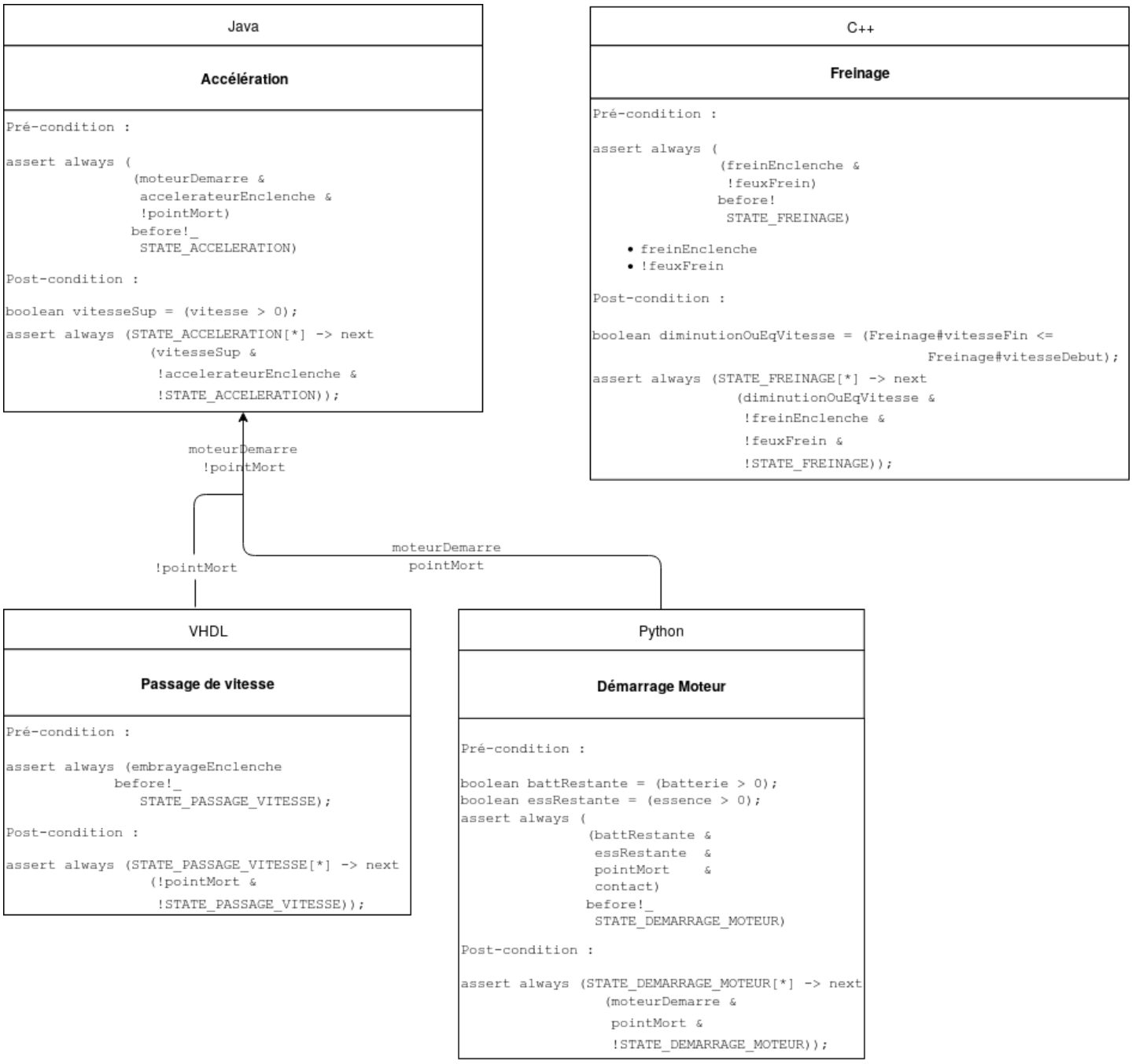


FIGURE 5.5 – Système résultant de la traduction des pré/post-conditions en PSL

5.1.8 Généricité du processus

Toute cette traduction peut être appliquée à n'importe quel diagramme d'état-machine.

Variables d'états

Chaque état qui correspond à une fonctionnalité du système doit avoir une variable PSL qui lui correspond (vrai si on est dans cet état).

Comme expliqué précédemment, on normalise l'écriture des variables d'états : même nom que l'état, écrit en majuscule, avec le préfixe `STATE_`.

Pré/Post-condition

Chaque condition ou protection sur les transitions correspondent à des pré-conditions des états à l'origine des transitions et à des post-conditions des états à la destination des transitions.

Les pré-conditions sont traduites avec le mot clé PSL `before`.

Les post-conditions sont traduites avec le mot clé PSL `next`.

5.2 Séquence

Dans cette partie nous allons recommencer l'application à un nouveau système : diagramme de séquence SysML.

Pour illustrer cette exemple, nous avons choisi un diagramme représentant un scénario de retrait d'argent à un Distributeur automatique de billets (DAB) crée pour l'exemple.

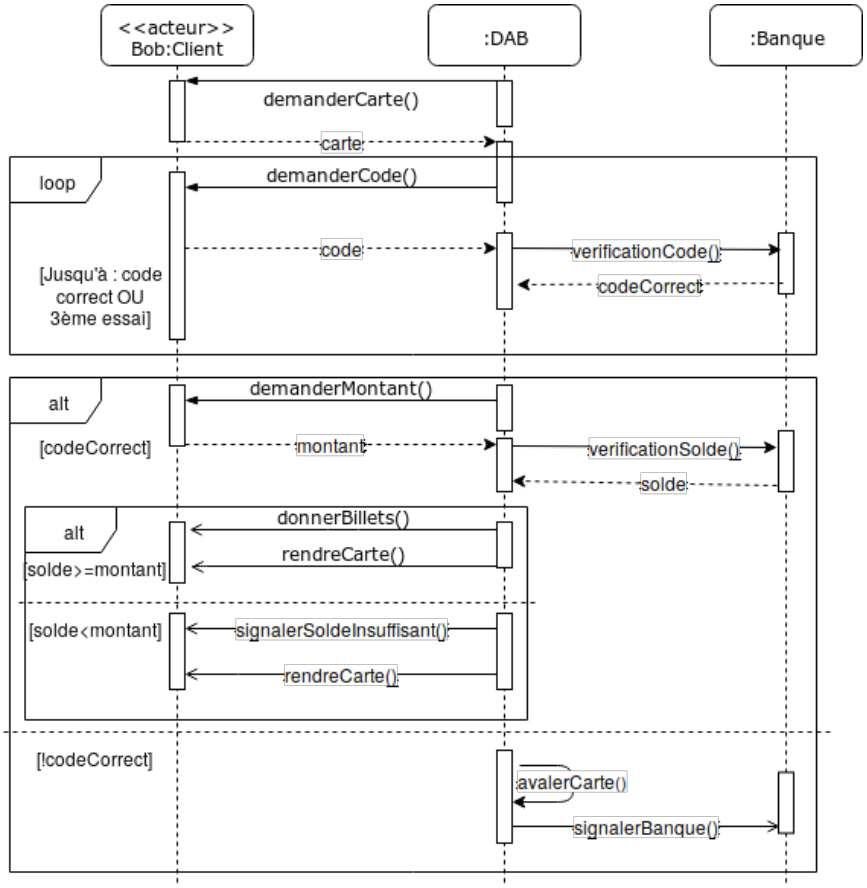


FIGURE 5.6 – Exemple de diagramme de séquence

5.2.1 Variables d'états

Comme précédemment, on définit des variables d'états :

- STATE_DAB : vrai si on est en train d'utiliser le DAB
- STATE_BANQUE : vrai si on utilise le service de banque

5.2.2 Variables globales

Comme pour le diagramme d'états-machine, nous avons besoin de variables pour définir nos pré et post-conditions. On commence par les variables globales au système, autrement dit, les variables qui vont définir nos pré et post-conditions sur les lignes de vie.

- `enMarche` : vrai si le DAB est raccordé à l'éléctricité, allumé et fonctionnel
- `fond` : (Entier) somme d'argent restant dans le DAB
- `serveurOn` : vrai si les serveurs de la banque sont allumés et fonctionnels

5.2.3 Propriétés globales

On ne définit pas de propriétés sur les acteurs du systèmes, balisé «`acteur`»

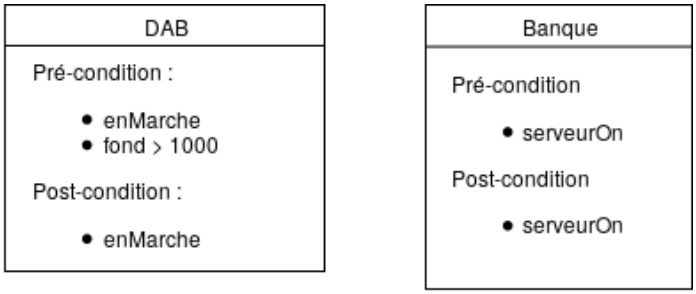


FIGURE 5.7 – Pré-post-conditions globale au système

5.2.4 Traduction PSL

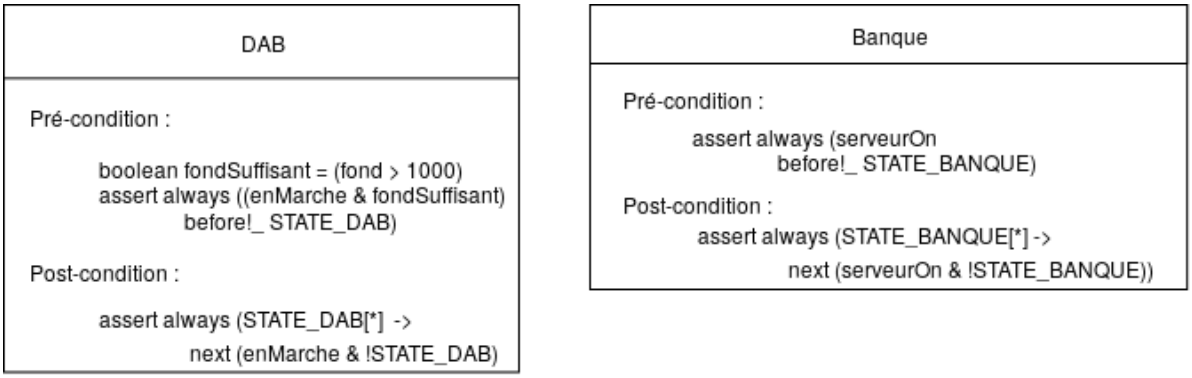


FIGURE 5.8 – Pré-post-conditions globale au système

5.2.5 Variables locales

On a maintenant besoin d'avoir le détail du fonctionnement de chaque composant. Pour cela, on va avoir besoin de deux nouveau types de variables :

Variables de contrôle

- `carteRecu` : vrai si la carte a été inséré par le client
- `codeRecu` : vrai si un code a été entré par le client
- `codeCorrectRecu` : vrai si la `BANQUE` à renvoyé la vérification du code au `DAB`.
- `codeCorrect` : vrai si le code entré est correct
- `montantRecu` : vrai si un montant a été entré par le client

- `soldeRecu` : vrai si la `BANQUE` a renvoyé le solde du client au `DAB`
- `solde` : (Entier) solde du client
- `montant` : (Entier) montant entré par le client
- `carteRendue` : vrai si la carte a été rendue au client

Variable d'exécution

Dans le cas d'un diagramme de séquence, nos propriétés vont être définies sur les méthodes. On va donc avoir de variables représentant celles-ci pour les traduire en propriétés PSL.

Pour différencier ces variables des booléen de contrôle, on ajoute le préfixe `CALL_` devans chacune d'elle.

- `CALL_demanderCarte` : vrai si la méthode demanderCarte() est en cours d'exécution
- `CALL_demanderCode` : vrai si la méthode demanderCode() est en cours d'exécution
- `CALL_verificationCode` : vrai si la méthode verificationCode() est en cours d'exécution
- `CALL_demanderMontant` : vrai si la méthode demanderMontant() est en cours d'exécution
- `CALL_verificationSolde` : vrai si la méthode verificationSolde() est en cours d'exécution
- `CALL_rendreCarte` : vrai si la méthode rendreCarte() est en cours d'exécution
- `CALL_signalerSoldeInsuffisant` : vrai si la méthode signalerSoldeInsuffisant() est en cours d'exécution
- `CALL_donnerBillet` : vrai si la méthode donnerBillet() est en cours d'exécution
- `CALL_avalerCarte` : vrai si la méthode avalerCarte() est en cours d'exécution
- `CALL_signalerBanque` : vrai si la méthode signalerBanque() est en cours d'exécution

5.2.6 Traduction PSL

demanderCarte() - Composant DAB

```
1    assert always ((STATE_DAB) before!_ CALL_demanderCarte);
2    assert always (CALL_demanderCarte[*] -> next (carteRecu & !CALL_demanderCarte));
```

Précondition : Ici, on est sur la première transition de notre diagramme. C'est pourquoi la transition ne possède qu'une variable en précondition, l'entité dont il dépend (Ici `STATE_DAB`).

Postcondition : Ici, la post-condition est classique, dans le `next`, on a la transition suivante et la négation de la transition en cours.

carteRecu - Composant DAB

```
1    assert always ((STATE_DAB & (CALL_demanderCarte)) before!_ carteRecu);
2    assert always (carteRecu[*] -> next (CALL_demanderCode & !carteRecu));
```

demanderCode() - Composant DAB

```
1  boolean infTroisEssai = nbEssai < 3;
2  assert always ((STATE_DAB & (carteRecu | codeCorrectRecu) & !codeCorrect &
       infTroisEssai) before!_ CALL_demanderCode);
3  assert always (CALL_demanderCode[*] -> next (codeRecu) & !CALL_demanderCode);
```

Précondition : précondition spécial de début de boucle expliqué plus tot. Quand on demande le code, soit le système a recu la carte juste avant, soit c'est le retour de la boucle (dernière instruction de la boucle) qui est juste avant, ici codeCorrectRecu. On ajoute aussi les booléens de contrôle de la boucle.

codeRecu - Composant DAB

```
1  boolean infTroisEssai = nbEssai < 3;
2  assert always ((STATE_DAB & CALL_demanderCode & !codeCorrect & infTroisEssai) before!
       _ codeRecu);
3  assert always (codeRecu[*] -> next (CALL_verificationCode & !codeRecu));
```

verificationCode() - Composant DAB

```
1  boolean infTroisEssai = nbEssai < 3;
2  assert always ((STATE_DAB & codeRecu &!codeCorrect & infTroisEssai) before!_
       CALL_verificationCode);
3  assert always (CALL_verificationCode[*] -> next (codeCorrectRecu & !
       CALL_verificationCode));
```

verificationCode() - Composant BANQUE

```
1  boolean infTroisEssai = nbEssai < 3;
2  assert always ((STATE_BANQUE & CALL_verificationCode & !codeCorrect & infTroisEssai)
       before!_ codeCorrectRecu);
3  assert always (CALL_verificationCode[*] -> next (codeCorrectRecu & !
       CALL_verificationCode));
```

Précondition : Ici, on change de composant. En effet, `verificationCode()` est commun au composant `DAB` et `BANQUE`. C'est dans ce composant qu'on ajoute la pré-condition sur `codeCorrectRecu` et on a une répétition de la post-condition de `CALL_verificationCode`.

CodeCorrectRecu - Composant DAB

```
1  boolean infTroisEssai = nbEssai < 3;
2  assert always ((codeCorrectRecu & !codeCorrect & infTroisEssai)[*] -> next (
       CALL_demanderCode & carteRecu));
3  assert always ((codeCorrectRecu & codeCorrect)[*] -> next (CALL_demanderMontant));
4  assert always ((codeCorrectRecu & !codeCorrect & !infTroisEssai)[*] -> next (
       CALL_avalerCarte));
```

Pré-condition : pré-condition de fin de boucle. En fonction des booleéns de contrôle de boucle, soit on revient au début de la boucle, soit on passe à la suite du programme.

demanderMontant() - Composant DAB

```
1    assert always ((STATE_DAB & codeCorrectRecu & codeCorrect) before!_
         CALL_demanderMontant);
2    assert always (CALL_demanderMontant[*] -> next (montantRecu & !CALL_demanderMontant))
         ;
```

Pré-condition : pré-condition de cas alternatifs. On ajoute aux pré-conditions, les booléens de contrôle de cas alternatifs (ici `codeCorrect`).

montantRecu - Composant DAB

```
1    assert always ((STATE_DAB & CALL_demanderMontant & codeCorrect) before!_ montantRecu)
         ;
2    assert always (motantRecu[*] -> next (CALL_verificationSolde & !motantRecu));
```

verificationSolde() - Composant DAB

```
1    assert always ((STATE_DAB & montantRecu & codeCorrect) before!_
         CALL_verificationSolde);
2    assert always (CALL_verificationSolde[*] -> next (soldeRecu & !CALL_verificationSolde
         ));
```

verificationSolde() - Composant BANQUE

```
1    assert always ((STATE_BANQUE & CALL_verificationSolde & codeCorrect) before!_
         soldeRecu);
2    assert always (CALL_verificationSolde[*] -> next (soldeRecu & !CALL_verificationSolde
         ));
```

soldeRecu - Composant DAB

```
1            boolean soldeSuffisant = (solde >= montant);
2            assert always (soldeRecu[*] -> next ((soldeSuffisant & CALL_donnerBillets) |
                 (!soldeSuffisant & signalerSoldeInsuffisant) & !soldeRecu));
```

donerBillets() - Composant DAB

```
1  boolean soldeSuffisant = (solde >= montant);
2  assert always ((STATE_DAB & soldeRecu & codeCorrect & soldeSuffisant) before!_
       CALL_donerBillets);
3  assert always (CALL_donnerBillets[*] -> next (CALL_rendreCarte & !CALL_donnerBillets)
       );
```

signalerSoldeInsuffisant() - Composant DAB

```
1  boolean soldeSuffisant = (solde >= montant);
2  assert always ((STATE_DAB & CALL_verificationSolde & codeCorrect & !soldeSuffisant)
       before!_ CALL_signalerSoldeInsuffisant);
3  assert always (CALL_signalerSoldeInsuffisant[*] -> next (CALL_rendreCarte & !
       CALL_signalerSoldeInsuffisant));
```

rendreCarte() - Composant DAB

```
1  assert always ((STATE_DAB & (CALL_donnerBillets | CALL_signalerSoldeInsuffisant) &
       codeCorrect) before!_ CALL_rendreCarte);
2  assert always (CALL_rendreCarte[*] -> next (carteRendue & !CALL_rendreCarte));
```

avalerCarte() - Composant DAB

```
1  assert always ((STATE_DAB & CALL_verificationCode & codeCorrectRecu & !codeCorrect)
       before!_ CALL_avalerCarte);
2  assert always (CALL_avalerCarte[*] -> next (CALL_signalerBanque & !CALL_avalerCarte))
       ;
```

signalerBanque() - Composant DAB

```
1  assert always ((STATE_DAB & CALL_avalerCarte & !codeCorrect) before!_
       CALL_signalerBanque);
```

5.2.7 Généricité du processus

Toute cette traduction peut être appliquée à n'importe quel diagramme de séquence.

Variables d'états

Chaque ligne de vie qui correspond à un composant du système doit avoir une variable PSL qui lui correspond (vraie si les variables globales du composants sont vraies).

Comme expliqué précédemment, on normalise l'écriture des variables d'états : même nom que l'état, écrit en majuscule, avec le préfixe `STATE_`.

Variables de contrôle

Ces variables correspondent à toutes les réponses du diagramme de séquence ainsi que toutes les variables de contrôle dont on a besoin (Dans notre exemple : `codeCorrect, solde, montant`).

Variables d'exécution

Ces variables correspondent à tous les messages du diagramme de séquence. Comme expliqué, on normalise l'écriture des variables d'exécution : même nom que le message, avec le préfixe `CALL_`.

Pré/Post-condition

Les pré-conditions sont traduites avec le mot clé PSL `before`.

Les post-conditions sont traduites avec le mot clé PSL `next`.

Pré-condition classique : Hors d'une frame, la pré-condition est classique. A gauche du before, on a la variable d'état de la ligne de vie correspondante (on répète la pré-condition si il y a deux lignes de vie) ainsi que la transition précédente (si elle existe). A droite du before, on a notre transition courante.

Post-condition classique : Hors d'une frame, la post-condition est classique. A gauche du next, on a la transition courante, suivit de `[*]` pour se dédouaner du temps d'exécution de la transition. A droite du next, on a la transition suivante ainsi que la négation de la transition courante.

Pré-condition de frame Alternative : Dans une pré-condition d'une transition qui se trouve à l'intérieur d'une frame d'alternative, on rajoute le ou les booléens de la frame (à gauche du before).

Pré-condition de la première transition d'une boucle : Elle doit avoir a gauche du before, la transition qui la précède OU la dernière transition de la boucle dont elle fait partie.

Post-conditions de la dernière transition de la boucle : Elle doit en avoir deux : Si la condition de boucle est vraie, on revient à la première transition de la boucle, sinon, on passe à la transition suivante (et on sort de la boucle).

Troisième partie

Planification et méthodologie

6 Planification

La planification de la mission a été réalisée à l'aide d'un diagramme de gantt que voici :

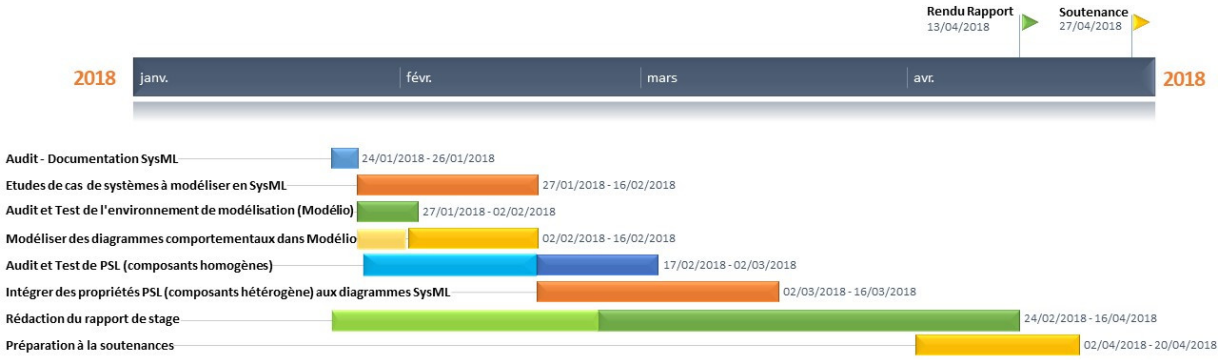


FIGURE 6.1 – Planification de la mission - diagramme de gantt

SysML L'audit et la documentation de sysML c'est fait en plus ou moins 2 semaines. Nous nous sommes concentré sur les diagrammes de comportement et notamment le diagramme d'activités qui est la seul nouveauté apporté par SysML depuis UML. Pour les 3 autres diagrammes (États-Machine, Cas d'utilisation, Séquence) nos connaissances en UML nous ont permis d'avancer assez vite et la documentation a surtout été pour nous des rappels.

Etudes de cas SysML Nous nous somme basé sur le document *OMG SysMLTM Tutorial* de **L'INCOSE** (voir bibliographie) pour nos études de cas. Pour les 4 diagrammes comportementaux, ce tutorial présente le fonctionnement simple d'une voiture. Nous avons utilisé ce cas d'utilisation pour notre étude et nos exemples.

Modélio - Modélisation Nous avons choisit d'utiliser l'environnement de modélisation UML *Modélio* qui supporte aussi le SysML. Cette partie du travail n'a été que de modéliser les exemples décrits à la partie précédente.

Etude de PSL Dans cette partie du travail, nous nous somme documenté sur le langage PSL grâce à de nombreux documents présent dans la bibliographie. Cela nous a permis de nous familiariser avec la syntaxe de ce langage et d'étudier des exemples d'utilisation.

Intégration PSL - SysML Cette partie est le coeur du sujet et arrive donc dans les derniers temps de notre planning.

Rapport La rédaction du rapport de stage a commencé dès le début du projet à l'aide de notes et continue jusqu'au rendu.

Préparation à la soutenance Enfin bien sur, nous passerons par une phase de création de slides et de préparation à la soutenance qui sera réalisée fin mai.

7 Méthodologie

Le stage a été réalisé en binôme et le travail a été partagé équitablement.

Les recherches et la réalisation, n'ont pas été séparé à travers le binôme. Toutes les tâches ont été réalisé à deux, en parallèle. Cela nous a permis de discuter et de comprendre ensemble tous les sujets. Cela nous a semblé primordiale de privilégier la communication au sein du groupe plutôt que de partir chacun de notre coté.

Nous avons avancé ce projet au rythme d'une journée hebdomadaire. Le projet a été ponctué de visites et de discussions avec M. Attiogbé afin de poser des questions, de nous recadrer et de partir dans les bonnes directions.

Notre rapport a été rédigé grâce à Latex. Cela nous a permis de collaborer facilement. De plus, nous avons aussi utilisé google Drive, afin de nous partager plus facilement nos sources. Nombreuses recherches ont été effectuées à la bibliothèque universitaire, ainsi que sur internet, tout en prêtant attention à la fiabilité de nos sources.

Conclusion

Après toutes les expérimentations que nous avons effectué, nous remarquons que les diagrammes SysML ont différentes façons de s'adapter à un système. Ces diagrammes ont servis de représentation des systèmes hétérogènes, qui correspondaient à des logiciels composés de différents modules. En fonction du logiciel concerné, différentes approches peuvent être étudiées : si les modules (hétérogènes) sont actifs en simultanés, alors il serait préférable d'utiliser un diagramme de séquence ; dans le cas contraire, un diagramme d'état-machine pourrait facilement représenter la situation. D'autres représentations auraient pu être utilisées, mais celles-ci étaient d'avantage pratique pour les travaux que nous avons effectué. En effet, les diagrammes SysML ne sont qu'un exemple pour illustrer des systèmes hétérogènes. On peut imaginer des propriétés PSL sur des composants programmés dans plusieurs langages différents, sur des composants web, etc...

Tout au long de nos travaux, nous n'avons jamais été contraints par les différents types de modules. En effet, nous avons réussi à leur appliquer des propriétés PSL, sans nous attarder sur d'éventuels problèmes d'incompatibilité. C'est grâce à ces travaux que nous avons compris toute la flexibilité de PSL : nous avons vu qu'il s'applique de la même manière à différents langages, ce qui leurs permet d'établir des contrats sans difficulté.

Étant donné que ce projet été proposé par l'équipe AeLoS, il a été très orienté architecture logicielle, ce qui correspond parfaitement à la spécificité de notre Master. En effet, les diagrammes SysML ont été de bonnes représentations des architectures, représentant tous les modules d'un logiciel. Cela nous a donc permis de nous spécialiser d'avantage dans ce domaine que nous avons choisi.

Après toutes nos expérimentations, nous nous sommes rendus compte que ces intégrations étaient automatisables. Tous ces travaux manuels que nous avons effectués pourraient être gérer plus simplement par un logiciel dédié à ce domaine. Nous n'avons malheureusement pas eu le temps de créer nous même cela, mais il aurait été intéressant d'approfondir ce sujet.

C'était la première fois que nous faisions de la recherche, et que nous découvrions ce domaine. En dehors de la recherche, des solutions existantes peuvent être réutilisées et adaptées, mais dans ce cadre, c'est à nous même de fournir une solution et des explications. Cela nous a permit d'en apprendre d'avantage sur le travail des chercheurs, notamment sur la méthodologie et les conventions utilisée pour un papier de recherche et une conférence.

Nous tirons un bilan très positif de ce stage de recherche. La mission qui nous a été confiée a fait appel à de nombreuses compétences que nous avons pu mettre au service d'un projet concret. Notre premier année de Master c'est achevée avec une expérience dans la recherche à laquelle nous avons été satisfait de participer.

Ce stage est un réel atout nous permettant d'enrichir et d'étayer notre projet professionnel ainsi que de valider notre choix pour l'informatique. Il a confirmé notre préférence pour le travail en équipe et c'est pourquoi, nous souhaitons continuer nos études en informatique au sein de l'université de sciences, dans le but de décrocher un Master.

Bibliographie

IUT

[1] *Cours et TD d'UML ; J. Christian ATTIOGBE*

[2] *Cours et TD de culture et communication ; Sébastien CAZALAS*

[3] *Cours et TD de Réutilisation ; Dalila TAMZALIT*

FACULTE

[4] *Cours et TD d'introduction à la recherche ; Claude JARD*

[5] *Cours et TD d'introduction à la recherche ; Eric MONFROY*

[6] *Cours et TD d'Analyse, conception et mise en oeuvre de logiciels ; Gerson SUNYE*

[7] *Cours et TD de Design Pattern ; Florian RICHOUX*

PSL

[8] Property Specification Language - Reference Manual - Accelera

[9] Property Specification Language - Jasper Design Automation

[10] Property Specification Language - CE Sharif

SysML

[11] OMG Systems Modeling Language - Tutorial - INCOSE

[12] Pollet Yann, *Systèmes, architectures, intégration : pratique de l'intégration des systèmes et des logiciels avec SysML*, Paris Ellipses DL 2016, Références sciences, 2016, 431 p.

Glossaire

AeLoS Architectures et Logiciels Sûrs. 2

ALMA Architecture Logiciel. 2

contrats Ensemble de propriétés inviolables. 2

DAB Distributeur automatique de billets. 30

frame Définition des limites d'un diagramme, ici pour spécifier les : FOR, IF, ELSE, ... 36

GNU GNU's Not UNIX. 4

OMG Object Management Group. 22

PSL Property Specification Language. 2

SysML Systems Modeling Language. 2

timeout Délai maximum. 5

trigger Déclancheur d'une transition dans un diagramme state-machine. 16

Verilog Langage de description matériel de circuits logiques en électronique. 4

VHDL Hardware Description Language. 4