Université Catholique de Louvain

Ecole Polytechnique de Louvain

A framework for visualizing and manipulating structured execution graphs

(Un outil pour visualiser et manipuler des graphes d'exécutions structurées)

Promoteur : Prof. Charles **Pecheur**
Lecteur : Simon **Busard**
Lecteur : Prof. Kim **Mens**

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

Louvain-la-Neuve
Année 2014 - 2015

Résumé

Ce travail présente un outil qui permet d'afficher et de manipuler des graphes d'exécutions structurées. Ceux-ci représentent des chemins possibles dans un système et ils peuvent être structurés parce qu'il existe une hiérarchie. En réalité, ils proviennent du model checking qui permet de vérifier une propriété sur un système. Ceci nous donne une explication qui elle-même peut avoir des sous-explications faisant apparaître une hiérarchie. Cet outil permet d'afficher ces explications mais aussi d'appliquer un layout sur les graphes, naviguer dans la hiérarchie, afficher une trace sélectionnée dans le graphe, mettre en évidence les prédécesseurs et successeurs d'un nœud, ... Ce document contient l'explication du contexte dans lequel s'insère cet outil, son architecture, son utilisation, des exemples, un détail des fonctionnalités et enfin un ensemble de tests.

Le code de l'outil peut être trouvé à l'adresse suivante :

- https ://bitbucket.org/onauw/hierarchicalexplanationapi

Table des matières

Chapitre 1

Introduction

Il existe plusieurs techniques afin de vérifier un système qui est dynamique. Cela peut être un programme informatique, une modélisation d'un environnement, ... Ce que l'on cherche ici, c'est à trouver les bugs qui peuvent exister lorsque l'on conçoit un système ou alors simplement de s'assurer que le système respecte certaines caractéristiques. Pour trouver cette faille ou cette absence de faille, on va utiliser une technique existante : le model checking. On procède à une exploration exhaustive des différents états possibles du système afin de vérifier qu'une propriété est vraie pour l'entièreté du système. Cependant, ce n'est pas si simple, on veut comprendre pourquoi ce système possède ou ne possède pas cette caractéristique et on veut l'expliquer. Afin de bien saisir l'explication, une visualisation peut être utile, voire nécessaire parce que l'explication peut être complexe et pas uniquement une exécution simple du système où l'on atteint un état d'erreur après trois étapes.

C'est ici que ce travail intervient en proposant un outil visuel qui permet d'afficher et de manipuler des graphes. Ceux-ci sont complexes et permettent d'expliquer des propriétés assez riches sur des systèmes. Ils peuvent être hiérarchiques dans le sens où une explication principale d'une caractéristique peut nécessiter une explication plus simple sous-jacente. En outre, une complexité supplémentaire est ajoutée puisque l'outil doit être capable de gérer une génération incrémentale, petit à petit, de cette explication. Enfin, on parle de graphes d'exécutions structurés parce qu'il s'agit d'exécutions du système et ces graphes possèdent une certaine structure hiérarchique en fonction de l'explication.

Cette présentation de l'outil se divise en plusieurs parties. Premièrement, afin de bien saisir le contexte et le pourquoi d'un tel outil, une description du problème traité ainsi que la solution choisie vont être présentées. Ensuite, l'architecture résultant des choix effectués va être décrite. La partie suivante détaillera la communication qu'il existe avec l'outil visuel et comment l'utiliser. Par après, deux exemples de programmes utilisant l'outil visuel seront donnés. Par la suite, toutes les fonctionnalités offertes seront présentées et finalement, divers tests seront réalisés afin d'évaluer les graphes présentés par l'outil ainsi que les performances d'affichage.

Le code de l'outil peut être trouvé à l'adresse suivante :

- https ://bitbucket.org/onauw/hierarchicalexplanationapi

Chapitre 2

Contexte

Dans ce chapitre, nous verrons dans quel contexte a été réalisé ce mémoire, en réponse à quelle demande. Ensuite, pour comprendre les données traitées par l'outil visuel, il faut donner certaines bases en logique comme les opérateurs afin de comprendre les propriétés que l'on vérifie sur un modèle. Et enfin, nous exposerons les différentes possibilités qui existaient ainsi que la solution, en exposant ses avantages et inconvénients.

2.1 Exposition du problème

Dans le cadre de ce mémoire, il fallait fournir un outil qui permette de visualiser, manipuler et obtenir certaines informations sur des explications hiérarchiques tout en restant assez général afin d'avoir une étendue d'utilisation la plus large possible. De plus, il faut que cet outil puisse supporter une génération incrémentale de l'explication et donc une interactivité pour la mise à jour de celle-ci. Cependant, il faut comprendre d'où proviennent ces explications hiérarchiques et ce qu'elles expliquent.

Tout d'abord, il faut comprendre ce que l'on entend par explication hiérarchique. Une explication est composée d'un côté, de nœuds qui représentent des états et d'un autre côté, d'arêtes dirigées qui représentent des transitions entre ces différents états. Ceux-ci possèdent des variables qui évoluent au travers des transitions. Cette évolution permet de montrer une caractéristique. La hiérarchie intervient parce qu'un nœud peut lui-même demander une explication. Par exemple, on veut montrer qu'un compteur (une seule variable qui augmente) peut atteindre une valeur de 2. La figure 2.1 montre la valeur qui passe de 0 à 2 avec les états 2, 3 et 1 et les transitions en trait plein. Ceci est une explication. Par contre, l'état 4 atteint avec une transition en pointillé est une sous-explication de l'état 1. Elle montre pourquoi l'état 1 vaut bien 2 et donc on a juste un état où la valeur du compteur est à 2.

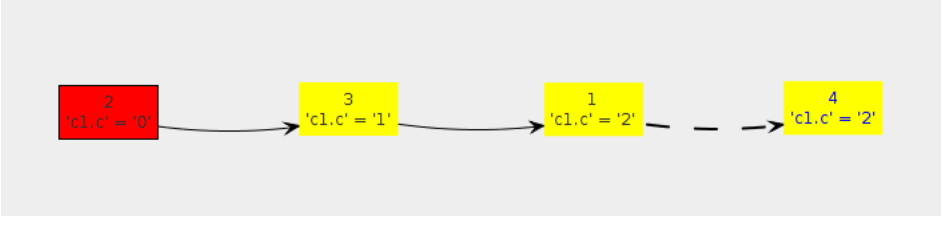


FIGURE 2.1 – Explication hiérarchique.

Maintenant que l'on a vu un exemple très basique d'explication hiérarchique, il faut comprendre comment générer de telles données et dans quel contexte. Pour cela, il faut savoir ce qu'est le model checking. Il s'agit d'une famille de techniques de vérification automatique de systèmes dynamiques. Il s'agit de vérifier algorithmiquement si un modèle donné, le système lui-même ou une abstraction du système, satisfait une spécification, souvent formulée en termes de logique temporelle. Pour faire simple, on représente un système par un modèle et on veut vérifier certaines propriétés dans celui-ci. Ces propriétés sont exprimées grâce à des logiques (temporelle, épistémique, ...) qui comprennent des opérateurs. Dans ce cas-ci, l'outil utilisé pour générer les explications est PyNuSMV. Il s'agit d'un model checker écrit en Python qui se base sur NuSMV qui est lui-même un model checker avec méthode symbolique (BDD - Binary Decision Diagram) [6].

Il y a donc deux points très importants en model checking : le modèle et les propriétés. Ce modèle représente un système que l'on peut décrire. Dans ce cas-ci, il est composé de deux compteurs qui peuvent aller jusque 2 (valeur maximum). Les actions possibles pour ceux-ci sont de ne rien faire ou alors d'incrémenter leur valeur. Il faut rajouter qu'un seul compteur effectue une action sur un laps de temps. Ils n'effectuent pas d'actions simultanées, c'est chacun à son tour. Voici le code qui modélise ce système.

```
MODULE Counter(run)
    VAR
        c: 0 .. 2;
    IVAR
        act: {inc, skip};
    INIT
        c = 0
    TRANS
        run & act = inc -> next(c) = (c + 1) mod 3
    TRANS
        ! run | act = skip -> next(c) = c

MODULE main
    IVAR
        run: {rc1, rc2};
    VAR
        ran: {rc1, rc2};
        c1: Counter(run = rc1);
        c2: Counter(run = rc2);
    TRANS
        next(ran) = run
    FAIRNESS
        ran = rc1
    FAIRNESS
        ran = rc2
```

On peut voir que l'on a bien deux compteurs, rc1 et rc2. Les variables du système sont donc ceux-ci mais aussi une variable ran afin de savoir quel compteur effectue une action. La "fairness" permet de dire que chaque compteur pourra effectuer des actions. Un compteur est simple, uniquement une variable et deux actions, skip et inc, pour ne rien faire ou incrémenter d'une unité le compteur.

2.2 Logique

Maintenant que l'on a vu comment définir un modèle, il nous reste à donner des outils permettant d'exprimer des propriétés que l'on pourra vérifier dans ce modèle grâce à PyNuSMV. C'est exactement ce que la logique nous apporte. Dans cette section, les opérateurs logiques sont présentés en fonction de la logique à laquelle ils appartiennent. De plus, la section suivante présente un exemple utilisant ces opérateurs.

2.2.1 CTL

Prenons un exemple de formule afin de comprendre ce qu'elles peuvent exprimer. Si nous prenons $EF\ c1.c = 2$, cette propriété exprime le fait qu'il existe un chemin où $c1.c = 2$ devient finalement vrai. Nous avons donc la possibilité d'exprimer des choses sur les états comme "il existe un état", "un état ou un autre", " état implique un autre", ... Ce sont les formules d'états. Mais nous avons aussi les formules sur chemins qui permettent de donner une notion temporelle et de dire quand un état doit être vrai. De plus, en logique CTL, il faut toujours grouper les opérateurs sur les chemins avec **A** et **E**. Cela permet de donner un chemin et de spécifier son état.

Les opérateurs de la logique CTL (Computation tree logic) :

— Formules sur les états :

- true et false.
- Les opérateurs usuels de la logique propositionnelle : $\neg\phi$, $\phi \wedge \phi$, $\phi \vee \phi$, $\phi \Rightarrow \phi$ et $\phi \Leftrightarrow \phi$.
- **A** ϕ : All, ϕ doit être vrai sur tous les chemins à partir de l'état courant.
- **E** ϕ : Exists, il existe au moins un chemin à partir de l'état courant où ϕ est vrai.

— Formules sur les chemins :

- **X** ψ : Next, ψ doit être vrai dans le prochain état.
- **G** ψ : Globally, ψ doit être vrai sur l'entièreté du chemin subséquent.
- **F** ψ : Finally, ψ doit finalement être vrai(quelque part sur le chemin subséquent).
- ψ **U** ψ' : Until, ψ doit être vrai au moins jusqu'a ce que ψ' le soit. Ce qui implique que ψ' devienne vrai.
- ψ **W** ψ' : Weak until, ψ est vrai jusqu'à ψ' est vrai. La différence avec le point précédent est qu'il n'y a pas de garantie que ψ' devienne vrai un jour.

2.2.2 CTL avec fairness

Si l'on ajoute la fairness à CTL, il faut considérer uniquement les chemins équitables. Ces contraintes de fairness nous donnent un ensemble d'états par contrainte. Pour chaque contrainte, on veut que le chemin passe par tous les membres de son ensemble d'états infiniment souvent. Pour les opérateurs **A** et **E**, on considère uniquement ce type de chemin. Par exemple, nous avons l'opérateur **EFf** ϕ qui nous dit qu'il existe finalement un chemin fair qui satisfait ϕ.

2.2.3 ARCTL

Action-Restricted CTL étend la logique CTL en rajoutant le fait qu'un quantificateur sur le chemin (**A** et **E**) est restreint par une formule propositionnelle sur les actions. Par exemple, $\mathbf{A}_{a|b}\mathbf{G}\ \phi$ signifie que tous les états atteignables au travers des actions a ou b satisfont ϕ. Les modèles changent un peu des modèles CTL puisqu'on ajoute la notion d'action sur les transitions [5].

2.2.4 CTLK

Cette logique étend aussi CTL en rajoutant la connaissance. On a donc un opérateur supplémentaire $\mathbf{K}_{ag}\ \phi$ où ag est un agent. Cela signifie que l'agent ag connaît ϕ dans un état s si et seulement si ϕ est vrai dans tous les états atteignables qui sont indiscernables à partir de s par ag. Cette logique fournit aussi d'autres opérateurs épistémiques (group knowledge $\mathbf{E}_g$, distributed knowledge $\mathbf{D}_g$ et common knowledge $\mathbf{C}_g$) mais ils ne sont pas développés ici. Les modèles sont ici aussi différents puisque ce sont des modèles multi-agents qui ajoutent des agents dans le système ainsi que les relations qui définissent ce que ces agents observent [5].

2.2.5 ATL

Alternating-time Temporal Logic est une logique pour raisonner sur les stratégies d'un sous-ensemble d'agents dans un système et les buts qu'ils peuvent atteindre en utilisant ces stratégies. ATL considère que les agents ont une connaissance parfaite du système et de son exécution. Si cette connaissance est partielle, on parle alors de logique ATL_{ir}. La logique CTL est un sous-ensemble de la logique ATL.

Cette logique introduit deux nouveaux opérateurs, $< A >$ et $[A]$ où A appartient à l'ensemble des joueurs. Ici aussi, il faut les grouper avec les opérateurs sur chemins. Ils remplacent donc A et E de la logique CTL. $< A > \phi$ signifie que A possède une stratégie pour gagner dans ce jeu. Tous les chemins forcés par cette stratégie satisfont ϕ. $[A]\ \phi$ signifie lui que les joueurs de A n'ont pas de stratégie pour rendre ϕ faux [3],[7].

2.2.6 ATLK_{IrF}

Cette logique-ci ajoute la fairness à ATL. Nous avons donc le f pour signaler que l'on ne regarde que les chemins qui sont équitables.

2.2.7 ATLK_{irF}

ATL avec information imparfaite et fairness. L'information imparfaite permet de modéliser des systèmes où les agents ne connaissent pas tout de celui-ci ou de son exécution. Par exemple, si on veut parler d'un jeu carte classique, le joueur ne connaît pas les cartes de ses adversaires et donc l'information qu'il possède est incomplète [7].

2.2.8 Exemple

Voici un exemple de liste de propriétés que l'on peut essayer de vérifier avec PyNuSMV dans le modèle des compteurs.

1. Il existe un chemin où $c1.c = 2$ devient finalement vrai.
2. Il existe un chemin où $c1.c = 2$ devient finalement vrai et il existe un chemin où $c2.c = 2$ devient finalement vrai.
3. Il existe un chemin où $c1.c = 2$ devient finalement vrai et $c2.c = 2$ aussi.
4. Pour tous les chemins, $c1.c <= 2$ est vrai.
5. Il existe un chemin fair démarrant en l'état initial du système.
6. Pour tout état atteignable, il existe un chemin pour atteindre un état où $c1.c = 0$.
7. Il existe un chemin où finalement $c1$ connait $c1.c = 2$.
8. Pour tout état atteignable où $c1.c = 2$, $c1$ sait que $c1.c = 2$.
9. Il existe un chemin menant à un état où $c1$ sait que $c2.c = 2$.
10. $c1$ possède une stratégie pour obtenir finalement que $c1.c = 2$ est vrai.
11. $c1$ possède une stratégie qui tient compte des chemins équitables pour obtenir finalement que $c1.c = 2$ est vrai.
12. $c1$ possède une stratégie sans avoir une information parfaite et qui tient compte des chemins équitables pour obtenir finalement que $c1.c = 2$ est vrai.
13. $c1$ et $c2$ possèdent une stratégie sans avoir une information parfaite et qui tient compte des chemins équitables pour obtenir finalement que $c1.c = 2$ et $c2.c = 2$ est vrai.

```
// CTL properties
1. EF c1.c = 2
2. (EF c1.c = 2 & EF c2.c = 2)
3. EF (c1.c = 2 & c2.c = 2)
4. AG c1.c <= 2
5. EFf True
6. AG EF c1.c = 0
// CTLK properties
7. EF K<c1>c1.c = 2
8. AG (c1.c = 2 -> K<c1>c1.c = 2)
9. EF K<c1>c2.c = 2
// ATL properties
10. <c1> F c1.c = 2
11. <c1> Ff c1.c = 2
// ATL_ir properties
12. <c1>i Ff c1.c = 2
13. <c1c2>i Ff (c1.c = 2 & c2.c = 2)
```

2.3 Complexité des explications hiérarchiques

Dans cette section, nous présenterons un exemple d'explication d'une propriété du modèle des compteurs afin de montrer que ces données peuvent être conséquentes.

AG EF c1.c=0

Pour tout état atteignable, il existe un chemin pour atteindre un état où $c1.c = 0$. On a donc comme résultat, une explication avec tous les états possibles du modèle au centre et ils possèdent tous une sous-explication pour montrer qu'ils peuvent atteindre $c1.c = 0$. Malgré le fait que ce modèle soit assez simple, le nombre d'états et de sous-explications peut déjà être grand. Potentiellement, on peut avoir tout le modèle avec la complexité des sous-explications en plus. Les données contenues par une explication peuvent donc être riches et complexes. Il est donc nécessaire d'avoir un outil pour rendre cette richesse plus gérable.

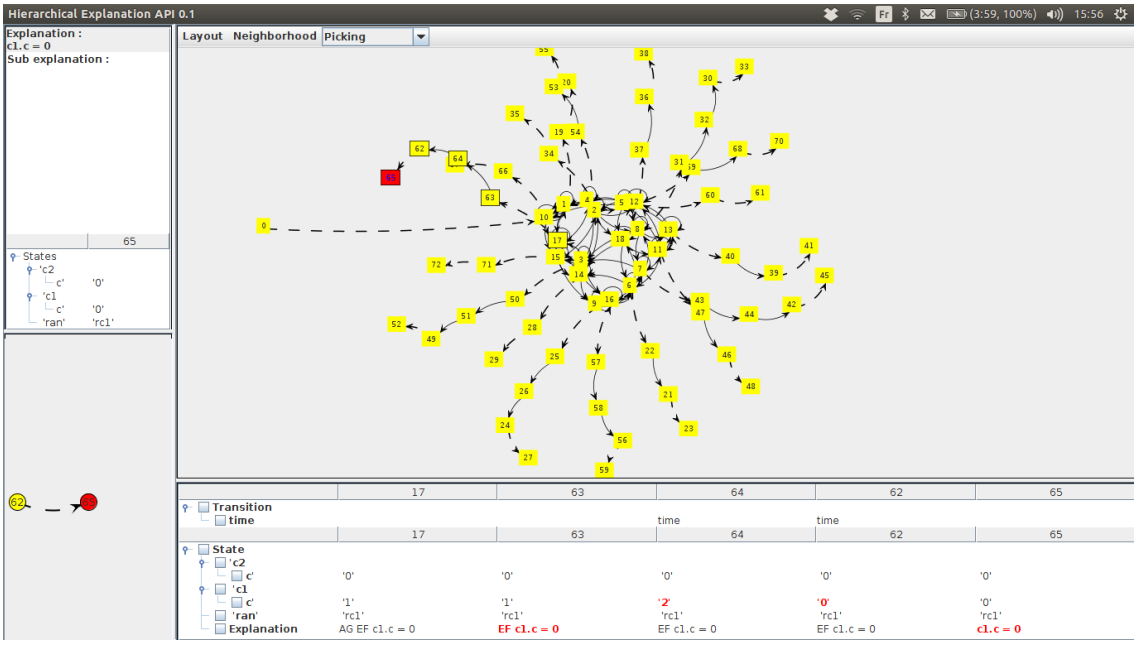


FIGURE 2.2 – AG EF c1.c=0 - complexité d'une explication.

Génération interactive

Étant donné la complexité et la taille qui peut être très grande pour les explications hiérarchiques, une fonctionnalité intéressante que l'on souhaite pour notre outil, c'est l'interactivité. C'est-à-dire que l'on veut pouvoir générer les explications au fur et à mesure pour alléger le calcul. On veut donc pouvoir fournir la structure de données de manière incrémentale. Pour limiter cette interactivité et sa complexité, deux façons de rajouter de l'information ont été retenues : soit on ajoute une sous-explication entière, soit on ajoute tous les successeurs d'un nœud.

2.4 Choix d'une librairie graphique et impact

Les données sur lesquelles on travaille, les explications hiérarchiques, sont difficiles à visualiser et manipuler. C'est pourquoi il est nécessaire d'avoir un outil visuel qui permet cela et facilite la tache d'exploration et de manipulation pour comprendre une explication. Pour combler ce besoin d'un outil pour les explications hiérarchiques, il y a deux possibilités : soit implémenter l'outil de A à Z ou alors trouver une librairie graphique existante qui

permette déjà l'affichage et la manipulation de graphes. Le choix de chercher une librairie est plus simple et permet de fournir un outil plus abouti.

Le but était donc de trouver une bonne librairie graphique qui fournisse des graphiques de qualité et qui permette les interactions, que l'on puisse déplacer les nœuds, appliquer un layout, sélectionner un nœud ou une trace, changer les couleurs, zoomer/dézoomer, rajouter des nœuds ou des arêtes par après pour l'interactivité, ... Deux possibilités sont apparues en fonction des librairies trouvées : Soit rester en python puisque PyNuSMV est en python, soit s'ouvrir aux autres langages mais en trouvant une solution pour intégrer PyNuSMV. Le choix a été fait en fonction des librairies et de leurs possibilités plus qu'au niveau du langage. Celui-ci s'est porté sur JUNG qui fait exactement ce que l'on veut et même encore plus. D'autres logiciels existent comme GraphViz en java ou NetworkX et IGraph en python mais JUNG apparaissait comme étant le plus complet et les exemples fournis sur leur site internet montraient qu'on avait toutes les fonctionnalités nécessaires pour développer une telle application graphique.

Cependant, pour pouvoir choisir JUNG comme librairie de base, il y a un souci puisque celle-ci est implémentée en Java alors que PyNuSMV est en Python. Il existe une solution pour combiner ces 2 types de code. Il s'agit de Jython mais malheureusement cet outil n'est pas encore fonctionnel avec Python3. Donc sans pouvoir combiner le code, il a fallu trouver un autre moyen pour communiquer entre Java avec JUNG d'un côté et PyNuSMV de l'autre. La façon de procéder choisie pour faire ceci est de lancer un process Python à partir de Java grâce à un "ProcessBuilder" et de communiquer par le biais d'un stream texte. Cette communication correspond à l'envoi d'explication Hiérarchique. Cela entraine donc un parsing pour faire passer la structure de données fournie par PyNuSMV sur le stream texte mais aussi un parsing du côté de Java pour récupérer ces données et les mettre dans une nouvelle structure représentant une explication hiérarchique.

Enfin, le dernier choix à faire était de définir quelle librairie visuelle de Java utiliser, JavaFX (qui est le nouveau standard) ou SWING (l'ancien). Étant donné que les composants JUNG sont des composants SWING, on peut les intégrer directement. Cependant comme JavaFX est la nouveauté et qu'il est possible d'y intégrer des composants SWING, il fallait quand même l'essayer mais le résultat était très lent et donc le choix s'est porté sur SWING.

Au niveau des avantages et inconvénients du choix de JUNG comme librairie graphique, on peut mentionner que ce choix apporte pas mal de bonnes choses et de la facilité au niveau de la conception du logiciel d'affichage car cette librairie fournit des fonctionnalités bien utiles. Les graphes sont gérés directement par cette librairie, il faut juste définir le graphe, les états et transitions et ensuite l'affichage (graphe, zoom/dezoom, mouvement du graphe, rajout de nœuds ou arêtes, ...) est directement fourni. De plus, il offre une customisation facile au moyen de "Transformer" qui permet de changer l'apparence (couleur, type de trait, ...). Et enfin, JUNG fournit aussi des layouts qui sont très utiles pour mieux disposer le graphe mais aussi la possibilité d'en créer soi-même. Par contre, le gros inconvénient de cette solution est qu'il faut passer toutes les explications par le stream texte, ce qui implique un parsing des deux côtés. Il faut écrire selon un format fixé du côté Python et récupérer la structure de données du côté Java en fonction de ce format. Il y a donc un possible ralentissement de l'application si les données à passer sont trop importantes.

2.5 Travaux existants sur des graphes

Deux travaux déjà réalisés à l'UCL, le TLACE-visualizer [5] et le mémoire réalisé par Cédric Delforge sur la visualisation dans LTSA, ont permis de prendre des idées afin d'afficher au mieux les explications et leur contenu. On peut notamment citer l'affichage d'un chemin dans le graphe au moyen d'un tableau dans le TLACE-visualizer ou la gestion des layouts pour LTSA [8].

D'autres programmes graphiques existants ont inspiré ce travail comme le logiciel de conception de diagramme yEd avec sa vue montrant les prédécesseurs et successeurs [2]. Et pour finir, le logiciel GOAL (Graphical Tool for Omega-Automata and Logics) avec l'idée de pouvoir sélectionner et mettre en évidence les prédécesseurs et successeurs directement dans l'explication principale [1]. Il est aussi intéressant de remarquer que trois des quatre outils mentionnés ici ont été implémentés avec JUNG (yEd ne l'est pas).

Chapitre 3

Architecture

Au niveau de l'architecture du logiciel d'affichage graphique, elle est composée de deux parties : un module et le programme d'affichage en lui-même (voir figure 3.1). On entend par module, tout programme java qui veut utiliser l'interface graphique pour afficher une explication hiérarchique (voir section module simple et module interactif par exemple). C'est le module qui dirige l'échange. Tout d'abord, ce module doit créer un objet `HierarchicalExplanation` du modèle, il représente une explication hiérarchique avec tous ses états, transitions et sous-explications (voir section modèle ci-après). Il a aussi la possibilité d'implémenter un `MessageListener` afin d'écouter les messages que le logiciel graphique lui envoie en fonction des interactions de l'utilisateur.

Ensuite l'interaction d'un module avec le logiciel graphique se fait par la classe `API` qui nous fournit tous les outils pour remplir le modèle (avec l'explication hiérarchique préalablement créée), ouvrir une fenêtre (pour afficher l'explication graphiquement), permettre la mise à jour de l'explication avec une sous-explication ou d'envoyer et afficher les successeurs d'un état (dans le cadre d'une utilisation interactive de l'application pour les deux dernières options, voir section communication API).

Cette API va utiliser les composants de la partie vue (package view) pour afficher une explication hiérarchique. Il est aussi possible de la mettre à jour en fonction des demandes de l'utilisateur. Ceci est possible grâce au jungController qui manipule le modèle fourni par la vue afin d'obtenir des graphes JUNG qui peuvent ensuite être affichés. L'architecture du logiciel est de style MVC (Modèle - vue - contrôleur) avec la classe `API` qui pilote le tout. Les sections suivantes vont détailler ces trois parties.

Cette architecture possède un gros point positif, c'est la séparation entre le module et l'application visuelle. Cela nous permet de changer facilement le programme qui l'utilise et que celle-ci soit totalement indépendante. Cela nous permet de faire plusieurs modules de tests pour couvrir l'ensemble des fonctionnalités offertes par l'API comme une utilisation directe ou interactive avec nos deux modules présentés dans le chapitre sur les exemples.

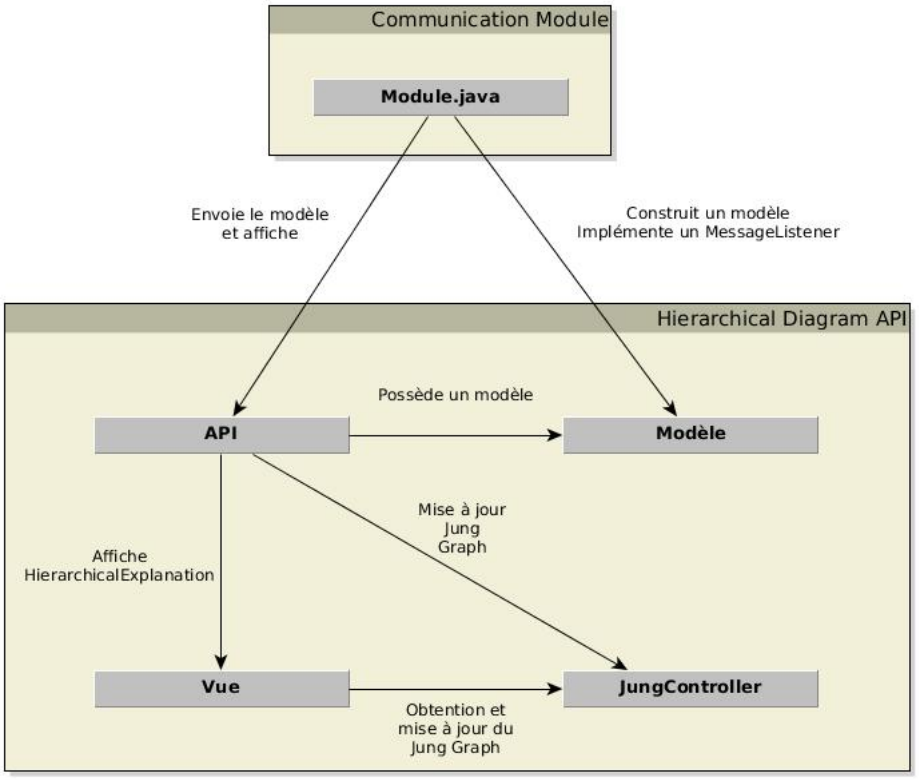


FIGURE 3.1 – Design du fonctionnement du logiciel.

3.1 Modèle

Le modèle est composé de cinq classes différentes afin de représenter les diagrammes hiérarchiques. La figure 3.2 présente ces cinq classes. Au niveau du choix des structures de données, celui-ci s'est porté sur des `ArrayList` pour les ensembles car elles sont simples d'utilisation et flexibles au niveau de la taille.

HierarchicalExplanation

Tout d'abord, nous avons la classe `HierarchicalExplanation` au sommet de la hiérarchie de la structure de données. C'est la classe principale qui contient toutes les autres. Celle-ci représente une explication dans son ensemble. Il possède un label qui permet de définir ce qui est expliqué au travers du graphe. Ensuite, une explication est composée principalement d'une liste d'états (classe `State`) et d'une liste de transitions (classe `Transition`). Au niveau de la hiérarchie, elle est définie de manière récursive. Un objet `HierarchicalExplanation` contient des états qui eux-mêmes contiennent des objets `HierarchicalExplanation` pour leurs sous-explications. Afin de garder une liaison entre une explication et son explication parente, on ajoute une transition initiale. L'état entrant de cette transition est soit nul s'il s'agit du graphique initial soit l'état parent de ce graphe. L'état sortant est quant à lui celui qui est attaché au nœud parent. Pour finir avec les explications hiérarchiques, il y a un booléen `isComplete` qui permet de savoir si une explication est remplie ou pas, ce qui va intervenir dans l'interactivité.

State

Premièrement, un état possède un identifiant qui est un `String` pour une plus grande généralité. Ensuite, il a aussi un attribut (voir attribute, utilisé pour la généralité, si on veut changer les informations contenues par un état), une liste de ses sous-explications, un lien vers l'explication à laquelle il appartient et enfin, un booléen permettant de savoir si ses successeurs ont déjà été calculés (il intervient au niveau de l'interactivité, voir communication API pour plus de détails).

Transition

Les transitions contiennent bien évidemment un lien vers un état entrant et un état sortant. Elles possèdent aussi un booléen qui nous permet de savoir si cette transition est initiale ou non (c'est-à-dire qu'elle lie un état et sa sous-explication ou alors simplement deux états d'une explication). Tout comme les objets `State`, les transitions contiennent un attribut qui permet de mettre des données sur celle-ci. Et finalement elles contiennent aussi un lien permettant de savoir à quelle explication elles appartiennent.

Attribute

La classe `Attribute` est utilisée par les transitions et les états pour contenir de l'information liée à ceux-ci. Pour ce faire, il contiennent une liste de variables et une liste de labels qui sont de simples `String`. On peut noter que cet classe contient une méthode `toString` qui peut être utilisée par JUNG pour afficher des étiquettes sur les nœuds et transitions.

Variables

C'est une classe très simple qui représente un couple valeur - variable. Il possède donc uniquement deux champs `String`.

3.2 Vue

C'est la partie la plus importante et la plus grosse du programme puisqu'il s'agit d'un logiciel graphique. Elle correspond à la partie vue de l'architecture. Celle-ci est divisée en cinq packages qui regroupent les panneaux de la fenêtre, les "listener", la souris, les "transformer" et le layout.

view

Ce package contient donc toutes les classes relatives à l'affichage général, la fenêtre en elle-même (voir figure 3.3). Nous avons donc une classe principale `GUI` qui utilise les autres classes qui sont des `JPanel`. C'est celle-ci qui est responsable d'avoir une fenêtre qui soit modifiable au niveau de la taille de ses composants (utilisation `JSplitPane`) ainsi que d'enregistrer les panneaux qui écoutent l'objet `ObservableList` qui contient les états sélectionnés. Cela permet de sélectionner des états dans la fenêtre et d'afficher des informations sur ceux-ci dans d'autres panneaux de l'affichage. De plus, en fonctionnant ainsi, cela permet d'ajouter facilement un nouveau panneau pour afficher de nouvelles données (qui peuvent être ajoutées facilement grâce aux attributs). Il nous suffit de créer un nouveau `JPanel` et de l'intégrer dans la `GUI` et il peut facilement recevoir l'information des nœuds

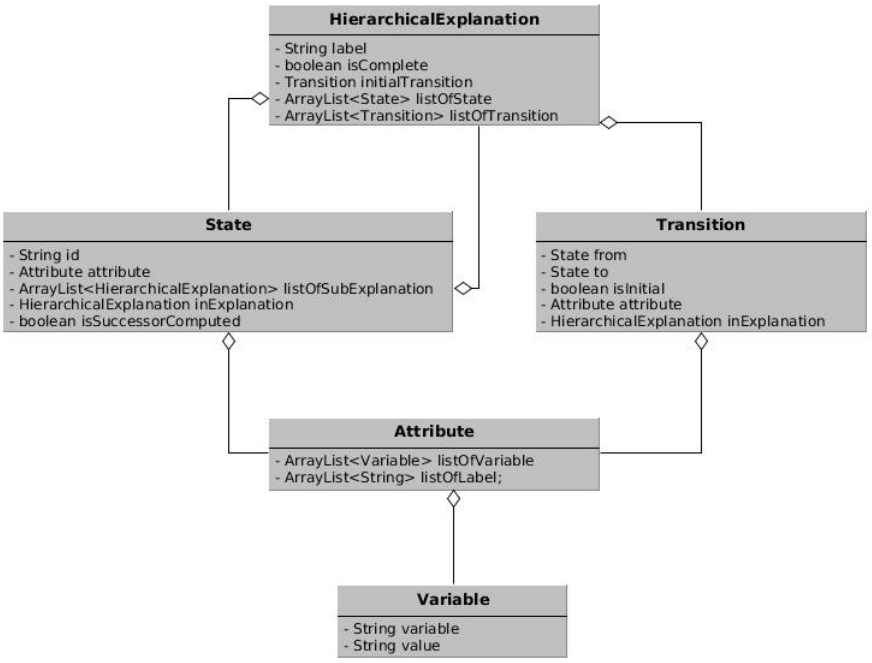


FIGURE 3.2 – Design du fonctionnement du logiciel.

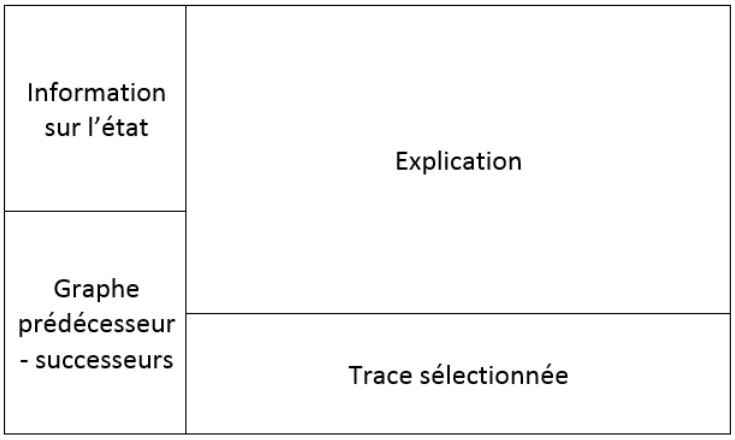


FIGURE 3.3 – Design de la fenêtre d'affichage d'une explication.

sélectionnés grâce à notre liste observable.

Ensuite nous avons les quatre panneaux qui composent la fenêtre :

- le `CenterPanel` pour l'affichage principal d'une explication.
- le `TableInfoPanel` qui permet de nous montrer une trace sélectionnée dans l'explication avec les informations contenues sur les états et transitions appartenant à cette trace. Ce panneau utilise l'outil `treeTable` pour afficher cette trace sous forme de tableaux.
- le `StateInfoPanel` avec lequel nous pouvons voir les informations relatives à un état qui est préalablement sélectionné sur l'explication principale.
- le `PredSuccPanel` qui affiche les prédécesseurs et successeurs du même état.

view.layout

C'est ici que l'on trouve les deux layouts implémentés, c'est-à-dire ceux qui ne sont pas fournis directement par JUNG. On trouve donc un layout hiérarchique qui permet d'afficher la structure principale, c'est-à-dire la décomposition en sous-explications sous forme d'arbre avec uniquement les transitions entre sous-explications qui apparaissent. Et un layout simple pour l'affichage des prédécesseurs et successeurs.

view.listener

Il contient toutes les classes permettant d'effectuer une action en fonction d'un événement particulier comme le `LayoutListener` qui change le layout de l'affichage d'une explication dans le `CenterPanel` en fonction du layout choisi dans la barre de menu. Il y a aussi un listener pour les checkboxs du tableau du `TableInfoPanel` afin que les bonnes branches de l'arbre soient expansées au bon moment, un pour le mode de sélection du graphe JUNG (comboMenu pour transforming ou picking), deux listeners pour l'affichage des labels sur les états et les transitions en fonction des checkbox sélectionnées dans le `TableInfoPanel` et enfin cinq qui permettent de gérer la navigation, interactive ou non, dans l'explication affichée sur le panneau principal.

view.mouse

Ce package comprend tout ce qui est lié aux événements de la souris comme le popup permettant la navigation entre les différents sous-graphes d'une explication. Nous avons aussi un plugin pour la sélection des états. Celui-ci permet d'avoir une trace grâce à l'algorithme de Dijkstra utilisé par JUNG et cette trace est mise dans une `ObservableList` qui permet aux autres panneaux d'afficher des informations en fonction des états sélectionnés.

view.transformer

Le framework JUNG utilise les "transformer" pour modifier l'apparence du graphe. Par exemple, changer la couleur d'un ou plusieurs nœuds, afficher les labels sur les transitions ou les états, le type de ligne utilisé pour une transition, la forme d'un nœud, ...

3.3 Jung Controller

Ce contrôleur contient deux types de méthodes. En premier, les deux méthodes qui fournissent un graphe JUNG à partir d'une explication hiérarchique. Une pour le graphe initial qui s'affiche à l'ouverture de la fenêtre et l'autre pour obtenir le graphique prédécesseur - successeur d'un état.

Ensuite il y a les méthodes qui permettent de mettre à jour l'affichage (`VisualizationViewer`), c'est-à-dire le graphe du panneau central en fonction des différentes interactions de l'utilisateur avec le graphe (afficher une sous-explication, le graphe parent d'un nœud, l'explication dans son entier, ajouter tous les successeurs d'un état, ...). Ces méthodes modifient le graphe JUNG en ajoutant ou retirant des états et des transitions.

Chapitre 4

Communication avec l'API graphique

Dans ce chapitre, nous allons voir comment un module peut utiliser et communiquer avec le logiciel graphique. Cette communication est importante pour l'utilisation interactive, lorsque l'on veut calculer notre explication au fur et à mesure que l'utilisateur l'explore. La première partie du chapitre introduit les cinq commandes de l'API, les trois dernières étant utiles uniquement pour une utilisation interactive et la deuxième partie va expliquer plus en détail comment fonctionne ce mécanisme permettant d'envoyer les données uniquement quand l'utilisateur les demande.

4.1 Utilisation API - Les 5 commandes

Tout d'abord, la première chose à faire pour pouvoir utiliser l'API d'affichage graphique d'une explication, c'est de générer celle-ci. En tout cas, au moins le début dans le cadre de l'utilisation de l'interactivité. Il faut donc commencer par construire un objet `HierarchicalExplanation` et le remplir d'états, de transitions ainsi que de sous-explications en fonction du problème traité. Ci-dessous, un exemple basique comprenant uniquement deux états et une transition, aucune sous-explication ni variable.

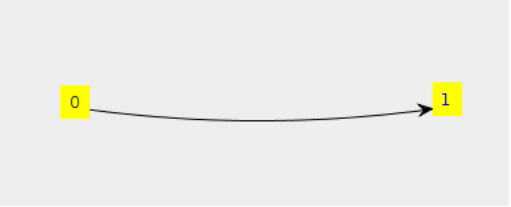


FIGURE 4.1 – Illustration du graphe généré.

```
HierarchicalExplanation he = new HierarchicalExplanation();

//set the State
State s0 = new State("0", null, null, he, true);
State s1 = new State("1", null, null, he, true);

ArrayList<State> listOfState = new ArrayList<State>();
```

```
listOfState.add(s0);
listOfState.add(s1);
he.setListOfState(listOfState);

//set the Transition
Transition t1 = new Transition(s0, s1,false, null, he);

ArrayList<Transition> listOfTransition = new ArrayList<Transition>();

listOfTransition.add(t1);
he.setListOfTransition(listOfTransition);

he.setComplete(true);
```

La deuxième étape consiste simplement à mettre cette structure hiérarchique fraîchement créée dans le logiciel graphique au moyen d'une commande simple (Les arguments des cinq commandes sont à titre indicatif pour montrer les paramètres dont elles ont besoin).

```
API.setModel(HierarchicalExplanation he);
```

Ensuite, il est alors temps de passer à l'étape présentant l'explication hiérarchique de façon visuelle. Cette commande ouvre la fenêtre de visualisation. On peut se contenter de ces deux commandes pour une utilisation directe du programme. On entend par directe, un module qui aurait calculé toute son explication hiérarchique et qui ne voudrait donc pas envoyer des informations au fur et à mesure.

```
API.displayGraph();
```

Afin d'utiliser l'interactivité, il faut écouter les besoins de l'application visuelle en données. Pour ceci, il faut créer un `MessageListener` qui va recevoir tous les messages et c'est lui qui est responsable des actions à entreprendre en fonction du message reçu.

```
API.setMessageListener(MessageListener m);
```

Nous avons une méthode qui sert à afficher une sous-explication une fois que celle-ci a été complétée. Elle intervient donc dans le cadre d'un calcul interactif d'une explication et permet au module de dire à l'API qu'il a fini de calculer la suite et qu'il peut donc l'ajouter à l'affichage. Cette méthode s'utilise donc dans un `MessageListener` en réponse à une demande afin de fournir la suite de l'explication (voir section fonctionnement de l'interactivité pour plus de détails).

```
API.displaySubExplanation(HierarchicalExplanation subExplanation);
```

Et pour finir, nous avons la possibilité d'envoyer les successeurs d'un état, dans une même explication ainsi que ses sous-explications. Cette méthode s'utilise également dans le MessageListener (voir section fonctionnement de l'interactivité pour plus de détails).

```
API.setSuccessors(
        State state,
        ArrayList<Transition> listOfSuccessors,
        ArrayList<HierarchicalExplanation> listOfSubExplanation);
```

4.2 Fonctionnement de l'interactivité

Tout d'abord, il faut comprendre le cadre du fonctionnement et ce que l'on recherche comme interactivité. Il existe deux cas différents, deux demandes différentes provenant de l'API. Dans le premier cas, on veut pouvoir afficher le graphe d'une explication et calculer une sous-explication seulement si celle-ci est demandée par l'utilisateur lors de sa navigation dans le graphe. Ensuite, dans le second cas, on veut pouvoir expanser le graphe d'une façon encore plus incrémentale, uniquement les successeurs d'un nœud. Afin de réaliser ceci, il nous faut donc un mécanisme pour écouter les besoins de l'application graphique lorsque celle-ci demande de l'information supplémentaire. C'est exactement ce que fait le `MessageListener` à l'intérieur de l'`interactivityComponent`. Commençons par l'explication de ce dernier, ensuite nous allons voir comment organiser un `MessageListener` pour traiter les deux types de demandes et pour finir, nous allons discuter de la manière d'informer le logiciel graphique de l'existence de ces données supplémentaires.

4.2.1 InteractivityComponent

Le module ne touche pas à ce composant mais c'est lui qui enregistre le `MessageListener` que le module envoie à l'application graphique. Il s'agit d'un objet contenant un string pour passer des messages, un `State` pour savoir quel état est concerné par la demande d'information et enfin une `HierarchicalExplanation` à compléter dans le cadre de demande d'une sous-explication.

Lorsque l'utilisateur demande des informations dont le logiciel graphique connaît l'existence mais qu'elles n'ont pas encore été calculées, le logiciel graphique change les informations du composant interactif ce qui entraîne un envoi de ces données (le message, l'état et la sous-explication) à tous ses listeners. Pour compléter ce mécanisme d'écoute, ce composant permet donc l'ajout de `MessageListener` qui sont des `EventListener` et permettent donc de savoir si l'API a envoyé un message.

4.2.2 Organisation/utilisation MessageListener

Il faut définir un `MessageListener` qui reçoit les messages de l'API et effectue certaines actions en fonction du message. Lorsque c'est une demande de sous-explication, il faut remplir la sous-explication fournie et afficher ces nouvelles données. Pour une expansion des successeurs d'un état, il faut renvoyer les successeurs dans la même explication que l'état concerné ainsi qu'une liste de ses sous-explications puisqu'il s'agit aussi de ses successeurs.

```
MessageListener listener = new MessageListener() {

   public void listenMessage(String newMessage,State state,
       HierarchicalExplanation subExplanation) throws IOException {

      //forme du message : SubExplanation/formule de la sous-explication demandee
      if(newMessage.contains("SubExplanation")){

         // TODO : remplir la sous-explication.
         fill(subExplanation);

         // notifier l API et afficher
         API.displaySubExplanation(subExplanation);
      }
      //forme du message : Expand
      else if (newMessage.contains("Expand")) {

         // TODO : calculer les successeurs.
         computeSuccessor(listOfSuccessors);
         computeSubExplanation(listOfSubExplanation);

         // renvoyer les successeurs
         API.setSuccessors(state,listOfSuccessors,listOfSubExplanation);
      }
      else {
         System.out.println("Message not understand.");
      }
   }
};
```

Et pour finir, il ne faut pas oublier la commande parmi les cinq correspondant à l'ajout de ce listener à notre `InteractivityComponent` dans l'API.

```
API.setMessageListener(MessageListener m);
```

4.2.3 Existence d'informations supplémentaires

Afin de pouvoir demander de l'information en plus, il faut que le logiciel graphique sache qu'il en existe. C'est ici qu'intervient le booléen *isComplete* pour les sous-explications. Il nous permet de savoir si une explication est déjà remplie ou s'il faut envoyer un message pour demander de la remplir. Donc, pour un module, la manière de dire au logiciel graphique qu'il existe une sous-explication mais qu'il veut la calculer plus tard est de déjà l'insérer dans son parent en mettant *isComplete* à false.

Pour noter la présence de successeurs pas encore présents dans le graphe, on utilise un mécanisme similaire. Il s'agit d'un booléen *isSuccessorComputed* présent dans l'état lui-même. Il sera mis à true une fois que le module aura envoyé sa réponse avec les successeurs.

On peut utiliser ces 2 mécanismes ensemble. Par exemple, lorsque l'on demande une sous-explication, on renvoie celle-ci en disant qu'elle est complète mais en mettant uniquement

un état pas encore calculé qu'on pourra expanser. Dans l'autre sens, lorsque l'expansion nous renvoie des sous-explications, il faut que celles-ci contiennent au moins l'état initial afin d'afficher la nouvelle transition mais on peut soit laisser uniquement cet état pas encore calculé ou alors directement renvoyer l'explication entière.

Chapitre 5

Exemple de module : simple et interactif

Dans ce chapitre, nous allons voir deux exemples de modules qui utilisent le logiciel graphique. Comme expliqué dans la partie architecture, le logiciel graphique fournit uniquement des commandes pour mettre, afficher et compléter une explication hiérarchique. C'est donc un module (un programme java) qui pilote le tout avec aussi le calcul d'une explication. Il gère toutes les données et le logiciel nous permet de les afficher. Ici nous avons donc deux modules qui génèrent des explications à partir de PyNuSMV et les affichent. Un qui travaille de façon direct et récupère toute l'explication en une fois et un qui permet de tester les fonctions interactives de l'application.

5.1 Architecture

L'architecture est la même pour les deux exemples (voir figure 5.1). Nous avons une classe Java qui lance un process Python. Ils communiquent entre eux par le biais d'un stream texte, comme on le ferait en lançant le programme Python dans un terminal. Le process Python calcule donc les explications grâce à PyNuSMV et écrit celles-ci sur sa sortie standard. Le programme java lit ce que Python lui envoie et effectue un parsing, en fonction de ce qu'il reçoit, il crée une `HierarchicalExplanation` de l'application graphique et peut l'afficher grâce à l'API. Cette communication, parsing de l'explication entre Python et java sera détaillée plus tard vu qu'elle diffère dans nos deux exemples d'utilisation, nous allons commencer par la génération d'une explication par PyNuSMV qui elle, est commune .

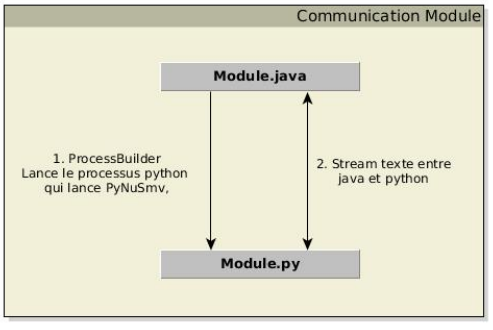


Figure 5.1 – Architecture des modules utilisant PyNuSMV

5.2 Génération d'une explication

Tout d'abord le process Python demande le choix d'un modèle dans lequel on va vérifier si une propriété est vraie ou non. Les lignes qui commencent par "process : " viennent de Python et sont transmises sur la sortie sortie standard java pour que l'utilisateur puisse choisir un modèle en tapant simplement le numéro de celui-ci suivi de Enter. Ce numéro sera écrit sur le stream texte et ainsi envoyé à Python. Ces lignes sont donc visibles par l'utilisateur sur la console java.

Ensuite il propose de choisir une propriété que l'on veut vérifier sur le modèle choisi. Ce choix s'effectue de la même manière que pour le modèle. Voici ci-dessous un exemple de cette communication pour le modèle des deux compteurs.

```
process : 1. Models.counters
process : 2. Models.cardgame
process : 3. Models.cardgamePostFair
process : 4. Models.transmission
process : 5. Models.transmissionPostFair
Choose an option ...
// 1
You chose : 1
process : You chose Models.counters.
process : 1. EF c1.c = 2
process : 2. (EF c1.c = 2 & EF c2.c = 2)
process : 3. EF (c1.c = 2 & c2.c = 2)
process : 4. AG c1.c <= 2
process : 5. EFf True
process : 6. AG EF c1.c = 0
process : 7. EF K<c1>c1.c = 2
process : 8. AG (c1.c = 2 -> K<c1>c1.c = 2)
process : 9. EF K<c1>c2.c = 2
process : 10. <c1> F c1.c = 2
process : 11. <c1> Ff c1.c = 2
process : 12. <c1>i Ff c1.c = 2
process : 13. <c1c2>i Ff (c1.c = 2 & c2.c = 2)
Choose an option ...
// 1
You chose : 1
process : You chose EF c1.c = 2.
process : Checking the property...
process : The property is True.
```

En réponse à ce choix de propriété, PyNuSMV effectue trois grosses actions. Tout d'abord, il construit le modèle. Ensuite, il calcule, vérifie si la propriété choisie est vraie et pour finir, il génère une explication, à partir d'un état initial qui satisfait la propriété ou sa négation selon la réponse de l'étape précédente, du pourquoi cet état satisfait cette propriété (voir chapitre Contexte pour plus de détail sur PyNuSMV).

Cette explication a une structure précise pour stocker les états, transitions et sous-explications. L'étape suivante est donc d'imprimer cette explication sur le stream texte selon un format fixé afin que le programme java puisse la réceptionner et la transformer en

une explication hiérarchique de l'application graphique. Et c'est donc ici que la différence se fait entre le module simple et interactif.

Voici la structure d'une explication fournie par Python :

— expl.States : un ensemble d'état.

— expl.relations : un dictionnaire avec pour clé le nom de la relation et pour valeur un ensemble de paires d'état.

— expl.labelling : un dictionnaire état - ensemble de labels.

— expl.subexplanations : un dictionnaire avec comme clé un état et comme valeur un ensemble de paire formule - sous-explication.

5.3 Module Simple

Cet exemple a une communication moindre, que ce soit au niveau du module en lui-même entre le programme principal Java et PyNuSMV ou alors au niveau des interactions entre le module et l'API graphique étant donné qu'ici on traite l'explication dans son entier en une fois et qu'on n'utilise pas l'interactivité fournie par l'interface graphique. Après avoir généré l'explication, la seule chose à faire est de lire ce que Python écrit dans le stream texte et de le traduire en une explication hiérarchique du modèle de l'application graphique.

On peut trouver ci-dessous un exemple de parsing (pour la propriété numéro 1, $EFc1.c = 2$, du modèle des compteurs), ce que Python va écrire dans le stream texte. Ici, ce n'est donc pas visible par l'utilisateur, c'est le module java qui réceptionne ces lignes et effectue les actions adéquates en fonction de ce qu'il lit. C'est-à-dire la création de listes d'états, de transitions, de sous-explications mais aussi d'attributs avec des variables et de rajouter le tout dans une explication hiérarchique afin de garder la même structure que celle que l'on reçoit. La tabulation a été rajoutée uniquement pour avoir une meilleure compréhension des différentes explications imbriquées l'une dans l'autre.

Tout d'abord, on commence par dire que c'est le début d'un graphe et ensuite on donne ses états (d'abord le nombre suivi des états avec leurs variables). Après ceci, on énumère les transitions s'il y en a. On commence par mentionner le nombre de transitions dans cette explication afin de savoir combien de fois boucler sur le reste. Puis le nombre de labels pour ce type de transition (puisque qu'un type de transition peut être représenté par plusieurs labels) suivi de ceux-ci. Et enfin, le nombre de transitions de ce type suivi des paires d'états.

Et pour finir les sous-explications en mentionnant l'état parent. Lorsqu'il y a une ligne "Label", cela indique que ce qui suit est une explication de l'état mentionné juste en dessous avec la ligne d'après étant le label, la formule de l'explication. On écrit directement l'explication en entier après. Ce parcours de la hiérarchie est donc un parcours en profondeur (DFS - Depth First Search) puisqu'on explore directement jusqu'au fond de la branche en développant directement une sous-explication lorsqu'on en rencontre une, le développement des autres branches (s'il y a plusieurs sous-explications pour un même état ou alors pour des états différents) se faisant après. D'où le fait de bien remettre la commande "Label - état - formule" avant chaque développement de sous-explication (Voir les annexes pour des exemples plus complexes avec plusieurs sous-explications).

Les commentaires sont présents uniquement à titre indicatif, ils n'apparaissent pas réellement lors du parsing.

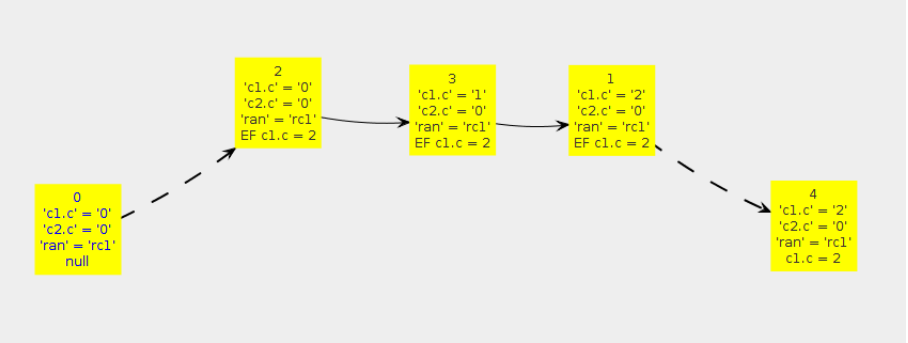


FIGURE 5.2 – Illustration du graphe généré par cet exemple de parsing.

```
Generating an explanation...
Graph- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f39d04c79b0>
States
1     // Nombre d'états
0-{'c1.c': '0', 'c2.c': '0', 'ran': 'rc1'}
Transitions
0     // Nombre de transitions.
SubExplanations
Set of expl of state
0     // Etat possédant une ou plusieurs sous-explications.
Label // Enumération des sous-explications.
0     // On remet l'état parce que c'est ici qu'on crée la sous-explication et de
    cette façon on sait toujours quel état possède la sous-explication décrite
    juste après.
EF c1.c = 2
<tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f39d04c7be0>
      Graph- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
          0x7f39d04c7be0>
      States
      3     // Nombre d'états
      1-{'c1.c': '2', 'c2.c': '0', 'ran': 'rc1'}  // Enumération des états.
      2-{'c1.c': '0', 'c2.c': '0', 'ran': 'rc1'}
      3-{'c1.c': '1', 'c2.c': '0', 'ran': 'rc1'}
      Transitions
      1     // Nombre de types différents de transitions.
      1     // Nombre de label décrivant ce type.
      time
      2     // Nombre de transitions de ce type.
      2/3     // Enumération des transitions.
      3/1
      SubExplanations
      Set of expl of state
      1
      Label
      1
```

```
     c1.c = 2
     <tools.mucalculus.explanation.Explanation object at 0x7f39d04c7a58>
          Graph- <tools.mucalculus.explanation.Explanation object at
              0x7f39d04c7a58>
          States
          1
          4-{'c1.c': '2', 'c2.c': '0', 'ran': 'rc1'}
          Transitions
          0
          SubExplanations
          No more graph
          End- <tools.mucalculus.explanation.Explanation object at
              0x7f39d04c7a58>
     End- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
         0x7f39d04c7be0>
End- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f39d04c79b0>
```

5.4 Module Interactif

Ce module est plus complexe en terme d'interactions. Cela commence avec une interaction entre le module et l'API graphique comme présenté dans le chapitre sur la communication et ensuite cette communication est répercutée entre le programme Java du module et Python qui fournit l'explication d'une propriété.

Comme dit précédemment, le départ est le même avec les choix d'une propriété et ensuite la génération d'une explication. Ce qui diffère dans ce cas-ci, c'est le fait que Python envoie uniquement la première explication (le nœud initial qui satisfait la propriété en fait) sans envoyer les sous-explications en entier, en envoyant uniquement l'état parent et le label afin que le programme java du module puisse initialiser ces sous-explications, ce qui permet à l'API graphique de savoir qu'elles existent et donc de les demander si l'utilisateur veut les voir en manipulant le graphe. Voici ce que Python envoie donc lorsqu'on lui demande une explication en mode interactif : uniquement l'explication initiale (C'est le même exemple que pour le module simple).

```
Generating an explanation...
Graph- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f16f4c90a90>
States
1
0-{'c1.c': '0', 'ran': 'rc1', 'c2.c': '0'}
Transitions
0
Set of expl of state
0
1
EF c1.c = 2
End- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f16f4c90a90>
What id/formula or Expand/id ?
```

Cependant, dans ce cas-ci, PyNuSMV se met en attente d'instruction pour envoyer la suite de l'explication. Suite à la réception d'une explication, le module java lance le programme graphique avec ce graphe initial. Comme c'est expliqué dans le chapitre sur la communication avec l'API, il existe deux types de demandes que le logiciel peut faire : soit une sous-explication, soit une expansion de nœud.

Lorsque l'utilisateur demande une sous-explication, le module java reçoit le message grâce au `MessageListener` de cette demande et écrit sur le stream texte ce qu'il veut. Ce à quoi Python répond en envoyant l'explication complète après l'avoir trouvée en cherchant dans l'explication que PyNuSMV a calculée. La ligne en commentaire correspond à ce que le module va écrire dans le stream pour avoir cette réponse. C'est donc bien le programme Java qui s'occupe d'écrire cette ligne. Cette explication peut être décortiquée de la même manière que dans le module simple à part pour les sous-explications, on ne le développe pas mais on mentionne quand même l'état parent et le label afin de montrer leur existence.

```
What id/formula or Expand/id ?
// 0/EF c1.c = 2
Graph- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f16f4c90cc0>
States
3
1-{'c1.c': '2', 'ran': 'rc1', 'c2.c': '0'}
2-{'c1.c': '0', 'ran': 'rc1', 'c2.c': '0'}
3-{'c1.c': '1', 'ran': 'rc1', 'c2.c': '0'}
Transitions
1
1
time
2
3/1
2/3
Set of expl of state
1
1
c1.c = 2
End- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f16f4c90cc0>
```

De plus, lorsque la demande est une expansion de nœud, le schéma est le même au niveau des messages échangés mais cependant la réponse ne doit pas être interprétée de la même façon. Nous avons donc plusieurs méthodes différentes qui lisent le stream dans le module java et traduisent l'information en explication hiérarchique ou en partie d'explication. Il y en a une pour recevoir une explication complète, une pour recevoir les successeurs et pour finir, une pour réceptionner les sous-explications dans le cadre d'une expansion, c'est à-dire avec uniquement l'état initial de cette sous-explication. Selon la demande qui est faite à Python par le stream texte, le module appellera soit la première méthode, soit les deux suivantes l'une après l'autre.

```
// Expand/2
1
time
3-{'c1.c': '1', 'ran': 'rc1', 'c2.c': '0'}
No more graph
End expand state
End expand explanation

// Expand/1
End expand state
c1.c = 2
4-{'c1.c': '2', 'ran': 'rc1', 'c2.c': '0'}
No more graph
End expand explanation
```

Voici deux exemples de réponses lors d'une demande d'expansion de nœud. De nouveau, le commentaire est l'instruction envoyée à Python. Dans le premier cas, il n'y a qu'un état dans la même explication qui est un successeur de l'état 2. On a donc le nombre de labels pour le type de transition suivi de ces formules. Ensuite nous avons l'état atteint par la transition avec ses variables et pour finir, le nombre de sous-explications suivi par les labels de celle-ci. On utilise la méthode de lecture de ces transitions à l'intérieur d'une même explication jusqu'à ce qu'on lise "End expand state".

Pour la deuxième instruction "Expand/1", c'est l'inverse, il n'y pas de successeur dans l'explication même mais l'état numéro 1 possède une sous-explication et donc un successeur qui est l'état initial de celle-ci. Python envoie ceci : d'abord la formule de l'explication et ensuite l'état initial avec ses variables suivi par ses sous-explications et ainsi de suite jusqu'à la lecture de "End expand explanation".

5.5 Remarque : simulation de l'interactivité

Dès lors, il apparaît évident que la seule différence entre nos deux exemples d'utilisation est que dans le premier cas, on envoie directement toute l'explication alors que dans le second, on envoie uniquement l'information qui est demandée. Ce qui nous amène à dire que l'on simule la génération incrémentale de l'explication mais que l'interactivité, elle, n'est pas simulée. En réalité, l'explication a été calculée dans son entièreté directement par PyNuSMV. On simule donc un calcul au fur et à mesure en envoyant par morceaux l'explication afin de tester les possibilités interactives de l'application graphique. L'interactivité était une fonction nécessaire de l'outil et donc cette simulation nous permet de la tester.

Chapitre 6

Logiciel - Fonctionnalités

Dans ce chapitre, nous allons voir toutes les fonctionnalités offertes par l'application visuelle. Pour ce faire, on va regarder panneau par panneau ce qu'il est possible de faire et surtout de voir. La figure 6.1 nous montre une vue de la fenêtre d'affichage composée des quatre panneaux. Sur la gauche, nous avons en haut l'information sur un état et en bas le panneau de voisinage. Et sur la droite, nous avons en haut l'explication et en dessous une trace.

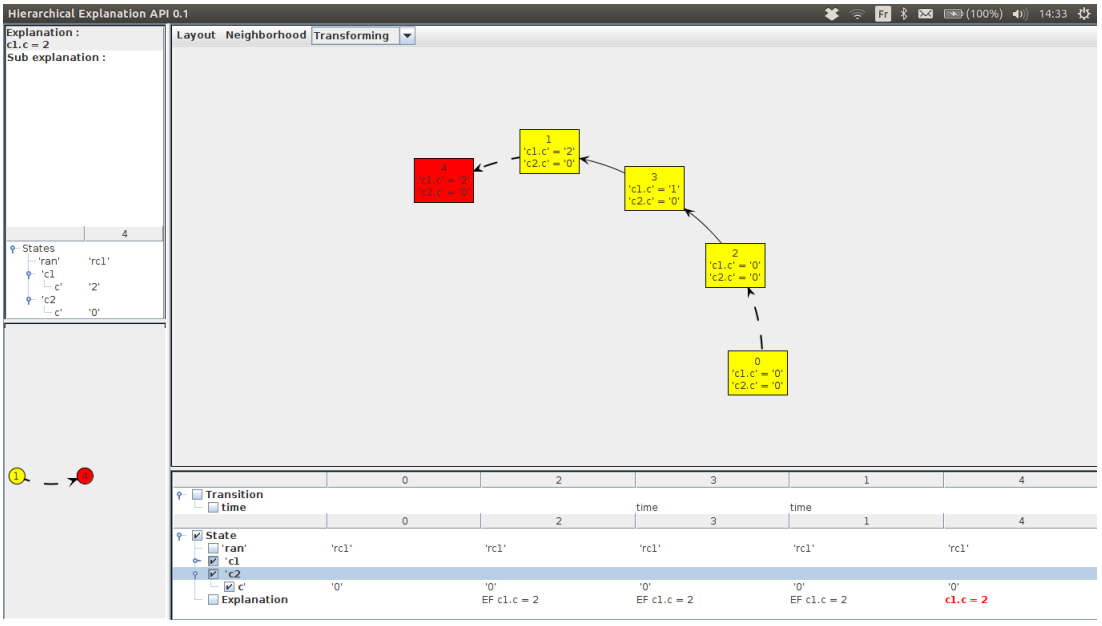


FIGURE 6.1 – Les 4 panneaux d'affichage.

6.1 Panneau central, l'explication

C'est dans cette vue principale que l'explication en elle même est affichée. Nous avons des possibilités de disposition du graphe, de zoom/dézoom, de mouvement, ...

6.1.1 Layout

Pour commencer, il y a les layouts qui permettent de disposer de différentes manières le graphe. Ici, on travaille sur la position des nœuds et des transitions afin de mieux voir

ce qui se passe. On peut aussi mentionner le fait que l'on peut appliquer ce layout sur l'entièreté de l'explication ou alors sur une sélection de nœuds (voir mode Picking). Il existe cinq layouts différents dans le programme, les quatre premiers étant fournis par JUNG.

KamadaKawai

Ce layout suit l'algorithme de Kamada-Kawai et est un force-based layout. On essaye de minimiser le nombre d'arêtes qui se croisent et d'avoir une distance équivalente entre tous les nœuds. Ce layout permet de bien disposer les états et de voir ce qu'il se passe.

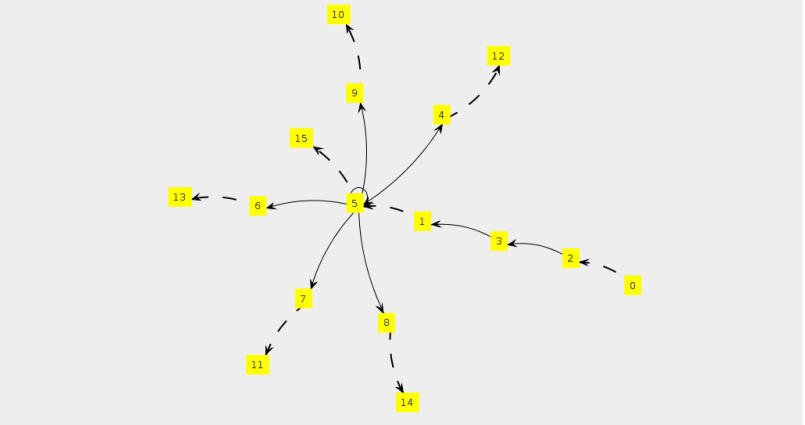


FIGURE 6.2 – KamadaKawai layout.

FruchtermanReingold

Il s'agit également d'un force-based layout. Ici encore, on parle de minimisation de croisements de segment et d'une distance équivalente entre les états mais on rajoute le fait de vouloir occuper l'espace disponible.

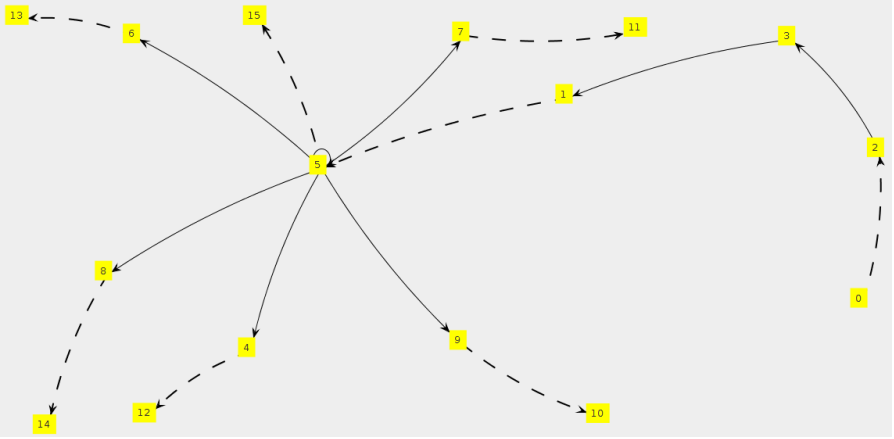


FIGURE 6.3 – FruchtermanReingold layout.

ISOM

L'ISOM (inverted self-organizing maps) layout inspiré de "self-organizing graph methods" de Bernd Meyer essaye de générer une distribution uniforme des nœuds afin de remplir l'espace disponible.

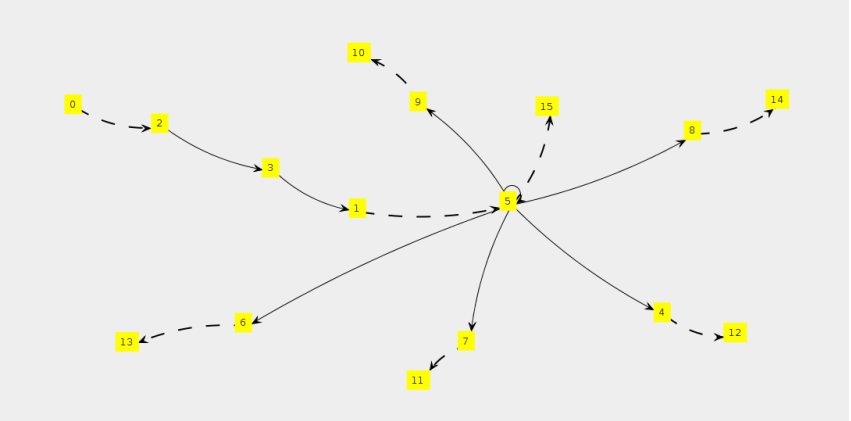


FIGURE 6.4 – ISOM layout.

Circle

Ce layout place les états sur un cercle à même distance mais n'essaye pas de minimiser les croisements d'arêtes. Il est moins intéressant.

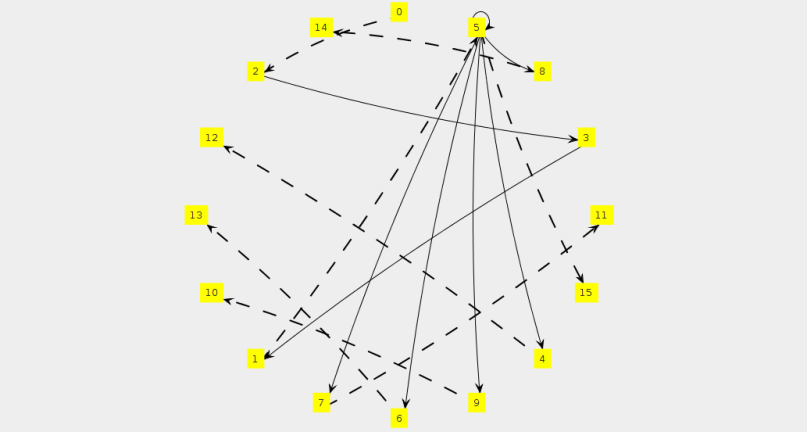


FIGURE 6.5 – Circle layout.

Hierarchical

Ce layout nous permet de voir la structure générale d'une explication. C'est-à-dire que les transitions entre états d'une même explication n'apparaissent pas, uniquement celles entre explications apparaissent. Nous pouvons donc voir la hiérarchie apparaître avec les liens entre les différentes explications et sous-explications. Cette hiérarchie est mise sous-forme d'arbre. Ce layout fonctionne avec deux paramètres, une profondeur et une largeur.

On effectue un parcours en profondeur (DFS - depth first search) avec des appels récursifs pour chaque sous-explication rencontrée qui permettent de garder et augmenter à chaque palier la variable pour la profondeur. Pour la variable de la largeur, celle-ci est globale et est incrémentée lors du retour de l'appel récursif si l'explication possède plus qu'une sous-explication.

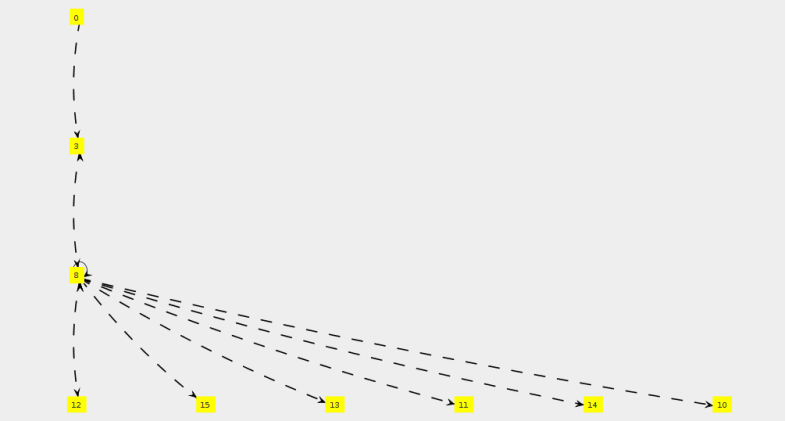


FIGURE 6.6 – Hierarchical layout.

6.1.2 Navigation

On entend par navigation le fait de se déplacer dans l'explication hiérarchique : ajouter/-retirer une sous-explication, ajouter l'explication parente, afficher uniquement une sous-explication, expanser les successeurs d'un état, ... Les options qui ne sont pas accessibles, si les informations sont déjà présentes, sont grisées et ne sont plus cliquables. On peut aussi noter que les traits pleins correspondent aux transitions d'une explication alors que les traits pointillés signifient une transition entre un état et sa sous-explication.

Il est important aussi de spécifier que l'utilisateur de l'interface graphique ne voit pas si une partie du graphique n'est pas encore calculée. Sa navigation n'en sera pas affectée, seul le calcul derrière changera. En effet, si c'est le cas, le logiciel continue d'être réactif et tout ce que l'utilisateur pourrait voir, c'est un délai dans le temps de réaction à afficher l'information demandée. Une manière pour lui de voir ce qui est dans l'explication est d'utiliser le click droit dans le cadre sans être sur un nœud et d'utiliser l'option "Show the entire explanation". Il pourra voir tout ce qui est actuellement calculé, présent dans l'explication hiérarchique. On peut diviser cette navigation en trois parties à part l'option pour afficher l'explication entière, les sous-explications, les parents et les successeurs.

Pour les sous-explications, on peut les afficher ou les cacher comme bon nous semble grâce à un click droit sur un état qui nous présente toutes les sous-explications s'il y en a. Ce label de sous-explication mène à la possibilité de faire "Add/Remove" selon ce qui est déjà affiché ou bien l'option "Only" qui permet d'afficher uniquement cette explication-là.

Au niveau de l'affichage de l'explication parente d'un nœud, à nouveau on obtient un popup avec un click droit sur un état avec la possibilité soit d'ajouter le parent s'il n'est pas encore présent sur le panneau ou alors d'afficher uniquement le parent avec l'option "Only".

L'option "Expand successors" permet d'expanser tous les successeurs d'un état, qu'il soit déjà calculé ou non, que ce soit des successeurs dans la même explication ou alors l'état initial d'une sous-explication de cet état.

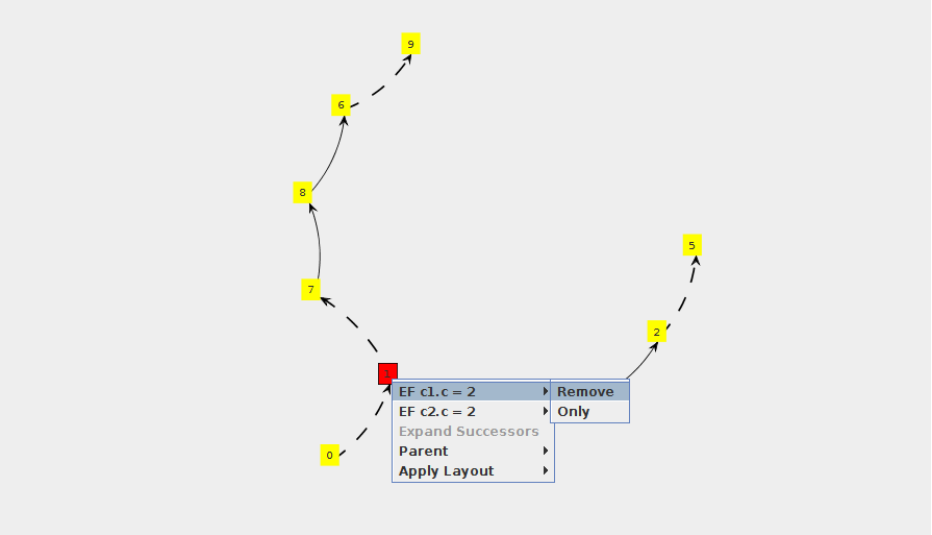


FIGURE 6.7 – Navigation popup menu.

6.1.3 Voisinage

Il existe une option dans la barre de menu qui s'appelle "Neighborhood". Celle-ci fait apparaître deux cases à cocher si l'on veut mettre en évidence (ils seront en vert et les autres états et transition en gris clair) les prédécesseurs et successeurs de l'état sélectionné. Les couleurs peuvent être changées dans les *Settings* de l'application graphique.

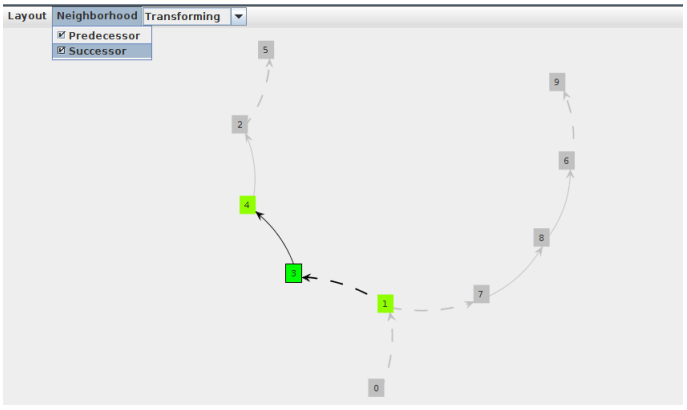


FIGURE 6.8 – Neighborhood dans la barre des menu.

6.1.4 Combo Menu

Il s'agit du menu déroulant dans la barre qui propose les options "Transforming" et "Picking" qui sont des modes pour la souris de JUNG [4]. Une chose à spécifier, c'est que l'application d'un layout sur certains états se fait grâce à la sélection de nœuds du Picking Mode.

Picking Mode :

— SourisBouton1 click sur un nœud pour le sélectionner.

— SourisBouton1+Shift click sur un nœud pour l'ajouter à la sélection.

— SourisBouton1+mouvement sur un nœud pour bouger la sélection.

— SourisBouton1+mouvement pour sélectionner les nœuds dans un rectangle.

— SourisBouton1+Shift+mouvement pour ajouter à la sélection les nœuds dans un rectangle.

Transforming Mode :

— SourisBouton1+mouvement pour bouger l'affichage.

— SourisBouton1+Shift+mouvement pour effectuer une rotation de l'affichage.

— SourisBouton1+ctrl(or Command)+mouvement pour cisailler l'affichage.

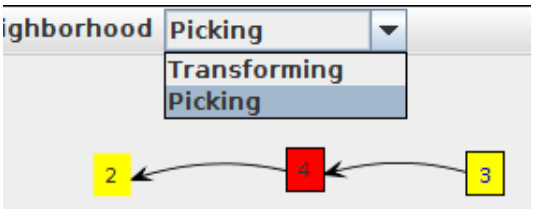


FIGURE 6.9 – Picking ou Transforming Mode + sélection d'un état

Dans les deux modes, il reste possible de sélectionner un état ou une trace. On a toujours la possibilité de cliquer sur un nœud qui devient alors rouge. Pour les traces, il faut alors cliquer sur un autre nœud et s'il existe un chemin (cherché avec l'algorithme de Dijkstra), tous les nœuds faisant partie de cette trace ont un bord noir et le dernier cliqué reste bien rouge. De plus, ce chemin se retrouve dans la liste observable des états sélectionnés.

6.2 Panneau d'information, un état

La figure ci-dessous nous présente ce panneau d'affichage sur la gauche qui contient les informations concernant l'état sélectionné en rouge. On y retrouve l'explication à laquelle il appartient, la liste de ses sous-explications et pour finir, un tableau avec un arbre pour afficher/cacher les variables de cet état. De base, cet arbre expanse automatiquement tous ses nœuds afin de voir toute l'information.

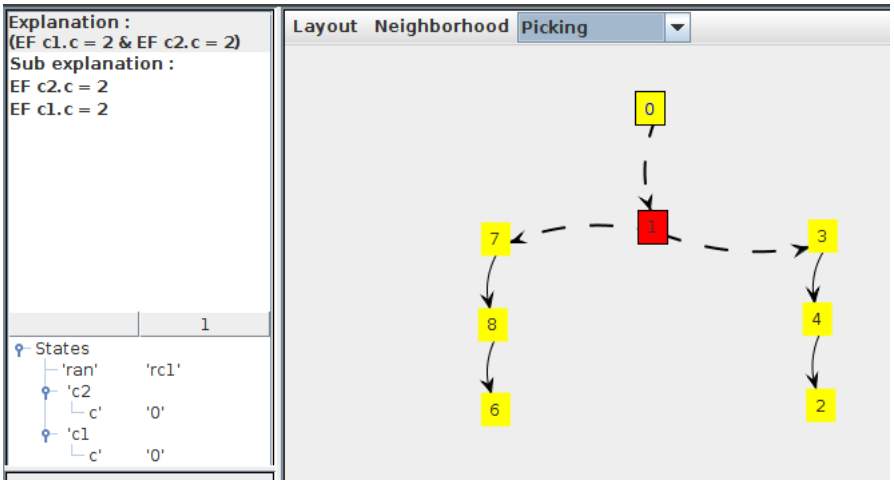


FIGURE 6.10 – Information sur un état.

6.3 Panneau de voisinage, les prédécesseurs et successeurs

Dans cette vue, on voit simplement les prédécesseurs et successeurs du nœud sélectionné dans le panneau principal. De nouveau, il affiche ceux qui sont dans la même explication aussi bien que les transitions entre deux sous-explications. Le code couleur et les traits sont les mêmes que dans la fenêtre principale. Ce graphe possède son propre layout qui affiche en ligne sur la gauche les prédécesseurs, au milieu l'état concerné et enfin à droite les successeurs.

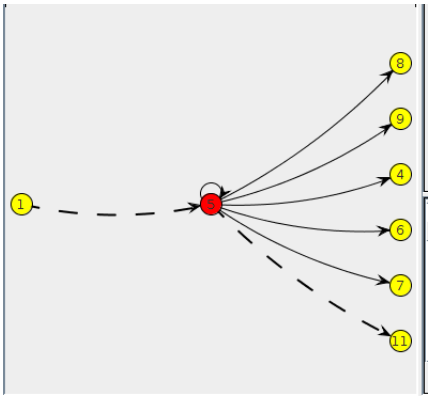


FIGURE 6.11 – Graphe prédécesseurs et successeurs.

6.4 Tree Table, une trace

Ce panneau est très important. Il nous permet d'analyser une trace dans une explication. La sélection du path se fait grâce à l'algorithme de Dijkstra comme déjà mentionné. Cette trace représente une exécution possible dans le problème donné, une succession d'états et de transitions. Le chemin se présente sous la forme de deux tableaux avec un arbre pour les variables qui permet d'afficher/cacher lorsqu'il y a une hiérarchie dans celle-ci.

Le premier tableau représente les transitions présentes le long du chemin. Le type de transition est placé dans la case de l'état d'arrivée de la transition.

Le deuxième tableau nous fournit les informations sur les variables des états présents dans la trace. Lorsqu'il y a un changement de variable d'un état à l'autre, celle-ci apparaît en rouge pour permettre une bonne visualisation de ce qui bouge le long du chemin.

La lecture se fait donc normalement, d'abord du haut vers le bas et puis de gauche à droite colonne par colonne. C'est donc pour ceci que les labels des transitions sont positionnés sur l'état d'arrivée. Par exemple, dans la figure ci-dessous, on peut lire la colonne 2, voir les variables de cet état et ensuite passer à la colonne 3. Dans celle-ci, on voit que la transition est de type "time" et ensuite que cela a changé la variable 'c1.c' en '1' au lieu de '0'. On peut continuer la lecture de la trace comme ceci jusqu'au bout.

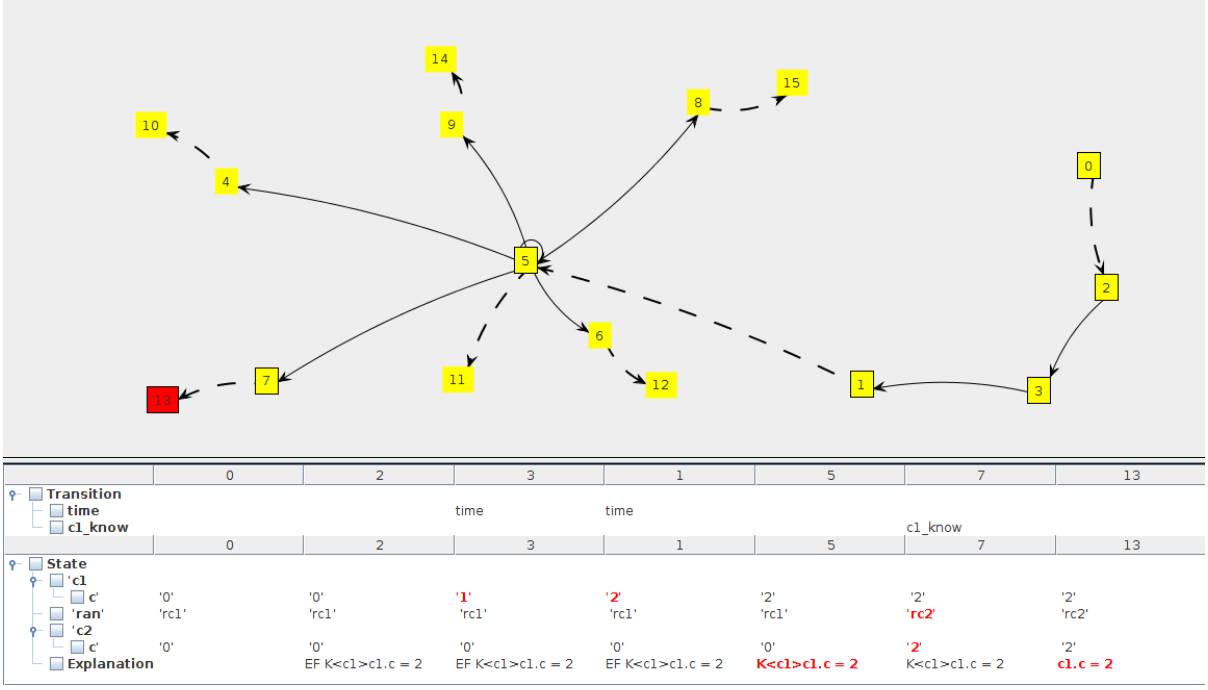


FIGURE 6.12 – Une trace dans une explication sélectionnée dans le panneau principal.

6.4.1 Sélection des variables affichées

C'est également dans cette vue que l'on offre la possibilité d'afficher les variables ou l'explication d'un état mais aussi les labels des transitions pour savoir de quel type elles sont. Cette sélection se fait au moyen d'une case à cocher devant chaque variable ou type de transition. Grâce à l'arbre, nous avons aussi la possibilité de cocher/décocher toutes les variables dans la hiérarchie.

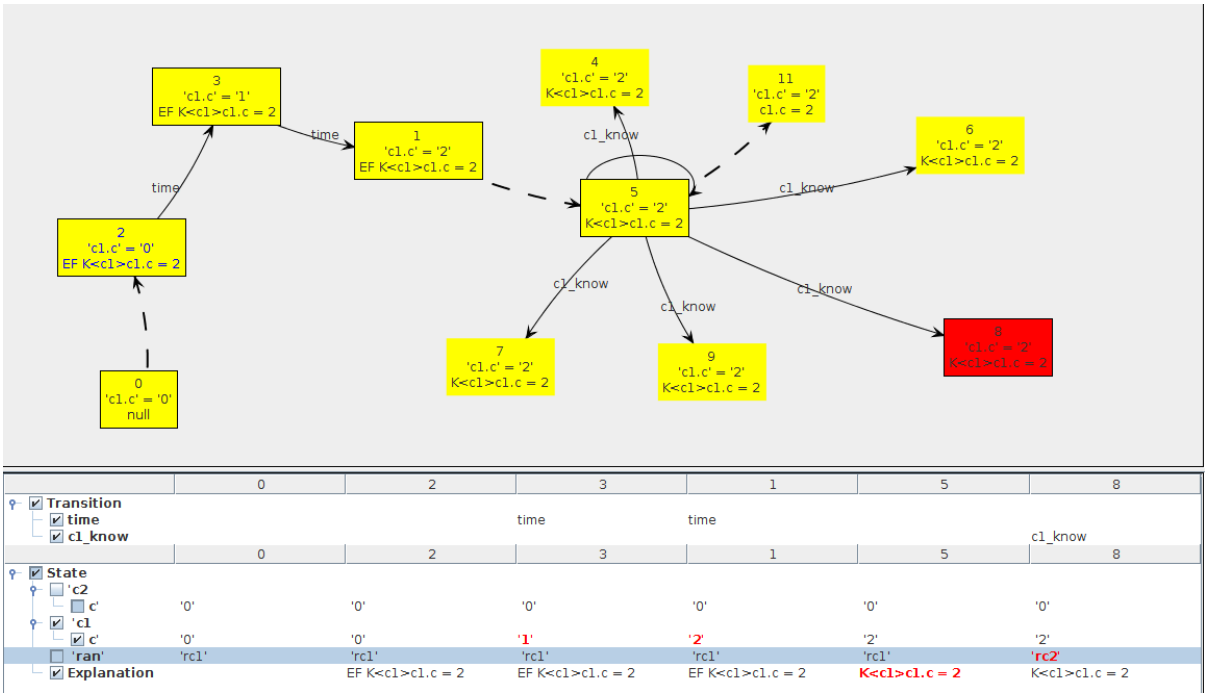


FIGURE 6.13 – Affichage de labels sur les états et transitions.

Chapitre 7

Test

Dans ce chapitre, nous verrons les cinq exemples de modèle fournis avec PyNuSMV : counters, cardgame, cardgamePostFair, transmission et transmissionPostFair. Nous ne testerons pas toutes les propriétés ce qui serait bien trop long. Cependant, nous essayerons d'illustrer l'utilité de pouvoir visualiser une explication et de tester les fonctionnalités de l'application visuelle ainsi que ses limites.

7.1 Counters

Cet exemple de système est composé de deux compteurs et d'un environnement. Chaque compteur peut s'incrémenter ou passer et l'environnement choisit quel compteur peut agir. Toutes les fois où un compteur choisit de s'incrémenter, sa valeur est incrémentée. Le compteur ne dépasse pas un maximum de 2 grâce à un modulo. La contrainte de fairness nous dit que chaque compteur doit être choisi par l'environnement infiniment souvent.

EFf true

Il existe un chemin fair démarrant en l'état initial du système. C'est-à-dire un chemin qui passe par toutes les contraintes ($ran = rc1$ et $ran = rc2$) de fairness infiniment souvent. L'état 5 nous montre pourquoi l'état initial est vrai et les états 1, 2, 3 et 4 nous montrent un chemin qui est équitable. Ce chemin est représenté par la boucle entre les états 3 et 4 où la variable ran change à chaque fois.

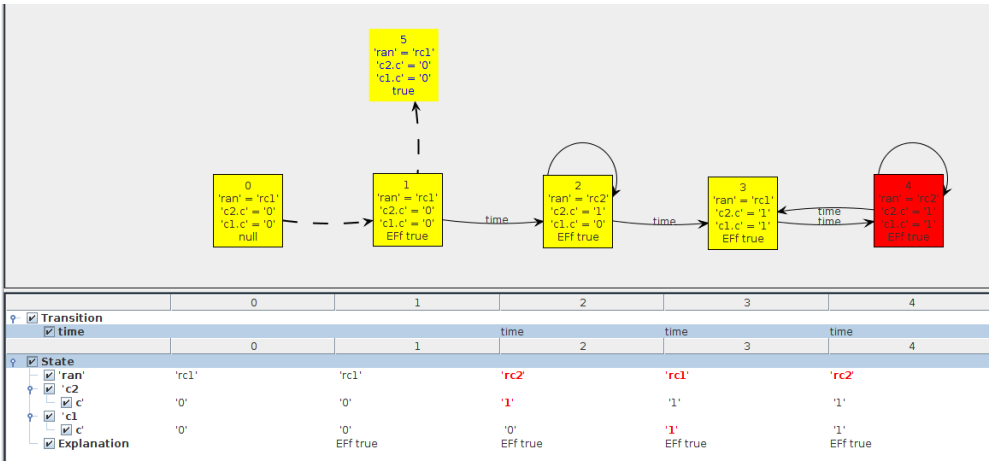


FIGURE 7.1 – EFf true - Explication

AG EF c1.c=0

Pour tout état atteignable, il existe un chemin pour atteindre un état où $c1.c = 0$. On a donc comme résultat, une explication avec tous les états possibles du modèle qui possèdent une sous-explication pour montrer qu'ils peuvent atteindre $c1.c = 0$. C'est ce qui nous donne cette forme en étoile avec des sous-explications tout autour d'un graphe central contenant tous les états possibles du modèle. Une trace est sélectionnée pour montrer un exemple de sous-explication permettant d'atteindre $c1.c = 0$ à partir d'un état où la valeur de $c1$ est pourtant à 1.

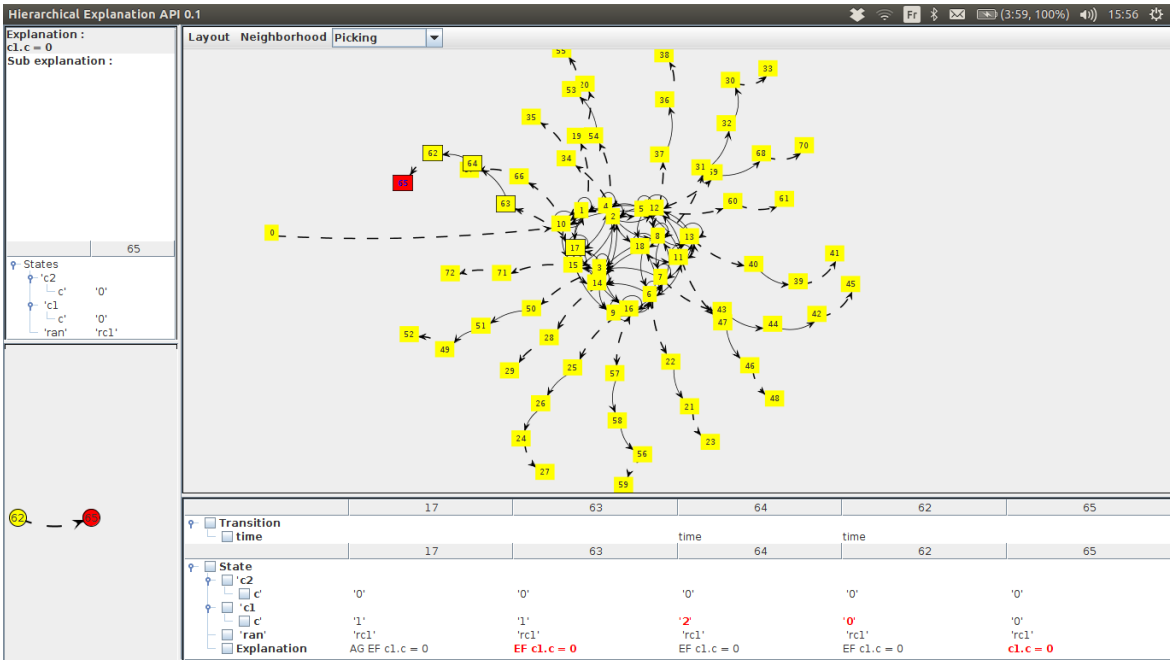


FIGURE 7.2 – AG EF c1.c=0 - KamadaKawai layout

On peut voir sur la figure 7.3 que le layout hiérarchique nous montre bien toutes les sous-explications pour chaque état du graphe principal. Cependant, on peut voir ici une première limite. C'est le fait qu'avec le nombre grandissant de sous-explications, ce layout s'étire fort horizontalement. Mais il nous permet quand même de voir la structure ce qui est son but premier.

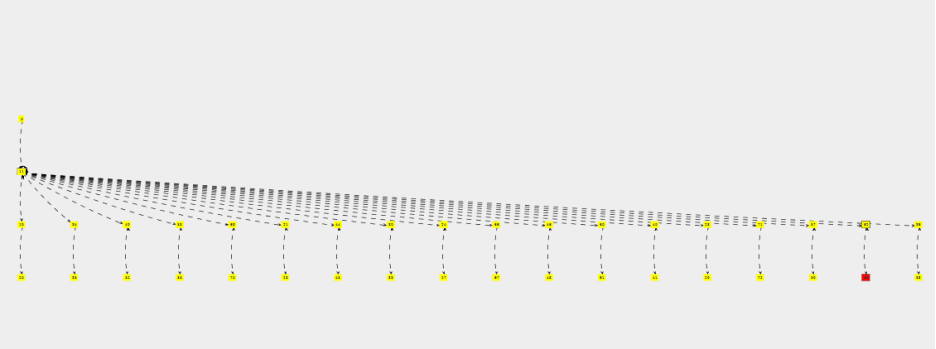


FIGURE 7.3 – AG EF c1.c=0 - Hierarchical layout

<c1c2>i Ff (c1.c = 2 & c2.c = 2)

$c1$ et $c2$ possèdent une stratégie sans avoir une information parfaite et qui tient compte des chemins équitables pour obtenir finalement que $c1.c = 2$ et $c2.c = 2$ est vrai. Cette propriété est vraie dans le modèle des compteurs parce que s'ils prennent comme stratégie de toujours s'incrémenter, on peut atteindre les états 16 et 19 qui vérifient ($c1.c = 2 \& c2.c = 2$). La trace sélectionnée nous le montre bien. Il y a deux états ,16 et 19, parce que nous en avons deux dans le modèle où $c1$ vaut 2 et $c2$ vaut 2 en même temps : $ran = rc1$ et $ran = rc2$. Une nouvelle limitation apparaît ici avec l'affichage des informations sur les transitions qui n'est pas toujours optimal avec JUNG.

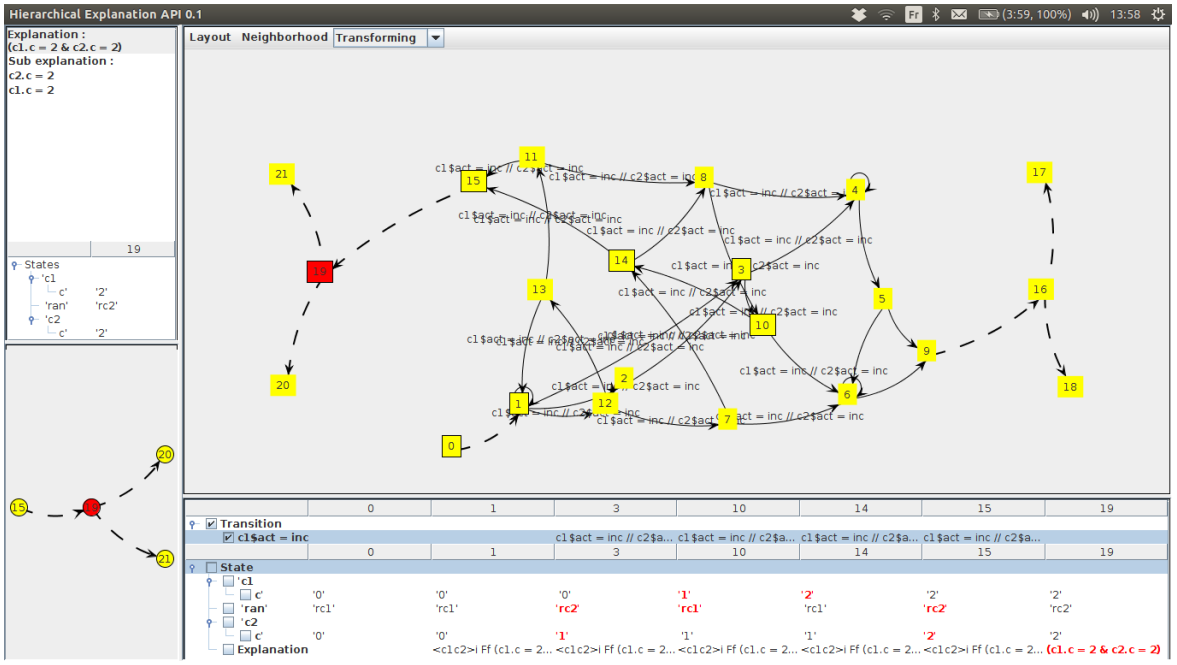
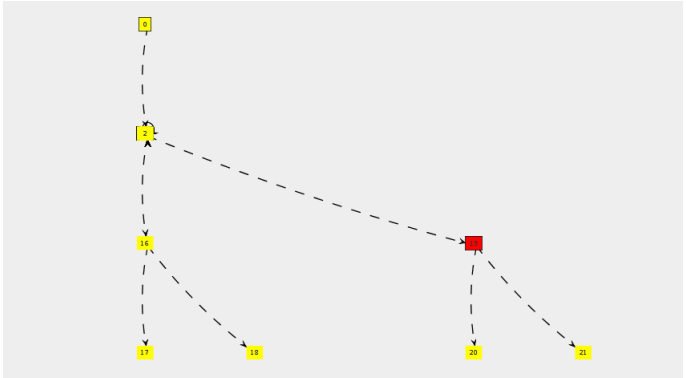


FIGURE 7.4 – <c1c2>i Ff (c1.c = 2 & c2.c = 2) - Explication

Le layout hiérarchique dans ce cas-ci fonctionne bien et nous montre la structure de l'explication avec les deux sous-explications finales (deux états qui vérifient la formule) qui elles-même expliquent deux formules dues au &.



4

FIGURE 7.5 – <c1c2>i Ff (c1.c = 2 & c2.c = 2) - Hierarchical layout

<c1> F c1.c = 2

Nous pouvons terminer les exemples pour le modèle des compteurs avec une propriété qui est fausse et voir ce qu'il se passe dans ce cas-là. Comme expliqué dans le contexte, lorsqu'une propriété n'est pas vérifiée, on prend alors sa négation et on génère un contre-exemple qui nous explique pourquoi elle est fausse. Dans la figure 7.6, on peut voir que les états 2 et 3 expliquent qu'il suffit d'empêcher $c1$ d'agir. On reste indéfiniment dans l'état 3 ce qui empêche $c1$ d'incrémenter sa valeur jusque 2. Les états 4 et 5 nous montrent bien que $\sim c1.c = 2$ est vrai.

La formule expliquée dans ce cas-ci en tant que contre-exemple est $[c1]G \sim c1.c = 2$. Elle nous dit que $c1$ ne possède pas de stratégie qui rendent $\sim c1.c = 2$ faux. En effet, si c'est à $c2$ d'agir, $c1$ ne sait rien faire. Il est important de noter que ce raisonnement est possible parce que la fairness n'est pas prise en compte dans cette propriété. Ce qui permet à $c2$ d'exécuter infiniment son action skip alors que s'il y avait de la fairness, $c1$ pourrait aussi s'exécuter infiniment souvent et donc prendre comme stratégie de s'incrémenter tout le temps. C'est exactement ce que nous dit la formule numéro 11, $<c1> Ff\ c1.c = 2$ (La génération de cet exemple ne fonctionne pas suite à un bug dans les états du côté de PyNuSMV).

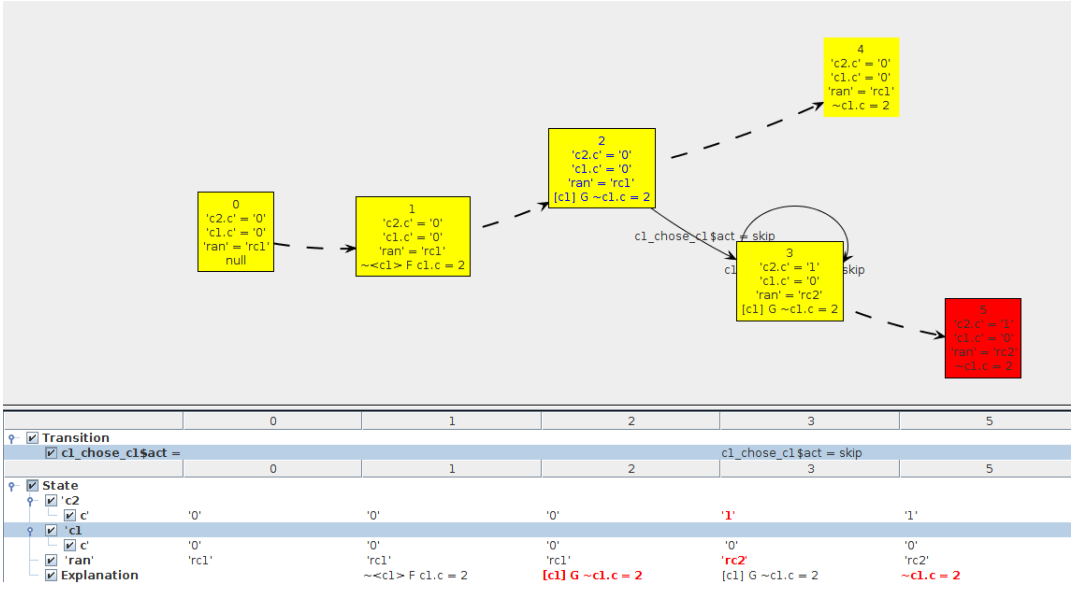


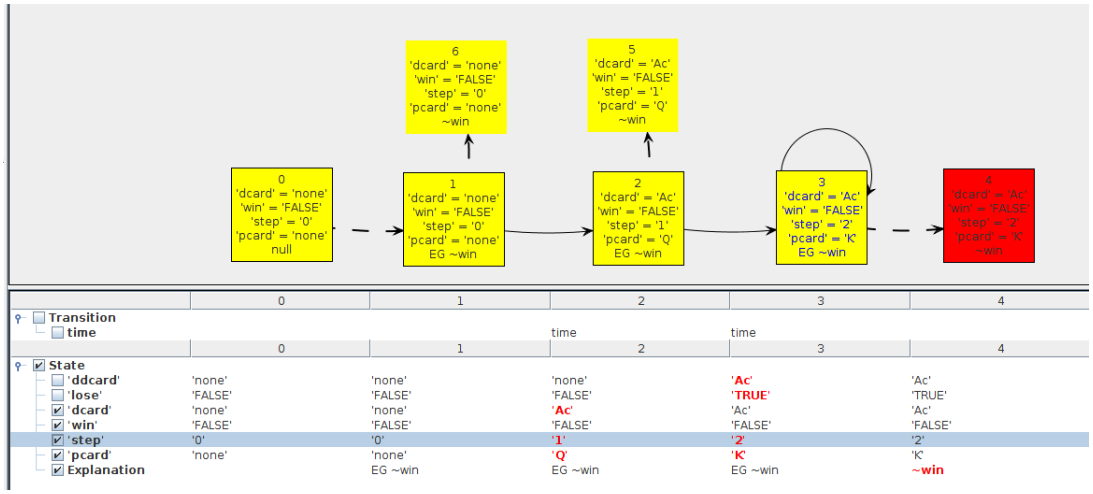
FIGURE 7.6 – <c1> F c1.c = 2 - Contre-exemple

7.2 Cardgame

Il s'agit d'un jeu de carte simple. Ce jeu est composé d'un joueur, d'un dealer et de trois cartes. Nous avons l'as, le roi et reine. L'as bat le roi, le roi bat la reine et la reine bat l'as, ce qui forme un triangle. Le jeu se joue en deux étapes. Tout d'abord, le dealer donne une carte au joueur, en garde une et met la troisième face cachée sur la table mais il voit toutes les cartes alors que le joueur ne voit que la sienne. Ensuite, le joueur peut garder ou échanger sa carte avec celle qui est sur la table. A la fin, le gagnant est celui qui possède la carte gagnante.

EG ∼win

Nous commencerons par un exemple simple afin de comprendre le jeu. Cette formule nous dit qu'il existe un chemin où dans tous les états, ∼ *win* est vrai. Pour expliquer cette propriété, il suffit de prendre une partie de cardgame où le player perd. Ceci est représenté par les états 1, 2 et 3. Le dealer a un as alors que dans le step 1 le joueur possède une dame mais celui-ci change de carte dans l'étape 2 ce qui permet au dealer de gagner la partie. Maintenant, il faut expliquer pourquoi chacun de ces trois états satisfaits ∼ *win* ce qui est fait par le biais des trois sous-explications. Elles sont triviales vu que la variable *win* reste à false si on n'est pas encore à la fin du jeu et que lorsque le jeu est fini, le joueur perd.



	0	1	2	3	4
Transition					
time			time	time	
	0	1	2	3	4
State					
'ddcard'	'none'	'none'	'none'	'Ac'	'Ac'
'lose'	'FALSE'	'FALSE'	'FALSE'	'TRUE'	'TRUE'
'dcard'	'none'	'none'	'Ac'	'Ac'	'Ac'
'win'	'FALSE'	'FALSE'	'FALSE'	'FALSE'	'FALSE'
'step'	'0'	'0'	'1'	'2'	'2'
'pcard'	'none'	'none'	'Q'	'K'	'K'
Explanation		EG ∼win	EG ∼win	EG ∼win	∼win

FIGURE 7.7 – EG ∼win - Explication

EG ∼step=3

Il s'agit du même genre de formule, une exécution du jeu où tous les états vérifient ∼ *step* = 3. On a donc le même exemple de déroulement du cardgame sauf qu'ici, les sous-explications nous montrent qu'on ne peut pas atteindre le step 3 dans ce jeu grâce à la variable *step*.

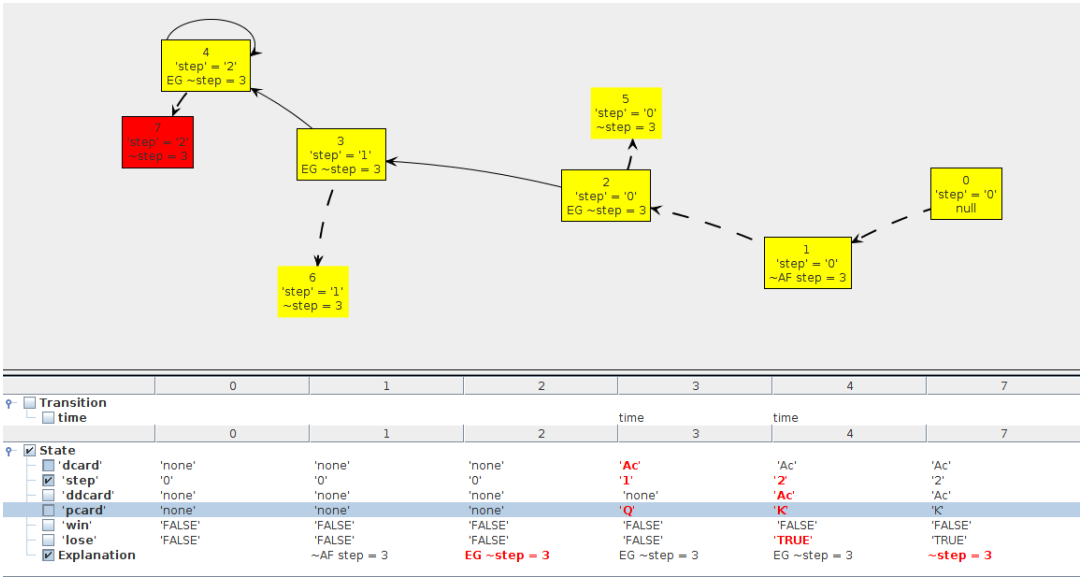


FIGURE 7.8 – EG ~step=3 - Explication

EX K<dealer>dcard = Ac

Cette formule nous dit qu'il existe un chemin où le prochain état est vrai. Pour qu'il soit vrai, il faut que le dealer connaisse sa carte qui vaut Ac. Les états 1 et 2 nous montrent un état et son suivant (EX). Ensuite, l'état 1 possède une sous-explication qui explique pourquoi $K < dealer > dcard = Ac$ est vrai. Pour cela, il faut regarder tous les états indistinguables pour le dealer où sa carte vaut Ac. Ce qui nous donne les états 3, 4, 5 et 6. Et enfin, les états 7 et 8 nous montrent que la carte du dealer vaut Ac dans les états possibles lorsque le dealer possède la carte Ac. C'est-à-dire les cas où le joueur possède un roi ou une dame.

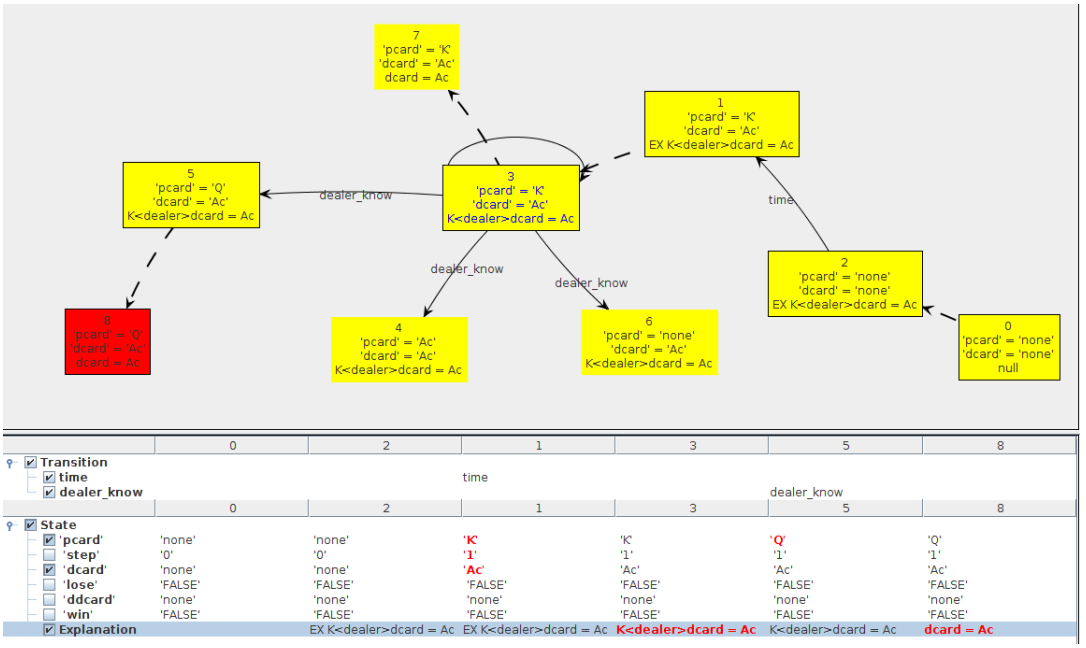


FIGURE 7.9 – EX K<dealer>dcard = Ac - Explication

7.3 CardgamePostFair

Dans cette version du cardgame, le jeu est joué encore et encore et la contrainte de fairness nous dit que le dealer doit donner les cartes de manière équitable et donc pas toujours faire la même distribution. Une variable est ajoutée afin de garder en mémoire les dernières actions jouées ce qui permet de mettre cette contrainte de fairness sur les états.

EF win

Cet exemple est le plus basique possible. Il nous dit tous simplement qu'il existe un chemin pour gagner le jeu. Donc dans ce jeu il est possible de gagner, on ne perd pas forcément. Le dealer donne un roi au joueur et un as à lui-même mais le joueur change de carte et gagne donc cette partie.

Ce qu'il est intéressant de remarquer ici, c'est que nous avons affiché toutes les variables des états et il s'agit de notre modèle qui en possède le plus. On peut dès lors remarquer que le graphe est très vite surchargé avec les nœuds qui se cachent l'un l'autre et les arêtes. Cependant, ce layout rapproche fort les nœuds mais cela permet d'illustrer ce problème d'affichage. L'avantage de l'application visuelle, c'est que l'on peut choisir séparément les variables que l'on affiche et ainsi permettre de pouvoir réduire ce problème en ne mettant que celles qui sont intéressantes pour le problème donné. De plus du côté du positif, on peut noter que les layouts continuent de bien se comporter même avec plus de nœuds à afficher.

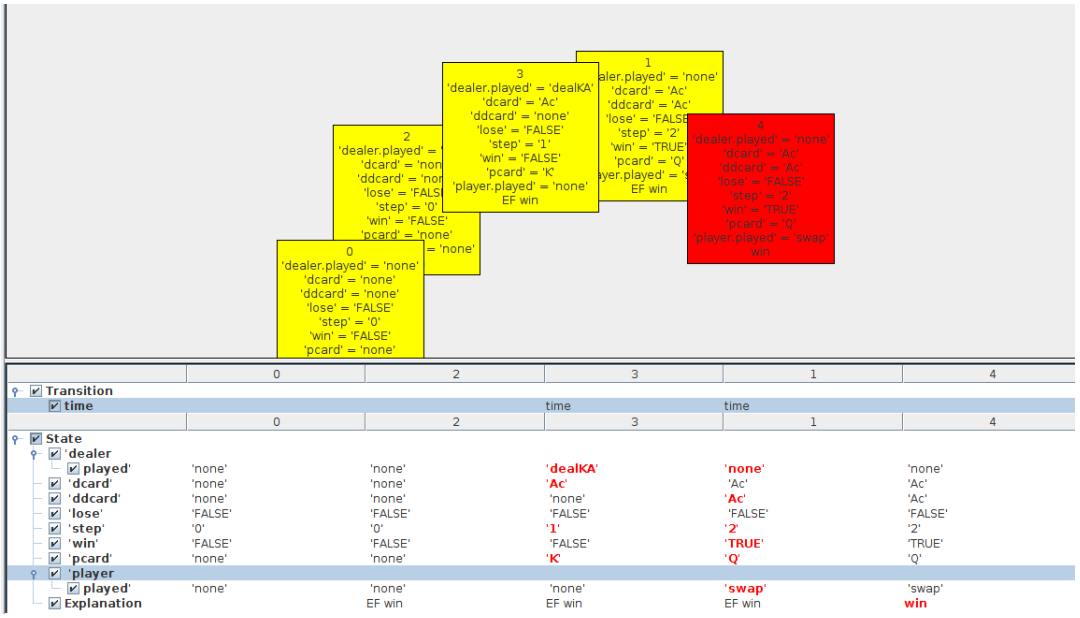


FIGURE 7.10 – EF win - KamadaiKawai layout avec affichage des variables

EGf ∼win

Il existe un chemin équitable où dans tous les états, ∼ *win* est vrai. Ce chemin fair est représenté par les six triangles attachés à l'état numéro 3. Chacun de ceux-ci correspond à une distribution différente des cartes par le dealer : QA, QK, AQ, AK, KA et KQ. Ce qui nous montre la fairness entre toutes les distributions possibles du jeu. Ensuite, pour terminer l'explication de la formule, chaque état du chemin fair possède une sous-explication qui nous dit que le joueur ne gagne pas.

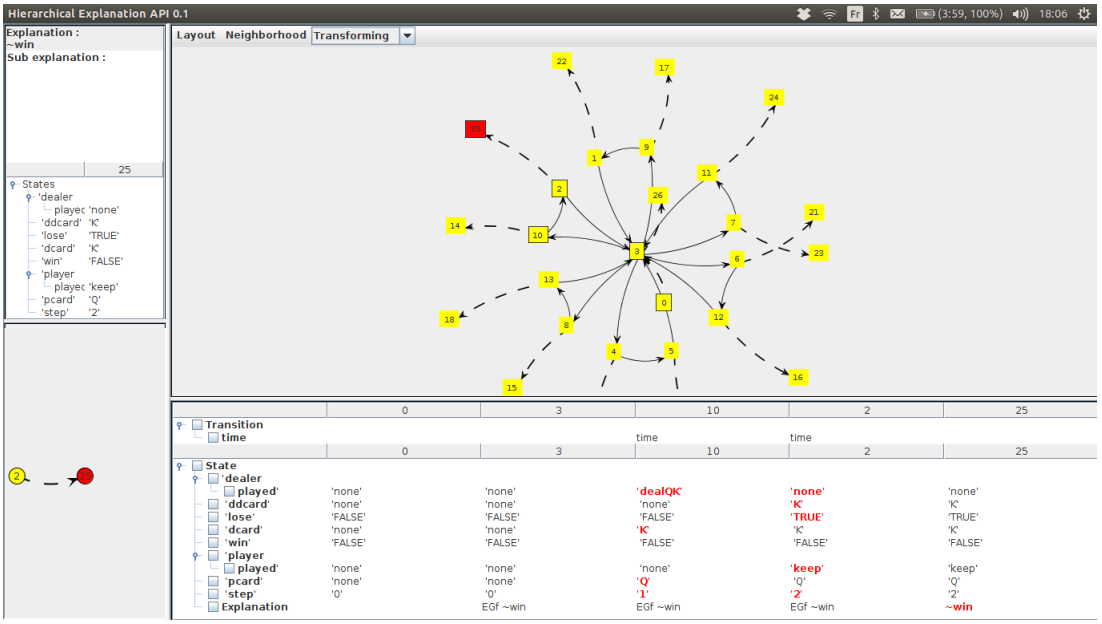


FIGURE 7.11 – EGf ∼win - Explication

Et on peut à nouveau tester l'affichage des variables avec une explication possédant plus d'états, ce qui devient vraiment illisible même avec la vue des prédécesseurs et successeurs.

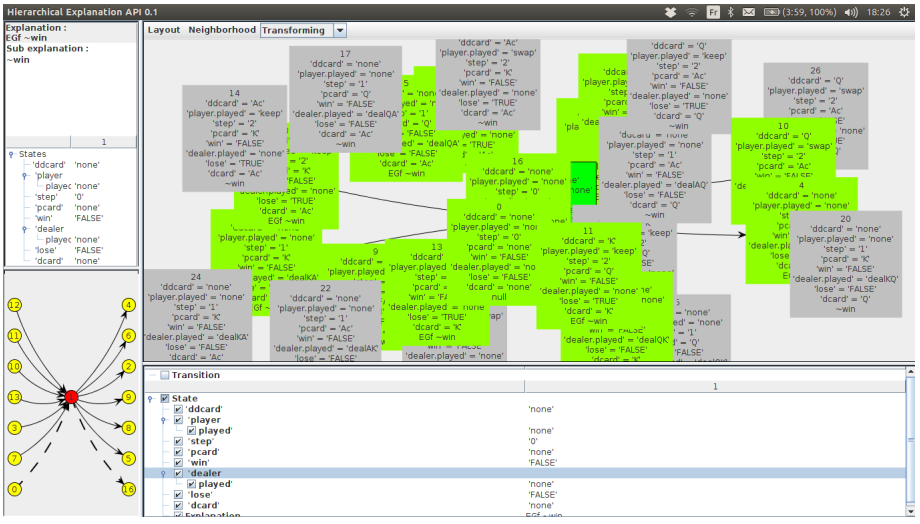


FIGURE 7.12 – EGf ∼win - Toutes les variables affichées

7.4 Transmission

Cet exemple est très simple. Il s'agit d'un système de transmission d'un bit. Il est composé d'un bit d'information, un envoyeur et un transmetteur. L'envoyeur peut demander d'envoyer le bit ou d'attendre alors que le transmetteur peut choisir de transmettre le bit ou de bloquer la transmission. Le bit est reçu si et seulement si l'envoyeur envoie le bit et que le transmetteur le transmet.

EG ∼received

Il existe un chemin sur lequel tous les états vérifient ∼received. Ce qui est vrai puisque l'on peut rester indéfiniment dans l'état d'attente pour l'envoyeur ou alors dans un état bloqué pour le transmetteur. Ce qui se voit dans l'état numéro 1. Il possède une sous-explication qui nous dit pourquoi l'état est vrai pour ∼received.

On ne voit pas très bien ce qu'il se passe dans ce modèle car les états possèdent uniquement une variable *received* et on ne voit pas l'envoyeur ou le transmetteur. Un problème apparaît cependant dans cet exemple simple. Lorsque qu'on a un état avec une seule transition vers lui-même, on ne sait pas sélectionner de trace avec l'algorithme de Dijkstra et par conséquent, on ne sait pas choisir d'afficher le label de cette transition.

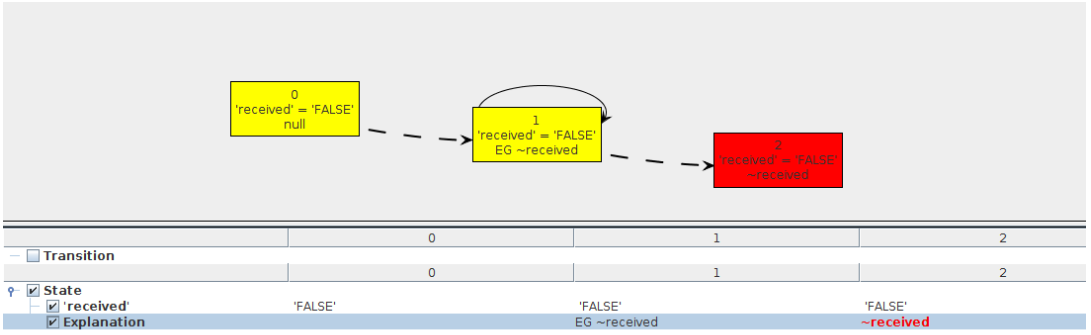


FIGURE 7.13 – EG ∼received - Explication

7.5 TransmissionPostFair

Dans cette version du système de transmission d'un bit, on rajoute une contrainte de fairness. Celle-ci stipule qu'un chemin est équitable lorsque le transmetteur autorise la transmission infiniment souvent. Des variables sont ajoutées afin de garder en mémoire les dernières actions. De cette manière, la contrainte de fairness est spécifiée sur les variables d'états.

EG ∼received

Cet exemple est simple mais il permet de faire le contraste avec la transmission normale. Avec le rajout des variables, on voit mieux ce qu'il se passe et l'état reste indéfiniment dans un état bloqué. Les états 3 et 4 servant à montrer ∼received.

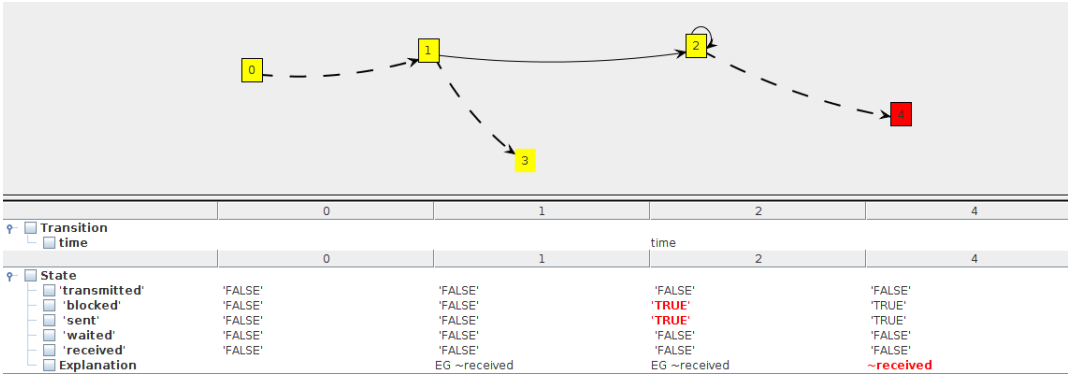


FIGURE 7.14 – EG ∼received - Explication

EGf ∼received

Si on ajoute la contrainte que le chemin doit être équitable, on a toujours une propriété vraie mais il faut une autre astuce pour y arriver. Ici, la boucle est entre les états 2 et 3. Il s'agit bien d'un chemin fair puisque transmitted passe de *true* à $false$. En outre, on respecte ∼received en mettant *true* sur *blocked* et *waited* à tour de rôle afin que le bit ne soit jamais transmis.

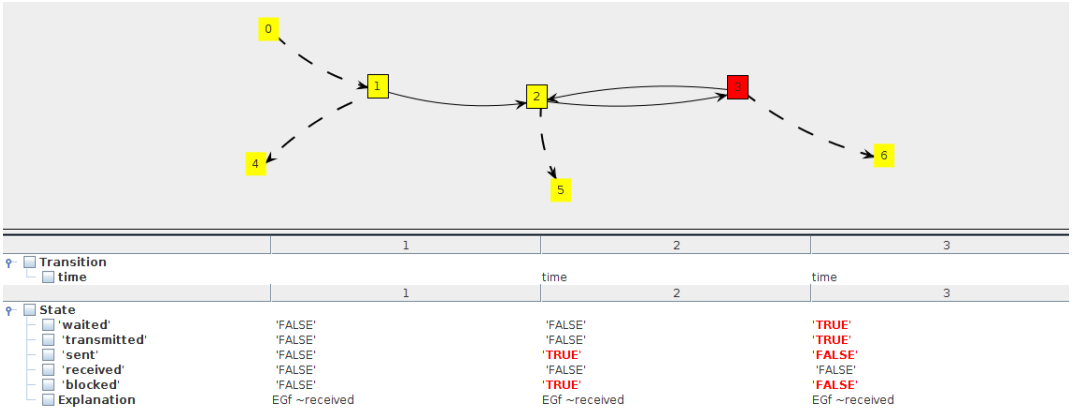


FIGURE 7.15 – EGf ∼received - Explication

<sender>i Ff received

Cette formule est plus complexe. L'envoyeur possède une stratégie pour que le bit soit finalement reçu. L'information est incomplète et le chemin est fair. Les nœuds en gris à droite correspondent aux transitions *sender_equiv*. Ce sont tous les états possibles au niveau des variables mais pas forcément dans le modèle car l'information est incomplète. Alors que les nœuds en vert sont ceux où l'envoyeur choisit comme stratégie de toujours envoyer et ce sont les états possibles du modèle. On entend par possible, les états où les variables opposées comme *sent*/*wait* et *transmitted*/*blocked* possèdent un *true* par couple sauf le 10 qui est l'état initial.

On voit que dans tous ces états du modèle, lorsque l'envoyeur choisit de toujours envoyer, on pourra atteindre un état où *received* vaut *true*. De plus, le chemin est fair puisque *transmitted* bouge. La véracité de cette propriété est intuitive puisque si *transmitted* est obligé de se mettre à *true* infiniment souvent, il est clair qu'en choisissant d'envoyer tous le temps, le bit sera reçu à un moment.

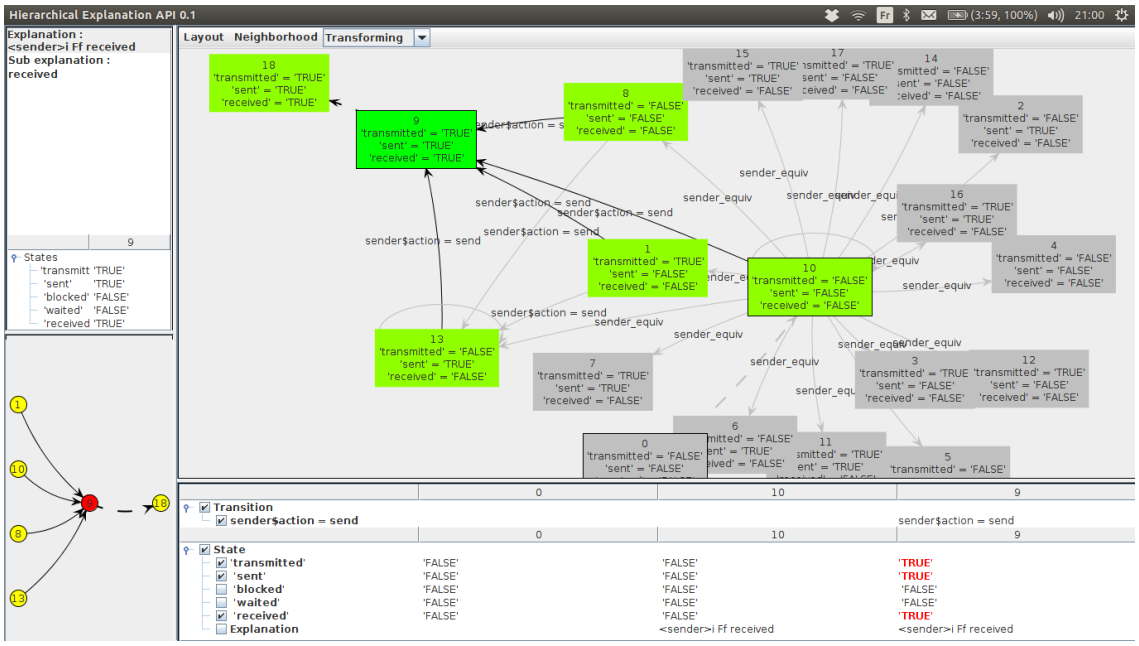


FIGURE 7.16 – <sender>i Ff received - Explication

7.6 Remarque

Les exemples 1.11, 3.7, 4.4, 4.5, 5.5 et 5.6, qui correspondent aux chiffres à taper pour choisir un modèle puis une propriété, ne fonctionnent pas suite à un bug dans les états du côté de PyNuSMV.

Les annexes contiennent la définition de tous les modèles sous forme de fichier smv.

Chapitre 8

Conclusion

Pour conclure ce travail, on peut dire, après avoir effectué les tests, que l'outil se comporte bien et permet une bonne visualisation des explications hiérarchiques. En outre, la compréhension de la complexité de celles-ci est plus facile et la visualisation de leurs richesses est incluse dans ces graphes et le tableau présentant une trace. Il reste encore quelques limites et l'outil n'est pas parfait mais il possède pas mal de fonctionnalités et son utilisation est simple. De plus, on peut ajouter que le possible problème de ralentissement à cause du stream texte n'est pas apparu mais les graphes ne contenaient qu'une centaine de nœuds maximum. De plus, l'architecture générale avec un module d'un côté et l'application visuelle de l'autre est un gros point positif afin de pouvoir utiliser cet outil avec d'autres programmes puisque celui-ci est indépendant. Et enfin, l'interactivité implémentée permet aussi une plus grande flexibilité au niveau de l'utilisation si on envisage un usage avec des programmes qui seraient assez lents au niveau de la génération des données.

8.1 Travail futur

On peut identifier une série de fonctionnalités qui ne sont pas encore présentes mais qui pourraient être utiles et améliorer le fonctionnement ou la visualisation.

- On peut imaginer améliorer le panneau affichant le graphe avec les prédécesseurs et successeurs d'un nœud en autorisant l'affichage de variables dans cette vue.
- On pourrait aussi ajouter le fait que l'on ne veut peut-être pas afficher les labels sur tous les nœuds ou toutes les transitions. Ajouter un mécanisme de sélection pour permettre un affichage encore plus personnalisé des variables d'états et de transitions.
- Pour le moment, lorsque l'application graphique demande de nouvelles informations, un thread est lancé pour effectuer cette demande et le calcul sans que la vue soit bloquée. Cependant, on ne possède pas de fonctionnalité pour arrêter ce calcul s'il prend trop de temps par exemple.
- La sélection des nœuds pour une trace se fait uniquement avec l'algorithme de Dijkstra. Celle-ci pourrait être plus personnalisée.

Bibliographie

[1] Goal - graphical tool for omega-automata and logics. `http://goal.im.ntu.edu.tw/wiki/doku.php`.

[2] yed graph editor : High-quality diagrams made easy. `https://www.yworks.com/en/products/yfiles/yed/`.

[3] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5) :672–713, September 2002.

[4] Greg Bernstein. Jung 2.0 tutorial, or how to achieve graph based nirvana in java. `http://www.grotto-networking.com/JUNG/JUNG2-Tutorial.pdf`, 2009.

[5] Simon Busard and Charles Pecheur. Rich counter-examples for temporal-epistemic logic model checking. volume 78 of *Electronic Proceedings in Theoretical Computer Science*, pages 39–53. Open Publishing Association, 2012.

[6] Simon Busard and Charles Pecheur. Pynusmv : Nusmv as a python library. volume 7871 of *LNCS*, pages 453–458. Springer-Verlag, 2013.

[7] Simon Busard, Charles Pecheur, Hongyang Qu, and Franco Raimondi. Reasoning about memoryless strategies under partial observability and unconditional fairness constraints. *Information and Computation*, 242(0) :128–156, 2015.

[8] Cédric Delforge and Charles Pecheur. Ltsa-delforge is an extension of ltsa adding extensive lts layout capabilities. `http://lvl.info.ucl.ac.be/Tools/LTSADelforge`.

Annexe A

Parsing de $(EF\ c1.c = 2\ \&\ EF\ c2.c = 2)$ dans le modèle des compteurs

```
Graph- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6d68>
States
1
0-{'c2.c': '0', 'ran': 'rc1', 'c1.c': '0'}
Transitions
0
SubExplanations
Set of expl of state
0
Label
0
(EF c1.c = 2 & EF c2.c = 2)
<tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6da0>
Graph- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6da0>
States
1
1-{'c2.c': '0', 'ran': 'rc1', 'c1.c': '0'}
Transitions
0
SubExplanations
Set of expl of state
1
Label
1
EF c1.c = 2
<tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6e10>
Graph- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6e10>
States
3
2-{'c2.c': '0', 'ran': 'rc1', 'c1.c': '2'}
```

```
3-{'c2.c': '0', 'ran': 'rc1', 'c1.c': '0'}
4-{'c2.c': '0', 'ran': 'rc1', 'c1.c': '1'}
Transitions
1
1
time
2
3/4
4/2
SubExplanations
Set of expl of state
2
Label
2
c1.c = 2
<tools.mucalculus.explanation.Explanation object at 0x7f4c223a6f98>
Graph- <tools.mucalculus.explanation.Explanation object at 0x7f4c223a6f98>
States
1
5-{'c2.c': '0', 'ran': 'rc1', 'c1.c': '2'}
Transitions
0
SubExplanations
No more graph
End- <tools.mucalculus.explanation.Explanation object at 0x7f4c223a6f98>
End- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6e10>
Label
1
EF c2.c = 2
<tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223ba9b0>
Graph- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223ba9b0>
States
3
6-{'c2.c': '2', 'ran': 'rc2', 'c1.c': '0'}
7-{'c2.c': '0', 'ran': 'rc1', 'c1.c': '0'}
8-{'c2.c': '1', 'ran': 'rc2', 'c1.c': '0'}
Transitions
1
1
time
2
7/8
8/6
SubExplanations
Set of expl of state
6
Label
6
c2.c = 2
<tools.mucalculus.explanation.Explanation object at 0x7f4c223a6748>
Graph- <tools.mucalculus.explanation.Explanation object at 0x7f4c223a6748>
States
1
```

```
9-{'c2.c': '2', 'ran': 'rc2', 'c1.c': '0'}
Transitions
0
SubExplanations
No more graph
End- <tools.mucalculus.explanation.Explanation object at 0x7f4c223a6748>
End- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223ba9b0>
End- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6da0>
End- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6d68>
```

Annexe B

Parsing de $< c1 > F\ c1.c = 2$ dans le modèle des compteurs

```
Graph- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6d68>
States
1
0-{'c2.c': '0', 'ran': 'rc1', 'c1.c': '0'}
Transitions
0
SubExplanations
Set of expl of state
0
Label
0
(EF c1.c = 2 & EF c2.c = 2)
<tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6da0>
Graph- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6da0>
States
1
1-{'c2.c': '0', 'ran': 'rc1', 'c1.c': '0'}
Transitions
0
SubExplanations
Set of expl of state
1
Label
1
EF c1.c = 2
<tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6e10>
Graph- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6e10>
States
3
2-{'c2.c': '0', 'ran': 'rc1', 'c1.c': '2'}
3-{'c2.c': '0', 'ran': 'rc1', 'c1.c': '0'}
4-{'c2.c': '0', 'ran': 'rc1', 'c1.c': '1'}
Transitions
```

```
1
1
time
2
3/4
4/2
SubExplanations
Set of expl of state
2
Label
2
c1.c = 2
<tools.mucalculus.explanation.Explanation object at 0x7f4c223a6f98>
Graph- <tools.mucalculus.explanation.Explanation object at 0x7f4c223a6f98>
States
1
5-{'c2.c': '0', 'ran': 'rc1', 'c1.c': '2'}
Transitions
0
SubExplanations
No more graph
End- <tools.mucalculus.explanation.Explanation object at 0x7f4c223a6f98>
End- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6e10>
Label
1
EF c2.c = 2
<tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223ba9b0>
Graph- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223ba9b0>
States
3
6-{'c2.c': '2', 'ran': 'rc2', 'c1.c': '0'}
7-{'c2.c': '0', 'ran': 'rc1', 'c1.c': '0'}
8-{'c2.c': '1', 'ran': 'rc2', 'c1.c': '0'}
Transitions
1
1
time
2
7/8
8/6
SubExplanations
Set of expl of state
6
Label
6
c2.c = 2
<tools.mucalculus.explanation.Explanation object at 0x7f4c223a6748>
Graph- <tools.mucalculus.explanation.Explanation object at 0x7f4c223a6748>
States
1
9-{'c2.c': '2', 'ran': 'rc2', 'c1.c': '0'}
Transitions
0
```

```
SubExplanations
No more graph
End- <tools.mucalculus.explanation.Explanation object at 0x7f4c223a6748>
End- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223ba9b0>
End- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6da0>
End- <tools.mucalculus.translation.atlk.HierarchicalExplanation object at
    0x7f4c223a6d68>
```

Annexe C

Fichier counters.smv

```
MODULE Counter(run)
    VAR
        c: 0 .. 2;
    IVAR
        act: {inc, skip};
    INIT
        c = 0
    TRANS
        run & act = inc -> next(c) = (c + 1) mod 3
    TRANS
        ! run | act = skip -> next(c) = c

MODULE main
    IVAR
        run: {rc1, rc2};
    VAR
        ran: {rc1, rc2};
        c1: Counter(run = rc1);
        c2: Counter(run = rc2);
    TRANS
        next(ran) = run
    FAIRNESS
        ran = rc1
    FAIRNESS
        ran = rc2
```

Annexe D

Fichier cardgame.smv

```
--- Simple card game
---
--- The game is composed of a player, a dealer and three cards: an Ace, a King
--- and a Queen; the Ace wins over the King, the King over the Queen and the
    Queen over the Ace. The game is played in two steps.
--- During the first step, the dealer gives one card to the player, keeps one
--- and put the last one, hidden, on table. The dealer sees all cards, while
--- the player only sees his.
--- During the second step, the player can ask to swap his card with the one
--- on table, or to keep it.
--- At the end, the winner is the one with the winning card.
---
---
--- Simon Busard <simon.busard@uclouvain.be>, 19/03/13

MODULE Player(step, pcard, ddcard)
    --- step is the id of the step in the game
    --- (0 for starting, 1 for first phase, 2 for the end)
    --- pcard is the card of the player (none, Ac, K, Q)
    --- ddcard is the card of the dealer, disclosed on table (none, Ac, K, Q)

    IVAR action : {none, keep, swap};

    --- Protocole
    TRANS
    action in case step = 1 : {keep, swap};
                step != 1 : {none};
           esac


MODULE Dealer(step, dcard, pcard)
    --- step is the id of the step in the game
    --- (0 for starting, 1 for first phase, 2 for the end)
    --- pcard is the card of the player (none, Ac, K, Q)
    --- dcard is the card of the dealer (none, Ac, K, Q)

    IVAR action : {none, dealAK, dealAQ, dealKA, dealKQ, dealQA, dealQK};

    --- Protocole
    TRANS
```

```
    action in case step = 0 : {dealAK, dealAQ, dealKA, dealKQ, dealQA, dealQK};
                step != 0 : {none};
            esac


MODULE main

    VAR step  : 0..2;
        pcard : {none, Ac, K, Q};
        dcard : {none, Ac, K, Q};
        ddcard : {none, Ac, K, Q};
        dealer : Dealer(step, dcard, pcard);
        player : Player(step, pcard, ddcard);


    INIT step = 0 & pcard = none & dcard = none & ddcard = none

    TRANS
    next(step) = case step < 2 : step + 1; TRUE : step; esac

    TRANS
    next(pcard) = case step = 0 : case dealer.action in {dealAK, dealAQ} : Ac;
                                      dealer.action in {dealKQ, dealKA} : K;
                                      dealer.action in {dealQK, dealQA} : Q;
                                      TRUE                              : none;
                                  esac;
                      step = 1 : case player.action = keep : pcard;
                                      player.action = swap :
                                       case (pcard = Ac & dcard = K) |
                                            (pcard = K & dcard = Ac) : Q;
                                            (pcard = Ac & dcard = Q) |
                                            (pcard = Q & dcard = Ac) : K;
                                            (pcard = Q & dcard = K) |
                                            (pcard = K & dcard = Q)  : Ac;
                                            TRUE                     : none;
                                       esac;
                                      player.action = none : none;
                                  esac;
                      step = 2 : pcard;
                esac

    TRANS
    next(dcard) = case step != 0 : dcard;
                      step = 0 : case dealer.action in {dealKA, dealQA} : Ac;
                                       dealer.action in {dealAK, dealQK} : K;
                                       dealer.action in {dealKQ, dealAQ} : Q;
                                       dealer.action = none          : none;
                                   esac;
                esac

    TRANS
    next(ddcard) = case step != 1 : ddcard;
                        step = 1 : dcard;
                 esac
```

```
DEFINE
win := step = 2 & ( (pcard = Ac & dcard = K) |
                    (pcard = K & dcard = Q) |
                    (pcard = Q & dcard = Ac) );
lose := step = 2 & ( (pcard = Ac & dcard = Q) |
                     (pcard = K & dcard = Ac) |
                     (pcard = Q & dcard = K) );
```

Annexe E

Fichier cardgamePostFair.smv

```
--- Simple card game
---
--- The game is composed of a player, a dealer and three cards: an Ace, a King
--- and a Queen; the Ace wins over the King, the King over the Queen and the
    Queen over the Ace. The game is played in two steps.
--- During the first step, the dealer gives one card to the player, keeps one
--- and put the last one, hidden, on table. The dealer sees all cards, while
--- the player only sees his.
--- During the second step, the player can ask to swap his card with the one
--- on table, or to keep it.
--- At the end, the winner is the one with the winning card.
---
--- In this version of the game, the game is played again and again,
--- and fairness constraints are added such that a fair path happens only if
--- the dealer fairly give the cards.
---
--- In this version of the game, additional state variables are added to track
--- the last played actions, and the fairness constraints are set only on state
--- variables (not on input variables), thanks to these additional variables.
---
--- Simon Busard <simon.busard@uclouvain.be>, 11/04/13

MODULE Player(step, pcard, ddcard)
    --- step is the id of the step in the game
    --- (0 for starting, 1 for first phase, 2 for the end)
    --- pcard is the card of the player (none, Ac, K, Q)
    --- ddcard is the card of the dealer, disclosed on table (none, Ac, K, Q)

    IVAR action : {none, keep, swap};

    VAR played : {none, keep, swap};

    INIT played = none

    --- Protocol
    TRANS
    action in case step = 1 : {keep, swap};
                  step != 1 : {none};
              esac
```

```
    TRANS next(played) = action


MODULE Dealer(step, dcard, pcard)
    --- step is the id of the step in the game
    --- (0 for starting, 1 for first phase, 2 for the end)
    --- pcard is the card of the player (none, Ac, K, Q)
    --- dcard is the card of the dealer (none, Ac, K, Q)

    IVAR action : {none, dealAK, dealAQ, dealKA, dealKQ, dealQA, dealQK};

    VAR played : {none, dealAK, dealAQ, dealKA, dealKQ, dealQA, dealQK};

    INIT played = none

    --- Protocol
    TRANS
    action in case step = 0 : {dealAK, dealAQ, dealKA, dealKQ, dealQA, dealQK};
                step != 0 : {none};
             esac

    TRANS next(played) = action


MODULE main

    VAR step  : 0..2;
        pcard : {none, Ac, K, Q};
        dcard : {none, Ac, K, Q};
        ddcard : {none, Ac, K, Q};
        dealer : Dealer(step, dcard, pcard);
        player : Player(step, pcard, ddcard);


    INIT step = 0 & pcard = none & dcard = none & ddcard = none

    TRANS
    next(step) = (step + 1) mod 3

    TRANS
    next(pcard) = case step = 0 : case dealer.action in {dealAK, dealAQ} : Ac;
                                      dealer.action in {dealKQ, dealKA} : K;
                                      dealer.action in {dealQK, dealQA} : Q;
                                      TRUE                              : none;
                                  esac;
                      step = 1 : case player.action = keep : pcard;
                                      player.action = swap :
                                       case (pcard = Ac & dcard = K) |
                                            (pcard = K & dcard = Ac) : Q;
                                            (pcard = Ac & dcard = Q) |
                                            (pcard = Q & dcard = Ac) : K;
                                            (pcard = Q & dcard = K) |
                                            (pcard = K & dcard = Q)  : Ac;
                                            TRUE                     : none;
                                       esac;
                                      player.action = none : none;
```

```
                                esac;
                        step = 2 : none;
                esac

    TRANS
    next(dcard) = case step = 0 : case dealer.action in {dealKA, dealQA} : Ac;
                                      dealer.action in {dealAK, dealQK} : K;
                                      dealer.action in {dealKQ, dealAQ} : Q;
                                      dealer.action = none          : none;
                                  esac;
                        step = 1 : dcard;
                        step = 2 : none;
                esac

    TRANS
    next(ddcard) = case step = 0 : none;
                         step = 1 : dcard;
                         step = 2 : none;
                 esac


    DEFINE
    win := step = 2 & ( (pcard = Ac & dcard = K) |
                        (pcard = K & dcard = Q) |
                        (pcard = Q & dcard = Ac) );
    lose := step = 2 & ( (pcard = Ac & dcard = Q) |
                         (pcard = K & dcard = Ac) |
                         (pcard = Q & dcard = K) );

FAIRNESS dealer.played = dealAK
FAIRNESS dealer.played = dealAQ
FAIRNESS dealer.played = dealKA
FAIRNESS dealer.played = dealKQ
FAIRNESS dealer.played = dealQA
FAIRNESS dealer.played = dealQK
```

Annexe F

Fichier transmission.smv

```
--- One bit transmission system
---
--- The system is composed of one bit of information, a sender
--- and a transmitter.
--- The sender can ask to send the bit or to wait, while the transmitter can
--- choose to transmit the bit or block the transmission.
--- The bit is received if and only if the sender sends the bit,
--- and the transmitter transmits it.
---
--- Simon Busard <simon.busard@uclouvain.be>, 23/03/13

MODULE Sender()

    IVAR  action : {send, wait};

    --- Protocol: nothing, can always do both actions


MODULE Transmitter()

    IVAR action : {transmit, block};

    --- Protocol: nothing, can always do both actions


MODULE main

    VAR received : boolean;
        sender : Sender();
        transmitter : Transmitter();

    INIT !received

    TRANS next(received) = (sender.action = send & transmitter.action = transmit
                          ? TRUE : received);
```

Annexe G

Fichier transmissionPostFair.smv

```
--- One bit transmission system
---
--- The system is composed of one bit of information, a sender
--- and a transmitter.
--- The sender can ask to send the bit or to wait, while the transmitter can
--- choose to transmit the bit or block the transmission.
--- The bit is received if and only if the sender sends the bit,
--- and the transmitter transmits it.
---
--- In this version of the system, fairness constraints are added such that
--- a fair path only happens when the transmitter allows the transmission
--- infinitely often (representing a fair channel).
---
--- In this version of the system, additional state variables are added to
--- track the last played actions, and fairness constraints are only specified
--- on state variables (not on input variables).
---
--- Simon Busard <simon.busard@uclouvain.be>, 23/03/13

MODULE Sender()

    IVAR  action : {send, wait};

    --- Protocol: nothing, can always do both actions


MODULE Transmitter()

    IVAR action : {transmit, block};

    --- Protocol: nothing, can always do both actions


MODULE main

    VAR received : boolean;
        sender : Sender();
        transmitter : Transmitter();
        sent : boolean;
        waited : boolean;
```

```
    transmitted : boolean;
    blocked : boolean;

INIT !received & !sent & !waited & !transmitted & !blocked

TRANS next(received) = (sender.action = send & transmitter.action = transmit
                      ? TRUE : received);

TRANS next(sent) = (sender.action = send)
TRANS next(waited) = (sender.action = wait)
TRANS next(transmitted) = (transmitter.action = transmit)
TRANS next(blocked) = (transmitter.action = block)


FAIRNESS transmitted
```