Université Catholique de Louvain

Ecole Polytechnique de Louvain

Outils d'analyse de modèles de systèmes interactifs ADEPT

Promoteur : Prof. Charles **Pecheur**
Co-promoteur : Sébastien **Combéfis**
Lecteur : Prof. Peter **Van Roy**

Mémoire présenté en vue de l'obtention du grade de
master 120 crédits en sciences informatiques
option
réseau et sécurité
par

Pierre **Nauw**

Louvain-la-Neuve
Année 2013 - 2014

Table des matières

Chapitre 1

Introduction

A notre époque, il existe un grand nombre de systèmes automatisés qui ne cesse de croître. Ce faisant, les gens sont de plus en plus contraints d'utiliser ces systèmes informatiques et d'interagir avec eux. De ce constat est née l'analyse de l'interaction homme machine ou HMI (Human Machine Interaction), qui avec le temps commence à prendre une part importante dans la vérification du bon fonctionnement de ces systèmes automatisés. Celle-ci est d'autant plus importante dans la vérification des systèmes complexes où impliquant la mise en danger d'êtres humains comme les pilotes automatiques d'avions ou les appareils médicaux.

Avec l'évolution de ce type d'analyses, plusieurs approches et divers outils permettant de réaliser de telles analyses ont vu le jour. C'est par exemple le cas de l'outil ADEPT qui permet de modéliser et de simuler l'exécution de divers systèmes automatisés. Il a par exemple été utilisé pour modéliser le pilote automatique d'un avion de type boeing 777. D'un autre coté, les chercheurs du département INGI ont mis au point un outil capable d'analyser un automate représentant un système automatisé et d'en extraire la vision qu'a l'opérateur de ce système. Or, l'analyse HMI repose notamment sur la comparaison entre le système comme il est dans la réalité et la manière dont ce système est perçu par l'opérateur.

Cependant, il n'existe pas encore d'outil capable de faire le lien entre un système virtualisé dans ADEPT et l'outil `jpf-hmi` développé par les chercheurs INGI. Ce document présente une proposition d'implémentation d'un troisième outil, capable de traduire un système virtualisé via ADEPT, en un automate analysable par l'outil `jpf-hmi`. Cette présentation se divise en plusieurs parties. Premièrement, afin de bien établir les bases du nouvel outil, une courte description des outils ADEPT et `jpf-hmi` va être donnée. Ensuite, le fonctionnement de l'outil ainsi que les choix ayant été réalisés quant à son implémentation seront décrits. La partie suivante présentera un système en particulier, le pilote automatique évoqué ci-dessus (**Autopilot**), ainsi que la marche à suivre pour le transformer en automate avec l'outil développé. Par après, un petit manuel d'utilisation de l'outil sera présenté contenant la marche à suivre pour le faire fonctionner correctement. Finalement, l'outil développé sera évalué grâce à divers exemples de transformation de systèmes, en terme d'efficacité ainsi qu'en terme de performances temporelles.

Chapitre 2

Contexte

2.1 ADEPT

ADEPT ou “Automation Design and Evaluation Prototyping Toolset” est un outil développé en Java qui a été conçu par la NASA et qui est utilisé pour développer des prototypes de programmes de dispositifs automatisés afin d'évaluer les interactions homme - machine (HMI). Ce chapitre présente la manière dont fonctionne ADEPT et comment interpréter les fichiers représentant les modèles qu'il génère.

2.1.1 Description de l'outil

ADEPT permet de créer des sortes de modèles automatiques virtuels que l'on appelle des modèles ADEPT. Chacun de ces modèles est composé de deux parties liées entre elles : une interface utilisateur interactive et un ensemble de tables logiques. L'interface utilisateur donne une représentation visuelle en deux dimensions du système modélisé. De plus, cette représentation est interactive dans le sens où certains de ses éléments permettent à l'utilisateur d'interagir avec le système. D'autre part, les tables logiques d'un système modélisé ont pour but de décrire son comportement. En effet, elles sont responsables de la mise à jour des éléments visuels contenus dans l'interface utilisateur, en fonction des actions effectuées par l'utilisateur et des événements qui pourraient survenir dans l'environnement du système. Il est important de noter que le contenu des tables logiques s'apparente à du code Java, ce qui permet à ADEPT d'exécuter et de tester le système modélisé.

Afin d'illustrer le concept de modèle ADEPT, jetons un coup d'oeil au modèle **Light** qui représente une simple lampe pouvant notamment être allumée ou éteinte lorsque l'on appuie sur le bouton “power” et disposant d'une durée de vie limitée. Ce modèle est illustré dans la figure 2.1 le montrant lorsqu'il est ouvert dans ADEPT. Comme on peut le voir dans celle-ci, la partie droite de la fenêtre contient l'interface visuelle permettant d'ajouter des éléments à la représentation 2D du modèle. Cette interface visuelle sert également d'interface interactive avec l'utilisateur lorsque le modèle est exécuté par ADEPT, grâce au bouton “TEST”. La partie centrale de la fenêtre est elle destinée à montrer le contenu de la table logique actuellement sélectionnée et à modifier le contenu de celle-ci. Finalement, la partie gauche contient un browser système, permettant de naviguer d'une table à l'autre ou d'un élément à l'autre du modèle. Il faut savoir que ADEPT permet la création de cinq types d'éléments, de cinq entités différentes. Ces différentes entités sont détaillées dans la section 2.1.2. Une fois le modèle créé et sauvegardé, ADEPT va générer un fichier de type

XML ayant une extension “.sgb”, représentant l’ensemble du modèle et de ses éléments. Ces fichiers font l’objet d’une étude plus poussée dans la section 2.1.4.

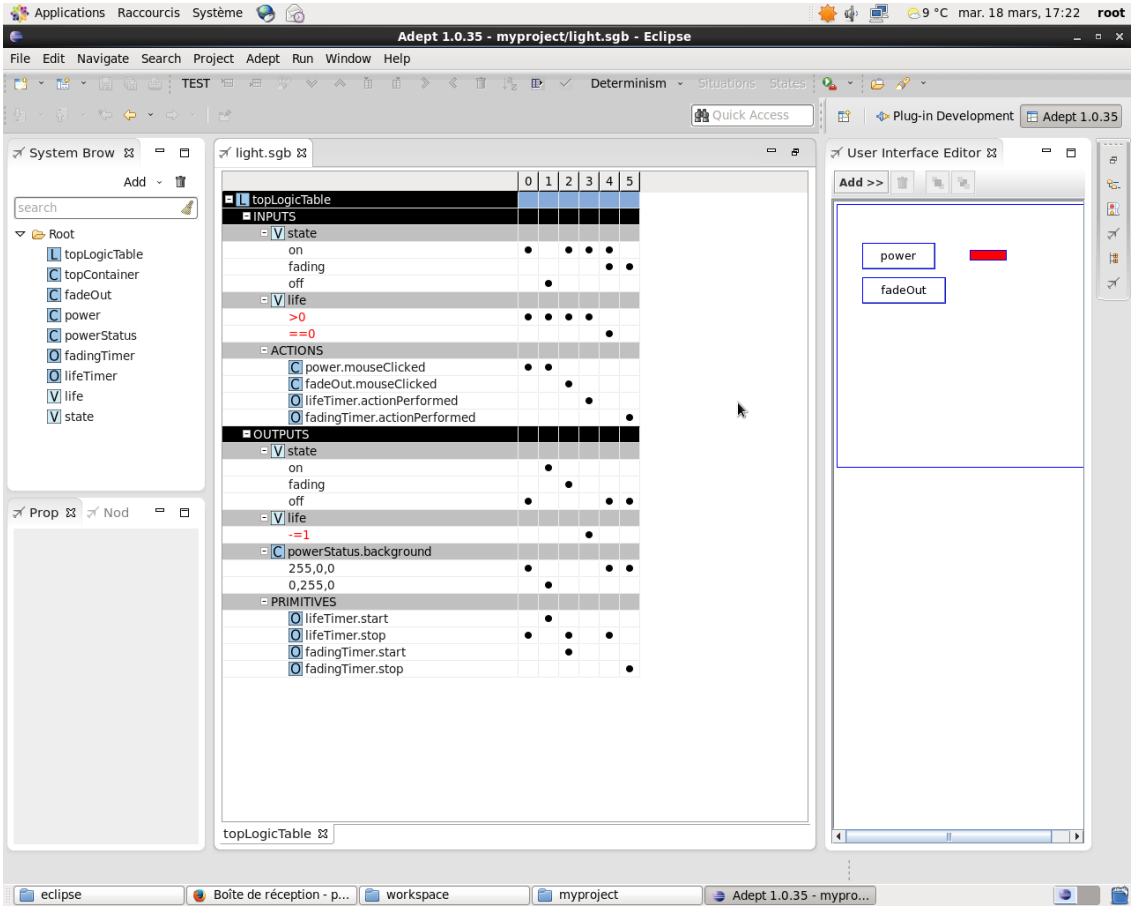


FIGURE 2.1 – Capture d’écran du modèle **Light** une fois ouvert avec ADEPT.

2.1.2 Les entités dans ADEPT

Comme on peut le remarquer dans la figure 2.1, chaque élément du browser système et la plupart de ceux des tables logiques sont précédés par une lettre majuscule. Ceux qui ne le sont pas représentent simplement les valeurs que peut prendre l’élément dans lequel ils se trouvent. Ces lettres représentent l’entité à laquelle appartient l’élément qu’elles précèdent. Comme cela a déjà été mentionné auparavant, ces entités sont au nombre de cinq et les particularités de chacune d’entre elles se trouvent ci-dessous.

C **Composant.** Cette entité représente les éléments visuels de l’interface utilisateur comme par exemple les boutons ou les labels. Ces éléments peuvent être trouvés sous forme d’actions (mouseClicked, mousePressed, ...) dans les lignes d’inputs des tables logiques, plus précisément comme valeur d’un élément spécial appelé ACTIONS. Ces actions appartiennent au premier des deux types d’actions existant dans ADEPT : les commandes. Cela signifie que ce sont des actions sur lesquelles l’opérateur à un impact, que c’est lui qui les déclenche en cliquant par exemple sur l’un des composants visuels. Ces composants peuvent également être trouvés associés à l’un de leurs attributs (background, text, ...) dans les lignes d’outputs des tables logiques. Cela

permet aux tables logiques de modifier les éléments de l'interface visuelle.

V **Variable.** Cette entité représente les variables classiques du système modélisé. Celles-ci sont en fait comme des variables Java et peuvent avoir les mêmes types que ces dernières. On les trouve à la fois en tant qu'élément ou comme valeur d'un élément et ce aussi bien dans les lignes d'inputs que d'outputs des tables logiques.

O **Objet.** Cette entité représente les événements qui peuvent survenir dans environnement du système modélisé. On peut les trouver dans les lignes d'inputs des tables logiques, aux côtés des actions sur les composants, comme valeur de l'élément spécial ACTIONS. Ces actions appartiennent au deuxième type d'actions existant dans ADEPT : les observations. Cela signifie que ce sont des actions sur lesquelles l'opérateur n'a aucune influence, qu'il ne fait que les observer.

F **Fonction.** Cette entité représente les fonctions que l'utilisateur peut ajouter au système modélisé. De la même manière que les variables, ces fonctions sont en fait des méthodes Java et se comportent donc comme telles. Dans une table logique, une fonction apparaît toujours comme valeur d'un élément, celui-ci pouvant tout aussi bien se trouver dans les inputs que dans les outputs. Elles peuvent par exemple être utilisées pour calculer une valeur en fonction de plusieurs variables du système.

L **Logique.** Cette dernière entité représente les éléments logiques, comme l'état de l'output d'une table logique ou alors l'appel à une table logique spécifique. Les appels aux autres tables logiques se trouvent toujours dans les lignes d'outputs des tables logiques et sous forme de valeurs de l'élément spécial LOGIC. Celui-ci est similaire à l'autre élément spécial déjà mentionné, l'élément ACTIONS.

2.1.3 Lecture des tables logiques

Comme dit dans [8], une table logique consiste en une liste d'inputs et d'outputs le long de l'axe Y, et en une série de colonnes de paires situation - comportement de l'automate le long de l'axe X. Si une table logique devait être traduite en langage parlé, la ligne INPUTS serait lue comme un "if" et la ligne OUTPUTS comme un "then". En dessous des ces deux lignes, les lignes grises pourraient être lues comme des "and" et les lignes blanches comme des "or". Il est important de noter que les outputs ne contiennent que des "and", ce qui signifie que pour un input bien défini, l'output de la table sera toujours le même et donc que le modèle est déterministe. Voici par exemple comment la colonne 0 de la figure 2.2 devrait être lue et comprise.

IF
state est *on*
AND
life est *strictly bigger than 0*
AND
action produite est *user click on the power button*
THEN
state devient *off*
AND
powersStatus background devient *255,0,0*

De plus, il faut savoir que les colonnes sont lues de gauche à droite, ce qui veut dire que l'on teste les inputs d'une colonne, uniquement si les précédentes ne conviennent pas. Vous remarquerez sans doute qu'il manque l'élément spécial PRIMITIVES dans l'illustration de la *topLogicTable* du modèle **Light**, ce qui est normal car l'élément PRIMITIVES gère le démarrage des timers. Or, la construction de l'automate lié au modèle ne nécessite pas de savoir quand les timers démarrent, seul le moment où ils arrivent à "zéro" et ils se déclenchent importe et est traduit en un événement pouvant survenir dans l'environnement du système. Ces événements sont les mêmes que ceux qui ont déjà été évoqués dans la section 2.1.2.

	0	1	2	3	4	5
L topLogicTable						
INPUTS						
V state						
on	•		•	•	•	
fading					•	•
off		•				
V life						
> 0	•	•	•	•		
== 0					•	
ACTIONS						
power.mouseclicked	•	•				
fadeOut.mouseclicked			•			
lifeTimer.actionPerformed				•		
fadingTimer.actionPerformed						•
OUTPUTS						
V state						
on		•				
fading			•			
off	•				•	•
V life						
-= 1				•		
C powerStatus.background						
255,0,0	•				•	•
0,255,0		•				

FIGURE 2.2 – La table *topLogicTable* issue du modèle **Light**.

2.1.4 Les fichiers SGBs

Un fichier sgb est un fichier XML contenant toutes les données relatives à la description d'un modèle ADEPT. Tout comme les fichiers XMLs, un fichier sgb contient une sorte d'arbre, composé de noeuds et de sous noeuds représentés entre "<" et ">". Chacun des fichiers de ce type possède une structure commune, comme celle présentée ci-dessous.

```
<AdeptProject>
       <DstPanelElement ... />
       <JarFileURLKey>...</JarFileURLKey>
       <MethodsKey>...</MethodsKey>
       <ComponentsKey>...</ComponentsKey>
       <ObjectsKey>...</ObjectsKey>
       <PropertiesKey>...</PropertiesKey>
       <IndexedPropertiesKey>...</IndexedPropertiesKey>
```

```
        <LogicsKey>...</LogicsKey>
        <TopLogicTable ... />
        <Node ... >...</Node>
        <SelectedLogicNodes>...</SelectedLogicNodes>
        <IndividualLogicNodes>...</IndividualLogicNodes>
        <Macro name="null"/>
</AdeptProject>
```

AdeptProject Ce noeud représente la racine du fichier sgb. Ils contient tous les autres noeuds, qui eux décriront le modèle ADEPT.

DstPanelElement Ce noeud-ci représente la description du conteneur principal de l'interface utilisateur, c'est à dire le panel de fond de cette interface. Il contient diverses informations relatives à ce panel, comme sa hauteur, sa largeur, sa couleur de fond, etc...

```
<DstPanelElement background="255,255,255" name="topContainer" width="300" font="Lucida Grande-PLAIN-13" visible="True" toolTipText="null" class="gov.nasa.arc.sgbe.remote.nodes.BeanNode" enabled="True" foreground="0,0,0" y="5" x="5" componentPostion="-1" beanClass="javax.swing.JLayeredPane" location="5, 5" height="300" size="300, 300"/>
```

JarFileURLKey Ce noeud contient d'autres noeuds servant à indiquer où trouver les librairies importées et utilisées par le modèle ADEPT.

```
<URL url="file:/Users/combefis/Documents/ADEPTworkspace/.metadata/.plugins/gov.arc.nasa.adept/SGBE.jar"/>
```

MethodsKey Ce noeud-ci est le noeud racine de tous les noeuds décrivant les fonctions définies dans le modèle. Ces noeuds font référence à des éléments appartenant à l'entité fonction F présentée dans la section 2.1.1. Chacun de ceux-ci contient le nom de la fonction qu'il représente, le type de la valeur qu'elle renvoie, le code java du corps de celle-ci et parfois une petite description de ce qu'elle fait. Voici par exemple le noeud décrivant la fonction *convertAirspeedTargetToMach* appartenant au modèle **Autopilot**. Le rôle de cette fonction est de transformer une valeur exprimée en m/s par une valeur en mach.

```
<Node returnType="double" name="convertAirspeedTargetToMach" description="null" code="return (airspeedTarget/(speedOfSound-((aircraftAltitude/1000)*2.1975)));" class="gov.nasa.arc.sgbe.remote.nodes.PrimitiveNode"/>
```

ComponentsKey Ce noeud contient tous les noeuds décrivant les éléments graphiques de l'interface visuelle du modèle. Ils appartiennent à l'entité composant C. Ces noeuds contiennent toutes les informations relatives au composant comme son nom, son positionnement, son type, sa police d'écriture, etc... Voici par exemple le noeud décrivant le bouton *power* du modèle **Light**.

```
<Node location="29, 44" beanClass="javax.swing.JButton" foreground="0,0,0" hideActionText="False" rolloverEnabled="False" visible="True" verticalAlignment="CENTER" verticalTextPosition="CENTER" opaque="False" enabled="True" y="44" x="29" borderPainted="True" text="power" class="gov.nasa.arc.sgbe.remote.nodes.BeanNode" font="Lucida Grande-PLAIN-13" size="82, 29" height="29" selected="False" width="82" horizontalAlignment="CENTER" toolTipText="null" horizontalTextPosition="TRAILING" background="238,238,238" componentPostion="0" name="power"/>
```

ObjectsKey Dans ce noeud-ci se trouve les noeuds décrivant les divers objets créés dans le modèle comme les timers. Ceux-ci appartiennent à l'entité objet [O]. Voici le noeud décrivant l'objet *lifeTimer* du modèle **Light**.

```
<Node name="lifeTimer" jarEntry="gov/nasa/arc/beans/TimerBean.class" class="gov.nasa.arc.sgbe.
    remote.nodes.BeanNode" delay="1000" repeats="True" initialDelay="1000" coalesce="True"
    beanClass="gov.nasa.arc.beans.TimerBean" jarURL="/Users/combefis/Documents/ADEPTworkspace
    /.metadata/.plugins/gov.arc.nasa.adept/SGBE.jar"/>
```

PropertiesKey Ce noeud est la racine de tous les noeuds décrivant les variables "classiques" du modèle. Tous ceux-ci appartiennent à l'entité variable [V] et contiennentt le nom de la variable qu'il représente, ainsi que son type et sa valeur initiale. Voici par exemple le noeud représentant la variable *state* du modèle **Light**.

```
<Node name="state" propertyType="String" value="off" class="gov.nasa.arc.sgbe.remote.nodes.
    PropertyNode"/>
```

LogicsKey Ce noeud contient tous les noeuds décrivant le comportement du modèle, c'est-à-dire le contenu des tables logiques. Toutes les tables sont décrites par un noeud servant d'entête et un noeud racine contenant tous les noeuds décrivant les différents éléments de la table. Voici par exemple l'entête et la racine de la table *topLogicTable* tirée du modèle **Light**.

```
<Table_Header Table_Name="topLogicTable"/>
<Node outputState="null" name="topLogicTable" class="gov.nasa.arc.sgbe.remote.sgb.nodes.
    LogicRootNode">
```

Ensuite, dans le noeud racine, on retrouve des noeuds spéciaux servant à distinguer les inputs, les outputs, les goals ou la catégorie de comportement. Les parties relatives aux goals et aux catégories de comportements ne seront pas étudiées car celles-ci, à la manière de l'élément spécial PRIMITIVES, n'ont aucune influence sur l'automate produit par un modèle ADEPT. Voici donc les noeuds racines des inputs et des outputs.

```
<Node name="INPUTS" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode">...</Node>
<Node name="OUTPUTS" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode">...</Node>
```

Le noeud racine d'une table contient également divers noeuds décrivant les différents comportements du modèle. Chacun de ces noeuds représente en fait une situation distincte, autrement dit une colonne de la table logique. Le premier noeud du fichier sgb représente la colonne 0, le deuxième la 1 et ainsi de suite. Ces noeuds contiennent un nom, une petite description du comportement et une spécification décrivant le contenu de la colonne qu'ils représentent. Ce que ces spécifications représentent concrètement fera l'objet d'une étude plus poussée dans la section 3.1.1. Voici les noeuds des colonnes 0 et 1 de la table *topLogicTable* du modèle **Light**.

```
<Node name="situation" specification="[[[1 0 0][1 0][1 0 0 0]][[0 0 1][0][1 0][0 1 0 0]]]"
    behavior="behavior" class="gov.nasa.arc.sgbe.remote.sgb.nodes.Situation"/>
<Node name="situation" specification="[[[0 0 1][1 0][1 0 0 0]][[1 0 0][0][0 1][1 0 0 0]]]"
    behavior="behavior" class="gov.nasa.arc.sgbe.remote.sgb.nodes.Situation"/>
```

Après cela, si l'on observe ce que contient le noeud racine des inputs, on peut trouver les noeuds relatifs aux paramètres de la table, aux différents inputs. Tous ces noeuds ont un nom et ont une référence vers un autre noeud déclaré auparavant dans le fichier. Si le paramètre est une variable classique, il fera référence au noeud représentant cette même variables dans les PropertiesKey, si le paramètre est l'output d'une table, alors il fera référence au noeud racine de la table associée, et enfin, si le paramètre est un composant, la référence se fera vers un noeud déclaré dans la partie ComponentsKey. Chacun de ces noeuds contient également des sous noeuds décrivant les différentes valeurs que peut prendre l'élément. Attention que ces sous noeuds peuvent également faire référence à d'autres éléments du modèle. Notez que tout ce qui vient d'être évoqué par rapport au noeud racine des inputs est transposable au noeud racine des outputs. Voici par exemple les noeuds représentants l'élément *state* dans les inputs de la table *topLogicTable* du modèle **Light**.

```
<Node name="state" referenceNodeName="state" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode
    ">
      <Node propertyEditor="sun.beans.editors.StringEditor" name="on" isValueNode="true"
          class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"/>
      <Node propertyEditor="sun.beans.editors.StringEditor" name="fading" isValueNode="true"
           class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"/>
      <Node propertyEditor="sun.beans.editors.StringEditor" name="off" isValueNode="true"
          class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"/>
</Node>
```

Finalement, parmi les noeuds représentant les inputs et les outputs d'une table, on peut également trouver des paramètres spéciaux. Nous les avons déjà évoqués plus haut, ce sont les paramètres spéciaux ACTIONS et LOGIC. Le noeud représentant le paramètre ACTIONS contient tous les noeuds décrivant les différentes actions possibles de la table. Ces noeuds contiennent le nom de l'action ainsi qu'une référence vers le noeud représentant le composant auquel l'action s'applique. Le noeud LOGIC contient lui des noeuds faisant référence à d'autres tables du modèle. Ces deux paramètres ont la même structure, comme celle du noeud ACTIONS de la table *topLogicTable* du modèle **Light** ci-dessous.

```
<Node name="ACTIONS" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode">
      <Node featureName="power.mouseClicked" name="power.mouseClicked" referenceNodeName="
          power" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"/>
      <Node featureName="fadeOut.mouseClicked" name="fadeOut.mouseClicked" referenceNodeName
          ="fadeOut" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"/>
      ...
</Node>
```

TopLogicTable Ce noeud a pour unique fonction d'indiquer que le prochain noeud sera le noeud décrit ci après, celui-ci portant le nom par défaut de "Node".

Node Ce noeud contient tous les noeuds décrivant le browser système qui a été évoqué dans la section 2.1.1. Ces noeuds peuvent soit représenter un dossier contenant divers sous noeuds, soit directement un élément du modèle comme un composant, une variable, une table, etc... Ces noeuds contiennent, un nom, une référence vers le noeud représentant l'élément, ainsi qu'un flag bbKey indiquant l'entité à laquelle l'élément appartient (ComponentsKey, PropertiesKey, etc). Voici par exemple un morceau du noeud Node du modèle **Light**.

```
<Node name="Root" referenceNodeName="Root" class="gov.nasa.arc.sgbe.ui.system.
    SystemReferenceNode" bbKey="SubSystemKey">
      <Node name="power" referenceNodeName="power" class="gov.nasa.arc.sgbe.ui.system.
          SystemReferenceNode" bbKey="ComponentsKey"/>
</Node>
```

Le code complet du fichier sgb représentant le modèle **Light** se trouve dans l'annexe A.

2.2 JPF-HMI

L'outil `jpf-hmi` a été développé par Sébastien Combéfis dans le cadre de sa thèse de doctorat. Comme il le dit lui-même dans [2], jpf-hmi est en quelque sorte une extension du Java PathFinder (JPF) qui analyse, transforme et vérifie les propriétés de divers automates. Cette section présente en quelques lignes cet outil, ses caractéristiques et les différentes opérations qu'il peut effectuer sur un automate donné.

2.2.1 Description de l'outil

Le rôle de l'outil `jpf-hmi` est de transformer des modèles systèmes en modèles mentaux ou d'effectuer quelques analyses sur ceux-ci. Comme décrit dans [4], un modèle système permet de décrire le comportement d'un système, tandis qu'un modèle mental décrit la connaissance d'un utilisateur par rapport à ce système, la manière dont l'utilisateur le perçoit. Ces modèles sont représentés par ce que l'on appelle des HMI-LTSs, qui sont en fait une sorte de systèmes de transitions à états (LTS) ou plus simplement des automates. A l'intérieur de l'outil, un HMI-LTS est en fait implémenté comme une liste de paires état, transition. De plus, un modèle peut être enrichi ou non. La différence entre un modèle système enrichi ou non est que le modèle enrichi possède une observation sur ses états, représentant l'observation qu'un opérateur peut faire lorsque le système est dans cet état.

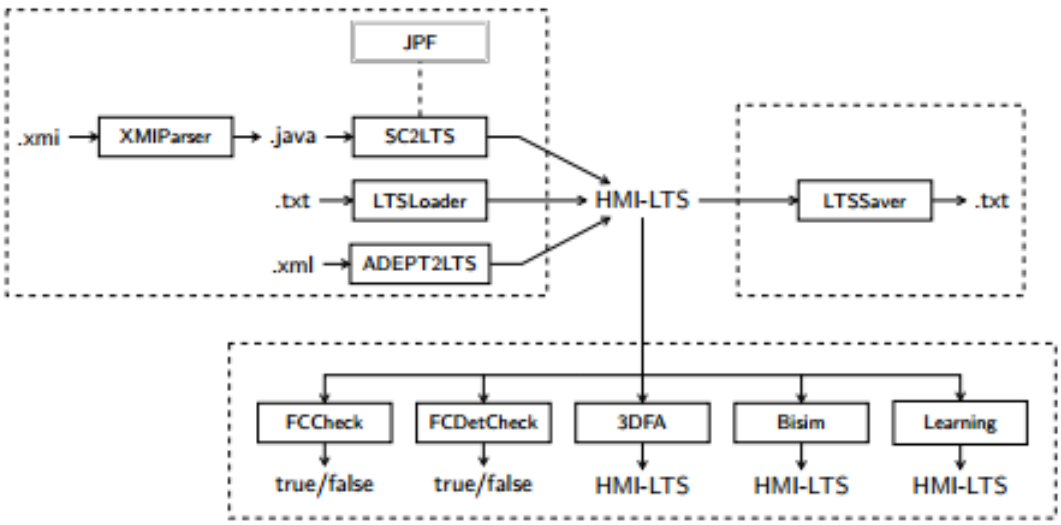


FIGURE 2.3 – Vue d'ensemble de l'outil `jpf-hmi`, tirée de [2].

Comme on peut le voir dans la figure 2.3 qui représente la structure de l'outil `jpf-hmi`, on peut se rendre compte que ce dernier permet de réaliser cinq opérations différentes sur un HMI-LTS. Les deux premières sont en fait des analyses, des tests sur certaines propriétés d'un modèle :

- FCCheck : Cette analyse permet de vérifier si un modèle et donc le HMI-LTS le représentant respecte une propriété appelée "full-control". Cette propriété met en évidence le fait que pour qu'une interaction homme-machine soit convenable, c'est à dire que l'homme ne soit pas surpris par le comportement de la machine, l'homme doit connaître toutes les opérations qu'il peut effectuer avec le système et doit pouvoir s'attendre à tout ce qui peut arriver.

- FCDetCheck : Cette opération vérifie la propriété de "Full-control determinism" d'un HMI-LTS.

Les trois opérations suivantes sont en fait les trois différentes manières que `jpf-hmi` met a disposition pour transformer un modèle système en un modèle mental minimal respectant la propriété de "full-control" :

- 3DFA : le modèle mental est construit selon une une approche basée sur un "Three-Valued Deterministic Finite Automaton". Ce processus se déroule en trois étapes. La première est ce que l'on appelle une 3NFA-complétion. La seconde étape est une "déterminisation" de l'automate, tandis que la dernière étape est elle une minimisation de ce dernier.

- Bisimulation : le modèle mental est construit par un mécanisme de réduction du modèle système. Ce processus se déroule en deux étapes. La première est une tau-complétion du modèle, c'est à dire que les transitions tau sont remplacées. La deuxième étape est la réduction proprement dite.

- Learning : dans cette dernière approche, le modèle mental est construit de manière incrémentale, état par état.

Finalement, la structure de `jpf-hmi` nous permet également de voir que cet outil fournit trois mécanismes différents pour importer les HMI-LTSs, ainsi qu'un autre pour les exporter :

- SC2LTS : Premièrement, les modèles peuvent être importés depuis un statechart, défini dans son format standard XMI.

- LTSLoader : `jpf-hmi` contient également une classe *LTSLoader* dont le rôle est d'importer un HMI-LTS depuis un fichier texte (.lts) contenant la liste des états et des transitions du modèle. D'autre part, le LTSLoader permet d'exporter les HMI-LTS sous le même format de fichier grâce à sa méthode *saveLTS()* (LTSSaver).

- ADEPT2LTS : Le dernier mécanisme permet d'importer un HMI-LTS à partir d'un modèle ADEPT et donc du fichier XML (.sgb) qui le représente.

Notez qu'une description plus complète du fonctionnement de l'outil `jpf-hmi` et des opérations qu'il permet de réaliser est donnée dans [2]. En réalité, la fonctionnalité permettant d'importer un HMI-LTS depuis un modèle ADEPT n'est pas implémentée. Le créateur de `jpf-hmi` utilisait bien des modèles ADEPT mais il les transformait manuellement. En effet, il ajoutait du code Java permettant de générer le modèle directement dans le code de `jpf-hmi`. Ce travail présente donc une proposition d'implémentation de cette fonctionnalité, qui permet d'importer les HMI-LTS depuis un modèle ADEPT de manière automatique. De plus, la nouvelle version de `jpf-hmi` qui va être présentée propose une petite interface permettant à l'utilisateur de facilement effectuer des opérations avec l'outil. Cette interface est détaillé dans la section 5 et permet de faire le lien entre les HMI-LTSs générés par ADEPT2LTS et les algorithmes de transformation d'un modèle système en un modèle mental déjà fournis par `jpf-hmi`.

Chapitre 3

ADEPT2LTS

L'outil ADEPT2LTS est une solution au problème qui est de transformer un fichier sgb décrivant un modèle ADEPT en un HMI-LTS équivalent utilisable par les algorithmes de l'outil `jpf-hmi`. Comme déjà dit précédemment, cet outil possède déjà un dispositif permettant l'extraction d'un HMI-LTS à partir d'un fichier texte (.lts) contenant les états et les transitions du modèle représenté. Pour ce faire, `jpf-hmi` utilise une classe appelée *LTSLoader* contenue dans la package *gov.nasa.jpf.hmi.models.util* qui contient une méthode pour importer un HMI-LTS et une autre pour en exporter un. Dés lors, pour conserver une certaine cohérence avec ce qui avait déjà été créé, une classe *ADEPT2LTS* a été ajoutée dans le même package. De plus, cette classe a été structurée de manière semblable à la classe *LTSLoader* dans le sens où elle contient elle aussi une méthode *loadLTS()* qui renvoie un HMI-LTS. Par contre, la comparaison s'arrête là car ajouter la possibilité de retraduire un HMI-LTS en un fichier sgb, de manière analogue à la méthode saveLTS, n'aurait aucun sens et semble fort complexe. La transformation d'un fichier sgb en un HMI-LTS exploitable par l'outil `jpf-hmi` est représentée à la figure 3.1 et se déroule en trois grandes étapes :

- Traduction du fihcier sgb en un fichier Java simulant l'automate
- Compilation de ce fichier Java.
- Appel à la méthode *simulate()* du fichier Java permettant la simulation du modèle.

La première étape est gérée par un outil appelé le Translator, dont le fonctionnement est décrit dant la section 3.1. Le rôle de cet outil est de générer un fichier Java qui va représenter le modèle traduit et permettre sa simulation. En effet, comme déjà mentionné auparavant, ADEPT se base sur des objets du langage Java ce qui en faisait le langage le plus approprié pour permettre la simulation de l'exécution d'un modèle. De plus, le seul moyen de récupérer l'automate représentant un modèle, c'est d'exécuter ce modèle et d'observer les états qu'il atteint et les transitions qu'il effectue. Le contenu de ce fichier Java généré fait l'objet d'une étude plus poussée et est décrit à la section 3.2. La deuxième étape est décrite dans la section 3.3 et consiste en la compilation du fichier Java. Cette compilation va générer une nouvelle classe dont le contenu pourra être utilisé par ADEPT2LTS grâce au caractère réflexif du langage Java. L'appel au contenu de cette nouvelle classe est la troisième étape du processus qui est décrite dans la section 3.4. ADEPT2LTS va en fait faire appel à la méthode *simulate()* de la classe générée car c'est cette méthode qui simule le modèle et retourne les états et transitions atteints par le système lors de cette simulation.

Effectuer la traduction de la sorte permet de séparer un processus complexe en plusieurs processus plus simples et plus faciles à comprendre. De plus, cela rend la détection des erreurs plus facile lorsqu'elles arrivent car chaque étape peut être testée séparément. L'utilisation d'un fichier Java annexe lors de la traduction était nécessaire pour permette de virtualiser le modèle et de l'exécuter afin de connaître l'automate découlant de celui-ci. Dès lors, le seul moyen d'accéder au contenu de ce fichier était la réflexivité. Une autre solution aurait pu être d'incorporer directement les algorithmes d'analyses fournis par `jpf-hmi` à l'intérieur du fichier Java généré de sorte que les analyses voulues s'effectuent pendant l'exécution du fichier Java. Cependant, cette solution ne permettait pas de conserver un certain parallélisme avec ce qui avait déjà été fait, qui consistait notamment à distinguer clairement une phase d'importation d'un modèle et ensuite seulement une phase d'analyse, comme on peut le voir clairement dans la figure 2.3.

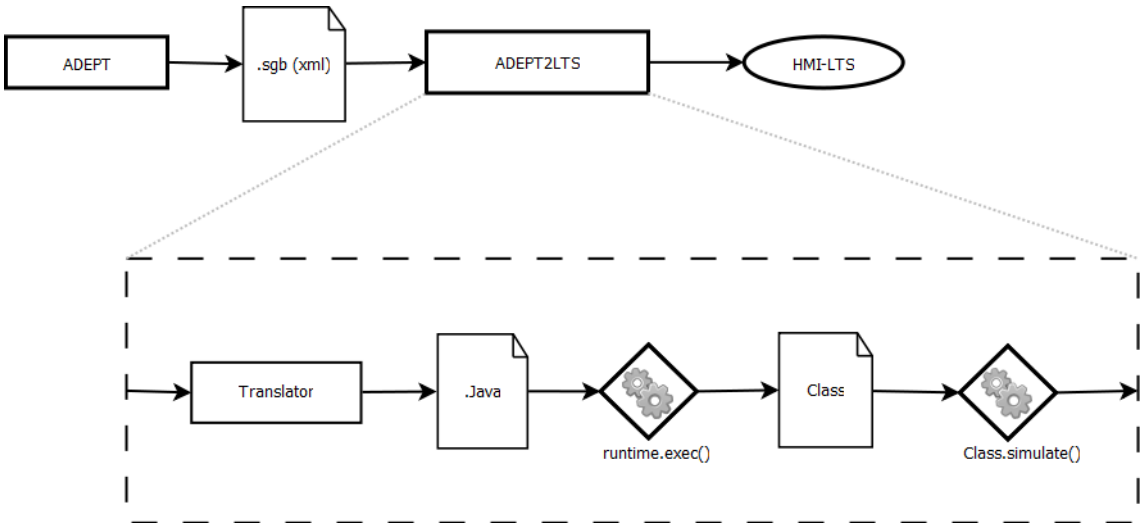


FIGURE 3.1 – Vue d'ensemble de l'outil ADEPT2LTS.

3.1 Translator

L'outil Translator est une classe Java qui prend en entrée un fichier sgb décrivant un modèle ADEPT et qui génère un fichier Java dont le but est de le simuler. En effet, pour connaître les états et transitions franchies par un modèle ADEPT, il est nécessaire de simuler une exécution de celui-ci et d'observer son comportement. Comme cela est visible dans la vue d'ensemble du fonctionnement de l'outil donné à la figure 3.2, l'opération de traduction se déroule en trois étapes distinctes. Une fois de plus, le travail de l'outil a été divisé et confié à d'autres sous-outils de manière à simplifier le processus. De nouveau, cette division permet, lorsqu'une erreur se produit, de distinguer plus aisément l'origine de cette erreur, chaque étape étant testable séparément. La première étape consiste à lire le fichier sgb et à extraire les informations utiles relatives au modèle qu'il représente. L'outil responsable de cette étape est appelé le SGBParser et son fonctionnement est détaillé dans la section 3.1.1. Deuxièmement, l'outil Translator va trier les informations extraites grâce au SGBParser de sorte à pourvoir donner au JavaWriter ce dont il a besoin et sous la forme la plus efficace possible. Ce JavaWriter est l'outil réalisant la troisième étape de la traduction et est utilisé par la méthode *write()* de la classe *Translator*. Il a donc pour but d'écrire les informations qu'on lui donne dans un fichier Java. Cette troisième étape est

détaillée dans la section 3.1.3.

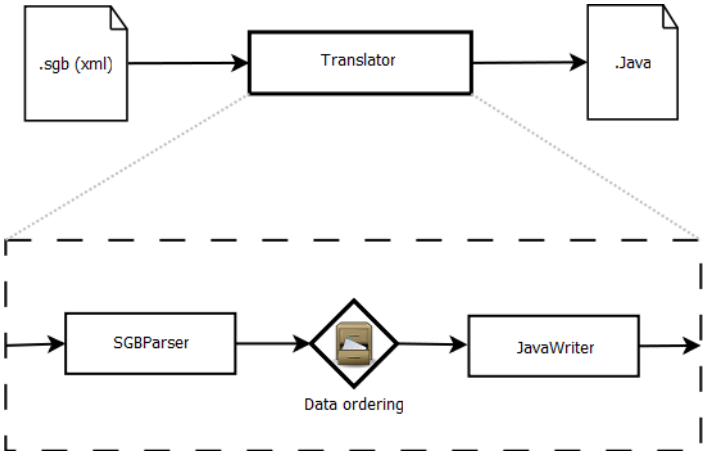


FIGURE 3.2 – Vue d'ensemble de l'outil Translator.

3.1.1 Première étape : le SGBParser

Le SGBParser est un outil qui, comme son nom l'indique, aura pour objectif de "parser" un fichier sgb donné. Étant donné qu'un fichier sgb est en réalité un fichier au format XML, le SGBParser va se baser sur un parseur traditionnel et existant de fichiers XMLs. Dans ce cas ci, le parseur Java utilisé est un parser de type DOM (Document Object Model), dont la spécificité est de renvoyer toutes les informations contenues dans le fichier XML sous la forme d'un arbre. Cependant, le format de cet arbre ainsi que la manière d'accéder aux informations qu'il contient ne sont pas très pratiques dans le cas ou les données contenues dans le fichier sont structurées comme dans un fichier sgb. C'est de ce constat que découle le rôle précis du SGBParser : lire les données contenues dans le fichier sgb et les mettre sous une forme plus adaptée. Concrètement, la classe *SGBParser* récupère une structure de données sous forme d'arbre via le parseur DOM, lit les informations qui y sont contenues et les place finalement dans une structure de données plus adaptée à la récupération des informations importantes contenues dans un fichier sgb. Cette structure de données est représentée dans la figure 3.3 et est détaillée ci dessous.

La nouvelle structure de données contenue dans le SGBParser est une fait une série de quatre listes. Initialement, le SGBParser avait été conçu de manière à contenir une liste pour chaque élément de l'architecture d'un fichier sgb, comme présenté dans la partie 2.1.4, mais seuls quatre de ces éléments sont utiles dans le processus de traduction. Comme cela a déjà été dit, le noeud <LogicsKey> contient tout ce qui est relatif au comportement d'un modèle, or c'est exactement ce comportement que l'on doit étudier pour obtenir l'automate représentant un modèle. Par contre, les noeuds représentant les différents inputs et outputs des tables ne sont pas différents en fonction qu'ils représentent une variable classique ou un composant visuel. C'est pour cela que Le SGBParser va également récupérer le contenu des noeuds <ComponentsKey> et <PropertiesKey>, qui de plus contiennent les valeurs initiales des variables qu'ils représentent. Finalement, le noeud <MethodsKey> sera également nécessaire afin de connaître les fonctions du modèles.

Le SGBParser va donc transformer et stocker chacun des noeuds utiles en leur équivalent sous forme d'une instance d'une classe Java. Par exemple, le noeud <ComponentsKey> est

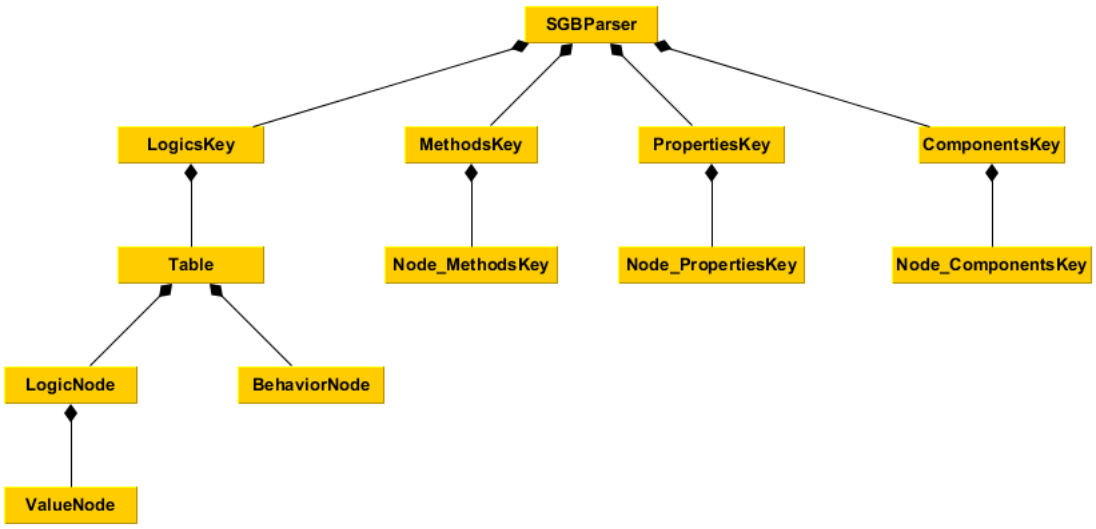


FIGURE 3.3 – Diagramme de classe UML simple représentant la structure de données.

transformé en une instance de la classe *ComponentsKey*, qui elle même contient une liste d'instances de la classe *Node_ComponentsKey* qui sont obtenues par transformation des noeuds fils du noeud <ComponentsKey>. De manière analogue, l'outil SGBParser génère des instances des classes *PropertiesKey* et *MethodsKey*. Au niveau du noeud <LogicsKey>, c'est un peu plus compliqué et ce à cause des différents types de sous-noeuds qu'il contient. Comme cela a déjà été dit auparavant, ce noeud contient d'abord une liste de sous-noeuds représentant les tables logiques du modèle. Chacun de ces noeuds sera transformé en une instance de la classe *Table*. Ensuite, chaque instance de cette classe *Table* contient quatre listes d'instances de la classe *LogicNode* (behaviorCategory, outputs, goals, inputs) dont le but est de séparer les éléments d'une table en fonction de leur rôle dans celle-ci. Cette classe *LogicNode* représente en fait un élément de la table logique, que ce soit une variable **V**, un composant **C**, un output d'une table **L** ou même un des éléments spéciaux comme ACTIONS et LOGIC. Il existe également une cinquième liste composée d'instances de la classe *BehaviorNode* qui représentent les noeuds décrivant le contenu des colonnes de la table logique, des différentes situations possibles.

Cependant, cette première traduction n'est pas suffisante car lors de cette étape, les informations ne sont pas triées ni modifiées mais simplement recopiées sous une autre forme et nécessitent donc d'être triées avant d'être passées à l'outil JavaWriter. Cependant, cette forme est plus compréhensible, indique clairement où se trouve telle ou telle information et rend finalement les informations plus facilement accessibles. Ci-dessous se trouve une illustration de la simplification et de la clarification des moyens d'accès aux informations contenues dans un fichier sgb. Cet exemple met en place une comparaison entre les lignes de codes utilisées pour récupérer la valeur initiale de la variable "state" issue du modèle **Light**.

```
1 // Parseur DOM traditionnel
2 ((Element) Document.getDocumentElement().getChildNodes().item(5).getChildNodes.item(0)).getAttribute
      (''value'');
3 // SGBParser
4 SGBParser.getPropertiesKey.get(0).getValue();
```

3.1.2 Deuxième étape : Le tri

Cette seconde étape aura pour but de trier les informations récupérées via l'outil SGB-Parser, de recoller les morceaux en quelque sorte. Par exemple, afin de virtualiser les tables logiques de la meilleure manière qu'il soit pour les préparer à l'écriture dans le fichier Java, il va falloir récupérer des informations placées à divers endroits dans la structure de données générée par la classe *SGBParser*. Pour stocker toutes les informations dont l'outil JavaWriter aura besoin pour générer le fichier Java, la classe *Translator* va utiliser toutes une série de listes et de classes représentant certains éléments. En fait, Translator va utiliser exactement quatre classes annexes :

- LogicTable − Cette classe a pour but de virtualiser une table logique. Celle-ci est présentée un peu plus en détails ci-après.
- LogicVariable − Cette classe représente une variable logique du modèle V, dans une table. Celle-ci possède quatre attributs : le nom de la variable, son type, sa valeur initiale, ainsi qu'une liste des opérations qui lui sont applicables dans la table.
- LogicElement − Cette classe représente simplement une ligne de la première colonne d'un table logique, celle contenant les éléments et les valeurs, la colonne -1 en quelque sorte.
- Function − Cette troisième classe représente simplement une fonction d'un modèle. Chacune d'elle a trois attributs : le nom de la méthode, le type de la valeur renvoyée et le contenu du corps de celle-ci.

LogicTable

Cette classe est celle qui virtualise une table logique et a donc un rôle central. Chaque instance de la classe *LogicTable* est définie par son nom et son état d'output initial. Ces tables possèdent deux vecteurs Java composés d'instances de la classe *LogicElement*, l'un pour les inputs, l'autre pour les outputs. Cette classe représente simplement une valeur d'une table, associée à un parent qui est en fait l'élément auquel elle se rapporte. De plus chaque table contient deux listes de vecteurs de type boolean, l'une pour les inputs, l'autre pour les outputs. Ces vecteurs d'objets boolean représentent en fait les colonnes de la table dont l'emplacement des bullets est défini par les boolean ayant "true" comme valeur. Cette manière de faire a permis de représenter aisément et efficacement une table logique et ses colonnes au travers des différents vecteurs utilisés. En effet le résultat obtenu est simplement une sorte de tableau dont les colonnes sont des vecteurs.

La manière dont sont construits les vecteurs représentant les positions des bullets dans les différentes colonnes des tables logiques est intéressante. En effet, comme déjà évoqué dans la section 2.1.4, les colonnes dans les fichiers sgb sont représentées comme une succession de "0" et de "1", séparés par des "[" et des "]" permettant d'indiquer à quelles lignes de la table les numéros font référence. Il existe trois niveaux de crochets, chacun servant à différencier les parties d'une colonne. Le premier niveau de crochets sert simplement à délimiter la colonne entière : `"[colonne]"`. Ensuite, le deuxième niveau est utilisé pour séparer les inputs des outputs, mais aussi dans certains cas les goals ou les behaviorCategory : `"[[behaviorCategory][goals][inputs][outputs]]"`. La possible présence de goals ou autres pose problème lorsqu'il faut retrouver l'emplacement des inputs et outputs dans la représentation. La solution à ce problème était simplement d'ajouter deux variables aux tables issues du SGBParser servant à indiquer le positionnement

des inputs et des outputs dans les colonnes. Finalement, le troisième niveau de crochets permet de séparer les différents éléments présents comme inputs ou outputs dans la colonne : `"[[[in1][in2][in3]][[out1][out2]]]"`. Dès lors, pour récupérer le contenu des colonnes, il suffit simplement de traverser le string la représentant, caractère par caractère en utilisant un compteur sur les crochets afin de déterminer à quel niveau on se trouve et s'il on est dans les inputs ou outputs. Cette opération est effectué par la méthode *behaviorsCreation()* dont le code source est donné à l'annexe F.

3.1.3 Trosième étape : le JavaWriter

L'outil JavaWriter est donc l'entité chargée de générer un nouveau fichier Java portant le même nom que le modèle en traduction et d'y écrire. Le rôle du fichier Java généré est de virtualiser un modèle de sorte à pouvoir le simuler, afin de pouvoir observer les différents états et transitions qu'il traverse et donc de générer un automate représentant son comportement. Une fois que l'on appelle la classe *JavaWriter*, il faut lui fournir le nom du modèle ainsi que son emplacement afin que l'outil crée automatiquement le fichier Java. Cependant, le fichier est toujours vide et il faudra le remplir. Pour ce faire, l'outil JavaWriter met à disposition diverses méthodes permettant d'écrire dans ce fichier prédéfini. Chacune de ces méthodes permet la traduction d'une partie bien précise du modèle en son équivalent en code Java. La classe *JavaWriter* possède par exemple une méthode *writeActions()* servant à déclarer dans le fichier Java toutes les actions pouvant avoir lieu lors de l'exécution du modèle, une méthode *writeFunctions()* permettant la traduction des fonctions F, ou encore *writeVariableDeclarations()* qui permet de déclarer les variables classiques du modèle V dans le fichier Java. Une description complète du fichier Java généré, ainsi que de certaines règles de traduction utilisées se trouvent dans la section 3.2.

3.2 Fichier Java

Ce fichier Java est donc le résultat de l'appel à la méthode *write()* de la classe *Translator*. Cette méthode va utiliser les informations triées au préalable lors de la phase de tri et les passer à l'outil JavaWriter afin de générer le fameux fichier Java dont la structure est détaillée à la section 3.2.2. La méthode *write()* consiste en une succession d'appels aux différentes méthodes de la classe *JavaWriter* telles que *writeActions()*, *writeVariableDeclarations()*, etc. Chacune des ces méthodes est en fait responsable de la génération d'un élément de la structure du fichier Java. Une de ces méthodes, la méthode *writeSimulate()*, a notamment pour but de générer la méthode *simulate()* décrite dans la section 3.2.1. Une autre méthode importante, la méthode *writeTable()*, est responsable de la traduction des tables logiques, et donc du comportement du modèle, en code Java. Cette opération est a la fois la tâche centrale et la plus complexe de la traduction. C'est pourquoi elle a fait l'objet de l'établissement de plusieurs règles de traduction décrites dans la section 3.2.3.

3.2.1 Méthode simulate()

La méthode *simulate()* d'un fichier Java virtualisant un modèle ADEPT est donc la méthode qui est appelée afin générer le HMI-LTS correspondant. Son but est donc bien de virtualiser le modèle, de le simuler, d'en collecter les états et les transitions puis de les renvoyer sous la forme d'un HMI-LTS, représenté dans le code source comme une instance de la classe *LTS<ExtendedState,Transition>*.

Pour simuler un modèle, la toute première étape est de récupérer l'état initial de celui-ci. La manière dont les états et les actions sur ceux-ci sont représentés en code Java est détaillée ci-après dans la section 3.2.2. Une fois que l'on a l'état de départ, celui-ci est ajouté à une liste d'états à explorer et une liste d'états déjà visités. Ensuite, tant que la liste des états à explorer n'est pas vide, on va explorer le premier état de cette liste en testant l'effet de chaque action présente dans le modèle sur cet état. Le tuple état de départ, action, état d'arrivée forme une transition qui est ajoutée à la liste des transitions si elle ne s'y trouve pas déjà. Pour finir, chaque état résultant est ajouté à la liste des états visités, si ils ne s'y trouvent pas déjà, ainsi que dans la liste des états à explorer. Enfin, pour construire le HMI-LTS, on va parcourir la liste des états visités et celle des transitions précédemment créées afin de générer les états enrichis du modèle et les transitions entre ceux-ci et d'ajouter tout cela au HMI-LTS qui sera finalement renvoyé par la méthode. Le code complet de la méthode *simulate()* se trouve à la fin de l'annexe B.

3.2.2 Structure

Chacun des fichiers Java générés par la méthode *write()* de l'outil Translator possède exactement la même structure, le même squelette, et ce par un souci de lisibilité et de cohérence entre les différents modèles traduits. Tout d'abord, chaque fichier Java généré commence par deux imports nécessaires à l'écriture de fichiers et l'utilisation de listes. Ensuite, chaque fichier est composé d'une classe globale, qui représente l'ensemble du modèle et qui porte donc son nom.

```
1  public class Light {
```

Ensuite, on déclare toutes les actions pouvant se produire dans le modèle. Celles-ci sont déclarées à l'aide d'une énumération comme dans l'exemple ci-dessous, montrant la déclaration des actions du modèle **Light**. De plus, on y ajoute une action spéciale appelée "no action". Cette action se produit justement lorsque aucune des actions du modèle n'a lieu. C'est le cas notamment dans la colonne 4 de la *topLogicTable* du modèle **Light**. La manière dont cette colonne se traduit en code Java est donnée par après.

```
1  private enum Action {
2         POWER_MOUSECLICKED ("power.mouseClicked"),
3         FADEOUT_MOUSECLICKED ("fadeOut.mouseClicked"),
4         LIFETIMER_ACTIONPERFORMED ("lifeTimer.actionPerformed"),
5         FADINGTIMER_ACTIONPERFORMED ("fadingTimer.actionPerformed"),
6         NO_ACTION ("no action");
7
8         private final String action;
9
10        private Action (String action) {
11                this.action = action;
12        }
13
14        public String toString() {
15                return action;
16        }
17 }
```

Juste en dessous et de manière similaire, on retrouve les déclarations des états d'output des différentes tables du modèle. Pour chacune des tables ayant un état d'output utilisé dans le modèle, on retrouve une énumération comme celle ci-dessous, qui est la déclaration de l'état d'output de la table *lateralTargetSystemTable* du modèle **AutoPilot**.

```
private enum lateralTargetSystemTable {
       MOVE_PRESELECTED_LATERAL_TARGET_TO_THE_RIGHT ("move preselected lateral target to the right")
            ,
       MOVE_PRESELECTED_LATERAL_TARGET_TO_THE_LEFT ("move preselected lateral target to the left"),
       MOVE_SELECTED_LATERAL_TARGET_TO_THE_RIGHT ("move selected lateral target to the right"),
       MOVE_SELECTED_LATERAL_TARGET_TO_THE_LEFT ("move selected lateral target to the left"),
       LATERAL_TARGET_IS_STATIC ("lateral target is static");

       private final String lateraltargetsystemtable;

       private lateralTargetSystemTable (String lateraltargetsystemtable) {
              this.lateraltargetsystemtable = lateraltargetsystemtable;
       }

       public String toString() {
              return lateraltargetsystemtable;
       }
}
```

Une fois tout cela déclaré, chaque fichier Java représentant un modèle contient une sous classe *State*, qui aura pour but de représenter l'état dans lequel se trouve un modèle.

```
private static class State implements Cloneable
```

Cette classe *State* ayant pour but de représenter l'état d'un modèle, celle-ci devra forcément avoir comme attribut les différents éléments qui permettront de distinguer les états du modèle. Les premiers attributs du modèle qui sont déclarés et initialisés sont les variables de celui-ci [V]. Voici par exemple la déclaration des variables du modèle **Light**.

```
private String state = "off";
private int life = 2;
```

Ensuite ce sont les états d'output des différentes tables du modèles qui sont déclarés et initialisés. Voici une de ces déclarations, liée à l'output de la table *lateralTargetSystemTable* issue du modèle **AutoPilot**.

```
private lateralTargetSystemTable lateraltargetsystemtableOutput = lateralTargetSystemTable.
    LATERAL_TARGET_IS_STATIC;
```

Finalement, ce sont les attributs des composants [C] du modèle qui sont déclarés et initialisés. Voici par exemple l'attribut "background" du composant "powerStatus" tiré du modèle **Light**.

```
private String powerStatus_background = "255,0,0";
```

Désormais, afin de pouvoir comparer les différents états du modèles les uns par rapport aux autres, il faut qu'ils possèdent une méthode *equals()* qui va tester l'égalité de leurs attributs un à un. Voici par exemple cette méthode issue du modèle **Light**.

```
public boolean equals (Object o) {
       if(o instanceof State){
              State s = (State) o;
              return true
              && state == s.state
              && life == s.life
              && powerStatus_background.equals(s.powerStatus_background)
              ;
```

```
        } else {
                return false;
        }
}
```

Ensuite, il faut ajouter une méthode permettant d'imprimer les états. Voici la méthode *toString()* à nouveau extraite du modèle **Light**.

```
public String toString() {
        return ("["
                + "state=" + state
                + ",life=" + life
                + ",powerStatus_background=" + powerStatus_background
                + "]"
        );
}
```

Juste après cette dernière, on retrouve une méthode fort semblable appelée *stateObservation()*. Cette méthode permet de définir quelles variables sont observables par un opérateur qui interagi avec le modèle.

Finalement, on ajoute une dernière méthode à cette sous classe *State* permettant de la cloner : la méthode *clone()*. Voici celle issue du modèle **Light**.

```
public Object clone() {
        Object clone = null;
        try{
                clone = super.clone();
        } catch (CloneNotSupportedException exception) {}

        return clone;
}
```

Ensuite, et ce afin de pourvoir simuler le comportement du modèle, chaque table va être réécrite en code Java, sous forme d'une méthode. Chaque table sera une méthode et prendra comme paramètre une action (Action) et un état (State). Le contenu des tables logiques est traduit sous forme de code Java grâce aux règles sémantiques de traduction énoncées dans la section 3.2.3. Dans le code Java, chacune des colonnes de la table logique deviendra un "if-statement", dont la condition sera la traduction des inputs de cette colonne en Java, et le contenu la traduction des outputs de cette même colonne. Voici par exemple la traduction en code Java de la table *topLogicTable* issue du modèle **Light** et de ses colonnes 0 et 4.

```
private static void topLogicTable(Action action, State to) {
        // Situation 0
        if((to.state.equals("on")) && (to.life>0) && (action == Action.POWER_MOUSECLICKED)){
                to.state = "off";
                to.powerStatus_background = "255,0,0";

                actionPassed = true;
        }
        ...
        // Situation 4
        else if((to.state.equals("on") || to.state.equals("fading")) && ((to.life==0)) && (action ==
             Action.NO_ACTION)){
                to.state = "off";
                to.powerStatus_background = "255,0,0";

                noActionPassed = true;
```

```
16              }
17          ...
18  }
```

Le code complet du fichier Java représentant le modèle **Light** se trouve dans l'annexe B.

3.2.3 Règles de traduction d'une table logique

Cette section contient donc les quelques règles de traductions relatives au contenu des tables logiques. Ces règles découlent de l'observation des possibilités offertes par le logiciel ADEPT pour la confection des tables logiques et sont inspirées des règles sémantiques déjà établies dans [2].

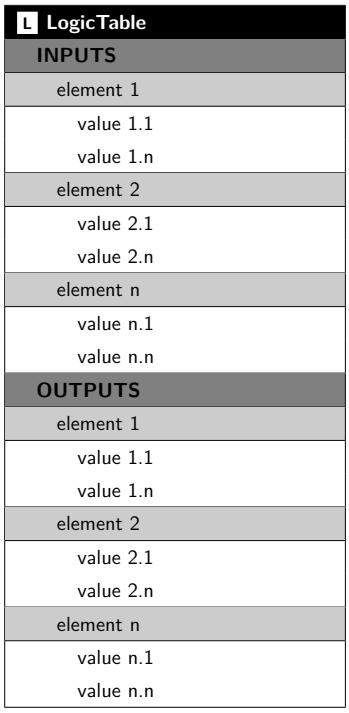


FIGURE 3.4 – Structure des tables logiques.

INPUTS

Comme déjà dit précédemment, les inputs d'une table logique peuvent se traduire en un "if-statement" en Java. Donc, chaque bullet présente dans la partie inputs d'une colonne représenterait une condition, c'est à dire une paire élément - valeur. Dés lors, en se basant sur la figure 3.4 qui représente la structure générale d'une table logique dans ADEPT, chaque input de colonne prendrait une forme semblable à celle-ci :

```
1  if(element1 == value1.x && (element2 == value2.x || ... || element2 == value2.y) && ... && elementn
       == valuen.x)
```

Chaque bullet noire, chaque condition est en fait un test sur une valeur d'un élément. Chaque condition aura justement une forme qui sera déterminée par cet élément et cette valeur qu'elle représente. Voici quelques règles de traduction, une bullet étant représentée par *cond*(element, value).

- *cond*(ACTIONS, C component.event) => action == Action.COMPONENT_EVENT

  cond(ACTIONS, O timer.event) => action == Action.TIMER_EVENT

  Si la condition porte sur la valeur d'une action, on teste simplement par rapport à l'équivalent de cette valeur dans l'énumération Action présente dans le code Java généré à partir du modèle.

- *cond*(L table.outputState, state value) => tableOutput == table.STATE_VALUE

  Si la condition porte sur la valeur d'un output d'une table logique, on teste simplement par rapport à l'équivalent de cette valeur dans l'énumération des états d'outputs possibles de la table.

- *cond*(V String variable, expr) => variable.equals(expr)
  cond(V Boolean variable, expr) => variable == expr
  cond(V Number variable, V variable) => variable == variable
  cond(V Number variable, number) => variable == number
  cond(V Number variable, expr) => variable expr

  La troisième et dernière possibilité est que la condition porte sur une variables classique. La traduction va alors dépendre du type de la variable et ensuite du contenu de la valeur que l'on veut tester. D'une part, si le type de la variable est String, on va utiliser la méthode *.equals()* avec avec la valeur comme paramètre pour déterminer si la condition est vraie ou fausse. Ensuite, si la variable est de type boolean, on va utiliser le symbole "==" pour comparer la valeur. Finalement, si le type de la variable est un nombre, c'est-à-dire int, double ou float, on va également utiliser le symbole "==" pour tester une valeur qui est un nombre ou une autre variable. Par contre, si la valeur est une expression plus complexe contenant des symboles comme "<", ">=", "&&", etc, alors la condition sera simplement une concaténation de la valeur à l'élément.

OUTPUTS

Dans la traduction Java d'une table logique, si les inputs d'une colonne forme un "if-statement", les outputs de cette même colonne forment alors le corps de cet "if-statement", c'est-à-dire les opérations à effectuer si celui-ci est valide. Dans ce cas-ci, les bullets forment toujours une paire élément, valeur mais la traduction de chaque bullet donnera lieu à un "statement" en Java. Si l'on se base à nouveau sur le tableau à la figure 3.4, chaque corps de "if-statement" serait de la forme suivante :

```
element1 = value1.x;
element2 = value2.x;
elementn = valuen.x;
```

Cependant, tout comme pour les inputs, la traduction des outputs se fait selon certaines règles dans lesquelles chaque bullet est représentée par *stmt*(element, value).

- *stmt*(LOGIC, L table) => table();

  La première possibilité est que l'élément soit l'élément spécial LOGIC, cela veut dire que les valeurs seront des appels à d'autres tables. Donc, l'opération devient un appel à la méthode liée à la table présente comme valeur.

- *stmt*(L table.outputState, state value) => tableOutput = table.STATE_VALUE;

  Deuxièmement, si l'élément est l'état d'output d'une table, la variable Java contenant cet état sera modifiée et on lui assignera l'équivalent de la valeur dans les états d'output possibles de la table.

- *stmt*(C component.field, F method) => component_field = method();
  stmt(C component.field, V variable) => component_field = variable;
  stmt(C component.field , expr) => component_field = String expr;

  Ensuite, si l'élément est l'attribut d'un composant visuel, alors on lui assigne soit l'appel à la méthode liée à la valeur si celle-ci indique une fonction, le contenu de la variable liée à la valeur si celle-ci indique une variable ou alors on lui assigne un String contenant la valeur si celle-ci est une expression.

- *stmt*(V variable , F function) => variable = function();
  stmt(V variable , V variable2) => variable = variable2;
  stmt(V String variable , expr) => variable = String expr;
  stmt(V Boolean variable , expr) => variable = expr;
  stmt(V Number variable , number) => variable = number;
  stmt(V Number variable , expr) => variable expr;

  Finalement, la dernière possibilité est que l'élément soit une variable classique. Dès lors, la traduction va d'abord dépendre du type de la valeur et ensuite du type de l'élément (String, Boolean ou Number). Tout d'abord, si la valeur est une fonction, alors on assignera simplement à la variable l'appel à cette méthode. Ensuite, si la valeur est elle-même une autre variable, on assignera à l'élément le contenu de la variable indiquée dans la valeur. Il faut dès à présent regarder au type de la variable indiquée en élément. Si celle-ci est un String, alors on lui assignera un String contenant l'expression indiquée comme valeur. Si c'est un Boolean, on lui assignera simplement la valeur de l'expression. Si la variable est de type int, double ou float, et que la valeur contient un nombre (positif ou négatif), alors on lui assignera

ce nombre. Finalement, si la variable est un nombre et que la valeur contient une expression complexe commençant par un symbole tel que + ou - n'étant pas suivi d'un chiffre, alors on concatène simplement l'expression à la variable dans le code Java.

Le code complet du fichier Light.java se trouve à l'annexe B.

3.3 Compilation du fichier Java

Cette opération est donc la deuxième étape de la traduction après la génération du fichier Java par l'outil Translator. Elle consiste en la compilation de ce programme Java généré grâce à la méthode *Runtime.getRuntime().exec()*. Cette méthode permet d'exécuter la commande passée en paramètre. Dans notre cas, la commande va compiler le fichier Java lié au modèle et placer le fichier class résultant là ou se trouve le code source de la classe *JpfHmi*, son emplacement ayant été obtenu au préalable grâce au caractère réflexif du langage Java. De plus, un mécanisme permettant d'afficher les erreurs de compilation éventuelles a été ajouté afin d'informer l'utilisateur en cas de problème. Le code source de cette deuxième étape se trouve dans la classe *ADEPT2LTS* dont le code source se trouve à l'annexe E.

3.4 Appel à la méthode *simulate()*

Cette troisième et dernière étape consiste donc en l'appel à la méthode *simulate()* évoquée dans la section 3.2.1 et qui se trouve donc dans le fichier class issu de la compilation du fichier Java précédemment généré. Cette méthode permet donc de simuler le comportement du modèle représenté et de renvoyer l'automate correspondant. Pour faire appel à cette méthode, il aura fallu une fois de plus faire usage du caractère réflexif du langage Java et de sa méthode *Class.forName()* renvoyant la classe dont le nom est donné en paramètre. Cependant, cette méthode ne parvient à trouver que les classes étant dans l'entourage direct de la classe qui y fait appel, ce qui explique le fait que le fichier Java compilé soit placé au même endroit que le code source de la classe *JpfHmi*. Le code complet de cette étape se trouve dans le code source de la classe ADEPT2LTS donné à l'annexe E.

3.5 Fichiers texte

Lors de la traduction d'un modèle ADEPT, ils est possible de garder une trace du HMI-LTS généré et ce grâce à cinq fichiers texte qui représentent l'automate produit. La génération de ces fichiers est optionnelle et se fait lors de la simulation du modèle ADEPT, c'est à dire dans la méthode *simulate()* de la classe liée à ce modèle. Voici une description de ces cinq fichiers texte.

state Ce fichier contient la description de tous les états possibles de l'automate, c'est-à-dire tous les états que le modèle va atteindre durant son exécution. Chaque ligne du fichier représente un état et chacune d'entre elles commence par un numéro qui définit à quel état visible, observable l'état appartient. Les états visibles représentent la manière dont l'opérateur voit, observe le système. Par exemple, si aucune variable du système n'est observable (on ne parle pas ici uniquement des variables dites classiques V), alors tous les états seront précédés d'un "0" indiquant qu'ils appartiennent tous

au même état observable. En effet, l'opérateur ne peut distinguer aucun des états vu qu'il ne connaît aucune de leurs valeurs, celles-ci n'étant pas visibles. Au contraire, si toutes les variables sont observables, tous les états seront précédés d'un numéro différent, vu que l'opérateur peut observer toutes les valeurs des différents états. Chaque état est représenté selon une sémantique bien définie :

element1 name=element1 state, ... ,elementN name=elementN state

Voici par exemple les deux premiers états contenus dans le fichier *Lightstate.txt* représentant les états du modèle **Light**.

```
0:[ state=off, life =2,powerStatus_background=255,0,0]
1:[ state=on,life=2,powerStatus_background=0,255,0]
...
```

transitionAction Dans ce second fichier, nous pouvons trouver une liste des actions effectivement utilisées durant la simulation du modèle que le fichier représente. Le fichier contient simplement les noms des actions, une action par ligne, comme dans le fichier *LighttansitionAction.txt* dont quelques lignes sont données en exemple ci-dessous. Vous noterez qu'est présente également l'action "no action", qui comme cela a déjà été évoqué, signifie qu'aucune action n'a lieu.

```
power.mouseClicked
fadeOut.mouseClicked
...
no action
```

transition Ce fichier contient la liste de toutes les transitions qui sont franchies durant la simulation du modèle que le fichier représente. Le fichier contient une transition par ligne. Chaque transition contenue dans le fichier est représentée par un numéro d'action, un état de départ et finalement un état d'arrivée selon la sémantique suivante :

action number ;*starting state number* ;*finishing state number*

Le nombre assigné à un état renseigne la position de celui-ci dans le fichier "state". Le même système est en vigueur pour les numéros correspondants aux actions et au fichier "transitionAction". Voici par exemple quelques une des transitions contenues dans le fichier *Lighttransition.txt* issu du modèle **Light**. Il est important de noter que le "tau" est en fait l'identifiant de l'action "no action".

```
0;0;1
0;1;0
1;1;2
...
tau;6;7
```

machine Ce fichier contient du code Java. Ce code Java, s'il est exécuté, permet de générer le HMI-LTS lié au modèle représenté. Chaque fichier machine a la même structure dont voici quelques extraits provenant du fichier *Lightmachine.txt* issu du modèle **Light**. L'observation de ces fichiers permet de mieux comprendre la manière dont on génère un HMI-LTS lors de la simulation d'un modèle car les techniques sont les mêmes. D'abord on déclare les actions du modèles.

```
Action a0 = new Action ("power.mouseClicked", ActionType.COMMAND);
```

Ensuite, on trouve les déclarations des actions liées aux observations sur les états.

```
Action o0 = new Action("O0", ActionType.STATE_OBSERVATION);
```

Pour en finir avec les actions, on déclare l'action spéciale "tau", qui est utilisée pour désigner qu'il n'y a aucune action utilisée (no action).

```
Action tau = new Action("tau", ActionType.TAU);
```

Après avoir déclaré toutes les actions, on initialise le HMI-LTS (LTS<Extended State,Transition>) et on y ajoute chacun des états de l'automate représentant le modèle. Afin de former des états enrichis, on ajoute simplement une action sur l'observation d'un état à l'état non enrichi.

```
LTS<ExtendedState,Transition> system = new LTS<ExtendedState,Transition>();
ExtendedState s0 = new ExtendedState ("S0", o0);
system.addState (s0);
```

Finalement, on ajoute au HMI-LTS chacune des transitions issues de l'automate.

```
system.addTransition (new Transition (a0), s0, s1);
system.addTransition (new Transition (a0), s1, s0);
```

stateObservationAction Ce fichier contient simplement un nombre indiquant le nombre d'actions liées aux observations sur les états, augmenté de un.

Si l'on se base sur le fichier contenant les états du modèle (state.txt) et sur celui contenant les transitions (transition.txt), on peut construire une représentation d'un modèle sous forme de diagramme comme celui de la figure 3.5. Cette représentation est issue de l'analyse des fichiers texte se trouvant dans l'annexe C et représentant l'automate obtenu de la traduction du modèle **Light**.

3.6 Fichiers de configuration

Lorsque l'on souhaite effectuer une analyse sur un modèle, ou en récupérer le modèle mental, on pourrait vouloir analyser le modèle dans son entièreté ou alors simplement une partie de celui-ci. De plus, dans certains cas comme le modèle **Autopilot**, il est tout simplement "impossible" de générer un HMI-LTS tant le modèle est vaste et complexe. En effet, la simulation de celui-ci est beaucoup trop coûteuse en terme de mémoire pour le matériel à notre disposition. Dès lors, il a fallu trouver une solution pour permettre de limiter la traduction d'un modèle à une partie de celui-ci. Une des solutions aurait pu être d'obliger l'utilisateur à réduire lui-même le modèle directement dans ADEPT mais cela n'aurait pas été pratique du tout. Une bien meilleure solution fût de permettre à l'utilisateur de joindre un fichier de configuration au modèle qu'il souhaite analyser. Ces fichiers de configuration se retrouve sous la forme d'un fichier XML dont l'extension est .conf et dont la structure est détaillée dans la section 3.6.1.

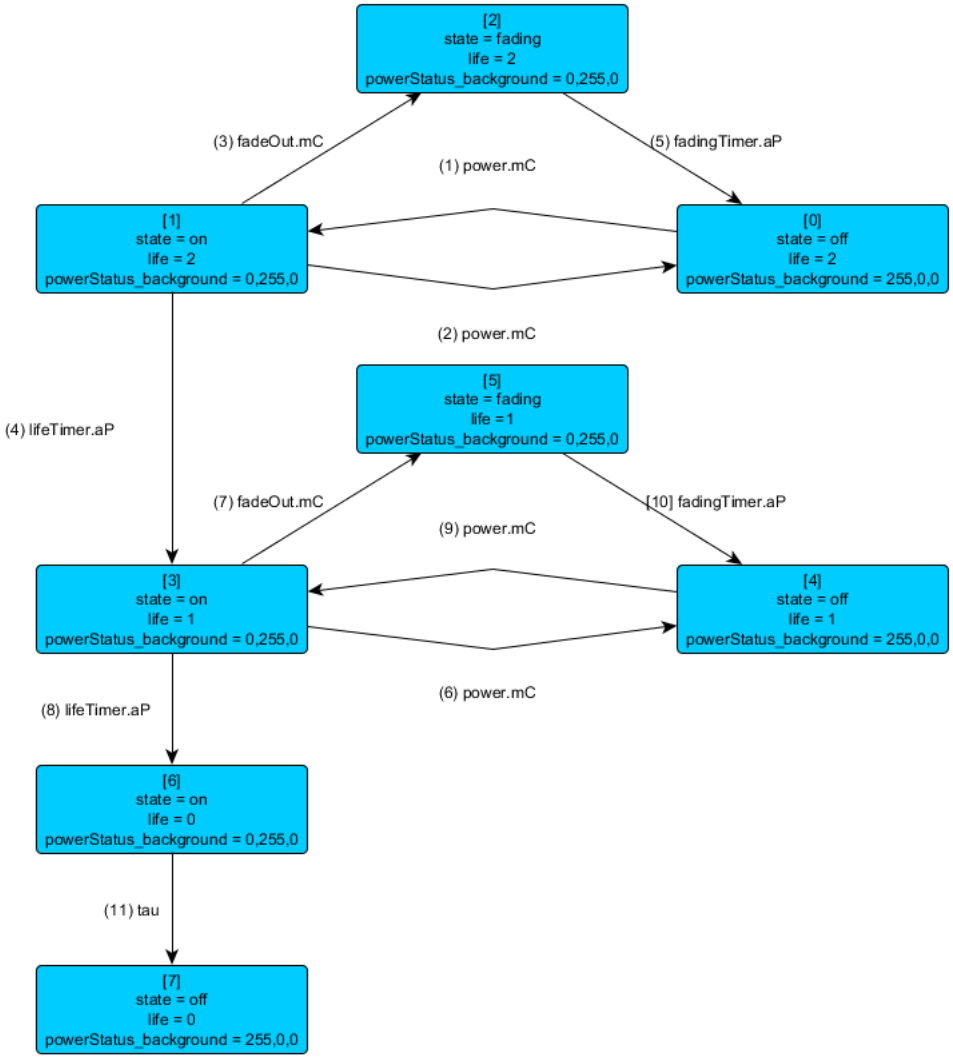


FIGURE 3.5 – Représentation de l'automate issu du modèle **Light**

Dès lors que la décision fût prise d'utiliser des fichiers de configuration pour limiter la taille et la complexité des modèles à sa guise, il a fallu réfléchir sur les caractéristiques des modèles ADEPT qui allaient pouvoir être utilisées ou non pour effectuer une analyse. Afin de permettre à l'utilisateur d'analyser un maximum de variantes d'un modèle, et donc de lui laisser un maximum de liberté, la totalité des éléments d'un modèle peuvent être mis de côté lors de la traduction de celui-ci. Le rôle de certains éléments peut également être modifié.

- **Tables** – Premièrement, il est apparu primordial de pouvoir limiter les tables logiques, de choisir quelles tables logiques on souhaitait utiliser pour une analyse. De plus, il paraissait important de permettre à l'utilisateur de choisir les colonnes (situations) qu'il souhaitait garder pour chaque table. Donc, l'utilisateur peut d'une part enlever certaines tables, et d'autre part modifier le comportement de celles qu'il garde en enlevant les colonnes de son choix. Il est important de noter que le fait d'utiliser une table signifie que l'on va également utiliser son état d'output à l'intérieur d'elle-même et dans les autres tables.

- **Variables** – Un deuxième point fût de permettre à l'utilisateur de choisir les variables dites classiques qu'il souhaitait conserver pour la transformation du modèle. En plus de pouvoir enlever complètement une variable, l'utilisateur peut également modifier les variables en leur assignant une valeur maximale et/ou une valeur minimale. Il peut également choisir quelles variables seront observables ou non, c'est à dire quelles variables seront utilisées pour distinguer deux états l'un de l'autre du point de vue de l'opérateur.

- **Composants** – De manière un peu similaire aux variables, la possibilité de limiter les composants visuels utilisés a été offerte à l'utilisateur. En effet, on peut retrouver dans diverses tables certains attributs des composants visuels du modèle. Ceux-ci font partie des éléments qui définissent l'état d'un modèle et il paraissait dès lors utile de permettre leur limitation. De plus, il est également possible de les rendre observables ou non, comme pour les variables classiques.

- **Outputs** – La possibilité de sélectionner quels outputs de tables logiques seront considérés comme faisant partie des variables du système a aussi été donnée à l'utilisateur. De ce fait, ceux-ci sont traités de la même manière que les composants et les variables. Il est donc également possible de les rendre observables ou non. Les outputs ont été séparés des tables pour permettre d'utiliser une table et son output normalement, sans que celui-ci soit considéré comme une variable du système et donc utilisé pour tester l'égalité de deux états. Il faut être attentif que dans ce cas ci c'est tout ce que les outputs sélectionnés font, ils ne sont pas responsables de l'utilisation directe de ces outputs dans les inputs où outputs des tables, qui eux dépendent directement des tables déclarées.

- **Actions** – Le système de fichiers de configuration offre également la possibilité de sélectionner les actions que l'on souhaite utiliser ou non pour la transformation du modèle. La limitation du nombre d'actions d'un modèle permet de réduire les possibilités dans le comportement du modèle et donc les transitions de l'automate obtenu après simulation de celui-ci. Cela peut donc également limiter le nombre d'états atteints par cette simulation et donc forcément le poids de celle-ci en terme de mémoire. De plus, il est également possible de choisir comment l'action doit être interprétée, c'est à dire si elle doit être vue par le système comme une commande ou comme une observation, un événement se produisant dans l'environnement du modèle. Il est important de noter que les actions initialement liées à un composant visuel, les commandes, ne le sont plus ici. Cela signifie que le fait de ne pas utiliser un composant n'affecte en rien les actions du modèle.

- **Fonctions** – Pour finir, la possibilité de limiter les fonctions utilisées a été ajoutée. Cependant, celle-ci n'a pas été ajoutée dans le but de diminuer le nombre de variables d'un état ou le nombre de transitions lors de la simulation d'un modèle mais plutôt pour éviter les problèmes de compilation avec le fichier Java généré. En effet, la non sélection d'une variable par exemple pourrait entraîner une erreur dans le corps d'une méthode si celle-ci fait appel à cette variable.

Une fois les caractéristiques des fichiers de configuration déterminées, il a fallu également réfléchir à un système pour récupérer les données contenues dans ces fichiers. Pour ce faire, un outil ConfParser a été ajouté à l'outil Translator. Cet outil va simplement récupérer

les informations contenues dans un fichier de configuration et les transmettre à l'outil Translator, qui va lui-même les donner à l'outil JavaWriter. En effet, c'est cet outil qui va, lors de la traduction du modèle vers son équivalent en fichier Java, vérifier que les informations qu'il est sur le point d'écrire dans le fichier doivent être utilisées et sont donc présentes dans le ConfParser.

3.6.1 Structure

Chaque fichier de configuration généré a la même structure. Comme cela a déjà été dit auparavant, le contenu des fichiers s'apparente à des balises de fichiers XML. Comme pour les fichiers de type sgb déjà évoqués à la section 2.1.4, les fichiers de configuration possèdent un noeud racine qui contient tous les autres noeuds définissant la configuration.

```
<AdeptConfig>
        ...
</AdeptConfig>
```

Le premier sous noeud est le noeud <Tables> qui contient les noeuds définissant quelles tables sont utilisées. Ce noeud contient donc une liste de noeuds <Table> contenant chacun un nom, un flag "used" servant à déterminer s'il faut l'utiliser ou non et une liste de noeuds <Situation>. Ces derniers noeuds représentent les colonnes de la table et possèdent également un flag "used", ce qui sera le cas de chaque élément du modèle représenté par une balise dans le fichier de configuration. Voici par exemple la représentation de la table *topLogicTable* du modèle **Light** et de ses deux premières colonnes.

```
<Tables>
        <Table name="topLogicTable" used="true">
                <Situation used="true">0</Situation>
                <Situation used="true">1</Situation>
                ...
        </Table>
</Tables>
```

Ensuite on retrouve le noeud <Variables> qui contient les noeuds relatifs aux variables dites classiques V. Chacun de ces sous noeuds contient le nom de la variable à laquelle il fait référence ainsi qu'un flag "observable" indiquant si la variable est observable, c'est à dire si elle est visible de l'opérateur. De plus, les noeuds contiennent des attributs "max" et "min" permettant de limiter les valeurs que peut prendre une variable lors de la simulation du modèle. Voici par exemple la partie relative aux variables du fichier de configuration issu du modèle **Light**.

```
<Variables>
        <Var used="true" observable="true" max="" min="">state</Var>
        <Var used="true" observable="true" max="" min="">life</Var>
</Variables>
```

De manière un peu similaire aux variables, on retrouve un noeud <Components> qui contient les composants visuels du modèle. Cependant, ceux-ci ne possèdent pas d'attributs "max" et "min". Voici la partie relative aux composants visuels du fichier de configuration tiré du modèle **Light**.

```
<Components>
        <Comp used="true" observable="true">powerStatus</Comp>
</Components>
```

On retrouve ensuite un noeud <Outputs> qui contient les noeuds définissant les outputs des tables logiques devant être utilisés comme variables du modèle. Pour rappel, on ne parle pas ici uniquement des variables dites classiques [V] mais bien de l'ensemble des éléments utilisés pour tester l'égalité de deux états du modèle. Tout comme les variables et les composants, ces noeuds possèdent un flag "observable". Voici par exemple le début de la partie relative aux outputs des tables logiques issue du modèle **Autopilot**.

```
<Outputs>
        <Out used="false" observable="false">verticalTargetSystemTableOutputs</Out>
        <Out used="true" observable="false">lateralTargetSystemTableOutputs</Out>
        ...
</Outputs>
```

Le noeud suivant est le noeud <Actions> qui contient les noeuds associés aux actions du modèle. Chacun des noeuds contient simplement le nom de l'action à laquelle il se rapporte, un attribut "actionType" permettant de spécifier le type de l'action et donc si celle-ci doit être interprétée comme une commande ou une observation. Par défaut dans le fichier, le contenu des attributs "actionType" est mis à sa valeur de base, c'est à dire que les actions sur les timers [O] sont des observations et que les actions sur les composants visuels [C] sont des commandes. Voici par exemple un morceau de la partie relative aux actions issue du modèle **Light**.

```
<Actions>
        <Act used="true" actionType="COMMAND">power.mouseClicked</Act>
        ...
        <Act used="true" actionType="OBSERVATION">lifeTimer.actionPerformed</Act>
        ...
</Actions>
```

Pour finir, on retrouve le noeud <Functions> qui contient les noeuds relatifs aux fonctions [F] du modèle. Voici par exemple le début de la partie relative aux fonctions tirée du modèle **Autopilot**.

```
<Functions>
        <Fun used="true">minProtectSpeedTarget</Fun>
        <Fun used="true">invertLateralTargetError</Fun>
        ...
</Functions>
```

L'entièreté du fichier de configuration lié au modèle **Light** se trouve à l'annexe D.

3.6.2 Exemple du modèle Light simplifié

Afin d'illustrer le fonctionnement des fichiers de configuration, le modèle **Light** va être simplifié sous une forme plus légère, c'est à dire en enlevant la variable "life" du modèle ainsi que le composant visuel "powerStatus" et les colonnes 3 et 4 de la table *topLogicTable*.

Au niveau du fichier de configuration, on obtient ceci pour les tables logiques et les situations (colonnes) :

```
<Table name="topLogicTable" used="true">
        <Situation used="true">0</Situation>
        <Situation used="true">1</Situation>
        <Situation used="true">2</Situation>
        <Situation used="false">3</Situation>
        <Situation used="false">4</Situation>
```

```
7         <Situation used="true">5</Situation>
8 </Table>
```

Au niveau des variables, cela va donner ceci :

```
1 <Var used="true" observable="true" max="" min="">state</Var>
2 <Var used="false" observable="false" max="" min="">life</Var>
```

Finalement, l'enlèvement du composant visuel "powerStatus" se fait de la même manière que l'enlèvement des variables. De plus, on peut également modifier les actions, vu que l'action "lifeTimer" n'aura plus lieu d'être mais ce n'est pas obligatoire car elle n'influence plus le comportement du modèle étant donné que les colonnes dans lesquelles elle intervenait ont été supprimées. Dès lors, avec un tel fichier de configuration, on obtient la table logique simplifiée représentée à la figure 3.6.

	0	1	2	5
L topLogicTable				
INPUTS				
V state				
on	•		•	
fading				•
off		•		
ACTIONS				
power.mouseclicked	•	•		
fadeOut.mouseclicked			•	
fadingTimer.actionPerformed				•
OUTPUTS				
V state				
on		•		
fading			•	
off	•			•

FIGURE 3.6 – Représentation de la table *topLogicTable* issue du modèle **Light** après simplification au moyen d'un fichier de configuration.

Au niveau du fichier Java qui sera généré, on va pouvoir observer quelques changements notables. En effet, à chaque endroit du fichier ou la variable "life" ou le composant "powerStatus" apparaissaient initialement, les même lignes de code vont être retrouvées mais sous forme de commentaires, que ce soit dans la méthode *equals()* par exemple, ou lors de leurs déclarations. Il aurait bien sur été possible de choisir d'enlever complètement ces lignes de codes plutôt que de les mettre en commentaire, cela aurait pu raccourcir les fichiers Java qui peuvent parfois être très longs. Cependant, la mise en commentaire de ces lignes permet de visualiser plus facilement les effets du fichier de configuration sur le modèle initial, et facilite donc la mise en place du fichier de configuration souhaité. Dans les tables logiques, le travail est un peu différent à cause des conditions. En effet, dans celles-ci, l'utilisation d'une variable qui n'est pas utilisée dans le fichier de configuration mène au booléen "true", car celui-ci ne modifie pas le résultat de l'expression testée dans la condition. Voici par exemple la traduction en Java de la colonne 0 du modèle réduit.

```
// Situation 0
if((to.state.equals("on")) && (true) && (action == Action.POWER_MOUSECLICKED)){
        to.state = "off";
        // to.powerStatus_background = "255,0,0";

        actionPassed = true;
}
```

Le résultat de cette simplification du modèle **Light** donne lieu à un automate possédant trois états et quatre transitions. Une représentation de ce résultat se trouve à la figure 3.7.

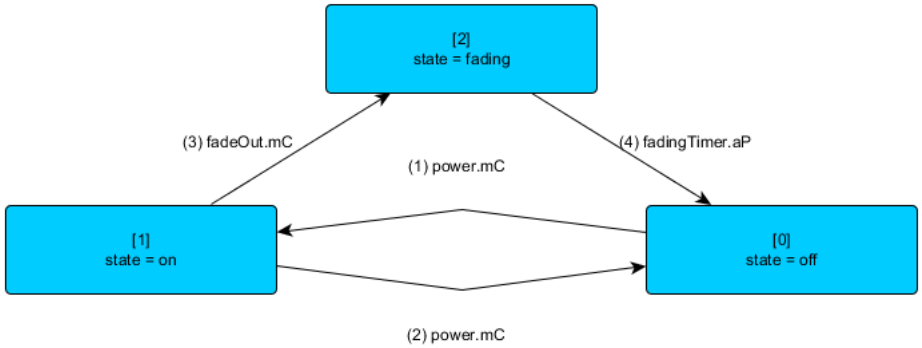


FIGURE 3.7 – Représentation de l'automate issu du modèle **Light** simplifié.

3.6.3 Génération des fichiers

Le problème avec cette manière de structurer un fichier de configuration sous forme de noeuds, c'est que ce nombre de noeuds est directement proportionnel au nombre d'éléments du modèle et donc à sa taille. Or comme cela a déjà été mentionné et comme cela va être illustré dans la section 4, certains modèles peuvent être très vastes et complexes, pouvant dés lors nécessiter des fichiers de configuration de plusieurs centaines de lignes. Pour pallier à ce problème de taille, un outil du nom de ConfGenerator a été ajouté à l'outil ADEPT2LTS, permettant de générer automatiquement un fichier de configuration depuis un modèle donné.

ConfGenerator

La classe *ConfGenerator* est donc une classe qui permet, de manière un peu similaire à la classe *JavaWriter*, de générer automatiquement un fichier de configuration lié à un modèle. Cette classe met à disposition diverses méthodes permettant d'écrire chacune des parties du fichier de configuration, en fonction des données qu'elles reçoivent en paramètre. Ces données, c'est en fait l'outil Translator qui va lui fournir et ce grâce à une nouvelle méthode qui lui a été ajoutée et qui sera elle-même appelée par l'outil ADEPT2LTS et sa méthode *generateConfigurationFile()*. Une caractéristique de l'outil ConfGenerator est qu'il prend comme paramètre un mode permettant de définir le mode utilisé pour la génération du fichier de configuration. Il existe deux modes : l'un permettant d'utiliser tout les éléments du modèle et donc de mettre les flags "used" de tout les noeuds à true, l'autre de n'en utiliser aucun et donc de mettre tous les flags "used" à false. Ce concept de modes permet de simplifier le travail de l'utilisateur lorsqu'il veut obtenir un fichier bien précis. En effet, s'il veut simplement enlever quelques éléments du modèles, il utilisera le mode mettant tout

les flags à true, tandis que s'il ne veut utiliser que quelques éléments du modèles, il pourra simplement utiliser le mode mettant les flags à false et puis mettre à true manuellement les quelques éléments dont il a besoin.

3.6.4 Limitations

L'utilisation de fichiers de configuration est très pratique pour manipuler les modèles et permet de les modeler de nombreuses façons. Cependant, il faut savoir que ce principe souffre de quelques limitations.

Une première limitation à laquelle il faut être attentif lorsque l'on veut traduire un modèle ADEPT comportant plusieurs tables est qu'il est toujours nécessaire d'utiliser la table *topLogicTable*, même si on veut modéliser une table complètement différente. En effet, lors de la simulation d'un modèle, c'est cette table qui est le point de départ et qui fait appel aux autres tables si elles existent. De manière similaire, il faut faire attention a utiliser toutes les tables menant à la table que l'on souhaite traduire, sinon la simulation ne pourra jamais atteindre cette dernière.

Une seconde limitation du principe des fichiers de configuration découle du fait qu'il n'est pas possible de sélectionner soi-même quel input d'une table fera office d'action, de transition dans l'automate généré. En effet, seules les actions définies comme telle dans le modèle ADEPT peuvent être utilisées comme transitions dans l'automate le représentant. Pour rappel, ces actions sont soit des actions liées à un composant visuel, c'est à dire des commandes, soit des actions associées à des timers, c'est à dire des observations.

Une autre chose importante à prendre en compte lorsque l'on modifie un fichier de configuration, c'est que les flags "used" et "observable" sont en quelques sortes liés. En effet, si par exemple une variable est mise comme observable, celle-ci ne sera prise en compte que si le flag "used" est mis à true également. Sans cela, le fait de la rendre observable ne changera rien, elle sera toujours considérée comme non observable. Même si rendre une variable non utilisée observable n'aurait aucun sens, une erreur est vite arrivée et cette manière de faire permet d'éviter les erreurs de compilation pouvant apparaître dans le fichier Java.

Un phénomène pouvant se produire et pouvant poser quelques problèmes imprévus est le phénomène dit de la cascade. Cela signifie que lorsque l'on décide de ne pas utiliser une variable, tout ce qui touche à cette variable va également être mis de côté. Par exemple, si une ligne contient l'assignation d'une variable non utilisée à un composant visuel utilisé, toute la ligne est mise en commentaire et du coup l'état du composant ne change pas. L'enlèvement d'une variable peut donc parfois avoir plus de conséquences qu'on ne le pense sur le comportement d'un modèle.

Lorsque l'on modifie un fichier de configuration, il faut être prudent avec les fonctions, qui sont directement victimes du phénomène de cascade présenté juste avant. En effet, lorsque l'on veut utiliser une fonction, il faut s'assurer que toutes les variables présentes dans cette fonction soient bien utilisées. En effet, si l'une des variables de la méthode n'est pas utilisée, il faut se poser la question de savoir que renvoyer à la place du contenu de cette méthode. Faut il renvoyer "null" ? Cela ne fonctionnerait pas avec les int, double etc... Renvoyer une valeur en fonction du type du return de la fonction ? Oui mais quelle

valeur ? renvoyer un 0 par exemple pourrait fausser complètement le comportement du modèle. Dès lors le système, lorsqu'il va détecter une variable non utilisée dans le corps de la méthode, va imprimer un message de sorte à indiquer à l'utilisateur dans quelle méthode se trouve son erreur et un commentaire apparaîtra à la ligne concernée dans le fichier Java généré. Si cette ligne est le return de la méthode, cela entraînera également une erreur de compilation.

Chapitre 4

Le cas du pilote automatique

A plusieurs reprises auparavant, le modèle **Autopilot** a été mentionné. Celui-ci est en fait un modèle créé par les chercheurs de la NASA Ames représentant le pilote automatique d'un avion de type Boeing 777. Une illustration de ce modèle lorsqu'il est ouvert dans ADEPT est présenté dans la figure 4.1 qui a été tirée de [2]. Le pilote automatique d'un avion est un outil très complexe ce qui rend le modèle **Autopilot** assez difficile à comprendre et surtout très vaste. En effet, les tables logiques décrivant ce modèle sont au nombre de 39 et le fichier sgb le représentant contient plus de 2500 noeuds.

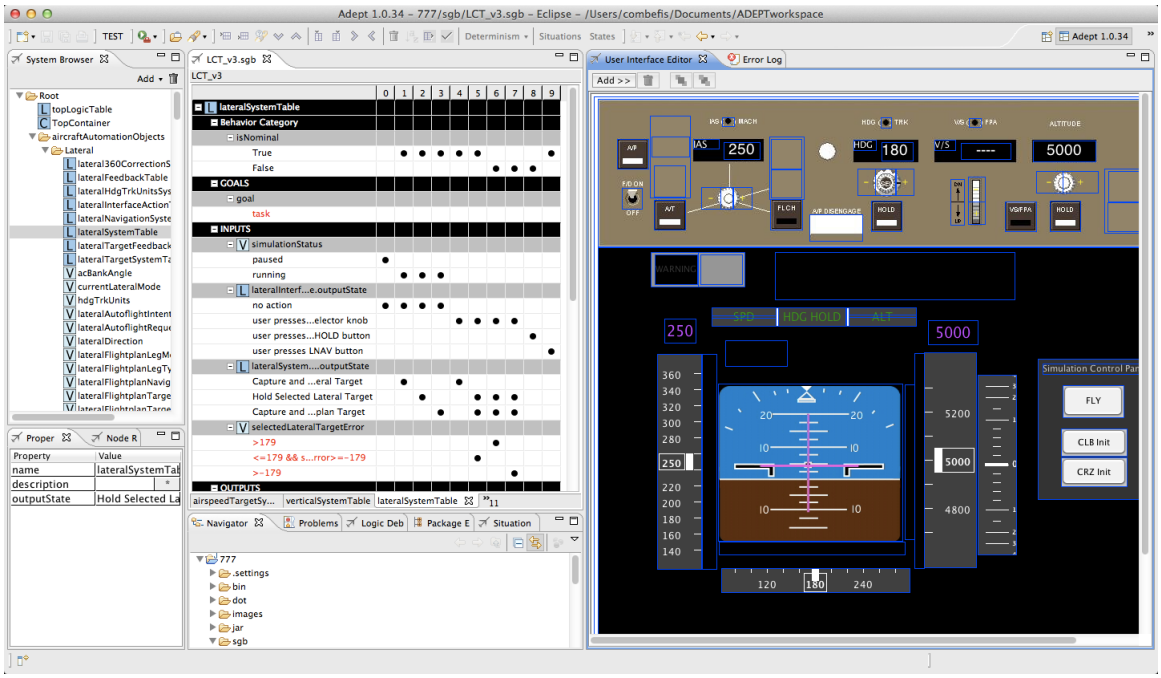


FIGURE 4.1 – Capture d'écran du modèle **Autopilot** une fois ouvert avec ADEPT.

4.1 Structure

Les tables du modèle sont structurées de sorte à pouvoir facilement distinguer les actions effectuées par l'utilisateur sur l'interface, les décisions internes de logique prises par le système et les réponses du système à l'utilisateur. Cette structure particulière est appelée la structure ASF en référence aux termes "action-system-feedback". Celle-ci implique que

les tables du modèles soient séparées en trois familles de table : les “ActionTable”, les “SystemTable” et les “FeedbackTable”. La figure 4.2 illustre bien le rôle des différents types de table et les interactions qu’il existe entre elles dans une telle structure.

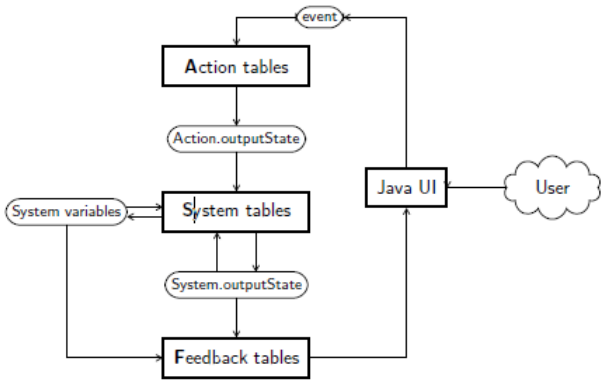


FIGURE 4.2 – Représentation de l’exécution des tables dans une structure ASF issue de [2].

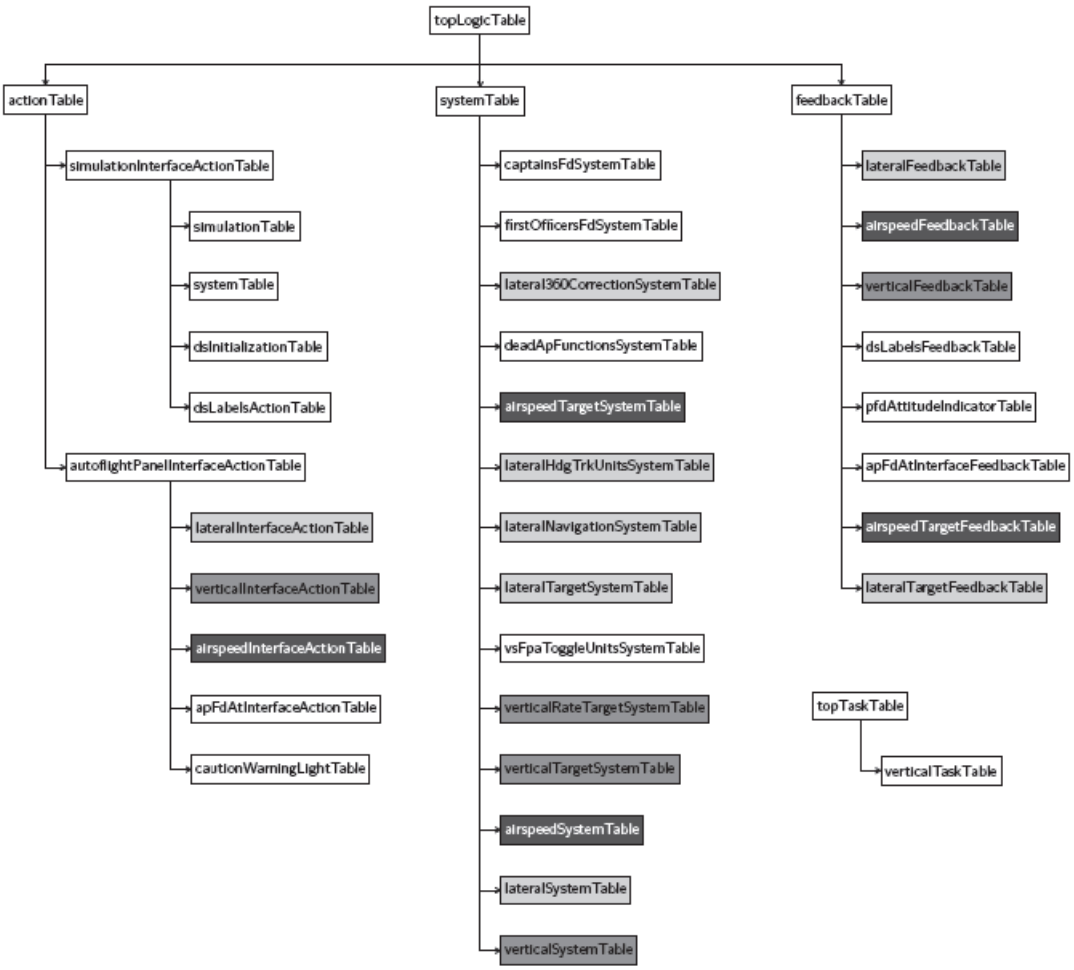


FIGURE 4.3 – Structure des tables du modèle **Autopilot**, issue de [2].

Il est également intéressant de remarquer que la plupart des tables du modèle **Autopilot** sont subdivisées en trois grandes parties. La première partie concerne les tables liées au comportement vertical de l'avion, c'est à dire son altitude, le fait qu'il descende ou qu'il monte. La seconde partie est composée des tables décrivant la vitesse de l'avion. Finalement, une troisième grande partie définit les tables gérant le comportement latéral de l'avion, c'est à dire la direction vers laquelle il se dirige. Dans la figure 4.3, qui représente la structure complète des tables du modèle **Autopilot**, les tables concernées par cette division ont été foncées. Les tables les plus foncées appartiennent à la partie relative à la vitesse, les moins foncées à la partie relative à la direction et entre les deux, les tables relatives à l'altitude. De plus, chaque table logique est nommée explicitement en fonction de celle des trois parties à laquelle elle appartient.

4.2 Traduction d'une table

Afin d'illustrer le modèle **Autopilot** et une partie de son fonctionnement, nous allons tenter d'en extraire une table en particulier, la table *airspeedTargetSystemTable*, et d'obtenir de celle-ci son modèle mental. Pour ce faire, une partie de cette table va être isolée du reste du modèle de la même manière que la méthode décrite à la page 229 de [2]. Cette table est représentée dans la figure 4.4.

	0	1	6
L airspeedTargetSystemTable			
INPUTS			
L airspeedInterfaceActionTable.outputState			
user rotates Airspeed selector knob clockwise	•		
user rotates Airspeed selector knob counterclockwise		•	
no action			•
V selectedSpeedTarget			
< 255	•		
> 245		•	
OUTPUTS			
L airspeedTargetSystemTable.outputState			
increase IAS airspeed target	•		
decrease IAS airspeed target		•	
V selectedSpeedTarget			
++	•		
–		•	
V airspeedTarget			
V selectedSpeedTarget	•	•	

FIGURE 4.4 – Une version réduite de la table airspeedTargetSystemTable issue du modèle **Autopilot**.

4.2.1 Fichier de configuration

Pour obtenir une telle table, il faudra bien évidemment manipuler le fichier de configuration lié au modèle **Autopilot**. Donc la première étape sera de générer ce fichier de configuration et cela en mode "false" afin que rien ne soit utilisé. En effet, le modèle Autopilot est tellement vaste qu'il est bien plus aisé et rapide de renseigner ce que l'on a besoin plutôt que l'inverse. Ensuite, il faut indiquer dans ce fichier les tables que l'on souhaite

utiliser ainsi que les colonnes de celles-ci. Pour recréer la situation voulue, il va en fait falloir utiliser six tables du modèle et non une seule. Cette contrainte découle directement de la limitation du principe des fichiers de configuration présentée dans 3.6.4 relative à l'utilisation de la table *topLogicTable*, et du fait que le modèle **Autopilot** soit sous la forme d'une structure ASF.

- *airpseedTargetSystemTable* – La première table à utiliser est évidemment la table que l'on souhaite extraire. De plus, il faut utiliser ses colonnes 0, 2 et 6 si l'on veut obtenir la même table que celle présentée à la figure 4.4.
- *topLogicTable* – La nécessité d'utiliser cette table et sa colonne 0 découle directement de la limitation présentée dans la section 3.6.4. En effet, c'est elle qui fait appel aux tables *systemTable* et *actionTable*.
- *systemTable* – Cette table et sa colonne 0 sont utilisées car elles permettent d'accéder à la table *airspeedTargetSystemTable*.
- *actionTable* – Cette table et sa colonne 0 sont utilisées car elles permettent d'accéder à la table *autoflightPanelInterfaceActionTable*.
- *autoflightPanelInterfaceActionTable* – Tout comme les deux tables précédentes, il faut utiliser cette table et sa colonne 0 car cette dernière permet l'accès à la table *airspeedInterfaceActionTable*
- *airspeedInterfaceActionTable* – Cette dernière et sixième table ainsi que ses colonnes 1, 2 et 4 sont nécessaires car c'est sa valeur d'output qui est utilisée comme input de la table *airspeedTargetSystemTable*. Elle sert dès lors à apporter une variation dans cet input.

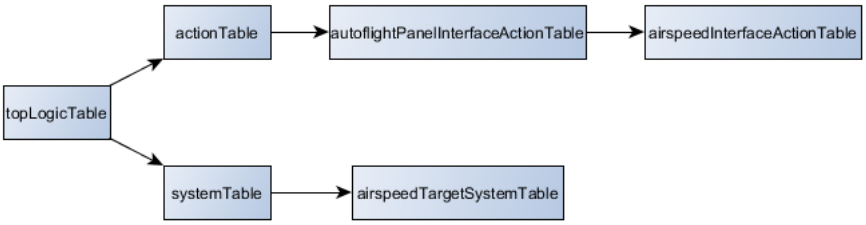


FIGURE 4.5 – Représentation des appels entre les tables.

Le fil de déclenchement des tables est représenté à la figure 4.5. Une fois que les tables à utiliser ont été indiquées, il faut indiquer les variables et les outputs dont l'on souhaite se servir pour caractériser les états de l'automate que l'on obtiendra. Dans ce cas-ci, il y aura deux variables et un seul output à prendre en compte, comme on peut le voir dans la table située à la figure 4.4.

- selectedSpeedTarget – Cette première variable doit être limitée entre 248 et 252 et ne pas être observable afin de respecter ce qui a été fait dans [2] et pour obtenir les même résultats.

```
1 <Var used="true" observable="false" max="252" min="248">selectedSpeedTarget</Var>
```

- airspeedTarget – De la même manière que la première variable, cette variable doit également être non observable et limitée entre 248 et 252.

```
<Var used="true" observable="false" max="252" min="248">airspeedTarget</Var>
```

- airspeedTargetSystemTable.outputState – Cet output est utilisé car il est bien présent dans la table et est de plus utilisé comme variable pour tester l'égalité de deux états. Tous comme les variables, celui-ci est non observable.

```
<Out used="true" observable="false">airspeedTargetSystemTableOutputs</Out>
```

Finalement, il faut indiquer dans le fichier de configuration les actions que l'on souhaite utiliser comme transitions dans l'automate généré. Dans [2], on utilise les outputs de la table *airspeedInterfaceActionTable* comme transitions, ce qu'il n'est pas possible de faire avec un fichier de configuration, comme expliqué dans la section 3.6.4. Cependant, si l'on cherche un peu dans cette table, on se rend compte que les outputs dont on a besoin sont directement liés a deux actions du modèle ADEPT. De plus, ces actions sont chacune liées à une colonne de la table logique, ce qui explique pourquoi il a fallu indiquer auparavant qu'on souhaitait les utiliser. Cette liaison entre les actions et l'output dont on a besoin sont une illustration du fonctionnement d'un modèle structuré sous la forme ASF.

- airspeedTargetIncreaseTimer.actionPerformed – Cette action est directement liée à l'output "user rotates Airspeed selector knob clockwise". En effet, la colonne 1 de la table *airspeedInterfaceActionTable*, prend comme unique input cette action et donne ce dernier comme unique output, ce qui permet de modifier les inputs de la table *airspeedTargetSystemTable*. De plus, même si normalement cette action est liée à un timer et est donc une observation, il est nécessaire de l'indiquer comme étant une commande car dans [2], on considère que les actions prises comme input sont justement des commandes.

```
<Act used="true" type="COMMAND">airspeedTargetIncreaseTimer.actionPerformed</Act>
```

- airspeedTargetDecreaseTimer.actionPerformed – Cette action est directement liée à l'output "user rotates Airspeed selector knob counterclockwise". En effet, la colonne 4 de la table *airspeedInterfaceActionTable*, prend comme unique input cette action et donne ce dernier comme unique output, ce qui permet, comme pour l'action précédente, de modifier les inputs de la table *airspeedTargetSystemTable*. De plus, pour les même raisons que la première action, il faut indiquer cette action comme étant une commande.

```
<Act used="true" type="COMMAND">airspeedTargetDecreaseTimer.actionPerformed</Act>
```

4.2.2 Résultats

Une fois le fichier de configuration complété comme ci dessus, on peut alors extraire le HMI-LTS lié à la table logique *airspeedTargetSystemTable* simplifiée du modèle **Autopilot**. Pour vérifier l'exactitude et le bon fonctionnement du fichier de configuration, il est utile de jeter un oeil aux fichiers texte générés se trouvant dans l'annexe G. Le fichier des états contient 9 états et le fichier des transitions contient 27 transitions, dont 9 transitions "tau". L'automate décrit dans ces fichiers a ensuite été mis sous forme de diagramme pour obtenir la représentation donnée à la figure 4.6. Dans celle-ci, le symbole "++" représente l'action airspeedTargetIncreaseTimer.actionPerformed, le symbole "--" représente l'action airspeedTargetDecreaseTimer.actionPerformed, "out" désigne l'output de la table

airspeedTargetSystemTable et finalement, “spd” désigne la variable selectedSpeedTarget. Remarquez que les 11 transitions ayant un même état comme point de départ et d’arrivée n’ont pas été reprises. De plus, notez que les numéros des états et des transitions sont ceux qui leurs ont été attribués dans les fichiers texte décrivant l’automate qui sont disponibles à l’annexe G. Cette représentation, lorsqu’elle est comparée à celle donnée dans [2] nous donne la confirmation que le fichier de configuration créé est correct et fonctionnel.

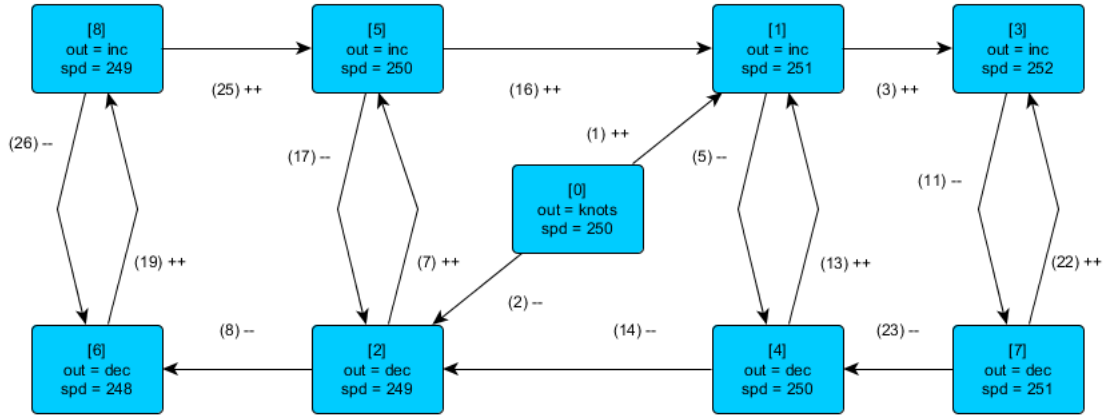


FIGURE 4.6 – Représentation du modèle système lié à la table airspeedTargetSystemTable (4.4).

La différence obtenue dans les résultats au niveau des transitions du HMI-LTS généré s’explique facilement lorsque l’on est attentif au contenu du fichier Java qui avait été écrit manuellement pour générer les résultats obtenus dans [2]. Ce fichier Java est disponible dans le répertoire “Archives/first-syst-abstr-1obs” de l’annexe CD-ROM. La raison de cette différence provient en fait de l’utilisation de deux variables appelées “actionPassed” et “noActionPassed”. Ces variables sont utilisées lors de la simulation d’un modèle pour savoir si l’action en cours s’est réellement déroulée, c’est à dire si elle a satisfait l’une des conditions d’une des tables traversées. Or, dans le cas d’un modèle structuré sous la forme ASF, une action satisfait toujours au moins une condition, celle liée à cette même action dans les tables relatives aux actions. Une action sera donc considérée comme s’étant vraiment produite, même si celle-ci n’aura fait que modifier l’état d’output de la table *airspeedInterfaceActionTable* et qu’elle n’aura satisfait aucune condition dans la table *airspeedTargetSystemTable*. Au contraire, dans le fichier Java généré manuellement, les actions sont testées directement dans la table *airspeedTargetSystemTable*, ce qui signifie qu’une action sera considérée comme ayant réellement eu lieu seulement si elle satisfait à l’une des conditions de cette table. Cette différence est à la base du fait que la version générée automatiquement donne deux transitions supplémentaires que sont les transitions 10 et 20. De plus, on peut s’apercevoir que la variable “noActionPassed” qui est censée être mise à true lorsque l’action qui satisfait la condition est l’action “no action”, a simplement été oubliée dans le fichier généré manuellement. Cette deuxième différence explique les neuf transitions liées à des “tau” présentes dans le fichier généré de manière automatique et pas dans celui généré manuellement.

Ensuite, toujours pour continuer le parallélisme avec ce qui a déjà été fait dans [2], une transformation a été effectuée sur le modèle afin d’en obtenir le modèle mental minimal,

dont la représentation se trouve à la figure 4.7. Les noms donnés aux états sont ceux qui ont été obtenus par une transformation selon la technique de bismulation. De plus, ce processus a été répété lorsque la variable selectedSpeedTarget était observable, ce qui a donné la représentation visible à la figure 4.8. De nouveau, ces résultats concordent parfaitement avec ceux présentés à la page 231 de [2].

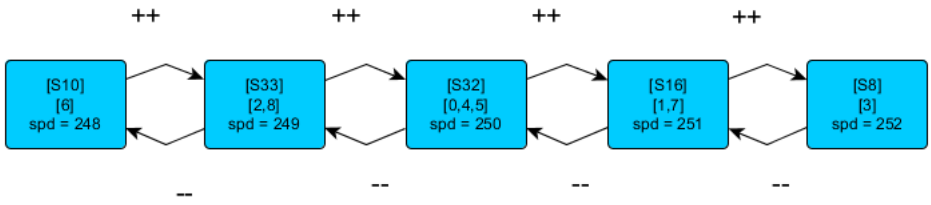


FIGURE 4.7 – Représentation du modèle mental lié à la table airspeedTargetSystemTable (4.4).

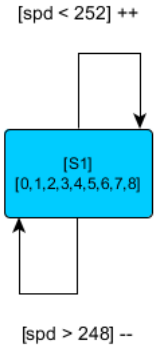


FIGURE 4.8 – Représentation du modèle mental lié à la table airspeedTargetSystemTable (4.4) lorsque la variable selectedSpeedTarget est observable.

Cet exemple a permis dans un sens d'illustrer une petite partie du fonctionnement du modèle **Autopilot**, mais il a surtout servi à illustrer la manière de procéder pour extraire une table d'un modèle grâce aux fichiers de configuration. De plus, cette expérience apporte un élément de preuve du bon fonctionnement de l'outil ADEPT2LTS, et de l'exactitude des résultats obtenus avec celui-ci. Remarquez que des évaluations de l'outil similaires à celle qui vient d'être faite sont effectuées dans la section 6.

Chapitre 5

Manuel d'utilisation

L'outil `jpf-hmi` a été légèrement retravaillé afin de permettre à l'utilisateur de réaliser les transformations de modèles de son choix au moyen de divers lignes de commandes. En effet, initialement, l'utilisateur devait modifier le code source de l'outil afin de transformer le modèle qu'il souhaitait. Dès à présent, grâce à la classe *JpfHmi*, l'utilisateur peut facilement effectuer toutes les sortes de transformations qu'il souhaite, que ce soit à partir d'un fichier XML représentant un modèle ADEPT (.sgb) ou un fichier texte représentant un automate (.lts).

Tout d'abord, pour pouvoir utiliser l'outil `jpf-hmi`, il faut indiquer dans la variable d'environnement CLASSPATH le chemin vers la classe principale de l'outil (jpf-hmi\build\main) ainsi que le chemin vers la librairie jargs (jpf-hmi\jargs.jar). Ensuite, il suffit de lancer la commande "java JpfHmi nom_du_modèle.extension" pour transformer le modèle passé en paramètre. Remarquez la nécessité de spécifier l'extension du fichier étant donné que l'outil se base sur cette extension pour déterminer s'il doit transformer le modèle avec la classe *ADEPT2LTS* ou la classe *LTSLoader*. Si l'utilisateur souhaite transformer un modèle ADEPT, il doit veiller tout d'abord à générer un fichier de configuration associé à ce modèle grâce aux options. L'outil `jpf-hmi` fournit désormais à l'utilisateur diverses options relatives à la transformation d'un modèle :

debug Cette option permet à l'utilisateur de générer ou non les fichiers texte présentés dans la section 3.5 et décrivant le HMI-LTS obtenu après transformation du modèle ADEPT. Utilisation : `[(--debug, -d)]`

verbose Cette option permet à l'utilisateur de visualiser le déroulement de toutes les étapes du processus de traduction. Utilisation : `[(--verbose, -v)]`

config Cette option, associée à un fichier, permet de spécifier le fichier de configuration que l'outil ADEPT2LTS doit utiliser lors de la traduction du modèle ADEPT vers le HMI-LTS équivalent. Si cette option n'est pas spécifiée, le fichier de configuration utilisé par défaut est celui portant le même nom que le modèle à traduire. Utilisation : `[(--config, -c) file_name]`

bisim Cette option permet à l'utilisateur d'indiquer à l'outil qu'il souhaite effectuer une transformation du HMI-LTS ou modèle système obtenu vers son modèle mental par la méthode de bisimulation (réduction). Le résultat de cette transformation est stocké dans un fichier texte portant le nom du fichier de configuration associé à "bisim" et ayant une extension ".mental" (fichierConfigurationbisim.mental). Utilisation : `[(--bisim)]`

learn Cette option permet à l'utilisateur d'indiquer à l'outil qu'il souhaite effectuer une transformation du HMI-LTS ou modèle système obtenu vers son modèle mental par la méthode de learning. Le résultat de cette transformation est stocké dans un fichier texte portant le nom du fichier de configuration associé à "learn" et ayant une extension ".mental" (fichierConfigurationlearn.mental) Utilisation : `[(--learn)]`

tdfa Cette option permet à l'utilisateur d'indiquer à l'outil qu'il souhaite effectuer une transformation du HMI-LTS ou modèle système obtenu vers son modèle mental par la méthode de . Le résultat de cette transformation est stocké dans un fichier texte portant le nom du fichier de configuration associé à "tdfa" et ayant une extension ".mental" (fichierConfigurationtdfa.mental) Utilisation : `[(--tdfa)]`

configGen Cette option, associée à un nom de fichier, permet à l'utilisateur de générer un fichier de configuration lié au modèle passé en paramètre. Le fichier généré portera le nom associé à l'option. Utilisation : `[(--configGen) file_name]`

configMod Cette option, associée à "true" ou "false", permet de définir le mode dans lequel l'utilisateur souhaite générer le fichier de configuration, comme celà a été présenté à la section 3.6. Lors d'une telle génération, si cette option n'est pas définie, le mode sera mis à "true" par défaut. Utilisation : `[(--configMod) true|false]`

Voici un petit exemple de l'utilisation de ces commandes sur le modèle **Light**. La première ligne va donc générer un fichier de configuration nommé LightConf.conf en mode "true". La seconde, quant à elle, va effectuer les trois types de transformations sur le HMI-LTS issu du modèle **Light** associé avec le fichier de configuration LightConf et montrer tout le processus vu que l'option verbose (-v) est présente. Cette commande va donc générer trois fichiers texte représentant les modèles mentaux obtenus : LightConfbisim.mental, LightConflearn.mental et LightConftdfa.mental. De plus, vu que la commande contient l'option debug (-d), cinq fichiers texte supplémentaires vont être générés, représentant le modèle système obtenu : LightConfstate.txt, LightConftransition.txt, etc. Notez que les deux lignes auraient pu être regroupées en une et donner le même résultat mais que celles-ci ont été séparées pour des raisons de lisibilité et de clarté.

```
1 $java JpfHmi --configGen=LightConf --configMod=true Light.sgb
2 $java JpfHmi -d -v --config=LightConf --bisim --tdfa --learn Light.sgb
```

Ces options et le nom des fichiers générés ont été conçus de sorte à permettre à l'utilisateur d'effectuer plusieurs opérations en une fois grâce à des scripts. En effet, le fait de nommer les fichiers contenant les résultats d'une opération avec le nom du fichier de configuration avec lequel ils ont été obtenus, permet de les lier à ce fichier de configuration et non au modèle utilisé. Il est en effet pratique de pouvoir travailler avec plusieurs fichiers de configuration en même temps sur le même modèle, tout en distinguant les différents résultats obtenus.

Chapitre 6

Evaluation

De manière similaire à l'exemple de la traduction de la table *airspeedTargetSystemTable* du modèle **Autopilot** présentée dans la section 4.2, ce chapitre illustre l'efficacité et le bon fonctionnement de l'outil ADEPT2LTS en fournissant l'évaluation du processus de traduction sur plusieurs modèles. Le premier d'entre eux est le modèle **F/D** (6.1) ou dit du "FlightDirector" qui est en fait une petite partie du modèle **Autopilot**. Le second exemple illustre le modèle **VTS** (6.2) ou dit du "Vehicle Transmission System" (6.4). Ensuite, on retrouve l'exemple du modèle **CountDown** (6.3), suivi du modèle **VCR** ou dit du "Video Cassette Recorder". Pour finir, l'exemple complexe du modèle **Autopilot** simplifié a également été illustré (6.5). Les résultats complets de chacune de ces évaluations se trouvent dans le répertoire "Modeles" de l'annexe CD-ROM.

6.1 Exemple du modèle F/D

Le modèle **F/D** est donc une petite partie très simple du modèle **Autopilot**, dont le comportement est décrit par deux tables : *captainsFdSystemTable* (fig. 6.1) et *firstOfficersFdSystemTable* (fig. 6.2). Cependant, comme pour l'exemple d'extraction d'une table donné à la section 4.2, il faut trouver et utiliser la table contenant les actions qui permettent de modifier les inputs des deux tables du modèle **F/D**. Ces actions sont mcpCaptainsFdSwitch.mouseClicked et mcpFoFdSwitch.mouseClicked qui sont contenues dans la table *apFdAtInterfaceActionTable*.

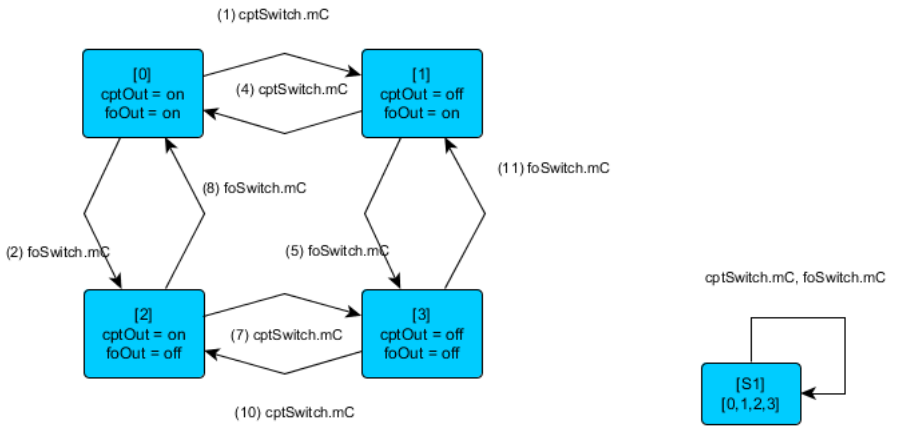
	0	1	2
L captainsFdSystemTable			
INPUTS			
L apFdAtInterfaceActionTable.outputState			
no action			•
user toggles captains fd switch	•	•	
L captainsFdSystemTable.outputState			
captains FD on	•		
captains FD off		•	
OUTPUTS			
L captainsFdSystemTable.outputState			
captains FD off	•		
captains FD on		•	

FIGURE 6.1 – La table *captainsFdSystemTable* issue du modèle **Autopilot**.

	0	1	2
L firstOfficersFdSystemTable			
INPUTS			
L apFdAtInterfaceActionTable.outputState			
no action			•
user toggles first officer fd switch	•	•	
L firstOfficersFdSystemTable.outputState			
first officers FD on	•		
first officers FD off		•	
OUTPUTS			
L firstOfficersFdSystemTable.outputState			
first officers FD off	•		
first officers FD on		•	

FIGURE 6.2 – La table *firstOfficersFdSystemTable* issue du modèle **Autopilot**.

La traduction de ce modèle grâce à l'outil ADEPT2LTS donne un HMI-LTS composé de 4 états et 12 transitions, dont la représentation se trouve à la figure 6.3a. Dans cette représentation, "cptSwitch.mC" représente l'action mcpCaptainsFdSwitch.mouseClicked, "foSwitch.mC" représente l'action mcpFoFdSwitch.mouseClicked, "cptOut" représente l'output de la table captainsFdSystemTable et finalement "foOut" représente l'output de la table firstOfficersFdSystemTable. Notez que seules les transitions ayant des états de départ et d'arrivée différents ont été représentées. Ensuite, la transformation du modèle par bisimulation mène au modèle mental représenté dans la figure 6.3b. Les résultats d'une traduction "manuelle" de ce modèle ont déjà été présentés dans [2], à la page 226, ce qui permet de juger de la validité des modèles système et mental obtenus avec la nouvelle version de `jpf-hmi`. Les résultats complets de ces transformations sont repris dans l'annexe H.



(a) Modèle système. (b) Modèle mental.

FIGURE 6.3 – Représentation du HMI-LTS lié au modèle **F/D**.

6.2 Exemple du modèle VTS

Ce modèle est bien connu car il est présenté dans divers articles notamment [2] et [4] dont il a été tiré. La première étape pour pouvoir tester la nouvelle version du `jpf-hmi` sur ce modèle était de mettre ce système sous forme d'un modèle ADEPT exploitable par ADEPT2LTS. La table logique qui a été construite via l'outil ADEPT pour représenter ce système est composée de seize colonnes et de huit différents états d'output. Une représentation de celle-ci se trouve à la figure 6.4.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L topLogicTable																
INPUTS																
ACTIONS																
pushUp.mouseClicked	•	•	•	•												
pullDown.mouseClicked					•	•										
upTimer.actionPerformed							•	•	•	•	•					
downTimer.actionPerformed												•	•	•	•	•
L topLogicTable.outputState																
Low1	•						•									
Low2	•							•				•				
Low3		•											•			
Medium1			•		•				•							
Medium2				•	•									•		
High1						•				•						
High2						•					•				•	
High3						•										•
OUTPUTS																
L topLogicTable.outputState																
Low1												•				
Low2							•						•			
Low3					•			•								
Medium1	•													•		
Medium2						•			•							
High1		•	•												•	
High2				•						•						•
High3											•					

FIGURE 6.4 – La table *topLogicTable* venant du modèle **VTS**.

Une fois le modèle **VTS** obtenu, celui-ci a été traduit par l'outil ADEPT2LTS afin d'en extraire un HMI-LTS représentant son comportement. L'automate ainsi obtenu se compose de 8 états et 20 transitions et est représenté dans la figure 6.5. Dans cette représentation, "push" fait référence à l'action pushUp.mouseClicked, "pull" à l'action pullDown.mouseClicked, "up" à upTimer.actionPerformed et "down" à downTimer.actionPerformed. Dès lors il était possible d'obtenir le modèle mental lié au **VTS**, ce qui a donné un automate à 5 états et 14 transitions représenté à la figure 6.6. Une fois de plus, les résultats obtenus grâce à l'outil ADEPT2LTS sont les mêmes que les résultats obtenus manuellement qui sont présentés dans [2] à la page 188 et dans [4]. Les résultats complets de cette évaluation se trouvent à l'annexe I.

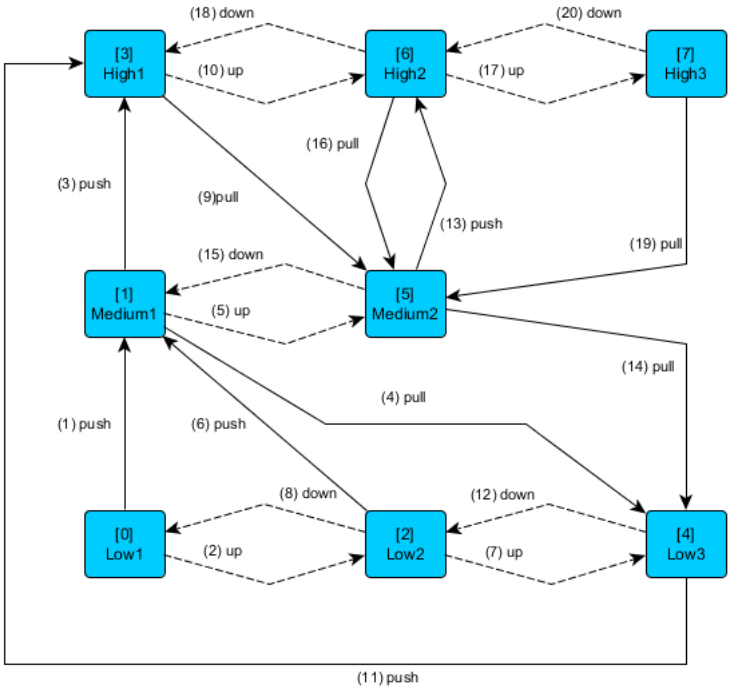


FIGURE 6.5 – Représentation du modèle système du Vehicle Transmission System

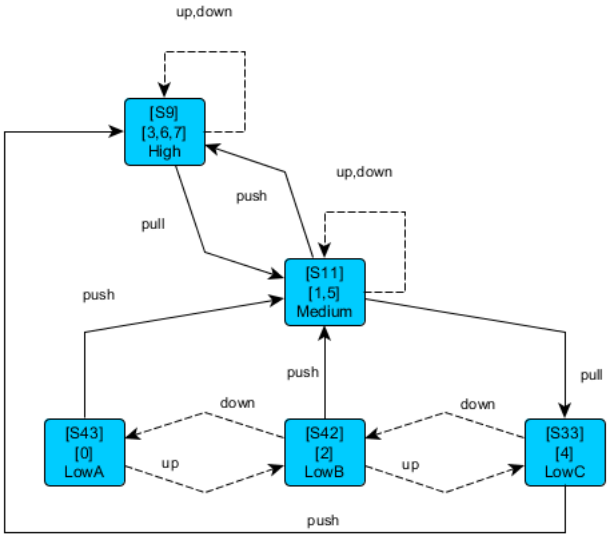


FIGURE 6.6 – Représentation du modèle mental du Vehicle Transmission System

6.3 Exemple du modèle CountDown

Le modèle **CountDown** a été tiré de la page 75 de [2]. Ce modèle représente un simple compte à rebours pouvant monter jusqu'à deux avant d'être lancé. Avant de pouvoir évaluer ADEPT2LTS sur ce modèle, il a d'abord fallu le modéliser sous forme d'un modèle ADEPT afin d'en obtenir un fichier sgb. Le résultat de cette modélisation est représenté dans la table logique donnée à la figure 6.7.

	0	1	2	3	4	5	6	7
L topLogicTable								
INPUTS								
ACTIONS								
Inc.mouseClicked	•	•						
Reset.mouseClicked			•					
Cancel.mouseClicked					•			
Start.mouseClicked					•			
Ring.mouseClicked								•
V value								
0	•							
1	•		•	•	•		•	
2		•	•	•		•		
V alarm								
True								•
False	•	•	•	•	•	•	•	
V running								
True					•	•	•	
False	•	•	•	•				•
OUTPUTS								
V value								
0		•	•		•		•	
++	•							
–						•		
V alarm								
True							•	
False								•
V running								
True				•				
False					•		•	

FIGURE 6.7 – La table *topLogicTable* issue du modèle **CountDown**.

Dès lors qu'une version du système sous forme d'un modèle ADEPT existait, il était possible d'obtenir son modèle système grâce à ADEPT2LTS. Cette traduction donne comme résultat un automate composé de 6 états et 11 transitions, dont la représentation est donnée dans la figure 6.8. Ces résultats sont les mêmes que ceux obtenus par la traduction manuelle effectuée dans [2]. Le modèle mental obtenu de ce modèle système par la technique de learning est représenté dans la figure 6.9. Les résultats complets de cette évaluation sont présentés dans l'annexe J.

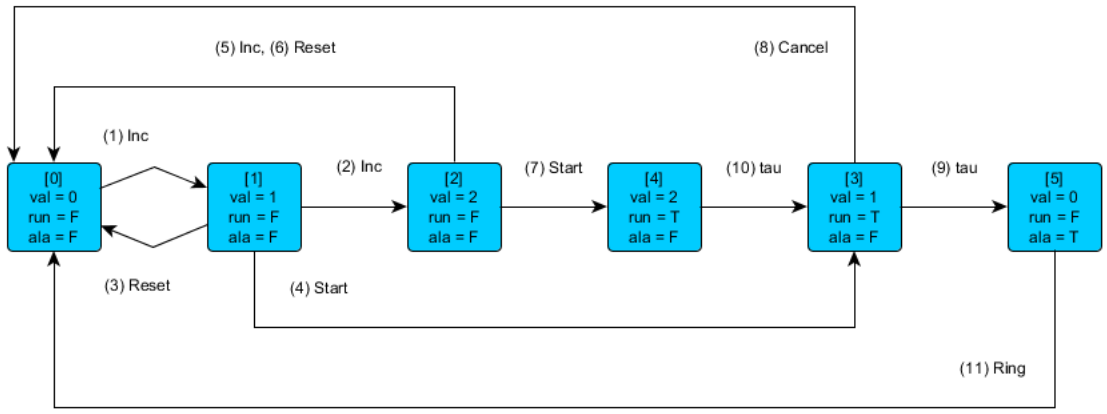


FIGURE 6.8 – Représentation du modèle système du **CountDown**.

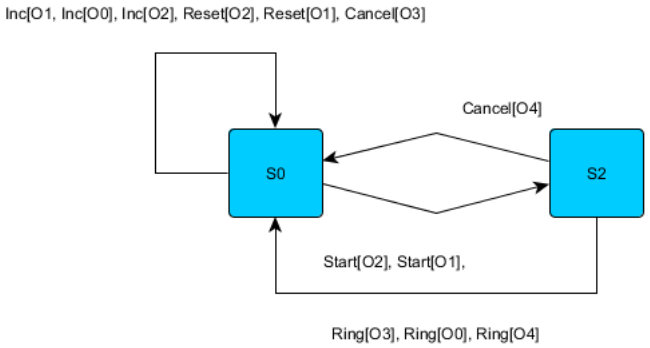


FIGURE 6.9 – Représentation du modèle mental du **CountDown**.

6.4 Exemple du modèle VCR

Le modèle du "Video Cassette Recorder" dit **VCR** est un modèle présenté dans [2] à la page 220. Cependant, la transformation qui va être réalisée ici est différente de celle réalisée manuellement dans [2] qui porte sur une version modifiée et réduite du **VCR**. La base du modèle est cependant la même et possède une seule table logique qui est la table *topLogicTable* présentée à la figure 6.10. Lorsque le modèle est traduit en un HMI-LTS, on obtient un automate composé de 3134 états et 25072 transitions. Le modèle mental correspondant obtenu comporte quant à lui 1 seul état et 8 transitions, lorsque aucune variable n'est observable.

Ces résultats sont très similaires à ceux du fichier "Archives/vcrDemoSingle.lts" fournit en annexe CD-ROM qui contient 3135 états et 25073 transitions. Ce fichier est en fait le résultat de la traduction manuelle du modèle **VCR** réalisée par Sébastien Combéfis. Cependant, si l'on observe les résultats qu'il a obtenu, on s'aperçoit que l'état initial de son HMI-LTS, que l'on peut apercevoir à la ligne 2 du fichier, est en fait un état erroné. En effet, il possède une variable "powerStatusLabel_background" à laquelle aucune valeur n'est assignée, or elle devrait être de "0,0,0", qui est la valeur initiale de cette variable, comme on peut le voir sur le fragment de code sgb décrivant le composant "powerStatusLabel" ci-dessous.

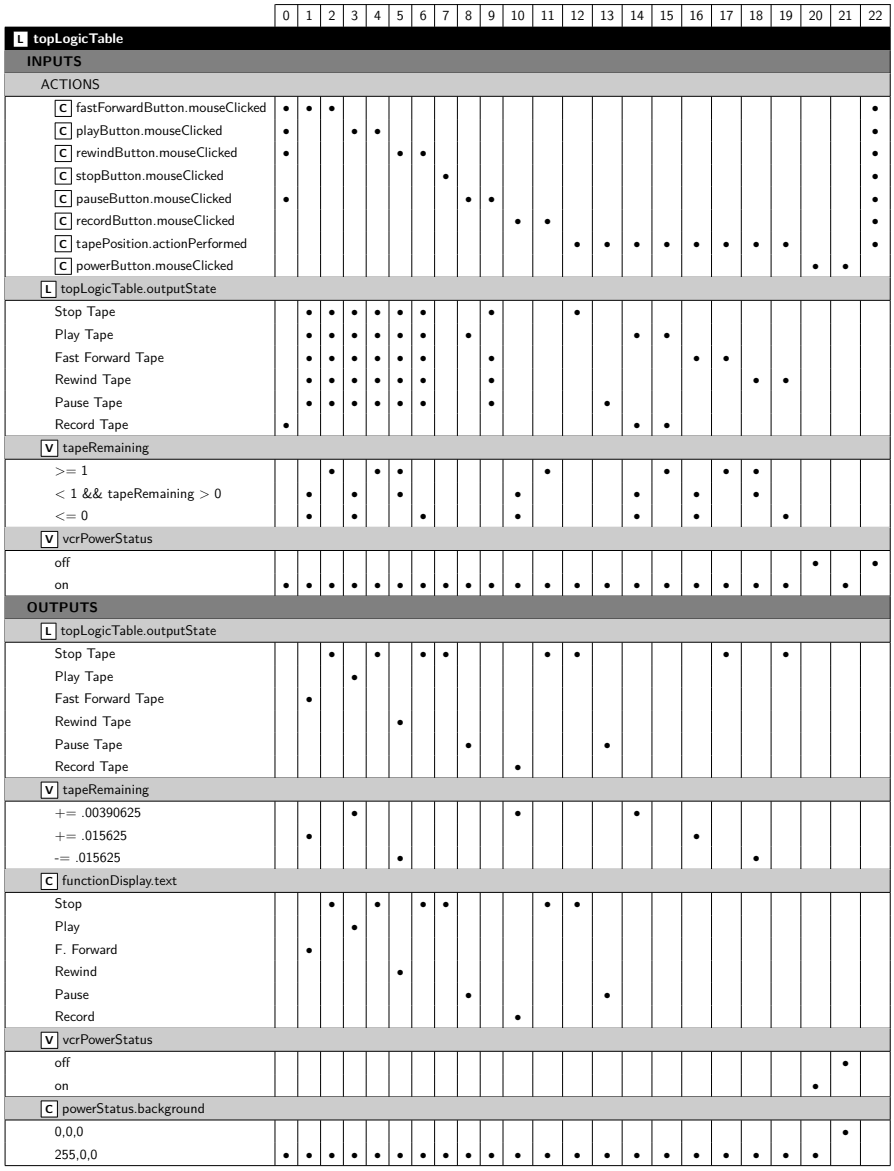


FIGURE 6.10 – La table *topLogicTable* du modèle **VCR**.

```
1 <Node location="19, 80" beanClass="javax.swing.JLabel" foreground="0,0,0" visible="True"
      verticalAlignment="CENTER" verticalTextPosition="CENTER" opaque="True" enabled="True" y="80" x=
      "19" text="" class="gov.nasa.arc.sgbe.remote.nodes.BeanNode" font="Dialog-PLAIN-11" size="71, 5
      " height="5" width="71" horizontalAlignment="LEADING" toolTipText="null" horizontalTextPosition
      ="TRAILING" \textbf{background="0,0,0"} componentPostion="0" name="powerStatusLabel"/>
```

En fait, le bon état initial de son HMI-LTS est l'état présent à la ligne 1266, qui est le même que l'état erroné, mis à part que la variable est bien initialisée dans celui-la. L'état présent à la deuxième ligne du fichier est donc l'état qui faisait penser à une erreur dans la traduction automatique. En tenant compte de cette erreur, on se rend compte alors que la traduction du modèle **VCR** par l'outil ADEPT2LTS donne de bons résultats en terme de nombre d'états et de transitions et peut être considérée comme une réussite. Les résultats complets de la transformation du **VCR** se trouve dans le répertoire "Modeles/VCR" de l'annexe CD-ROM.

6.5 Exemple du modèle Autopilot simplifié

Dans cet exemple, la traduction est effectuée sur une version simplifiée du modèle **Autopilot**. En effet, le comportement de cette version est uniquement décrit par les tables logiques liées à la vitesse de l'avion (airspeed) et à sa direction (lateral). Les tables étant utilisées sont donc *airspeedSystemTable*, *airspeedSystemTargetTable*, *lateralSystemTable* et *lateralTargetSystemTable*. En plus de ces tables, il a fallu ajouter les tables permettant de déclencher les inputs des celles-ci, c'est à dire les tables *airspeedInterfaceActionTable* et *lateramInterfaceActionTable*, tout comme cela avait été nécessaire dans l'exemple donné à la section 4.2. Une description plus complète de ce sous-modèle est donnée dans [2] à la page 232.

La transformation de cette version du modèle **Autopilot** donne 15 016 états et 165 176 transitions. Lorsque l'on compare ces résultats avec ceux obtenus manuellement dans [2], on s'aperçoit qu'ils sont complètement différents. Ce n'est pas pour autant que l'outil ADEPT2LTS ne fonctionne par correctement. En effet, dans ce cas-ci, la raison de cette grande différence se trouve dans l'interprétation de l'action "no action", signifiant justement qu'aucune action ne se produit.

Si l'on est attentif au fichier Java contenant la traduction manuelle du modèle, qui est disponible dans le répertoire "Archives/second-syst-abstr-1obs" de l'annexe CD-ROM, on peut s'apercevoir que l'auteur a mélangé les différentes interprétations de "no action". Par exemple, si l'on jette un coup oeil au bout de code ci-dessous, représentant les ligne 937 à 939 du fichier Java, on peut apercevoir une condition représentant les inputs de la colonne 8 de la table *lateralTargetSystemTable* (figure 6.11). Comme on peut le voir, l'auteur du code utilise directement les outputs de la table *lateralInterfaceActionTable* comme actions, ce qui permet d'éviter de devoir faire appel à cette table mais qui n'est pas possible de faire avec les fichiers de configurations (voir section 3.6.4). Cependant, cela ne change rien au problème. En effet, cette condition est la conséquence d'une autre interprétation de l'action "no action". L'auteur l'interprète ici comme signifiant que n'importe quelle action en cours du modèle pourrait satisfaire pour se trouver dans la situation 8, sauf les deux qui sont mises dans la condition. Or, "no action" signifie qu'il ne doit y avoir aucune action en cours pour satisfaire à cette situation. Ce qui est étrange c'est que dans le fichier Java, on retrouve tantôt l'une des interprétations, tantôt l'autre.

```
1 // Scenario 8
2       //else if (action == Action.NO_ACTION)
3       else if (action != Action.USER_ROTATES_LATERAL_TARGET_SELECTOR_KNOB_CLOCKWISE
4               && action != Action.USER_ROTATES_LATERAL_TARGET_SELECTOR_KNOB_COUNTERCLOCKWISE)
```

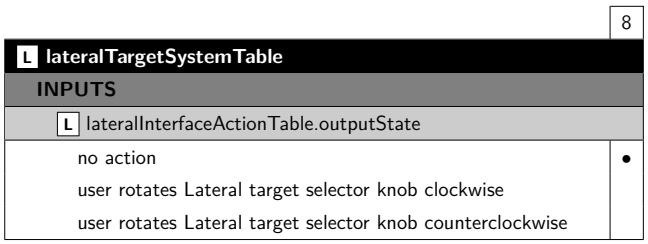


FIGURE 6.11 – Début de la colonne 8 de la table lateralTargetSystemTable.

6.6 Performances de l'outil

Cette section contient une tentative d'évaluation de l'outil ADEPT2LTS en terme de performances temporelles. Les performances de l'outil vont être comparée par rapport au performance totale de l'outil `jpf-hmi`. Les résultats obtenus en terme de temps sont repris dans la figure 6.12. Dans celle-ci, la colonne "# Etats" représente le nombre d'états composant l'automate issu du modèle et la colonne "# Noeuds" représente le nombre de noeuds que contient le fichier sgb de ce modèle. Ensuite la colonne "Trad." représente la phase de transformation d'un fichier sgb, c'est-à-dire la lecture de celui-ci et l'écriture du fichier Java lui correspondant. Pour rappel, cette phase est gérée par l'outil Translator et est présentée dans la section 3.1. La colonne "Sim." représente quant à elle la phase de simulation du modèle et de génération du HMI-LTS le décrivant, qui est présentée dans la section 3.4. Finalement, la colonne "Trans." représente le processus appliqué à un modèle système afin d'en extraire un modèle mental par une technique de bisimulation. Les résultats obtenus proviennent d'une moyenne de dix transformations successives, sauf dans le cas du modèle **Autopilot** pour lequel une seule a été réalisée.

	# Etats	# Noeuds	Trad. (ms)	Sim. (ms)	Trans. (ms)
F/D	4	2566	861	35	16
VTS	8	89	105	37	34
CountDown	6	96	108	42	31
VCR	3134	162	178	44 s	2 s
Autopilot	15 016	2566	964	50 m	15 s

FIGURE 6.12 – Performances de l'outil JPF-HMI.

Lorsque l'on analyse ces résultats, on peut tout d'abord se rendre compte que le temps nécessaire à la transformation d'un fichier sgb en un fichier Java semble dépendre principalement à la taille de ce fichier sgb, et ce de manière linéaire. En effet, plus le nombre de noeuds grandit, plus le temps nécessaire à la traduction semble élevé, mais sans que cela "explose". De plus, même si le modèle **F/D** et le modèle **Autopilot** ont des états différents à cause de leur fichier de configuration, ils se basent sur le même fichier sgb, ce qui explique leur similarité en terme de temps de traduction. Cependant, on peut voir que le fichier de configuration influence également légèrement ce temps, vu que le temps pour le modèle **Autopilot** est tout de même plus élevé.

Ensuite, en terme de simulation, on peut voir que le temps explose littéralement quand le nombre d'états augmente, qu'il semble évoluer de manière exponentielle par rapport aux états, ce qui est plutôt logique au vue de son rôle. Par contre, pour la transformation, les temps semblent plutôt évoluer de manière linéaire par rapport au nombre d'états du modèle. Le temps de simulation plus élevé pour le modèle **CountDown** que pour le modèle **VTS**, qui comporte pourtant plus d'états, semble indiquer que le nombre de transitions joue aussi un rôle dans ce résultat, vu que le modèle **CountDown** possède quelques transitions en plus. Finalement, plus un modèle grandit, plus le temps nécessaire à l'extraction du fichier Java permettant la simulation d'un modèle ADEPT semble négligeable par rapport au temps nécessaire à la simulation de celui-ci et à la transformation du modèle système obtenu en un modèle mental.

Chapitre 7

Conclusion

L'ensemble des améliorations qui ont été apportées à l'outil `jpf-hmi` et qui ont été présentées dans ce document se trouvent dans le package "gov.nasa.jpf.hmi.models.util.pierre" du code source de l'outil ainsi que dans les classes *ADEP2LTS* et *JpfHmi*. Le package "pierre" a été nommé comme cela de sorte à distinguer clairement ce qui existait déjà de ce qui a été ajouté à l'outil.

La nouvelle version de `jpf-hmi` est donc dotée d'un traducteur automatique de modèle ADEPT. Par rapport aux traductions manuelles anciennement effectuées, l'outil ADEPT2LTS apporte pas mal d'avantages. Premièrement, et c'est certainement le plus important, les temps obtenus pour les transformations de modèles ADEPT ne dépassent pas la seconde, même pour le modèle **Autopilot**. Manuellement, cela devait prendre des heures pour écrire le fichier Java lié à ce modèle qui compte plus de 1500 lignes. Ensuite, l'automatisation du processus enlève le facteur erreur que peut toujours commettre un humain lors de la traduction. Cette automatisation du processus a également apporté un peu plus de rigueur dans la représentation des modèles en fichier Java. En effet, lors des traductions manuelles, certains éléments des modèles ADEPT pouvaient prendre des formes légèrement différentes une fois traduits en Java. Par contre, le désavantage de l'outil ADEPT2LTS est la flexibilité. En effet, malgré la mise en place d'un fichier de configuration plutôt efficace, la traduction manuelle offrira toujours plus de flexibilité, même dans le respect des normes de traduction établie dans ADEPT2LTS.

Cette nouvelle version de `jpf-hmi` est également accompagnée d'une petite interface facile à utiliser, simple et suffisamment intuitive. Celle-ci permet, au moyen de scripts, d'effectuer plusieurs analyses successives ce qui peut être très pratique dans certains cas. Associés au fichiers de configurations, l'interface est rendue très efficace. En effet, ceux-ci permettent d'évaluer une grande partie des scénarios possibles et imaginables à partir d'un seul modèle ADEPT. Ceux-ci, malgré leur simplicité d'utilisation, se sont montrés très efficaces lors de l'évaluation de l'outil ADEPT2LTS.

L'évaluation de l'outil développé a également montré qu'il était difficile de tester l'efficacité de ce genre d'outil. Pour les petits modèles, on peut être certain de la validité de ADEPT2LTS car les résultats collent avec ceux des traductions manuelles et il sont aisément vérifiables mentalement. Cependant, pour les plus grands modèles, il est impossible de certifier la validité des automates obtenus, tant ceux ci sont vastes. De plus, la validité des automates obtenus par traduction manuelle des grands modèles n'est pas elle même prouvée.

En terme de performances temporelles, la seule partie sur laquelle l'implémentation de l'outil ADEPT2LTS pourrait avoir un impact est celle relative à la traduction d'un modèle en un fichier Java. En effet, les parties liées à la simulation du modèle et à la transformation de l'automate obtenu sont directement issues de ce qui avait déjà été réalisé dans la version initiale de `jpf-hmi`. Or, au vue des résultats obtenus à la section 6.6, le temps nécessaire à la traduction semble élevé par rapport aux temps de simulation et de transformation pour de petits modèles. Cela indique qu'une amélioration en terme de performances temporelles serait certainement possible et bénéfique pour l'outil. Par contre, si l'on regarde les résultats obtenus avec des modèles plus vastes, on s'aperçoit que le processus de traduction semble bien réagir par rapport au nombre de noeuds contenus dans le fichier sgb. Les modèles issus du fichier sgb le plus vaste (Autopilot.sgb) ont même un rapport temps, nombre de noeuds, inférieur aux autres modèles. De plus, au fur et à mesure que le nombre d'états d'un modèle augmente, le temps nécessaire à la traduction devient minime voir négligeable par rapport aux temps obtenus pour les deux autres phases.

Finalement, l'objectif de départ de l'outil ADEPT2LTS, qui était de traduire un modèle ADEPT au format sgb en un automate semble atteint, la validité de celui-ci ayant été prouvée dans la plus grande partie des modèles testés.

Bibliographie

[1] The automation design and evaluation prototyping toolset (adept) users guide version 1.0.19.

[2] Sébastien Combéfis. *A Formal Framework for the Analysis of Human-Machine Interactions*. PhD thesis, Université catholique de Louvain, November 2013.

[3] Sébastien Combéfis, Dimitra Giannakopoulou, Charles Pecheur, and Michael Feary. Learning system abstractions for human operators. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering*, MALETS '11, pages 3–10, New York, NY, USA, 2011. ACM.

[4] Sébastien Combéfis and Charles Pecheur. A bisimulation-based approach to the analysis of human-computer interaction. In *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '09, pages 101–110, New York, NY, USA, 2009. ACM.

[5] Sébastien Combéfis, Dimitra Giannakopoulou, Charles Pecheur, and Michael Feary. A formal framework for design and analysis of human-machine interaction. In *2011 IEEE International Conference on Systems, Man, and Cybernetics*. IEEE, 2011.

[6] Jean-Michel DOUDOUX. Développons en java - 38. dom (document object model). `http://www.jmdoudoux.fr/java/dej/chap-dom.htm`.

[7] Michael Feary. Automatic detection of interaction vulnerabilities in an executable specification. In *Proceedings of the 7th international conference on Engineering psychology and cognitive ergonomics*, EPCE'07, pages 487–496, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] Michael S. Feary. A toolset for supporting iterative human–automation interaction in design. Technical Report 20100012861, NASA Ames Research Center, March 2010.

[9] Dimitra Giannakopoulou, Neha Rungta, and Michael Feary. Automated test case generation for an autopilot requirement prototype. In *SMC*, pages 1825–1830. IEEE, 2011.

[10] Oracle. Javadoc. `http://docs.oracle.com/javase/7/docs/api/`.

Annexe A

Fichier Light.sgb

```
<?xml version="1.0"?>
<!DOCTYPE sgb [
<!ENTITY amp    "&#38;#38;"> <!-- AMPERSAND -->
<!ENTITY quot   "&#x0022;"> <!-- QUOTATION MARK -->
]>
<AdeptProject>
   <DstPanelElement background="255,255,255" name="topContainer" width="300" font="Lucida Grande-
        PLAIN-13" visible="True" toolTipText="null" class="gov.nasa.arc.sgbe.remote.nodes.BeanNode"
        enabled="True" foreground="0,0,0" y="5" x="5" componentPostion="-1" beanClass="javax.swing.
        JLayeredPane" location="5, 5" height="300" size="300, 300"></DstPanelElement>
   <JarFileURLKey>
      <URL url="file:/Users/combefis/Documents/ADEPTworkspace/.metadata/.plugins/gov.arc.nasa.adept/
          SGBE.jar"></URL>
   </JarFileURLKey>
   <MethodsKey></MethodsKey>
   <ComponentsKey>
      <Node nodeReferenceName="topContainer"></Node>
      <Node location="151, 52" beanClass="javax.swing.JLabel" foreground="0,0,0" visible="True"
          verticalAlignment="CENTER" verticalTextPosition="CENTER" opaque="True" enabled="True" y="
          52" x="151" text="" class="gov.nasa.arc.sgbe.remote.nodes.BeanNode" font="Lucida Grande-
          PLAIN-13" size="41, 11" height="11" width="41" horizontalAlignment="LEADING" toolTipText="
          null" horizontalTextPosition="TRAILING" background="255,0,0" componentPostion="1" name="
          powerStatus"></Node>
      <Node location="29, 44" beanClass="javax.swing.JButton" foreground="0,0,0" hideActionText="
          False" rolloverEnabled="False" visible="True" verticalAlignment="CENTER"
          verticalTextPosition="CENTER" opaque="False" enabled="True" y="44" x="29" borderPainted="
          True" text="power" class="gov.nasa.arc.sgbe.remote.nodes.BeanNode" font="Lucida Grande-
          PLAIN-13" size="82, 29" height="29" selected="False" width="82" horizontalAlignment="
          CENTER" toolTipText="null" horizontalTextPosition="TRAILING" background="238,238,238"
          componentPostion="0" name="power"></Node>
      <Node location="29, 83" beanClass="javax.swing.JButton" foreground="0,0,0" hideActionText="
          False" rolloverEnabled="False" visible="True" verticalAlignment="CENTER"
          verticalTextPosition="CENTER" opaque="False" enabled="True" y="83" x="29" borderPainted="
          True" text="fadeOut" class="gov.nasa.arc.sgbe.remote.nodes.BeanNode" font="Lucida Grande-
          PLAIN-13" size="94, 29" height="29" selected="False" width="94" horizontalAlignment="
          CENTER" toolTipText="null" horizontalTextPosition="TRAILING" background="238,238,238"
          componentPostion="2" name="fadeOut"></Node>
   </ComponentsKey>
   <ObjectsKey>
      <Node name="lifeTimer" jarEntry="gov/nasa/arc/beans/TimerBean.class" class="gov.nasa.arc.sgbe.
          remote.nodes.BeanNode" delay="1000" repeats="True" initialDelay="1000" coalesce="True"
          beanClass="gov.nasa.arc.beans.TimerBean" jarURL="/Users/combefis/Documents/ADEPTworkspace
          /.metadata/.plugins/gov.arc.nasa.adept/SGBE.jar"></Node>
      <Node name="fadingTimer" jarEntry="gov/nasa/arc/beans/TimerBean.class" class="gov.nasa.arc.sgbe
          .remote.nodes.BeanNode" delay="0" repeats="True" initialDelay="2000" coalesce="True"
          beanClass="gov.nasa.arc.beans.TimerBean" jarURL="/Users/combefis/Documents/ADEPTworkspace
          /.metadata/.plugins/gov.arc.nasa.adept/SGBE.jar"></Node>
   </ObjectsKey>
   <PropertiesKey>
      <Initial_Values>
```

```
24          <Node name="state" propertyType="String" value="off" class="gov.nasa.arc.sgbe.remote.nodes.
                PropertyNode"></Node>
25          <Node name="life" propertyType="int" value="2" class="gov.nasa.arc.sgbe.remote.nodes.
                PropertyNode"></Node>
26       </Initial_Values>
27    </PropertiesKey>
28    <IndexedPropertiesKey>
29       <Initial_Values></Initial_Values>
30    </IndexedPropertiesKey>
31    <LogicsKey>
32       <Table_Header Table_Name="topLogicTable"></Table_Header>
33       <Node outputState="null" name="topLogicTable" class="gov.nasa.arc.sgbe.remote.sgb.nodes.
             LogicRootNode">
34          <Node name="INPUTS" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode">
35             <Node name="state" referenceNodeName="state" class="gov.nasa.arc.sgbe.remote.sgb.nodes.
                   SgbNode">
36                <Node propertyEditor="sun.beans.editors.StringEditor" name="on" isValueNode="true"
                      class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
37                <Node propertyEditor="sun.beans.editors.StringEditor" name="fading" isValueNode="true"
                      class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
38                <Node propertyEditor="sun.beans.editors.StringEditor" name="off" isValueNode="true"
                      class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
39             </Node>
40             <Node name="life" referenceNodeName="life" class="gov.nasa.arc.sgbe.remote.sgb.nodes.
                   SgbNode">
41                <Node propertyEditor="sun.beans.editors.IntEditor" name=">0" isValueNode="true" class="
                      gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
42                <Node propertyEditor="sun.beans.editors.IntEditor" name="==0" isValueNode="true" class=
                      "gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
43             </Node>
44             <Node name="ACTIONS" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode">
45                <Node featureName="power.mouseClicked" name="power.mouseClicked" referenceNodeName="
                      power" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
46                <Node featureName="fadeOut.mouseClicked" name="fadeOut.mouseClicked" referenceNodeName=
                      "fadeOut" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
47                <Node featureName="lifeTimer.actionPerformed" name="lifeTimer.actionPerformed"
                      referenceNodeName="lifeTimer" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"><
                      /Node>
48                <Node featureName="fadingTimer.actionPerformed" name="fadingTimer.actionPerformed"
                      referenceNodeName="fadingTimer" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"
                      ></Node>
49             </Node>
50          </Node>
51          <Node name="OUTPUTS" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode">
52             <Node name="state" referenceNodeName="state" class="gov.nasa.arc.sgbe.remote.sgb.nodes.
                   SgbNode">
53                <Node propertyEditor="sun.beans.editors.StringEditor" name="on" isValueNode="true"
                      class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
54                <Node propertyEditor="sun.beans.editors.StringEditor" name="fading" isValueNode="true"
                      class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
55                <Node propertyEditor="sun.beans.editors.StringEditor" name="off" isValueNode="true"
                      class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
56             </Node>
57             <Node name="life" referenceNodeName="life" class="gov.nasa.arc.sgbe.remote.sgb.nodes.
                   SgbNode">
58                <Node propertyEditor="sun.beans.editors.IntEditor" name="-=1" isValueNode="true" class=
                      "gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
59             </Node>
60             <Node featureName="powerStatus.background" name="powerStatus.background"
                   referenceNodeName="powerStatus" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode">
61                <Node propertyEditor="gov.nasa.arc.sgbe.model.beans.editors.ColorEditor" name="255,0,0"
                       isValueNode="true" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
62                <Node propertyEditor="gov.nasa.arc.sgbe.model.beans.editors.ColorEditor" name="0,255,0"
                       isValueNode="true" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
63             </Node>
64             <Node name="PRIMITIVES" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode">
65                <Node featureName="lifeTimer.start" name="lifeTimer.start" referenceNodeName="lifeTimer
                      " class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
66                <Node featureName="lifeTimer.stop" name="lifeTimer.stop" referenceNodeName="lifeTimer"
                      class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
```

```
67              <Node featureName="fadingTimer.start" name="fadingTimer.start" referenceNodeName="
                    fadingTimer" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
68              <Node featureName="fadingTimer.stop" name="fadingTimer.stop" referenceNodeName="
                    fadingTimer" class="gov.nasa.arc.sgbe.remote.sgb.nodes.SgbNode"></Node>
69           </Node>
70        </Node>
71        <Node name="situation" specification="[[[1 0 0][1 0][1 0 0 0]][[0 0 1][0][1 0][0 1 0 0]]]"
              behavior="behavior" class="gov.nasa.arc.sgbe.remote.sgb.nodes.Situation"></Node>
72        <Node name="situation" specification="[[[0 0 1][1 0][1 0 0 0]][[1 0 0][0][0 1][1 0 0 0]]]"
              behavior="behavior" class="gov.nasa.arc.sgbe.remote.sgb.nodes.Situation"></Node>
73        <Node name="situation" specification="[[[1 0 0][1 0][0 1 0 0]][[0 1 0][0][0 0][0 1 1 0]]]"
              behavior="behavior" class="gov.nasa.arc.sgbe.remote.sgb.nodes.Situation"></Node>
74        <Node name="situation" specification="[[[1 0 0][1 0][0 0 1 0]][[0 0 0][1][0 0][0 0 0 0]]]"
              behavior="behavior" class="gov.nasa.arc.sgbe.remote.sgb.nodes.Situation"></Node>
75        <Node name="situation" specification="[[[1 1 0][0 1][0 0 0 0]][[0 0 1][0][1 0][0 1 0 0]]]"
              behavior="behavior" class="gov.nasa.arc.sgbe.remote.sgb.nodes.Situation"></Node>
76        <Node name="situation" specification="[[[0 1 0][0 0][0 0 0 1]][[0 0 1][0][1 0][0 0 0 1]]]"
              behavior="behavior" class="gov.nasa.arc.sgbe.remote.sgb.nodes.Situation"></Node>
77     </Node>
78  </LogicsKey>
79  <TopLogicTable nodeReferenceName="topLogicTable"></TopLogicTable>
80  <Node name="Root" referenceNodeName="Root" class="gov.nasa.arc.sgbe.ui.system.SystemReferenceNode
        " bbKey="SubSystemKey">
81     <Node name="topLogicTable" referenceNodeName="topLogicTable" class="gov.nasa.arc.sgbe.ui.system
           .SystemReferenceNode"></Node>
82     <Node name="topContainer" referenceNodeName="topContainer" class="gov.nasa.arc.sgbe.ui.system.
           SystemReferenceNode"></Node>
83     <Node name="fadeOut" referenceNodeName="fadeOut" class="gov.nasa.arc.sgbe.ui.system.
           SystemReferenceNode" bbKey="ComponentsKey"></Node>
84     <Node name="power" referenceNodeName="power" class="gov.nasa.arc.sgbe.ui.system.
           SystemReferenceNode" bbKey="ComponentsKey"></Node>
85     <Node name="powerStatus" referenceNodeName="powerStatus" class="gov.nasa.arc.sgbe.ui.system.
           SystemReferenceNode" bbKey="ComponentsKey"></Node>
86     <Node name="fadingTimer" referenceNodeName="fadingTimer" class="gov.nasa.arc.sgbe.ui.system.
           SystemReferenceNode" bbKey="ObjectsKey"></Node>
87     <Node name="lifeTimer" referenceNodeName="lifeTimer" class="gov.nasa.arc.sgbe.ui.system.
           SystemReferenceNode" bbKey="ObjectsKey"></Node>
88     <Node name="life" referenceNodeName="life" class="gov.nasa.arc.sgbe.ui.system.
           SystemReferenceNode" bbKey="PropertiesKey"></Node>
89     <Node name="state" referenceNodeName="state" class="gov.nasa.arc.sgbe.ui.system.
           SystemReferenceNode" bbKey="PropertiesKey"></Node>
90  </Node>
91  <SelectedLogicNodes>
92     <NodeReference nodeReferenceName="topLogicTable"></NodeReference>
93  </SelectedLogicNodes>
94  <IndividualLogicNodes></IndividualLogicNodes>
95 </AdeptProject>
```

Annexe B

Fichier Light.java

```
1  import java.util.*;
2  import java.io.*;
3  import gov.nasa.jpf.hmi.models.ExtendedState;
4  import gov.nasa.jpf.hmi.models.LTS;
5  import gov.nasa.jpf.hmi.models.Transition;
6  import gov.nasa.jpf.hmi.models.Action;
7  import gov.nasa.jpf.hmi.models.ActionType;
8
9  public class Light {
10
11         private enum Action {
12                 POWER_MOUSECLICKED ("power.mouseClicked"),
13                 FADEOUT_MOUSECLICKED ("fadeOut.mouseClicked"),
14                 LIFETIMER_ACTIONPERFORMED ("lifeTimer.actionPerformed"),
15                 FADINGTIMER_ACTIONPERFORMED ("fadingTimer.actionPerformed"),
16                 NO_ACTION ("no action");
17
18                 private final String action;
19
20                 private ActionType getType(){
21                         ActionType type = ActionType.COMMAND;
22                         if(this.action.equals("lifeTimer.actionPerformed")) type = ActionType.
                               OBSERVATION;
23                         if(this.action.equals("fadingTimer.actionPerformed")) type = ActionType.
                               OBSERVATION;
24                         return type;
25                 }
26                 private Action (String action) {
27                         this.action = action;
28                 }
29
30                 public String toString() {
31                         return action;
32                 }
33         }
34
35
36         private static class State implements Cloneable {
37
38                 // [V] variables
39                 private String state = "off";
40                 private int life = (int) 2;
41
42                 // [L]<table>.outputState variables
43
44                 // [C] components
45                 private String powerStatus_background = "255,0,0";
46
47                 public boolean equals (Object o) {
48                         if(o instanceof State){
49                                 State s = (State) o;
```

```
50                              return true
51                              && state == s.state
52                              && life == s.life
53                              && powerStatus_background.equals(s.powerStatus_background)
54                              ;
55                      } else {
56                              return false;
57                      }
58              }
59
60
61              public String toString() {
62                      return ("["
63                                      + "state=" + state
64                                      + ",life=" + life
65                                      + ",powerStatus_background=" + powerStatus_background
66                                      + "]"
67                      );
68              }
69
70
71              public String stateObservation() {
72                      return ("["
73                                      + "state =" + state
74                                      + ",life =" + life
75                                      + ",powerStatus_background =" + powerStatus_background
76                                      + "]"
77                      );
78              }
79
80
81              public void trimStateVariables() {
82              }
83
84
85              public Object clone() {
86                      Object clone = null;
87                      try{
88                              clone = super.clone();
89                      } catch (CloneNotSupportedException exception) {}
90                      return clone;
91              }
92      }
93
94
95      /// TABLES ///
96
97      private static void topLogicTable(Action action, State to) {
98              // Situation 0
99              if((to.state.equals("on")) && ((to.life>0)) && (action == Action.POWER_MOUSECLICKED)){
100                     to.state = "off";
101                     to.powerStatus_background = "255,0,0";
102
103                     actionPassed = true;
104             }
105             // Situation 1
106             else if((to.state.equals("off")) && ((to.life>0)) && (action == Action.
                    POWER_MOUSECLICKED)){
107                     to.state = "on";
108                     to.powerStatus_background = "0,255,0";
109
110                     actionPassed = true;
111             }
112             // Situation 2
113             else if((to.state.equals("on")) && ((to.life>0)) && (action == Action.
                    FADEOUT_MOUSECLICKED)){
114                     to.state = "fading";
115
116                     actionPassed = true;
117             }
```

```
118                 // Situation 3
119                 else if((to.state.equals("on")) && ((to.life>0)) && (action == Action.
                        LIFETIMER_ACTIONPERFORMED)){
120                   to.life-=1;
121
122                   actionPassed = true;
123                 }
124                 // Situation 4
125                 else if((to.state.equals("on") || to.state.equals("fading")) && ((to.life==0)) && (
                        action == Action.NO_ACTION)){
126                   to.state = "off";
127                   to.powerStatus_background = "255,0,0";
128
129                   noActionPassed = true;
130                 }
131                 // Situation 5
132                 else if((to.state.equals("fading")) && (action == Action.FADINGTIMER_ACTIONPERFORMED))
                        {
133                   to.state = "off";
134                   to.powerStatus_background = "255,0,0";
135
136                   actionPassed = true;
137                 }
138         }
139         /// SIMULATION VARIABLES ///
140         private static PrintWriter writer;
141         private static List<State> toexplore = new ArrayList<State>();
142         private static List<State> visited = new ArrayList<State>();
143         private static List<List<String>> transitions = new ArrayList<List<String>>();
144         private static Set<Action> usedActions = new HashSet<Action>();
145         private static int exploreCnt = 0;
146         private static int transitionCnt = 0;
147         private static boolean actionPassed;
148         private static boolean noActionPassed;
149
150         public LTS<ExtendedState,Transition> simulate() throws IOException {
151                 LTS<ExtendedState,Transition> system = new LTS<ExtendedState,Transition>();
152
153                 PrintWriter writer = new PrintWriter ("Lightmachine.txt");
154                 State init = new State();
155                 toexplore.add(init);
156                 visited.add(init);
157                 while (! toexplore.isEmpty()){
158                         State from = toexplore.get(0);
159                         toexplore.remove(0);
160                         for (Action action : Action.values()){
161                                 State to = (State) from.clone();
162                                 actionPassed = false;
163                                 noActionPassed = false;
164                                 topLogicTable(action, to);
165                                 to.trimStateVariables();
166                                 if (actionPassed){
167                                         List<String> t = Arrays.asList (from.toString(), action.toString
                                                (), to.toString());
168                                         if (! transitions.contains (t)){
169                                                 transitions.add (t);
170                                                 usedActions.add (action);
171                                                 transitionCnt++;
172                                         }
173                                         if (! visited.contains (to)){
174                                                 visited.add (to);
175                                                 toexplore.add (to);
176                                         }
177                                 } else if(noActionPassed){
178                                         List<String> t = Arrays.asList (from.toString(), "no action", to
                                                .toString());
179                                         if (! transitions.contains (t)){
180                                                 transitions.add (t);
181                                                 usedActions.add (Action.NO_ACTION);
182                                                 transitionCnt++;
```

```
                                }
                                if (! visited.contains (to)){
                                        visited.add (to);
                                        toexplore.add (to);
                                }
                        }
                }
                exploreCnt++;
        }

        List<gov.nasa.jpf.hmi.models.Action> transitionActions = new ArrayList<gov.nasa.jpf.
            hmi.models.Action>();
        PrintWriter transitionActionWriter = new PrintWriter ("LighttransitionAction.txt");
        Map<String,Integer> actionsMap = new HashMap<String,Integer>();
        int i = 0;
        for (Action a : Action.values()){
                actionsMap.put (a.toString(), i);
                String actionSplitted[] = a.toString().split("\\.");
                transitionActions.add (new gov.nasa.jpf.hmi.models.Action (a.toString(), a.
                    getType()));
                writer.printf ("Action a%d = new Action (\"%s\", ActionType." + a.getType() +
                    ");\n", i, a);
                transitionActionWriter.printf ("%s\n", a);
                i++;
        }
        transitionActionWriter.close();

        List<gov.nasa.jpf.hmi.models.Action> stateObservationActions = new ArrayList<gov.nasa.
            jpf.hmi.models.Action>();
        PrintWriter stateObservationActionWriter = new PrintWriter("
            LightstateObservationAction.txt");
        Map<String,Integer> stateObservationsMap = new HashMap<String,Integer>();
        i = 0;
        for (State s : visited){
                if (stateObservationsMap.get(s.stateObservation()) == null){
                        stateObservationsMap.put(s.stateObservation(), i);
                        stateObservationActions.add (new gov.nasa.jpf.hmi.models.Action ("O" +
                            i, ActionType.STATE_OBSERVATION));
                        writer.printf("Action o%d = new Action(\"O%d\", ActionType.
                            STATE_OBSERVATION); // %s\n", i, i, s.stateObservation());
                        i++;
                }
        }
        stateObservationActionWriter.println(i + 1);
        stateObservationActionWriter.close();
        writer.println ("Action tau = new Action(\"tau\", ActionType.TAU);");
        writer.println ("LTS<ExtendedState,Transition> system = new LTS<ExtendedState,
            Transition>();");

        gov.nasa.jpf.hmi.models.Action tau = new gov.nasa.jpf.hmi.models.Action("tau",
            ActionType.TAU);

        List<ExtendedState> states = new ArrayList<ExtendedState>();
        PrintWriter stateWriter = new PrintWriter("Lightstate.txt");
        Map<String,Integer> statesMap = new HashMap<String,Integer>();
        i = 0;
        for (State s : visited){
                statesMap.put(s.toString(), i);
                states.add(new ExtendedState("S" + i, stateObservationActions.get(
                    stateObservationsMap.get (s.stateObservation())) ));
                system.addState(states.get(i));
                writer.printf ("ExtendedState s%d = new ExtendedState (\"S%d\", o%s);\n", i, i
                    , stateObservationsMap.get (s.stateObservation()));
                writer.printf ("system.addState (s%d);\n", i);
                stateWriter.printf ("%s:%s\n", stateObservationsMap.get (s.stateObservation())
                    , s.toString());
                i++;
        }
        stateWriter.close();
```

```
241                 PrintWriter transitionWriter = new PrintWriter ("Lighttransition.txt");
242                 for (List<String> t : transitions){
243                         if ("no action".equals (t.get (1))){
244                                 system.addTauTransition(new Transition(tau), states.get(statesMap.get(t
                                        .get(0))), states.get(statesMap.get(t.get(2))));
245                                 writer.printf("system.addTauTransition(new Transition (tau), s%s, s%s)
                                        ;\n", statesMap.get (t.get (0)), statesMap.get (t.get (2)));
246                                 transitionWriter.printf ("tau;%d;%d\n", statesMap.get (t.get (0)),
                                        statesMap.get (t.get (2)));
247                         } else {
248                                 system.addTransition(new Transition(transitionActions.get(actionsMap.
                                        get(t.get (1)))), states.get(statesMap.get(t.get(0))), states.get(
                                        statesMap.get(t.get(2))));
249                                 writer.printf("system.addTransition (new Transition (a%s), s%s, s%s);\n
                                        ", actionsMap.get (t.get (1)), statesMap.get (t.get (0)), statesMap
                                        .get (t.get (2)));
250                                 transitionWriter.printf ("%d;%d;%d\n", actionsMap.get (t.get (1)),
                                        statesMap.get (t.get (0)), statesMap.get (t.get (2)));
251                         }
252                 }
253                 transitionWriter.close();
254
255                 writer.printf ("return system;");
256                 writer.close();
257                 return system;
258         }
259 }
```

Annexe C

Fichiers texte du modèle Light

C.1 Lightstate.txt

```
0:[ state=off, life =2,powerStatus_background=255,0,0]
1:[ state=on,life=2,powerStatus_background=0,255,0]
2:[ state=fading, life =2,powerStatus_background=0,255,0]
3:[ state=on,life=1,powerStatus_background=0,255,0]
4:[ state=off, life =1,powerStatus_background=255,0,0]
5:[ state=fading, life =1,powerStatus_background=0,255,0]
6:[ state=on,life=0,powerStatus_background=0,255,0]
7:[ state=off, life =0,powerStatus_background=255,0,0]
```

C.2 Ligthtransitionaction.txt

```
power.mouseClicked
fadeOut.mouseClicked
lifeTimer.actionPerformed
fadingTimer.actionPerformed
no action
```

C.3 Lighttransition.txt

```
0;0;1
0;1;0
1;1;2
2;1;3
3;2;0
0;3;4
1;3;5
2;3;6
0;4;3
3;5;4
tau;6;7
```

C.4 LightstateObservationAction.txt

```
9
```

C.5 Lightmachine.txt

```
Action a0 = new Action ("power.mouseClicked", ActionType.COMMAND);
Action a1 = new Action ("fadeOut.mouseClicked", ActionType.COMMAND);
Action a2 = new Action ("lifeTimer.actionPerformed", ActionType.OBSERVATION);
Action a3 = new Action ("fadingTimer.actionPerformed", ActionType.OBSERVATION);
Action a4 = new Action ("no action", ActionType.COMMAND);
Action o0 = new Action("O0", ActionType.STATE_OBSERVATION); // [state =off,life =2,
    powerStatus_background =255,0,0]
Action o1 = new Action("O1", ActionType.STATE_OBSERVATION); // [state =on,life =2,
    powerStatus_background =0,255,0]
Action o2 = new Action("O2", ActionType.STATE_OBSERVATION); // [state =fading,life =2,
    powerStatus_background =0,255,0]
Action o3 = new Action("O3", ActionType.STATE_OBSERVATION); // [state =on,life =1,
    powerStatus_background =0,255,0]
Action o4 = new Action("O4", ActionType.STATE_OBSERVATION); // [state =off,life =1,
    powerStatus_background =255,0,0]
Action o5 = new Action("O5", ActionType.STATE_OBSERVATION); // [state =fading,life =1,
    powerStatus_background =0,255,0]
Action o6 = new Action("O6", ActionType.STATE_OBSERVATION); // [state =on,life =0,
    powerStatus_background =0,255,0]
Action o7 = new Action("O7", ActionType.STATE_OBSERVATION); // [state =off,life =0,
    powerStatus_background =255,0,0]
Action tau = new Action("tau", ActionType.TAU);
LTS<ExtendedState,Transition> system = new LTS<ExtendedState,Transition>();
ExtendedState s0 = new ExtendedState ("S0", o0);
system.addState (s0);
ExtendedState s1 = new ExtendedState ("S1", o1);
system.addState (s1);
ExtendedState s2 = new ExtendedState ("S2", o2);
system.addState (s2);
ExtendedState s3 = new ExtendedState ("S3", o3);
system.addState (s3);
ExtendedState s4 = new ExtendedState ("S4", o4);
system.addState (s4);
ExtendedState s5 = new ExtendedState ("S5", o5);
system.addState (s5);
ExtendedState s6 = new ExtendedState ("S6", o6);
system.addState (s6);
ExtendedState s7 = new ExtendedState ("S7", o7);
system.addState (s7);
system.addTransition (new Transition (a0), s0, s1);
system.addTransition (new Transition (a0), s1, s0);
system.addTransition (new Transition (a1), s1, s2);
system.addTransition (new Transition (a2), s1, s3);
system.addTransition (new Transition (a3), s2, s0);
system.addTransition (new Transition (a0), s3, s4);
system.addTransition (new Transition (a1), s3, s5);
system.addTransition (new Transition (a2), s3, s6);
system.addTransition (new Transition (a0), s4, s3);
system.addTransition (new Transition (a3), s5, s4);
system.addTauTransition(new Transition (tau), s6, s7);
return system;
```

Annexe D

Fichier de configuration du modèle Light

```
<?xml version="1.0"?>
<!DOCTYPE conf [
<!ENTITY amp    "&#38;#38;"> <!-- AMPERSAND -->
<!ENTITY quot   "&#x0022;"> <!-- QUOTATION MARK -->
]>
<AdeptConfig>
	<Tables>
		<Table name="topLogicTable" used="true">
			<Situation used="true">0</Situation>
			<Situation used="true">1</Situation>
			<Situation used="true">2</Situation>
			<Situation used="true">3</Situation>
			<Situation used="true">4</Situation>
			<Situation used="true">5</Situation>
		</Table>
	</Tables>
	<Variables>
		<Var used="true" observable="true" max="" min="">state</Var>
		<Var used="true" observable="true" max="" min="">life</Var>
	</Variables>
	<Components>
		<Comp used="true" observable="true">powerStatus</Comp>
	</Components>
	<Outputs>
	</Outputs>
	<Actions>
		<Act used="true" actionType="COMMAND">power.mouseClicked</Act>
		<Act used="true" actionType="COMMAND">fadeOut.mouseClicked</Act>
		<Act used="true" actionType="OBSERVATION">lifeTimer.actionPerformed</Act>
		<Act used="true" actionType="OBSERVATION">fadingTimer.actionPerformed</Act>
	</Actions>
	<Functions>
	</Functions>
</AdeptConfig>
```

Annexe E

La classe ADEPT2LTS

```
1  package gov.nasa.jpf.hmi.models.util;
2
3  import java.io.BufferedReader;
4  import java.io.IOException;
5  import java.io.InputStreamReader;
6  import java.lang.reflect.InvocationTargetException;
7  import java.lang.reflect.Method;
8  import java.util.Calendar;
9
10 import gov.nasa.jpf.hmi.models.LTS;
11 import gov.nasa.jpf.hmi.models.ExtendedState;
12 import gov.nasa.jpf.hmi.models.Transition;
13 import gov.nasa.jpf.hmi.models.util.pierre.Translator;
14
15 /*
16  * An ADEPT2LTS is used to read an ADEPT model (sgb file) and to return its HMI-LTS (LTS<
        ExtendedState, Transition>).
17  *
18  * @author Pierre Nauw (UCLouvain)
19  * @version Juin 01, 2014
20  */
21 public class ADEPT2LTS {
22
23        private String sgbFile;
24
25        /*
26         * Create a new ADEPT2LTS linked to a sgb file which the name is given as parameter
27         */
28        public ADEPT2LTS(String sgbFile){
29               this.sgbFile = sgbFile;
30        }
31
32        /*
33         * Generate a configuration file linked to the model in sgbFile and named as the conFile
                String given as parameter.
34         * The mode parameter defines the mode of the generation.
35         */
36        public void generateConfigurationFile(String confFile, String mode, boolean verbose){
37               String modelName = "";
38               String path = "";
39
40               String[] windows = sgbFile.split("\\\\");
41               String[] linux = sgbFile.split("/");
42               // Windows
43               if(windows.length > 1){
44                      modelName = windows[windows.length-1].split(".sgb")[0];
45                      int i = 0;
46                      while(i < windows.length -1){
47                             path += windows[i] + "\\";
48                             i++;
49                      }
```

```
50              }
51              // Linux or Mac
52              else if(linux.length > 1) {
53                      modelName = linux[linux.length-1].split(".sgb")[0];
54                      int i = 0;
55                      while(i < linux.length -1){
56                              path += linux[i] + "\\";
57                              i++;
58                      }
59              }
60              // Only model name passed as parameter
61              else {
62                      modelName = sgbFile.split(".sgb")[0];
63              }
64
65              Translator trans = new Translator(modelName, path, verbose);
66              if(verbose) System.out.println("===Generation of configuration file...");
67              trans.generateConf(confFile, mode, verbose);
68              if(verbose) System.out.println("===> Configuration file generated : " + confFile + ".
                    conf");
69      }
70
71      /*
72       * Return an LTS<ExtendedState, Transition> linked to the model in sgbFile.
73       * confFile defines the configuration file which will be used for the translation of the sgb
                file.
74       */
75      public LTS<ExtendedState, Transition> loadLTS(String confFile, boolean debug, boolean verbose
            ) {
76
77              String modelName = "";
78              String path = "";
79
80              String[] windows = sgbFile.split("\\\\");
81              String[] linux = sgbFile.split("/");
82              // Windows
83              if(windows.length > 1){
84                      modelName = windows[windows.length-1].split(".sgb")[0];
85                      int i = 0;
86                      while(i < windows.length -1){
87                              path += windows[i] + "\\";
88                              i++;
89                      }
90              }
91              // Linux or Mac
92              else if(linux.length > 1) {
93                      modelName = linux[linux.length-1].split(".sgb")[0];
94                      int i = 0;
95                      while(i < linux.length -1){
96                              path += linux[i] + "\\";
97                              i++;
98                      }
99              }
100             // Only model name passed as parameter
101             else {
102                     modelName = sgbFile.split(".sgb")[0];
103             }
104
105             if(verbose) System.out.println("Model : " + modelName);
106             if(verbose) System.out.println("Path : " + path);
107
108             // 1) Generation of Java file
109             if(verbose) System.out.println("=== Generation of .java file...");
110             Translator trans = new Translator(modelName, path, verbose);
111             trans.write(confFile, debug, verbose);
112             if(verbose) System.out.println("===> STEP 1 == DONE");
113
114             // 2) Compilation of Java file
115             if(verbose) System.out.println("=== Compilation of .java file...");
116             try {
```

```
                        String location = getClass().getProtectionDomain().getCodeSource().getLocation
                            ().getPath();
            String commandComp = "javac -d " + location + " " + path + modelName + ".java";
            Process comp = Runtime.getRuntime().exec(commandComp);
            BufferedReader compError = getError(comp);
            String compLine = "";
            boolean success = true;
            while ((compLine = compError.readLine()) != null) {
                System.out.println(compLine);
                success = false;
            }

            if(success){
                if(verbose) System.out.println("===> STEP 2 == DONE");
            } else {
                if(verbose) System.out.println("===> STEP 2 == FAIL");
                 System.exit(2);
            }
        } catch (IOException e) {
                 e.printStackTrace();
                if(verbose) System.out.println("===> STEP 2 == FAIL");
                 System.exit(2);
        }

                // 3) Getting the class linked to the model and call to the simulate() method.
                if(verbose) System.out.println("=== Simulation of the model...");
                LTS<ExtendedState,Transition> system = new LTS<ExtendedState,Transition>();
                try {
                        Class modelClass = Class.forName(modelName);
                        Method meth = modelClass.getMethod("simulate");
                        system = (LTS<ExtendedState, Transition>) meth.invoke(modelClass.newInstance()
                            );
                } catch (ClassNotFoundException e1) {
                        if(verbose) System.out.println("===> STEP 3 == FAIL");
                        e1.printStackTrace();
                } catch (NoSuchMethodException | SecurityException e) {
                        if(verbose) System.out.println("===> STEP 3 == FAIL");
                        e.printStackTrace();
                } catch (IllegalAccessException | IllegalArgumentException
                                | InvocationTargetException | InstantiationException e) {
                        if(verbose) System.out.println("===> STEP 3 == FAIL");
                        e.printStackTrace();
                }
                if(verbose) System.out.println("===> STEP 3 == DONE");
                return system;
        }

        /*
         * Getting the error
         */
        private static BufferedReader getError(Process p) {
                return new BufferedReader(new InputStreamReader(p.getErrorStream()));
            }
}
```

Annexe F

Méthode *behaviorsCreation()*

```
private void behaviorsCreation(Table table) {
	for(BehaviorNode node : table.getBehaviorList()) {
		Vector<Vector<Boolean>> vList = new Vector<Vector<Boolean>>();
		String s = node.getSpecification().substring(1, node.getSpecification().length());
		int i = 0;
		int count = 0;
		while(i < s.length()-1) {
			char c = s.charAt(i);
			if(c == '[' && count == 0) {
				Vector<Boolean> vTemp = new Vector<Boolean>();
				i++;
				count++;
				// Filling the vector
				while(count != 0) {
					c = s.charAt(i);
					switch(c){
						case '[':
							count++;
							break;
						case ']':
							count--;
							break;
						case '1':
							vTemp.add(true);
							break;
						case '0':
							vTemp.add(false);
					}
					i++;
				}
				vList.add(vTemp);
			}
		}
		Vector<Boolean> vIn = new Vector<Boolean>();
		Vector<Boolean> vOut = new Vector<Boolean>();
		vIn = vList.get(table.getInputsPosition());
		vOut = vList.get(table.getOutputsPosition());
		this.inputsValue.add(vIn);
		this.outputsValue.add(vOut);
	}
}
```

Annexe G

Modèle airspeedTargetSystemTable

G.1 Autopilotstate.txt

```
0:[, airspeedTarget=250.0,selectedSpeedTarget=250.0,airspeedTargetSystemTableOutput=use knots for airspeed target]
0:[, airspeedTarget=251.0,selectedSpeedTarget=251.0,airspeedTargetSystemTableOutput=increase IAS airspeed target]
0:[, airspeedTarget=249.0,selectedSpeedTarget=249.0,airspeedTargetSystemTableOutput=decrease IAS airspeed target]
0:[, airspeedTarget=252.0,selectedSpeedTarget=252.0,airspeedTargetSystemTableOutput=increase IAS airspeed target]
0:[, airspeedTarget=250.0,selectedSpeedTarget=250.0,airspeedTargetSystemTableOutput=decrease IAS airspeed target]
0:[, airspeedTarget=250.0,selectedSpeedTarget=250.0,airspeedTargetSystemTableOutput=increase IAS airspeed target]
0:[, airspeedTarget=248.0,selectedSpeedTarget=248.0,airspeedTargetSystemTableOutput=decrease IAS airspeed target]
0:[, airspeedTarget=251.0,selectedSpeedTarget=251.0,airspeedTargetSystemTableOutput=decrease IAS airspeed target]
0:[, airspeedTarget=249.0,selectedSpeedTarget=249.0,airspeedTargetSystemTableOutput=increase IAS airspeed target]
```

G.2 AutopilottransitionAction.txt

```
airspeedTargetIncreaseTimer.actionPerformed
airspeedTargetDecreaseTimer.actionPerformed
no action
```

G.3 Autopilottransition.txt

```
0;0;1
1;0;2
tau;0;0
0;1;3
1;1;4
tau;1;1
0;2;5
1;2;6
tau;2;2
0;3;3
1;3;7
```

```
tau;3;3
0;4;1
1;4;2
tau;4;4
0;5;1
1;5;2
tau;5;5
0;6;8
1;6;6
tau;6;6
0;7;3
1;7;4
tau;7;7
0;8;5
1;8;6
tau;8;8
```

G.4 Autopilotbisim.mental

```
===== Initial State
 - S1 [S3, S4, S5, S6, S7, S8, S2, S1, S0]

===== States (1)
 - S1 [S3, S4, S5, S6, S7, S8, S2, S1, S0]

===== Transitions (2)
 - S1 [S3, S4, S5, S6, S7, S8, S2, S1, S0] --[O0] airspeedTargetDecreaseTimer.actionPerformed--> S1 [S3,
       S4, S5, S6, S7, S8, S2, S1, S0]
 - S1 [S3, S4, S5, S6, S7, S8, S2, S1, S0] --[O0] airspeedTargetIncreaseTimer.actionPerformed--> S1 [S3, S4
       , S5, S6, S7, S8, S2, S1, S0]
```

Annexe H

Modèle F/D

H.1 FDstate.txt

```
0:[, captainsFdSystemTableOutput=captains FD on,firstOfficersFdSystemTableOutput=first officers FD on]
1:[, captainsFdSystemTableOutput=captains FD off,firstOfficersFdSystemTableOutput=first officers FD on]
2:[, captainsFdSystemTableOutput=captains FD on,firstOfficersFdSystemTableOutput=first officers FD off]
3:[, captainsFdSystemTableOutput=captains FD off,firstOfficersFdSystemTableOutput=first officers FD off]
```

H.2 FDtransitionAction.txt

```
mcpCaptainsFdSwitch.mouseClicked
mcpFoFdSwitch.mouseClicked
no action
```

H.3 FDtransition.txt

```
0;0;1
1;0;2
tau;0;0
0;1;0
1;1;3
tau;1;1
0;2;3
1;2;0
tau;2;2
0;3;2
1;3;1
tau;3;3
```

H.4 FDbisim.mental

```
===== Initial State
 - S1 [S3, S2, S1, S0]

===== States (1)
 - S1 [S3, S2, S1, S0]

===== Transitions (8)
 - S1 [S3, S2, S1, S0] --[O1] mcpFoFdSwitch.mouseClicked--> S1 [S3, S2, S1, S0]
 - S1 [S3, S2, S1, S0] --[O3] mcpFoFdSwitch.mouseClicked--> S1 [S3, S2, S1, S0]
```

```
10  - S1 [S3, S2, S1, S0] --[O2] mcpFoFdSwitch.mouseClicked--> S1 [S3, S2, S1, S0]
11  - S1 [S3, S2, S1, S0] --[O2] mcpCaptainsFdSwitch.mouseClicked--> S1 [S3, S2, S1, S0]
12  - S1 [S3, S2, S1, S0] --[O3] mcpCaptainsFdSwitch.mouseClicked--> S1 [S3, S2, S1, S0]
13  - S1 [S3, S2, S1, S0] --[O0] mcpCaptainsFdSwitch.mouseClicked--> S1 [S3, S2, S1, S0]
14  - S1 [S3, S2, S1, S0] --[O0] mcpFoFdSwitch.mouseClicked--> S1 [S3, S2, S1, S0]
15  - S1 [S3, S2, S1, S0] --[O1] mcpCaptainsFdSwitch.mouseClicked--> S1 [S3, S2, S1, S0]
```

Annexe I

Modèle VTS

I.1 VTSstate.txt

```
0:[topLogicTableOutput=Low1]
0:[topLogicTableOutput=Medium1]
0:[topLogicTableOutput=Low2]
0:[topLogicTableOutput=High1]
0:[topLogicTableOutput=Low3]
0:[topLogicTableOutput=Medium2]
0:[topLogicTableOutput=High2]
0:[topLogicTableOutput=High3]
```

I.2 VTStransitionAction.txt

```
pushUp.mouseClicked
pullDown.mouseClicked
upTimer.actionPerformed
downTimer.actionPerformed
no action
```

I.3 VTStransition.txt

```
0;0;1
2;0;2
0;1;3
1;1;4
2;1;5
0;2;1
2;2;4
3;2;0
1;3;5
2;3;6
0;4;3
3;4;2
0;5;6
1;5;4
3;5;1
1;6;5
2;6;7
3;6;3
1;7;5
3;7;6
```

I.4 VTSbisim.mental

```
===== Initial State
 - S43 [S0]

===== States (5)
 - S42 [S2]
 - S9 [S3, S6, S7]
 - S11 [S5, S1]
 - S33 [S4]
 - S43 [S0]

===== Transitions (14)
 - S9 [S3, S6, S7] --[O0] downTimer.actionPerformed--> S9 [S3, S6, S7]
 - S9 [S3, S6, S7] --[O0] pullDown.mouseClicked--> S11 [S5, S1]
 - S11 [S5, S1] --[O0] upTimer.actionPerformed--> S11 [S5, S1]
 - S42 [S2] --[O0] upTimer.actionPerformed--> S33 [S4]
 - S9 [S3, S6, S7] --[O0] upTimer.actionPerformed--> S9 [S3, S6, S7]
 - S33 [S4] --[O0] downTimer.actionPerformed--> S42 [S2]
 - S42 [S2] --[O0] pushUp.mouseClicked--> S11 [S5, S1]
 - S11 [S5, S1] --[O0] pushUp.mouseClicked--> S9 [S3, S6, S7]
 - S42 [S2] --[O0] downTimer.actionPerformed--> S43 [S0]
 - S11 [S5, S1] --[O0] downTimer.actionPerformed--> S11 [S5, S1]
 - S11 [S5, S1] --[O0] pullDown.mouseClicked--> S33 [S4]
 - S33 [S4] --[O0] pushUp.mouseClicked--> S9 [S3, S6, S7]
 - S43 [S0] --[O0] pushUp.mouseClicked--> S11 [S5, S1]
 - S43 [S0] --[O0] upTimer.actionPerformed--> S42 [S2]
```

Annexe J

Modèle CountDown

J.1 CountDownstate.txt

```
0:[alarm=false,running=false,value=0]
1:[alarm=false,running=false,value=1]
2:[alarm=false,running=false,value=2]
3:[alarm=false,running=true,value=1]
4:[alarm=false,running=true,value=2]
0:[alarm=true,running=false,value=0]
```

J.2 CountDowntransitionAction.txt

```
Inc.mouseClicked
Reset.mouseClicked
Cancel.mouseClicked
Start.mouseClicked
Ring.mouseClicked
no action
```

J.3 CountDowntransition.txt

```
0;0;1
0;1;2
1;1;0
3;1;3
0;2;0
1;2;0
3;2;4
2;3;0
tau;3;5
tau;4;3
4;5;0
```

J.4 CountDownlearning.mental

```
===== Initial State
 - S0

===== States (2)
 - S0
 - S2
```

```
7
8  ===== Transitions (12)
9    - S2 --[O3] Ring.mouseClicked--> S0
10   - S0 --[O1] Inc.mouseClicked--> S0
11   - S0 --[O0] Inc.mouseClicked--> S0
12   - S2 --[O3] Cancel.mouseClicked--> S0
13   - S0 --[O2] Reset.mouseClicked--> S0
14   - S2 --[O0] Ring.mouseClicked--> S0
15   - S0 --[O1] Start.mouseClicked--> S2
16   - S2 --[O4] Ring.mouseClicked--> S0
17   - S0 --[O1] Reset.mouseClicked--> S0
18   - S0 --[O2] Inc.mouseClicked--> S0
19   - S0 --[O2] Start.mouseClicked--> S2
20   - S2 --[O4] Cancel.mouseClicked--> S0
```