Model-checking et Interprétation de Modèles UML

Valentin Besnard (ESEO), Ciprian Teodorov (ENSTA Bretagne),
Matthias Brun (ESEO), Frédéric Jouault (ESEO),
Philippe Dhaussy (ENSTA Bretagne)

Novembre, 2018

1 Objectifs

Ce document vise à décrire les procédures d'installation et d'utilisation des outils permettant l'interprétation et la vérification formelle de modèles UML. UML (Unified Modeling Language) est un langage de modélisation utilisé dans le domaine de l'informatique pour décrire le comportement d'un système. Néanmoins, la plupart du temps, les diagrammes UML réalisés en phase de conception ne sont utilisés qu'à des fins documentaires. L'utilisation d'un interpréteur de modèle permet d'aller plus loin en exécutant ces modèles UML. En connectant un outil d'analyse de modèles (ici le model-checker Plug), il est possible de simuler ces modèles ainsi que de vérifier des propriétés formelles correspondants aux exigences système. Il est ainsi possible de valider et de vérifier le modèle du système dès la phase de conception en détectant au plus tôt des erreurs de conception ou des comportements indéterminés.

2 Architecture d'Analyse de Modèles

Pour analyser nos modèles UML, nous allons connecter l'outil Plug à l'interpréteur de modèles. Plug est un outil de vérification formelle permettant de faire de la simulation et du model-checking de propriétés formelles. La connexion de cet outil peut être réalisée en mode "standalone" (Figure 1) ou en mode "remote" (Figure 2) via un serveur de langage appelé emi-analysis.
En mode "standalone", Plug est directement relié au serveur de langage grâce à son runtime plug-runtime-via-tcp. Plug et le serveur de langage sont ensuite connectés via TCP à l'interpréteur de modèles (par défaut sur le port 12345).
En mode "remote", Plug est indépendant du serveur de langage. Il peut se connecter au serveur de langage via une connexion TCP (par défaut sur le port 12346). Le serveur de langage est ensuite connecté via TCP à l'interpréteur de modèles (par défaut sur le port 12345).
Il est également possible de simuler ou de model-checker un modèle s'exécutant sur une carte embarquée. Dans ce cas, il est nécessaire d'ajouter un composant entre le serveur de langage et l'interpréteur pour convertir les trames TCP en trame série (e.g., UART, USB). En effet, les microcontrôleurs embarqués n'ont pas toujours de périphériques permettant de gérer une connexion TCP, il est donc nécessaire d'utiliser un protocole série.

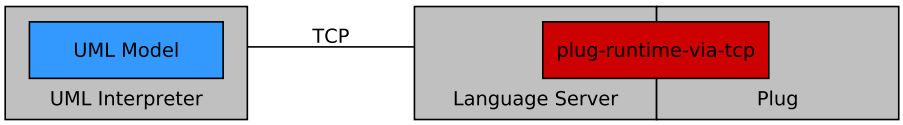

FIGURE 1 – Architecture en mode "standalone"

FIGURE 2 – Architecture en mode "remote"

3 Installation des Outils

3.1 Téléchargements des Outils

Télécharger les 3 outils nécéssaires : l'interpréteur de modèles, tUML qui va servir à concevoir les modèles, et Plug.

- Cloner le repository de l'interpréteur UML
  `git clone http://ValentinBesnard@forge.eseo.fr/git/uml-interpreter/`
- Cloner le repository de tUML
  `git clone http://ValentinBesnard@forge.eseo.fr/git/tUML/`
- Téléchager la version courante (OBP2 daily) de Plug sur le site :
  `https://plug-obp.github.io/`.
  Copier le JAR `uml-interpreter/plug/plug-runtime-via-tcp.jar` dans le répertoire `lib` de Plug. Dans `bin/obp2` de Plug, changer la variable `CLASSPATH` en : `CLASSPATH=$APP_HOME/lib/*`

3.2 Installation de Packages Supplémentaires

Installer les packages suivants :

- Pour la compilation (utilisation de gcc ou de clang)
  `apt install make`
  `apt install gcc` ou `apt install clang`
- Pour la simulation et le model-checking
  `apt install binutils`
  `apt install binutils-multiarch`
  `apt install dwarves`
- Pour la compilation sur cible embarquée ARM
  `apt install gcc-arm-none-eabi`
  `apt install gdb-arm-none-eabi` ou `apt install gdb-multiarch`
- Pour le flashage de l'interpréteur sur cible embarquée ARM
  `apt install opneocd` ou installation de stlink (`https://github.com/texane/stlink`)
- Pour la visualisation de graphes

```
apt install dot
```

3.3 Création des Variables d'Environnement

Renseigner les variables d'environnement suivantes dans le fichier ~/.bashrc (sous Linux) ou dans un terminal :

- Le chemin vers le répertoire de tUML dans la variable d'environnement TUML_HOME
  `export TUML_HOME="<path_vers_le_repertoire_tUML>"`
- Le chemin vers le répertoire de l'interpréteur dans la variable d'environnement EMI_HOME
  `export EMI_HOME="<path_vers_le_répertoire_de_UML_interpréteur>"`
- Ajouter au PATH le chemin vers le répertoires contenant les scripts de l'interpréteur
  `export PATH="$PATH:<path_vers_le_repertoire_de_UML_interpréteur>/scripts/"`

Si vous avez renseigné les variables d'environnement dans le fichier ~/.bashrc, il est nécessaire de recharger ce fichier en entrant dans votre terminal :

```
source ~/.bashrc
```

4 Création d'un Modèle UML

La création d'un modèle UML nécessite l'utilisation d'un formalism graphique (e.g., Papyrus [1]) ou d'un formalisme textuel. Dans le cadre de ce projet, nous avons choisi de travailler avec le formalisme textuel tUML. Ce formalisme permet de décrire l'architecture d'un système (e.g., avec des classes, des associations) ainsi que son comportement (e.g., avec des machines à états, des activités). Un sérialiseur permet ensuite de sérialiser les modèles tUML en UML (au format standard XMI).
L'interpréteur de modèles que nous allons utiliser ne supporte qu'un sous-ensemble d'UML pouvant être représenté par trois types de diagrammes :

- Diagramme de classes : Permet de représenter le système d'un point de vue structurel. Une classe peut être active ou passive. Une classe active possède un comportement alors qu'une classe passive sert seulement à encapsuler des données (i.e., elle n'a pas de comportement).
- Diagramme de composite structure : Permet d'instancier les différents objets (actifs ou passifs) du système et de décrire les liens qu'ils ont entre eux.
- Diagramme de machine à états : Permet de décrire la partie comportementale d'un object actif du système.

Une description plus détaillée de ce sous-ensemble est donnée en Appendice A.
Des exemples de modèles en tUML sont disponibles dans `uml-interpreter/models`.

Pour créer un modèle UML avec tUML, vous devez tout d'abord modéliser votre système (ici appelé "MyModel") dans un fichier `MyModel.tuml`. L'extension `.tuml` permet de spécifier qu'il s'agit d'un modèle tUML. Une fois le système modélisé en tUML, il faut le sérialiser en UML (`MyModel.uml`) et en langage C afin de pouvoir être chargé dans l'interpréteur de modèle. La commande permettant de réaliser la sérialisation est :

```
./uml_model_serialization.sh MyModel.tuml
```

L'opération de sérialisation va créer le fichier `MyModel.uml`, les fichiers de code source C dans le

1. `https://www.eclipse.org/papyrus/`

répertoire : `uml-interpreter/uml-interpreter-c/src/models/MyModel` et un répertoire `output` contenant des diagrammes en plantuml. Vérifier si votre modèle possède des erreurs et corriger votre modèle si c'est le cas jusqu'à ce que la sérialisation fonctionne.

5 Compilation de l'Interpréteur et du Modèle UML

Le chargement du modèle UML dans l'interpréteur est réalisé à la compilation. L'interpréteur et le modèle UML sont compilés et linkés ensemble pour produire un binaire exécutable. Cette opération de compilation peut être configurée par un certain nombre de paramètres renseignés dans `uml-interpreter/uml-interpreter-c/Makefile.config`.

Voici la liste de ces paramètres et leurs significations :

- **MODEL** : le nom du modèle à interpréter (e.g., `MODEL = MyModel`).
- **DEVICES** : les différents périphériques matériels à activer.
- **EVENT_POOL** : l'implémentation de l'event pool à utiliser. L'ordre de dispatching des évènements reçus par un objet actif est un point de variation sémantique d'UML. Différentes implémentations de l'event pool ont été définies afin de fournir différentes sémantiques possibles.
- **GUARD_LANGUAGE** : le langage d'action à utiliser pour la définition des gardes des transitions des machines à états.
- **EFFECT_LANGUAGE** : le langage d'action à utiliser pour la définition des effets des transitions des machines à états.
- **TARGET** : la cible sur laquelle l'interpréteur va être déployé :
  - host : sur le PC de développement
  - stm32f4discovery : sur une carte embarquée STM32f4 Discovery
  - at91sam7sek : sur une carte embarquée AT91SAM7S-EK
- **COMPILER** : le compilateur à utiliser.
- **ANALYSIS** : mettre à "yes" pour faire de l'analyse de modèles avec Plug ou à "no" sinon. En mode analyse de modèles, il est aussi intéressant d'activer "with diff" pour améliorer les performances et d'activer "with checking" pour faire du model-checking et pas seulement de la simulation.
- **TRANSLATION** : Utile pour visualiser le contenu d'une configuration avec Plug. Différents outils peuvent être utilisés :
  - pahole
  - objdump
  - dwarfdump
- **DEBUG** : pour afficher plus ou moins d'information sur la sortie standard.
- **SANITIZER** : pour activer des outils d'analyse dynamique de code.
- **TEST** : pour lancer la compilation des tests juste après la compilation de l'interpréteur.

Une fois les paramètres de compilation réglés selon vos besoins, vous pouvez :

- Lancer la compilation de l'interpréteur avec la commande : `make all`
- Supprimer les fichiers générés lors de la compilation avec la commande : `make clean`
- Lancer l'exécution sur le PC de dévelopment avec la commande : `make run`
- Lancer le debug avec la commande : `make debug`

Le flashage de l'interpréteur sur STM32 Discovery et la communication avec cette carte embarquée nécessite la création de plusieurs règles udev. Sous Linux, les règles udev doivent être ajoutées dans le répertoire `/etc/udev/rules.d`.
Pour le flashage de l'interpréteur sur STM32 Discovery, il faut ajouter la ou les règles udev correspondant à la version de stlink utilisée. Il est possible que ces règles soient installées dans le répertoire `/lib/udev/rules.d` lors de l'installation de stlink ou openocd. Si c'est le cas, il suffit juste de copier ces règles dans le répertoire `/etc/udev/rules.d`. Sinon, les règles udev suivantes doivent être ajoutées :

49-stlinkv1.rules

```
1  # stm32 discovery boards, with onboard st/linkv1
2  # ie, STM32VL
3
4  SUBSYSTEMS=="usb", ATTRS{idVendor}=="0483", ATTRS{idProduct}=="3744", \
5      MODE:="0666", \
6      SYMLINK+="stlinkv1_%n"
7
8  # If you share your linux system with other users, or just don't like the
9  # idea of write permission for everybody, you can replace MODE:="0666" with
10 # OWNER:="yourusername" to create the device owned by you, or with
11 # GROUP:="somegroupname" and mange access using standard unix groups.
```

49-stlinkv2.rules

```
1  # stm32 discovery boards, with onboard st/linkv2
2  # ie, STM32L, STM32F4.
3  # STM32VL has st/linkv1, which is quite different
4
5  SUBSYSTEMS=="usb", ATTRS{idVendor}=="0483", ATTRS{idProduct}=="3748", \
6      MODE:="0666", \
7      SYMLINK+="stlinkv2_%n"
8
9  # If you share your linux system with other users, or just don't like the
10 # idea of write permission for everybody, you can replace MODE:="0666" with
11 # OWNER:="yourusername" to create the device owned by you, or with
12 # GROUP:="somegroupname" and mange access using standard unix groups.
```

49-stlinkv2-1.rules

```
1  # stm32 nucleo boards, with onboard st/linkv2-1
2  # ie, STM32F0, STM32F4.
3  # STM32VL has st/linkv1, which is quite different
4
5  SUBSYSTEMS=="usb", ATTRS{idVendor}=="0483", ATTRS{idProduct}=="374b", \
6      MODE:="0666", \
7      SYMLINK+="stlinkv2-1_%n"
8
9  # If you share your linux system with other users, or just don't like the
10 # idea of write permission for everybody, you can replace MODE:="0666" with
```

```
# OWNER:="yourusername" to create the device owned by you, or with
# GROUP:="somegroupname" and mange access using standard unix groups.
```

Pour la communication avec la carte via un lien série, il est necéssaire d'ajouter un fichier `50-usbftdi.rules` dans /etc/udev/rules.d avec le contenu suivant :

```
ACTION=="add", SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_interface", \
ATTRS{idVendor}=="0483", ATTRS{idProduct}=="5740", \
DRIVER=="", \
RUN+="/sbin/modprobe -b cp210x"

ACTION=="add", SUBSYSTEM=="drivers", \
ENV{DEVPATH}=="/bus/usb-serial/drivers/cp210x", \
ATTR{new_id}="0483 5740"
```

Pour recharger les règles udev, entrez la ligne suivante dans un terminal :
`udevadm control -reload`
Une fois ceci effectué, il est possible de flasher la carte STM32 Discovery avec la commande `make flash`.
Vous devez également vous ajouter au groupe dialout en entrant les deux lignes suivantes (en replacant "`<username>`" par votre nom d'utilisateur) :
`sudo adduser <username> dialout`
`sudo usermod -a -G dialout <username>`

La communication avec Plug nécessite un composant intermédiaire `emi-tcp2serial` afin de convertir les trames TCP en trames série. D'un point de vue utilisateur, cela revient à se connecter en TCP à l'interpréteur par défaut sur le port 12345. Pour brancher la carte STM32 Discovery, il est nécessaire d'utiliser un câble USB vers série en effectuant le branchement suivant sur la STM32 :

- Fil blanc sur la broche PB6
- Fil vert sur la broche PB7
- Fil noir sur une des broches GND
- Fil rouge à **NE SURTOUT PAS BRANCHER**

Une fois le branchement effectué, vous pouver lancer la compilation de `emi-serial2tcp` en allant dans le répertoire du même nom et en faisant `make all`. Ensuite, vous pouvez lancer ce convertisseur TCP vers série en lancant la commande `./bin/tcp2serial`.

6 Analyse du Modèle avec Plug

Plug est un outil de vérification formelle permettant de faire de la simulation et du model-checking. Le model-checking est une technique permettant de vérifier la validité de certaines propriétés sur le modèle. Dans le cadre de ce projet, les propriétés seront écrites en utilisant le formalisme LTL (Linear Temporal Logic).

Pour lancer Plug, vous devez vous placer dans le répertoire `plug-obp-daily/bin` puis entrer dans un terminal la commande suivante :

`./obp2`
L'interface graphique de Plug va alors apparaitre (Figure 3).

6.1 Chargement des Paramètres de Configuration

Avant de pouvoir analyser le modèle, il faut configurer le runtime utilisé (ici plug-runtime-via-tcp). Pour le configurer, il est nécessaire de charger un fichier avec l'extension `.emi`. Le fichier `uml-interpreter/plug/config.emi` peut par exemple être utilisé.

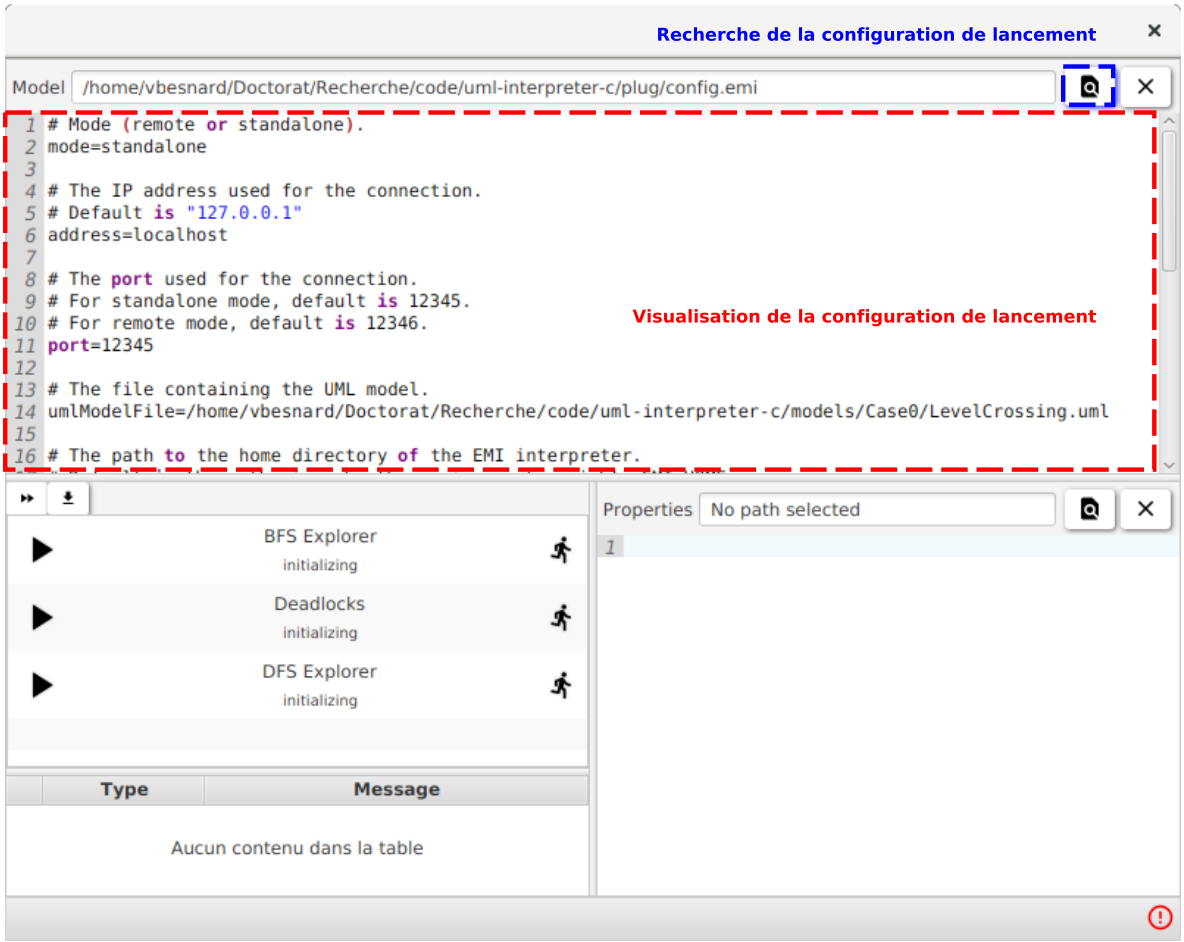

FIGURE 3 – UI de Plug pour le chargement de la configuration de lancement

Différents paramètres de configuration peuvent être réglés :

- **mode** : le mode de lancement (standalone ou remote) (cf section 2).
- **address** : l'addresse IP à laquelle Plug doit se connecter.
- **port** : le port auquel Plug doit se connecter (cf section 2).
- **umlModelFile** : le chemin vers le modèle UML à analyser (à adapter sur votre machine).
- **emiHome** : le chemin correspondant à la variable d'environnement EMI_HOME (à

adapter sur votre machine).

- **target** : la cible sur laquelle l'interpréteur s'exécute.
- **compiler** : le compilateur à utiliser pour compiler les propositions atomiques des propriétés.
- **makeTool** : l'outil à utiliser pour lancer les makefiles de compilation des propositions atomiques des propriétés.
- **dwarfTool** : l'outil à utiliser pour parser les informations dwarf nécessaires pour l'affichage de la configuration.

6.2 Model-checking

La Figure 4 illustre l'interface utilisateur utilisée pour le model-checking. Celle-ci se décompose en deux parties. La premère partie (à gauche) est utilisée pour lancer la vérification des propriétés LTL ou des algorithmes génériques de model-checking. Plug fournit trois algorithmes génériques :

- **BFS Explorer** : un algorithme pour explorer l'espace d'état en largeur.
- **Deadlocks** : un algorithme utilisé pour la détection de deadlocks.
- **DFS Explorer** : un algorithme pour explorer l'espace d'état en profondeur.

Si une propriété est violée (i.e., non-vérifiée), il est possible de cliquer sur le bonhomme rouge pour afficher la trace du contre-exemple dans le simulateur. La seconde partie de l'UI (à droite) est utilisée pour écrire les propriétés LTL à vérifier.

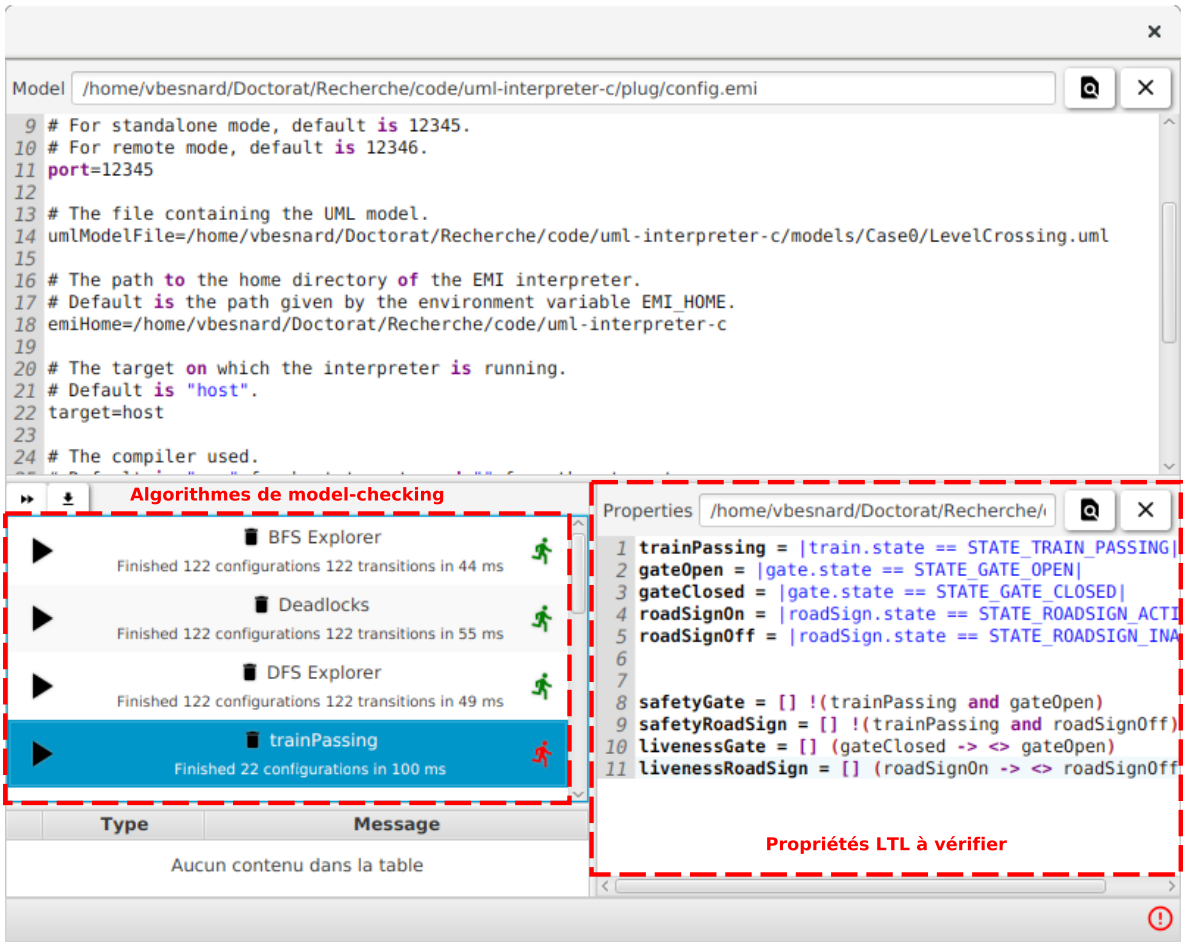

FIGURE 4 – UI de Plug pour le model-checking

6.3 Simulation

La Figure 5 illustre l'interface utilisateur utilisée pour la simulation. Cette interface se décompose en trois parties. La première (en haut à gauche) affiche la liste des transitions tirables pouvant être tirées par l"utilisateur. La seconde (en haut à droite) permet de visualiser le contenu de la configuration sélectionnée. Enfin la troisième (en bas) permet de visualiser le graphe des différentes traces explorées depuis le début de la simulation.

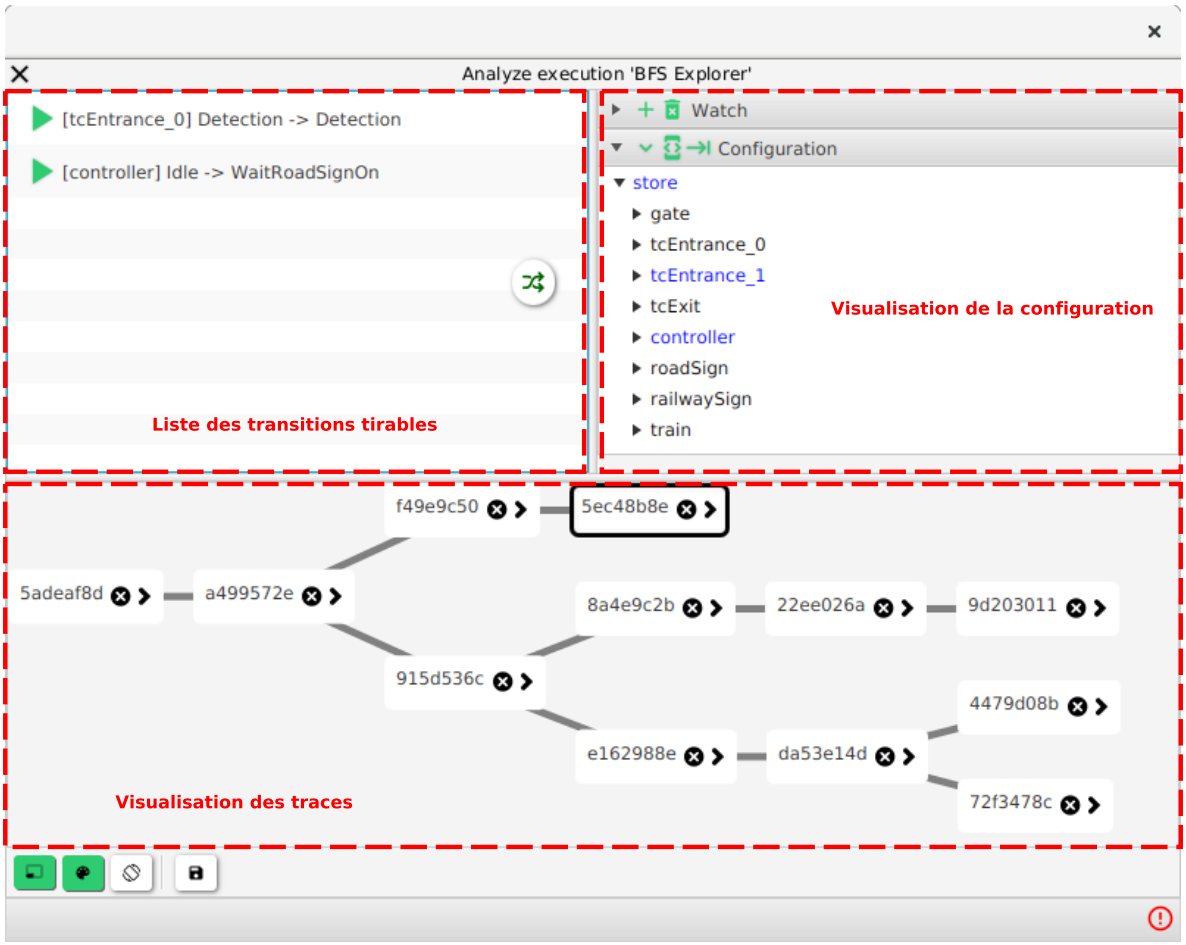

FIGURE 5 – UI de Plug pour la simulation

A Sous-ensemble UML Supporté

De façon plus détaillée, les concepts UML supportés par notre interpréteur sont :

- **Class** : Une *Class* classifies un ensemble d'objets et spécifie les concepts qui caractérisent la structure et le comportement de ces objets. Une classe peut avoir des attributs (provenant de ses associations ou des attributs primitifs (entiers ou booléens)), des réceptions, des interfaces, et elle peut également hériter d'une classe (seul l'héritage simple est supporté). Si une classe est active, elle peut également avoir un comportement définit à l'aide d'une machine à états.
- **Association** : Une *Association* définit un type pour un ensemble de liens. Une association permet de faire le lien entre des classes.
- **Property** : Une *Property* correspond à un attribut structurel (e.g., attribut d'une classe).
- **Connector** : Un *Connector* spéficie un lien qui permet la communication entre deux ou plusieurs instances.
- **ConnectorEnd** : Un *ConnectorEnd* est une extrémité d'un connecteur, qui permet de lier le connecteur à l'élément à connecter.
- **Signal** : Un *Signal* est la spécification d'un type de communiqation entre des objets dans lequel une réaction asynchrone est déclenchée chez le destinataire.
- **SignalEvent** : Un *SignalEvent* correspond à la réception d'une occurrence d'un signal asynchrone.
- **Reception** : Une *Reception* est une déclaration indiquant qu'une classe est préparer à réagir à la réception d'un signal.
- **PrimitiveType** : Un *PrimitiveType* définit un type de données prédéfini. Parmi les types primitifs, seuls les entiers et les booléens sont supportés.
- **Enumeration** : Une *Enumeration* est un type de données dans lequel les valeurs sont énumérées en tant que *EnumerationLiterals*.
- **EnumerationLiteral** : Un *EnumerationLiteral* est une value définie par l'utilisateur pour une énumération.
- **Interface** : Une *Interface* permet de définir un ensemble de services devant être implémentés par les classes qui implémentent l'interface. Dans ce sous-ensemble, une interface ne peut définir que des réceptions.
- **InterfaceRealization** : Une *InterfaceRealization* correspond à l'implémentation d'une interface par une classe.
- **Port** : Un *Port* permet de spécifier un point d'interaction entre une classe et son environnement. Les ports permettent notamment de transférer des évènements.
- **Behavior** : Un *Behavior* correspond au comportement d'une classe. L'interpréteur ne peut gérer qu'un type de comportement : les machines à états.
- **StateMachine** : Une *StateMachine* est une machine à états et permet de décrire le comportement d'une classe.
- **Region** : Une *Region* est l'objet de plus haut niveau contenu dans une machine à états et sert de containeur pour les noeuds et les transitions de cette machine à états.
- **Vertex** : Un *Vertex* correspond à un noeud d'une machine à états. Un vertex peut être de type *State* ou de type *Pseudostate*.
- **State** : Un *State* est un noeud d'une machine à états dans lequel des invariants sont vérifiées.

- **Pseudostate** : Un *Pseudostate* est un noeud d'une machine à états ne pouvant pas être considéré comme un état courant de celle-ci. Ils permettent de modéliser des comportements plus complexes au niveau des transitions des machines à états. Seuls les pseuodstates initial, choice, et junction sont supportés.
- **Transition** : Une *Transition* représente un arc entre un noeud source et un noeud cible.
- **Trigger** : Un *Trigger* spécifie un point spécifique dans lequel l'occurrence d'un évènement peut déclencher un effet dans un comportement (e.g., le tirage d'une transition).
- **Constraint** : Une *Constraint* exprime une condition ou une restriction sur un élement ou un ensemble d'éléments (e.g., une garde sur une transition d'une machine à états).
- **OpaqueExpression** : Une *OpaqueExpression* permet d'exprimer le calcul d'un prédicat dans un langage différent d'UML. Ceci est typiquement utilisé pour spécifier les gardes des transitions des machines à états.
- **OpaqueBehavior** : Un *OpaqueBehavior* est un comportement dont la spécification est donné dans un langage différent d'UML. Ceci est typiquement utilisé pour spécifier les effets des transitions des machines à états (e.g., l'envoi d'évènements, l'affectation d'une nouvelle valeur à un attribut).