TP Scade

Annie Ressouche
Inria-Sophia

24 Novembre 2011

Le but de ce TP est d'utiliser le logiciel SCADE, outil graphique pour spécifier des applications data flow et évènementielles (automates parallèles).

1 Utilisation de SCADE

1.1 Découverte de SCADE

Elle va se faire à l'aide de l'exemple COUNTER. La documentation du logiciel peut se consulter en ligne (HELP du menu).

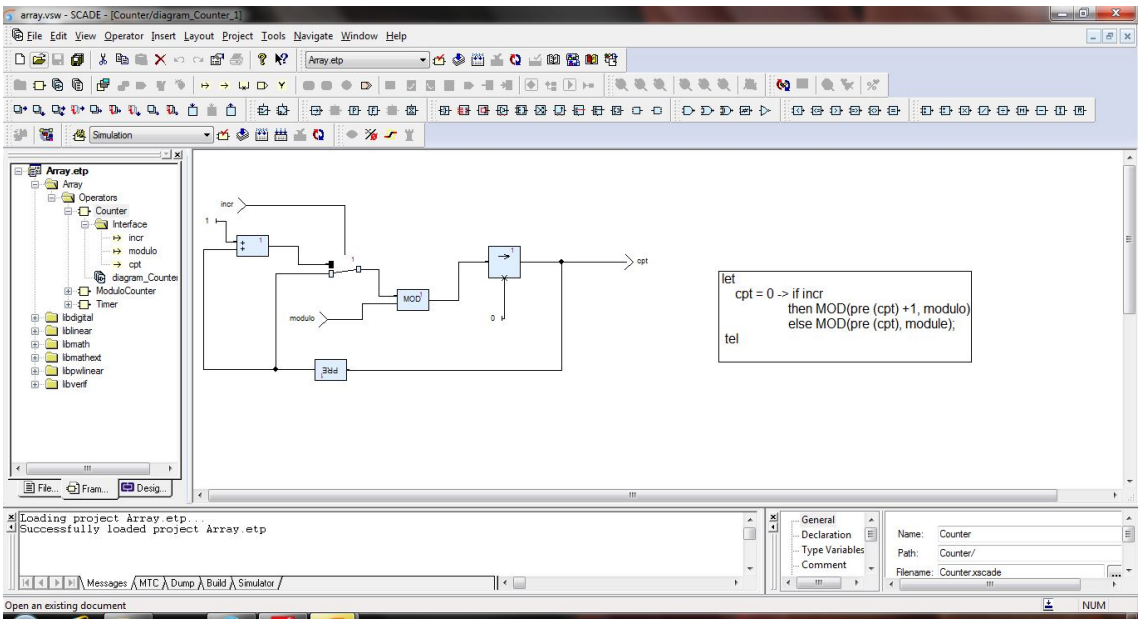

FIG. 1 – Scade Apparence

Voici le code de l'application COUNTER :

```
operator COUNTER (init, incr : int; reset: bool) returns (n:int)
 n = init -> if reset then init else pre(n)+incr;
```

Spécification

Tout d'abord, il faut créer un projet :>File>New ouvre un browser. Il faut donner un nom au projet (array sur l'exemple figure 1). Le fichier *array.vsw* est créé. La figure 1 représente l'espace de travail dans lequel l'application est spécifiée . Cet espace comporte essentiellement qautre fenêtres :

1. une fenêtre (en haut) comportant plusieurs barres d'outils utiles pour concevoir l'application :
   (a) une barre d'outils pour appeler les différentes fonctionnalités de SCADE : vérification de la cohérence sémantique du modèle (Quick Check), génération de code C, simulation
   (b) une barre d'outils pour dessiner des opérateurs "data flow" (pre, if, init,....) et des opérateurs pour dessiner des automates (state machine, state, ...). Dans cette barre d'outils, on trouve également des boutons pour définir les inputs et les outputs.
   (c) la dernière barre d'outils contient les opérateurs arithmétiques et booléens.

   Cette visualisation peut être changée dans le menu View.
2. une fenêtre (à gauche) représentant l'application avec plusieurs vues :
   – File View est utilisée pour visualiser les différents fichiers du projet : la spécification (Count.etp), l'organisation de l'application (Model Files) et les librairies d'opérateurs prédéfinis.
   – Framework View est la vue principale dans laquelle on doit construire l'application. Les différents opérateurs qui composent l'application sont décrits dans la partie Operator. Toutefois, si vous construisez une application hiérarchisée (avec des opérateurs appelant eux mêmes d'autres opérateurs que vous avez définis), la hiérarchie n'apparait pas dans cette vue, elle est visualisé dans File View.
   – Designer Verifier View est utilisée pour voir les outils de vérification définis pour valider l'application. Elle sera utile lorsque vous ferez des vérifications formelles.
3. une fenêtre centrale pour dessiner l'application
4. une fenêtre de log (en bas)
5. une autre fenêtre (en bas à droite) pour spécifier les caractéristiques des objets sélectionnés.

Pour dessiner le noeud Count, on se place dans Framework. Pour définir un nouvel opérateur, click droit dans Operator et ensuite Insert. En cliquant sur Operator, une fenêtre vous demande le nom de l'opérateur à définir. Sur l'exemple figure 1, nous avons choisi d'appeler le noeud Count. Count est défini avec une interface Interface et un corps. Pour dessiner le noeud Count : double click sur diagram_Count, on se place ensuite dans la fenêtre de dessin (au centre) et on utilise les opérateurs prédéfinis dans la barre d'outil, ceux des librairies ou bien ceux déjà définis dans l'application. Vous pouvez aussi définir un automate , en faisant glisser l'onglet "state machine" depuis la barre d'outils. Vous dessinez alors un noeud en Esterel graphique. Pour l'exemple présent, nous avons défini le noeud Count avec des opératuers "data flow". Pour ce faire les opérateurs and, not Init et pre ont été utilisés. Pour définir l'interface : click droit de la souris et Insert. On définit ainsi les inputs et les outputs. Il possèdent un nom et un type. Ce type peut être prédéfini (int par exemple) ou externe et défini dans l'onglet Types. Le type

par défault est booléen. On peut aussi utiliser un raccourci en faisant glisser directement le bouton input (resp output) depuis la barre d'outil.

On peut alors vérifier la cohérence sémantique de l'opérateur `Count` (`Quick Check`). On peut vérifier la cohérence séparement de chaque opérateur.

Simulation interactive

1. La simulation interactive est lancée à l'aide du bouton `Simulate` dans la barre d'outil [1].
2. La principale fenêtre de simulation montre le modèle simulé avec les valeurs en bleu sur les connections. Pour changer la valeur d'un input, il faut choisir `Instance View`, en bas dans la fenêtre de gauche. Ensuite, selectionner la variable et `f2` pour changer la valeur.
3. Trois autres fenêtres de visualisation sont disponibles :
   (a) une fenêtre de simulation pour affficher des informations sur la simulation, l'historique des actions faites, ...
   (b) une fenêtre "d'observation" (`watch`) pour observer les valeurs des variables. On peut également changer leur valeur dans cette fenêtre. Pour afficher la fenêtre, click sur le bouton `watch` dans la barre de menu et pour observer une variable dans cette fenêtre, click droit sur la variable dans la fenêtre principale, à gauche et ensuite sélectionner `watch`.
   (c) une fenêtre permettant une vue graphique de l'évolution des variables (`Graph`). Comme précedemment, pour l'afficher il faut sélectionner l'icone `graph` dans la barre de menu et pour y ajouter une variable, click droit sur la variable et sélectionner `graph`.
4. On peut exécuter un scénario significatif (contenant les situations intéressantes à explorer). On peut y inclure des `Reset` qui renvoient à l'état initial. Observer l'évolution des valeurs sur les connections et dans les différentes fenêtres (`watch` et `graph`).
5. On peut sauver un scenario (`Count.sss`) et le rejouer plus tard.

2 Porte de tramway

On se propose d'écrire un programme qui controle (1) l'ouverture et la fermeture automatique d'une porte de tramway, (2) le mouvement d'une passerelle extractible qui peut sortir et se poser sur le quai pour faciliter l'accès aux poussettes, fauteuils roulants,...

2.1 L'environnement du contrôleur

Il y a 3 éléments dans l'environnement du contrôleur :

1. **la porte et la passerelle :** le contrôleur leur envoie les commandes *fermerPorte*, *rentrerPass* et *ouvrirPorte*, *sortirPass*. L'état de la porte est connue en permanence grace au capteur *porteOuverte* et un capteur d'entrée *passSortie* indique l'état de la passerelle. On suppose

[1] Si la simulation n'est pas active : Project>Code Generator>Set Active Configuretaion>simulation (click sur le bouton set active)

que la porte fonctionne bien : elle s'ouvre et se ferme quand on lui demande, la seule inconnue étant le temps qu'elle met à réagir. Le contrôleur doit donc maintenir les ordres d'ouverture et de fermeture jusqu'à ce que la porte soit dans l'état désiré. La passerelle a un comportement similaire.

2. **l'usager :** envoie la demande *demandePorte*. Le contrôleur doit filtrer et enregistrer les demandes pour pouvoir les traiter en toute sécurité. Il peut également demander l'ouverture de la passerelle (*demandePass*). Cette demande signifie que l'usager demande aussi l'ouverture de la porte (cette demande peut être implicite) ;
3. **le tramway :** fournit au contôleur les informations sur son état :
   (a) *enStation* : signifie que le tramway est arrêté dans une station. Quand cet indicateur est vrai, la porte peut être ouverte et la passerelle sortie (si la demande a été faite) ;
   (b) *attentionDepart* : signifie que le ramassage des passagers est terminé et qu'il faut fermer fermer la porte. A partir de cet instant, le tramway va attendre du contrôleur le signal *portePassOk* signifiant que la porte est bien fermée et que la passerelle est rentrée ; ainsi il peut redémarrer. On fait des hypothèses fortes sur le fonctionnement du tramway : dès que la variable *enStation* devient vraie, elle le reste tant que *portePassOk* n'a pas été émis par le contôleur.

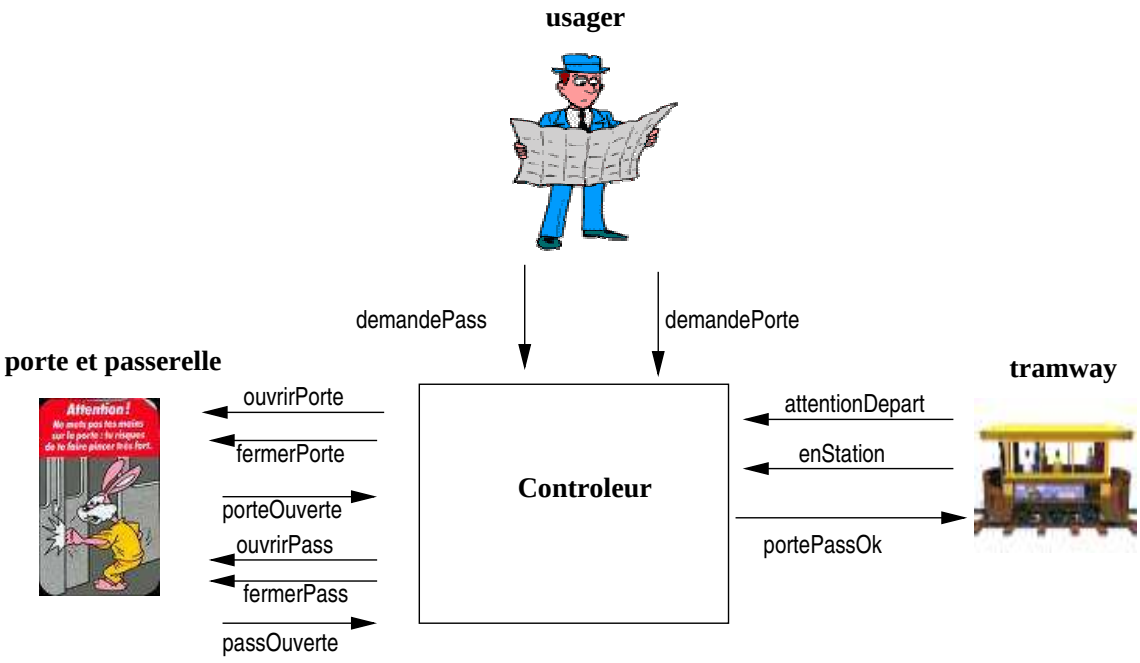

FIG. 2 – Le contrôleur de porte de tramway

2.2 Spécification du fonctionenemnt de la porte

Les interactions de la porte avec son environnement sont illustrées dans la figure2. Le but essentiel du contrôleur est de garantir la sécurité du tramway et des usagers : la porte doit toujours

être fermée en dehors des stations. Le tramway ne doit jamais rouler avec la passerelle sortie.

Le contrôleur doit satisfaire aux demandes des usagers dans la mesure du possible. En effet, un comportement sécuritaire serait de ne jamais ouvrir la porte ! Les demandes des utilisateurs sont prises en compte comme suit :

- hors station, *demandePorte* ou *demandePass* (ou les deux) vraie signifie que l'usager veut descendre au prochain arrêt, la demande sera traitée dans la prochaine station ;
- en station, et avant le signal *attentionDepart*, *demandePorte* provoque l'ouverture de la porte. Si la porte est déjà ouverte, *demandePass* ne peut être prise en compte car cela serait trop dangereux ;
- en station, après le signal *attentionDepart*, *demandePorte* et *demandePass* sont ignorées.

2.3 Implémentation en SCADE

Pour implémenter ce contrôleur avec SCADE, il est conseillé de s'appuyer sur une machine à états (state machine) possiblement hiérarchique. On peut aussi calculer des variables intermédiaires permettant de déclancher des transitions d'un état vers un autre. Par exemple : une variable *porte_pass_demandee* vraie quand la porte (ou la passerelle) a effectivement été demandée quand cette demande était recevable pourra être utile pour déclancher les transitions entre des états où la porte passe d'ouverte à fermée et ou la passerelle passe de rétractéé à sortie. Ces variables pourront être calculées dans des modèles spécifiques.

Ouverture de la porte et sortie de la passerelle

Pour implémenter l'ouverture de la porte et la sortie de la passerelle, on peut s'appuyer sur la variable introduite *porte_pass_demandee*. La commande *ouvrirPorte* est vraie dès que *enStation* et *porte_pass_demandée* sont vraies simultanément, et doit être maintenue jusqu'à l'ouverture effective. Même chose pour *sortirPass*.

Fermeture de la porte et de la passerelle

La commande *fermerPorte* doit étre envoyée quand :

1. le départ est immédiat ;
2. et la porte a été demandée (elle est donc soit ouverte soit en cours d'ouverture) ;
3. et la porte est ouverte.

fermerPorte doit être maintenue jusqu'à la fermeture effective. Le comportement est le même pour *renterPass* en tenant compte que lorsque la passerelle a été sortie, la porte a été ouverte.

portePassOk

Quand le départ est immédiat, le contrôleur doit envoyer le signal *portePassOk* au tramway pour lui permettre de redémarrer. Il y a deux cas possibles :

- la porte n'a pas été demandée et le tramway peut redémarrer ;
- la porte a été demandée et il faut attendre qu'elle soit fermée.

– la passerelle a été sortie et l'on doit attendre qu' elle soit rentrée (ce qui signifie que la porte a été ouverte et on se ramène au cas précédent)

2.4 Simulation et mise au point

Le simulateur de SCADE vous aidera à mettre au point votre implémentation. Des vérifications formelles seront nécessaires pour valider votre controleur

3 La vérification du fonctionnement de la porte de tramway et de la passerelle

3.1 Introduction

SCADE possède un `Design Verifier`, outil de model checking qui permet de prouver des propriétés de sécurité de fonctionnement. Dans SCADE, une propriété est exprimée à l'aide d'un opérateur qui est un observateur de la propriété à vérifier. Ce dernier écoute les entrées et les sorties de l'opérateur où la propriété doit être vérifiée (ou falsifiée si c'est une négation de propriété) et il émet une variable booléenne caractérisant la propriété (vraie ou fausse). Quand l' observateur est décrit, il faut créer un nouveau opérateur qui compose l'opérateur sur lequel on fait la vérification et l'observateur ; ensuite il faut brancher les entrées sorties écoutées par l'obervateur.

Pour faire la preuve d'une spécification complète avec une hiérarchie, on peut aussi prouver séparemment chaque composant de la hiérarchie et ainsi ne pas attendre que l'application complète soit vérifiée. Chaque fois qu'un composant est spécifié, vous pouvez vérifier son comportement en faisant des assertions sur les résultats des autres composants qui ont une influence sur lui. Ensuite quand on spécifie ces autres composants, on vérifie que les assertions faites sont effectivement vérifiées ; ainsi on peut les supprimer et faire les connections.

D'un point de vue pratique, appelons `Op` l'opérateur sur lequel on veut faire la vérification, `Obs` l'observateur de la propriété et `VerifOp` la composition de `Op` et `Obs`. Pour appeler le `Design Verifier`, il faut :

1. Dans la vue `Framework`, sélectionner l'output qui correspond à la propriété dans l'interface de `VerifNode`. Click droit et `Insert>ProofObjective`.
2. Dans la vue `Design verifier`, dans l'onglet `ProofObjective`, vous retrouvez votre propriété (vous pouvez la renommer) et vous pouvez aussi sélectionner `Analyze`. Cette commande lance le model checker interne de SCADE. La propriété est vérifiée .
3. En cas de réponse fausse un exemple montrant un chemin d'évaluation où la propriète est fausse est généré sous forme d'un scenario que l'on peut rejouer avec le simulateur.

Enfin, pour exprimer des propriétés de logique temporelle, une bibliothèque d'opérateurs logique `libverif` est fournie dans SCADE.

3.2 Les vérification demandées

On va prouver formellement que le contrôleur satisfait les propriété de sureté essentielles :

1. le tramway ne peut jamais rouler la porte ouverte ou la passerelle sortie.

2. la passerelle ne peut pas bouger (ni rentrer, ni sortir) quand la porte est ouverte.

Ces propriétés n'ont aucune chance d'être satisfaites quelque soit le comportement de l' environnement. On va donc prendre en compte des hypothèses sur le fonctionnement de la porte et de la passerelle et aussi du tramway (mais pas sur l'utilisateur qui peut faire ce qu'il veut !).

Dans un premier temps, on doit spécifier un observateur qui traduit la propriété.

Hypothèse sur la porte et la passerelle

On ne va pas tenir compte du délai de réaction de la porte et de la passerelle aux commandes d'activation (ce qui n'est pas possible pour le model checking). On va considérer que la porte ne s'ouvre uniquement que sur l'ordre *ouvrirPorte* et ne se ferme que sur la commande *fermerPorte*. La passerelle ne peut s'ouvrir qu'avec l'ordre *sortirPass* et ne se fermer que sur *renterPass*. Cette hypothèse n'est pas restrictive du point de vue du bon fonctionnement du contrôleur car si la propriété est satisfaite dans ce contexte, alors le contrôleur aura le comportement espéré. On va écrire une assertion qui stipule que

- la porte est initialement fermée ;
- elle ne s'ouvre que si *ouvrirPorte* est vraie ;
- elle ne se ferme que si *fermerPorte* est vraie ;

On décrit aussi les hypothèses sur le fonctionnement de la passerelle avec une assertion :

- la passerelle est initialement fermée ;
- elle ne sort que si *sortirPass* est vraie ;
- elle ne rentre que si *renterPass* est vraie ;

Hypothèses sur le tramway

On suppose que le tramway est initialement hors station. Il peut entrer en station à tout moment. En revanche, il ne peut repartir qu'après avoir émis *attentionDepart* et avoir reçu le signal *portePassOk*. Donc :

- il doit y avoir au moins une occurrence de *attentionDepart* entre les évènements d'arrivée et de départ de la station ;
- il doit y avoir au moins une occurrence de *portePassOk* entre l'arrivée et le départ de la station.

En SCADE, les hypothèses sont des assertions que l'on peut insérer dans le graphique quand elles sont simples sinon il faut leur associer un opérateur. En annexe, on donne le code lustre d' opérateurs utiles pour la vérification et qui ne sont pas dans la librairie de vérification. Vous devrez les définir dans votre projet si besoin est.

4 Annexe : des opérateurs logiques

Ces opérateurs permettent de décrire l'ordonnancement des évènements. Deux propriétés utiles pour exprimer les hypothèses faites sur l'environnement et les observateurs sont décrites par les opérateurs *always_from_to* et *once_from_to*. La première exprime qu'un évènement X doit impérativement être toujours vrai entre un évènement A et un évènement B. La seconde exprime

qu'un évènement X doit impérativement être vrai au moins une fois entre un évènement A et un évènement B (implies est dans la libverif).

```
node after (A: bool) returns (X: bool);
let
        X = false -> pre(A or X);
tel.


node always_since (C,A: bool) returns (X: bool);
let
        X =     if A then C
                        else if after(A) then C and pre(X)
                        else true;
tel.

node once_since (C,A: bool) returns (X: bool);
let
        X =     if A then C
                        else if after(A) then C or pre(X)
                        else true;
tel.

node always_from_to (X,A,B: bool) returns (X: bool);
let
        X = implies (after(A), always_since(C,A) or once_since(B,A));
tel.

node once_from_to (X,A,B: bool) returns (X: bool);
let
        X = implies (after(A) and B, once_since(C,A));
tel.
```