Laboratoire d'analyse et d'architecture des systèmes

MANUEL – JRdP

Fait par :
M'barek ZACRI

Été 2019

Table des matières

1 Couche décisionnelle : supervision avec réseaux de Petri

1.1 Choix du RdP comme outil de description de plan de navigation

Modélisation du plan de navigation : Le besoin de structuration du plan de navigation avec un outil formel dérive directement du niveau de complexité du plan. C'est le premier critère à considérer.
Un exemple de plan de navigation qui consiste à appeler les services simultanément ou successivement (séparées par des durées déterminées) n'annonce pas un besoin de structuration. Un simple script TCL (client TCL) ,comme nous avons vu dans le paragraphe 3.2.2, permettra de mettre en place ce plan. En revanche, trois principales raisons rendent l'utilisation d'un script TCL écrit d'une manière impérative plus délicate pour décrire le plan de navigation :

- Conditionnement des traitements : Avoir plusieurs statuts de différents services comme conditions entre les traitements du plan. Par exemple le traitement 2 s'exécute si le traitement 1 ,qui comporte plusieurs services à appeler, est bien achevé.
- La gestion d'erreurs : Pour pouvoir gérer une erreur levée par un service avec une certaine exception, nous avons besoin de spécifier le traitement de chaque cas de figure selon le plan.
  un switch sur les différents alternatives de traitements selon l'exception permettra de gérer cela, mais la complexité de la gestion peut rapidement le rendre délicat à utiliser. Imaginons que dans une gestion d'erreur on effectue un certain traitement, puis soit on relance les services erronés soit on lance d'autres services selon le résultat du traitement antérieur, on pourra toujours le coder d'une manière impérative mais un bon outil de modélisation facilite énormément la phase de conception.
- la modifiabilité du plan pendant la phase de développement : Pouvoir simplement modifier le plan pendant la phase de conception est très important. Surtout quand le projet avance en parallèle avec plusieurs intervenants. Un script écrit d'une manière impérative rends cette tâche plus délicate et moins portable.

Structurer le plan de navigation est donc très utile dès qu'il s'agit des plans de navigation à niveau moyen de complexité.

Choix de RdP : Plusieurs outils de modélisation sont utilisés pour décrire un plan de navigation (machine à états, Grafcet ...). Le choix des réseaux de Petri est due au pouvoir de cet outil à modéliser le parallélisme et la synchronisation entre processus. Ceci est fortement sollicité dans les plans de navigations à complexité considérable.

Conventions de description d'un plan de navigation par réseaux de Petri : On essaye ,dans cette paragraphe, de souligner deux principales conventions choisies pour pouvoir modéliser un plan de navigation d'une structure GenoM avec un réseau de Petri :

- Actions sur transition : Sur tirage d'une transition, on peut lancer un ou plusieurs services (activité, attribut ou fonction).
- Condition de transition : On peut spécifier comme condition de tirage d'une transition, le fait qu'un service lancé sur une autre transition ait un certain statut.

Le schéma ci-dessous représente ces deux simples principes :

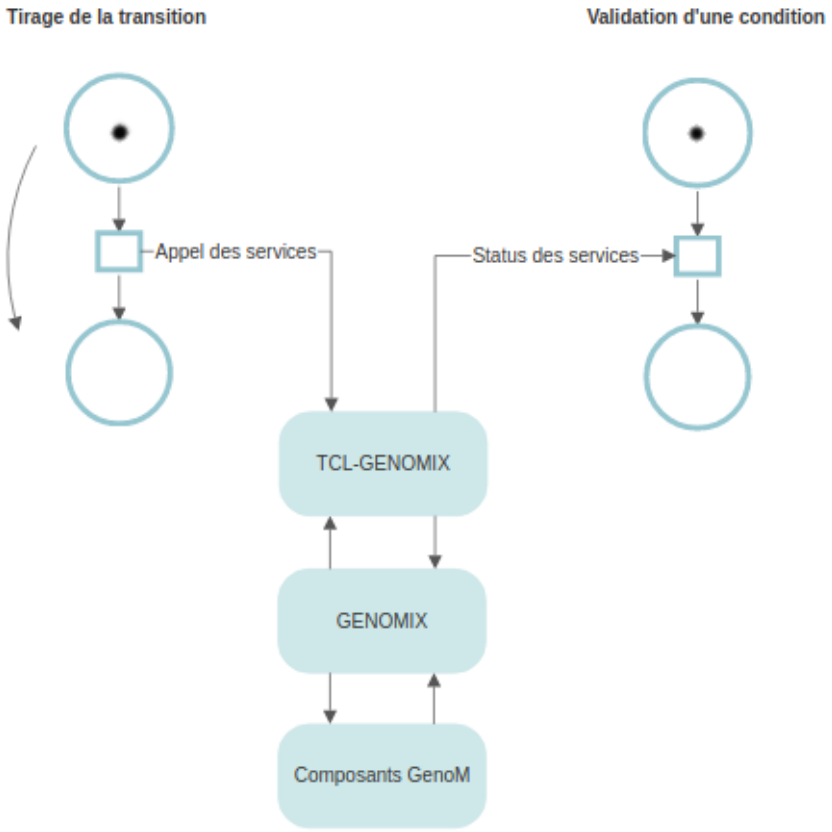

FIGURE 1 – Principes de supervision avec un RdP

Un exemple minimaliste de RdP de supervision , illustré dans la figure ci-dessous, décrit le plan de navigation qui consiste à lancer le service Move d'uN composant demo puis attendre que ce service soit sur "done" :

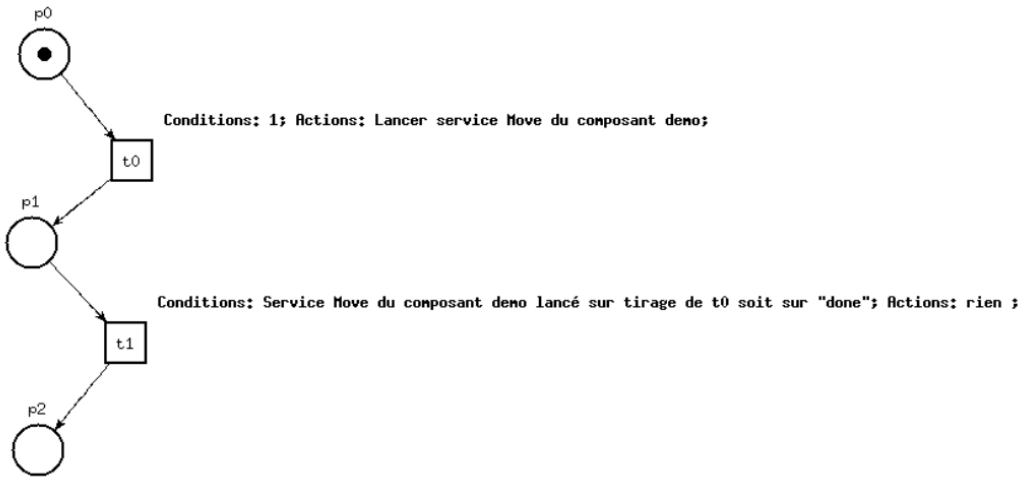

FIGURE 2 – Exemple simpliste de RdP de supervision

Les conditions et les transitions peuvent êtres plus compliqués que dans l'exemple précédent. On peut rédiger une formule logique sur différents services. On peut aussi lancer plusieurs

services sur tirage d'une transition. On aura comme conditions de la transition t1 par exemple : statut service Move lancé sur t0 == "done" OR statut service Goto lancé sur t0 == "done".

On vient de découvrir la notion de requête qui sera utilisée par la suite. La requête permet de suivre le statut d'un service et elle est labellisé selon la convention suivante : *nomducomposant_nomduservice_tn°* avec n° le numéro de la transition sur laquelle le service est lancé. Le numéro de la transition permet de faire la différence entre deux instances du même service d'un composant qui sont lancés à partir de différentes transitions. Un exemple de requête sera alors demo_Move_t0. La condition sur t1 s'écrit maintenant : statut requête demo_Move_t0 == "done" OR statut requête demo_Goto_t0 == "done". Par la suite on écrit la condition plus simplement comme suit : (demo_Move_t0 "done" OR demo_Goto_t0 "done")

Pour enrichir la capacité de modélisation des réseaux de Petri, on propose d'intégrer des formules logiques sur des variables dans les conditions de tirage des transitions. On pourra ainsi écrire des conditions de type : (demo_Move_t0 "done" OR variable == 1). Une suite logique de ce choix sera de pouvoir modifier ces variables dans les actions des transitions.

Un exemple simple de RdP avec l'ensemble des conventions est donné ci-dessous :

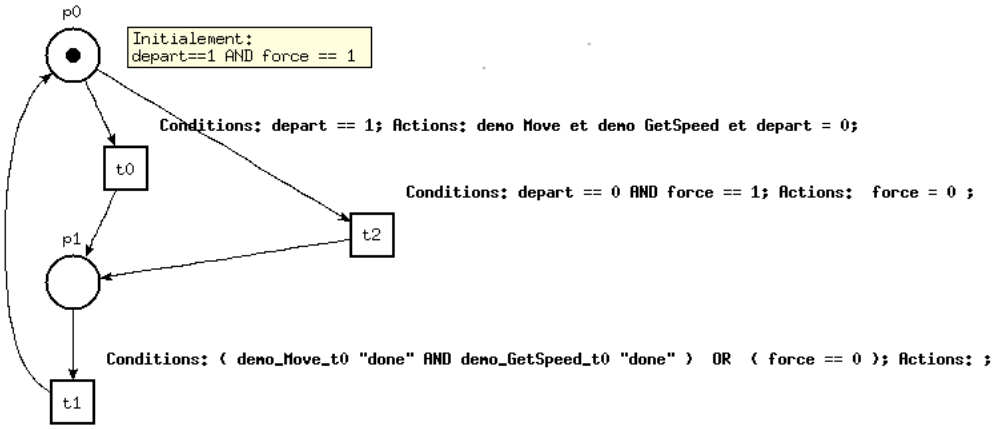

FIGURE 3 – Exemple de RdP de supervision

Dans l'exemple précédent , qui n'a pas de sens à part d'illustrer les conventions, la transition t0 est tirée (les services Move et GetSpeed du composant demo sont lancés et la variable depart prend la valeur 0).La transition t1 est ensuite tirée quand les deux services de la transition t0 arrivent À "done" . La transition t2 est ensuite validée (puisque depart == 0 et force == 1) et le tirage de la transition met la variable force à 0. La transition t1 est validée puisque force == 0 et donc tirée.

1.2 l'Outil JRdP pour configuration des clients TCL

L'outil JRdP permet de configurer graphiquement les réseaux de Petri selon les règles définies dans 4-1. JRdP et Le RdP configuré représentent un client TCL (puisque JRdP est écrit en TCL et utilise TCL-genomix pour l'envoi des requêtes). Les composants à orchestrer doivent être compilés et installés. JRdP permet de les charger et connecter les ports afin de mettre en place la structure à orchestrer à travers le RdP configuré.

1.2.1 Mode d'emploi

JRdP utilise l'outil nd de tina comme interface graphique permettant de tracer les réseaux de Petri et les enrichir par les conditions et les actions de chaque transition.

Un exemple de configuration de plan de navigation à travers l'interface nd est donné sur la Figure 5 qui suit.

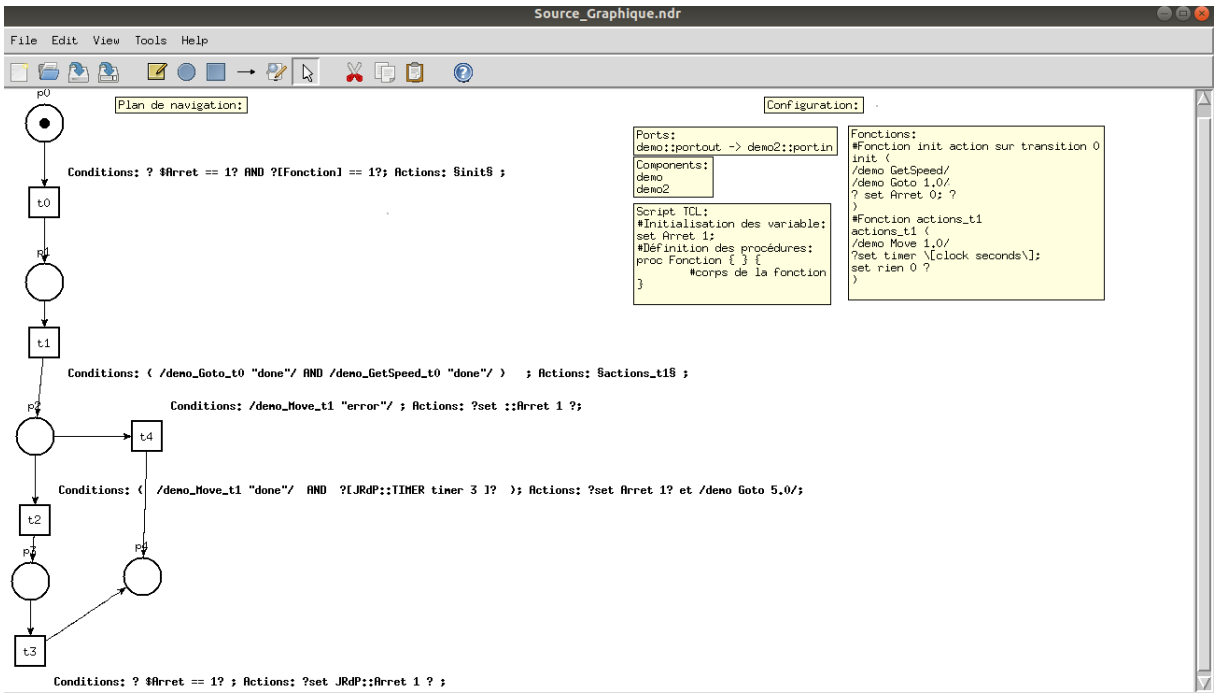

FIGURE 4 – Interface graphique avec nd

Étapes d'utilisations Les étapes d'utilisation de JRdP sont données comme suit :

- Compiler et installer la structure GenoM à orchestrer.
- Configurer le plan de navigation à travers nd. Cf 4-1 Explication des conventions de l'interface graphique.
  Il est utile de noter que pendant la phase de conception tous les outils de nd peuvent êtres utilisés. La chose qu'on recommande pour valider la structure du réseau de Petri (structural analysis) ou simuler l'évolution de la structure (stepper simulator).
- Sauvegarder le fichier .ndr de la configuration dans le dossier de JRdP sous le nom source.ndr.
- Lancer le fichier tcl Joueur_Rdp.tcl avec eltclsh ou tclsh en se plaçant dans le dossier du JRdP (Très important).
- Suivre les consignes du Joueur.
- Pendant la phase de conception, une fenêtre de l'outil nd stepper simulator s'ouvre pour permettre de suivre l'évolution du marquage en dynamique :

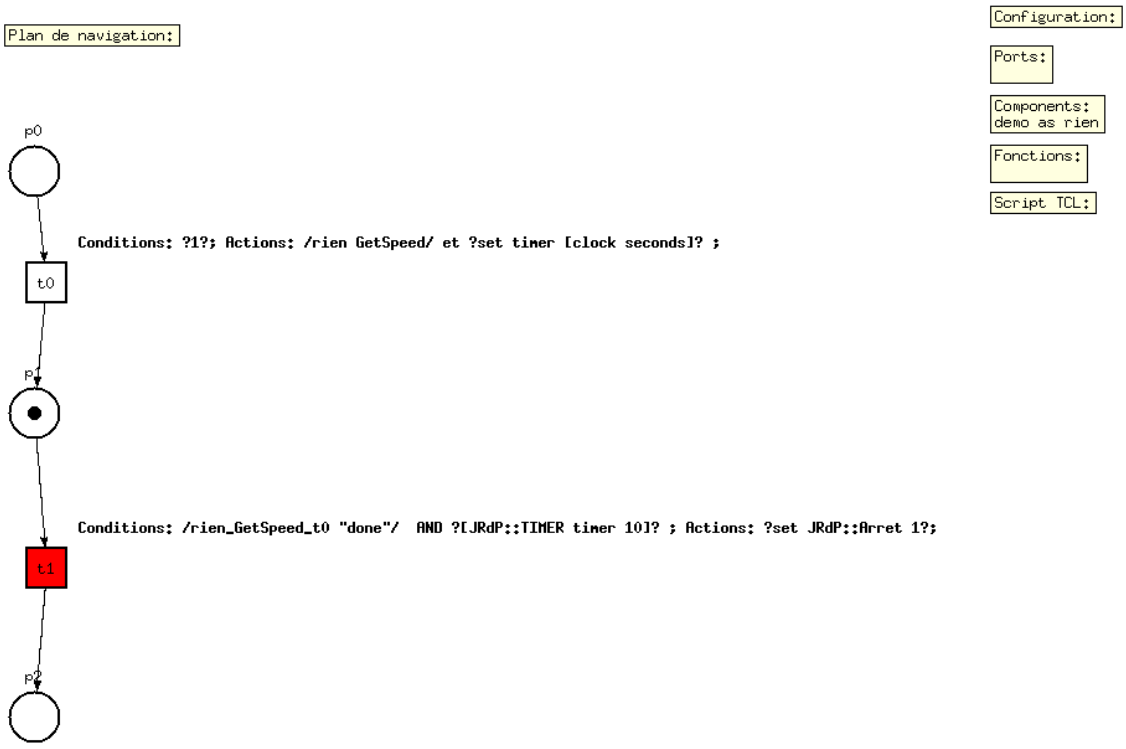

FIGURE 5 – Interface graphique avec nd

Remarque : Les transitions sensibilisées sont marqués en rouge. Appuyez sur ces transitions fait tirer la transition et engendrera ,par la suite, une erreur du stepper simulator quand JRdP demandera de la tirer (puisque elle n'est plus sensibilisée).

— Le résultat du jeu est enregistré dans le fichier Logs.txt qui se trouvera dans le dossier du joueur.

— Une fois le développement du plan de navigation est terminé. JRdP fournit une version du Joueur non interactive avec la même configuration que celle crée de manière interactive. La version s'appelle JRdP_Embadded et elle peut être lancé avec eltclsh de la même manière que précédemment.

Voix de développement : Parmi les évolutions qui peuvent être apportés à JRdP :

— Jeu off-line graphiquement : Il est aussi intéressant de pouvoir relire l'historique du jeu (une sorte de boite noire) pour une déboggage graphique ultérieur.

— Pilotage des variables crées par l'utilisateur dans l'interface nd par un niveau de supervision plus haut pour pouvoir commander les robots (à travers une tablette par exemple et donc une IHM bien adaptée à l'utilisateur normal qu'on considère très chargé pour voir des RdP).

Dépendances à installer JRdP a comme prérequis les dépendences suivant :

— L'outil nd de TINA (License Binaire Freeware) : version > 3.5.0 .
http ://projects.laas.fr/tina/home.php.

— ELTCLSH (License 2-clause-bsd) : https ://www.openrobots.org/wiki/eltclsh

— TCL-GENOMIX (License 2-clause-bsd) :
https ://git.openrobots.org/projects/tcl-genomix
ou avec robotpkg : robotpkg/supervision/tcl-genomix

— GENOMIX (License 2-clause-bsd) :
//git.openrobots.org/robots/genomix.git ou robotpkg/net/genomix

— VECTCL (OpenSource) :
https ://auriocus.github.io/VecTcl/design/60.html
Remarque : Le package VECTCL peut être facilement enlevé en codant les fonctions utilisés dans vexpr (dans Data.tcl) en pur tcl.

Les dépendances liées à GenoM3 ne sont pas inclus dans cette liste. Cf 3-1-2 pour la configuration de l'environnement.

Explication des conventions de l'interface graphique

Utilisation des notes Les bloc notes fournis par nd sont utilisées pour configurer les fonctionnalités citées ci-dessous. La première ligne de la note est utilisée pour identifier le type de fonctionnalité. Il est donc non modifiable.

— *Note commençant par Ports* : Ce type de note permet de connecter les ports. Le syntaxe est le suivant : composant1 : :portout -> composant2 : :portin . Ce syntaxe traduit la phrase suivante : composant1 fournit comme port de type out est connecté au port de type in du composant2.

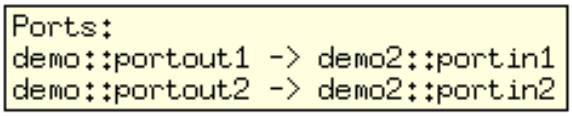

Figure 6 – Note des Ports

— *Note commençant par Components* : Elle permet de charger les composants écrits dans la note.

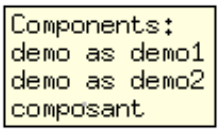

Figure 7 – Note des Components

Comme dans l'exemple de la figure, il est aussi possible d'avoir plusieurs instance du même composant selon différents noms. Le syntaxe est le suivant : composant as nom_instance. Si on lance un composant sans spécifier son nom d'instance (ou de le mettre au même nom du composant), il n'est plus possible d'instancier plusieurs instances du même composant avec différents noms.
Évolution : Il est aussi intéressant de pouvoir lancer plusieurs demon genomix sur différents hôtes [1] .

— *Note commençant par Script TCL* : C'est un script TCL qui s'exécute avant le démarrage de la boucle de JRdP. Il sert à initialiser des variables ou à définir des fonctions. On peut utiliser ces variables et ces fonctions dans les conditions et les actions des transitions. Cf Utilisation des labels.

1. https ://git.openrobots.org/projects/tcl-genomix/gollum/README#connecting-to-a-server-and-loading-clients_handle-load

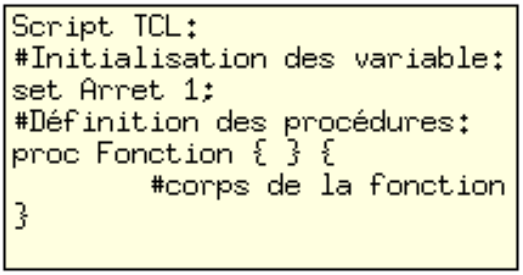

FIGURE 8 – Note des Script TCL

Note : Veillez à télécharger la dernière version de nd (version > 3.5.0) qui corrige un comportement non désirable par rapport à l'évaluation du script contenu dans les notes. Dans le fichier .ndr les lignes des notes commencent par "a" au lieu de "n".

— *Note commençant par Fonctions* : Cette note permet d'écrire des macros (encapsulation) qui permettent de rédiger plusieurs actions sur le tirage d'une transition sans encombrer le label du transition. Cf Utilisation des labels.

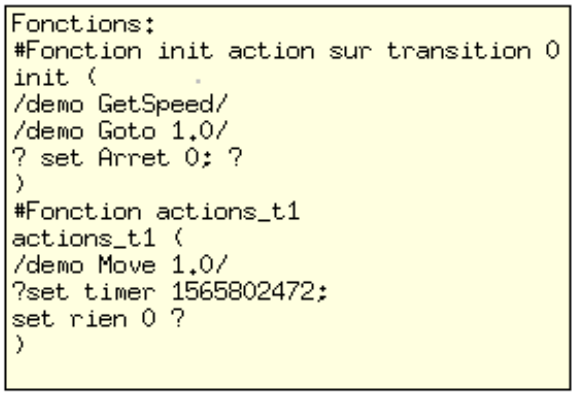

FIGURE 9 – Note des Fonctions

Le syntaxe est le suivant :nom_macro (contenu_macro). Le contenu du macro (actions sur transitions) sera expliqué en détail dans le paragraphe Utilisation des labels. Il est important de laisser un espace entre le nom du macro et la première parenthèse, sinon une erreur sera levée.
L'appel d'un macro dans les actions d'une transition suit la convention suivante : §nom_macro§. Un exemple d'utilisation du macro init défini sur la Figure 8 est donné comme suit :

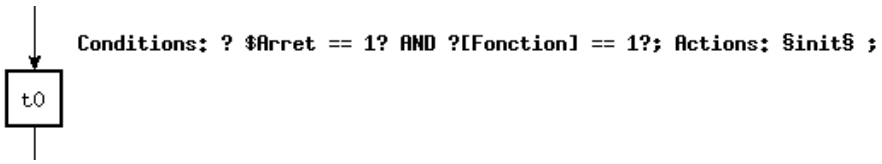

FIGURE 10 – Utilisation du macro init

Utilisation des labels Les labels des transitions permettent de rédiger les conditions et les actions associées aux transitions. Le syntaxe utilisé est le suivant :
Conditions : lesconditions ; Actions : lesactions ;

1. Deux types d'actions sur transitions sont prises en charge par JRdP :
   (a) *Lancement de services :* On peut lancer un service avec le syntaxe suivant :/composant service paramètres/ . Les slashs permettent de délimiter l'appel du service. Par exemple : (/composant1 service1 paramètres/ et /composant2 service2 paramètres/). Si le service ne prend pas de paramètres, le syntaxe est réduit à /composant service/.

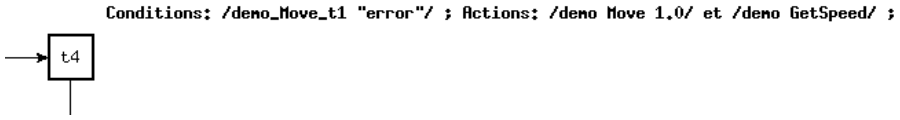

FIGURE 11 – Actions de type lancement de services

   Note : Un chemin de développement de JRdP sera de pouvoir tenir en compte des options de lancement des services fournies par TCL-genomix (-timeout principalement comme exemple).

   (b) *Exécution de script TCL :* On peut exécuter un script TCL sur tirage de la transition. Pour délimiter le script, on utilise la convention suivante : ?Script_TCL ?.

```
Conditions: /demo_Move_t1 "error"/ ; Actions: ?set ::Arret 1 ? ;
```

FIGURE 12 – Actions de type exécution de script

   L'ordre d'exécution des actions de la même transition est aléatoire mais insignifiant.

2. Deux types de conditions sont prises en charge par JRdP :
   (a) *Conditions sur statut de service :* Comme dans 4-1-conventions, on peut rédiger une condition sur un statut spécifique d'un service lancé en utilisant la requête de ce service. Pour délimiter ce type de conditions, on utilise les slashs. Le syntaxe est le suivant : /requête "statut_désiré"/. Dans le cas où le statut désiré est "error", on peut spécifier le nom de l'exception (par exemple demo : :INVALID_PARAM). Le syntaxe devient :/requête "error" "nom_exception"/. Si on ne spécifie pas le nom de l'exception, seulement le statut "error" est considéré.

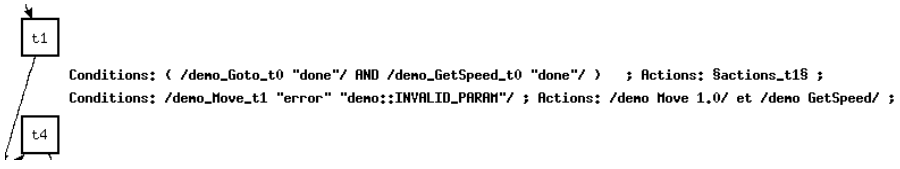

FIGURE 13 – Conditions sur statut de service

   On peut rédiger une condition avec plusieurs flags (différents statuts de services) en formulant la condition logiquement. Par exemple : (flag1 AND flag2) OR flag3 avec flag1 est /requête "statut_désiré"/ par exemple.
   Remarque : Pour valider l'interprétation du fichier .ndr, on peut utiliser le fichier de configuration généré. On peut comparer par exemple le nombre de services dans la formule logique de condition de chaque transition au nombre de fonction sensibilise_transition_service (par exemple).

(b) *Conditions sur un script TCL :* On peut rédiger une condition (un flag plus précisément) avec un script TCL. Cela permet d'avoir des conditions sur une valeur de variable ou sur un retour d'une fonction définie dans la note Script TCL. Le syntaxe est le suivant : ?Script_TCL_formant_une condition ?. Par exemple : ? var == 1 ? OR ? [fonction] == 5 ? . JRdP fournit plusieurs fonctions utiles à rédiger des conditions :

i. La fonction TIMER : La fonction TIMER permet de valider un flag de condition d'une transition après une durée comptée du moment de sensibilisation de la transition. La fonction TIMER prend comme paramètre une référence de temps, la durée (en secondes) à compter et la précision. Le syntaxe d'appel de la fonction est le suivant : JRdP : :TIMER réf_de_temps durée précision. La fonction TIMER retourne 1 si $-precision <= duree + ref\_de\_temps <= precision$, elle retourne 0 sinon. La référence de temps doit être déclaré dans les actions des transitions qui mènent vers la place sensibilisant la transition. Le syntaxe de déclaration de la référence de temps est comme suit : ?set réf_de_temps [clock seconds] ? .

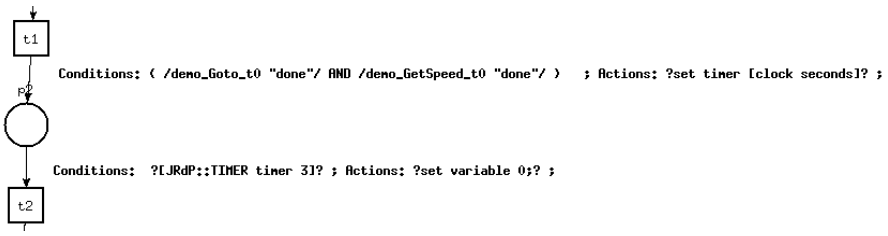

Figure 14 – Actions de type exécution de script

Si on ne spécifie pas la précision, elle est mise automatiquement à 0. Les crochets permettent de récupérer le retour de la fonction TIMER (Puisque c'est un script TCL).
Note : Une évolution possible de cette fonction est de pouvoir utiliser des références de temps prédéfinies par JRdP et qu'elles soient associées à chaque place. On aura pas besoin de déclarer la référence de temps pour chaque transition précédant la place qui sensibilise la transition. On peut utiliser la notion de front montant (avec une variable intermédiaire) sur le marquage des places pour définir les références de temps. L'appel de TIMER deviendra comme suit : ?[JRdP : :TIMER p2 3] ?

ii. La fonction marquage_place : Elle permet de récupérer le marquage d'une place. Le syntaxe est les suivant : marquage_place num_place . On peut écrire des conditions de type : ? [JRdP : :marquage_place 4] == 1 ? .

iii. Une fonction de lecture des ports : C'est une évolution intéressante de JRdP. Elle permettra de simplifier la structuration de la couche fonctionnelle en remontant des traitements au niveau de la supervision.

iv. La fonction result_req : Elle permet de retourner le résultat d'une requête (qu'elle prend comme paramètre). Par exemple ? [JRdP : :result_req demo_GetSpeed_t0] == : :demo : :FAST ?.

Pour arrêter la boucle du JRdP, JRdP fournit la variable Arret. On peut mettre cette variable à 1 sur tirage d'une transition pour arrêter la boucle complètement. Le syntaxe est le suivant : ? set JRdP : :Arret 1 ; ? . **Note :** le petit "d" miniscule de JRdP

est trainé à cause d'une mauvaise habitude durant le codage. Mais il doit être en miniscule (plus simple que changer le nom du namespace :)).

À savoir pour bien utiliser l'outil Parmi les choix d'implémentation qui doivent êtres assimilés par l'utilisateur de JRdP, on trouve :

1. *La réutilisation de la même instance de requête pour valider un flag :*
   Problème : un flag de type condition sur statut de service est validé et le statut du service est sauvegardé pour les autres flags qui l'utilisent. Si on re-boucle sur une place qui sensibilise la même transition, le flag se re-valide à cause de la sauvegarde de statut effectuée précédemment. Le comportement voulu est la prise en compte de la sauvegarde que dans le cas où on ne l'a pas utilisée précédemment. Le comportement "naturel" de l'exemple de la figure suivante est donc le tirage dans l'ordre des transitions suivantes : t0 , t1 , t2.

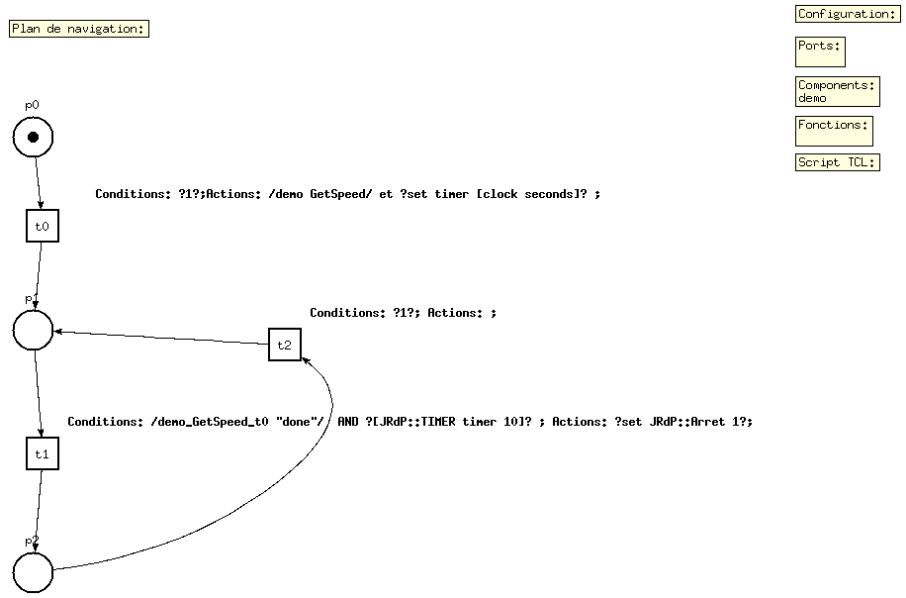

Figure 15 – Problème de bouclage

Solution : Sauvegarde d'un identifiant de l'instance de requête qui a déjà validé le flag de la condition.Sur chaque appel d'un service, un identifiant d'appel (conteur) est enregistré localement à la requête. Si on utilise une instance de cette requête pour valider (mettre à 1) un flag (de type condition sur statut de service) d'une condition, cette instance est sauvegardé (au moment de tirage de la transition) localement au flag pour qu'elle ne sera pas réutilisé pour le re-valider. Cela permet de devoir attendre une nouvelle instance de la même requête pour reformuler la condition.
Remarque : Dans le cas où on attend le statut "sent" dans le flag (Par exemple /demo_Move_t0 "sent"/), On a généralement intérêt à ré-valider le flag sur la même instance de requête. Donc ne pas avoir besoin de relancer le service de la requête puisque elle toujours sur sent. Un choix d'implémentation est donc fait pour permettre de ré-valider un flag sur la même instance de requête si le statut désiré est bien "sent".
Note : Les instances d'une requête sont identifiés par un conteur incrémenté sur chaque appel de cette requête. Ceci est dit, il est important d'anticiper le problème de débordement du conteur si le robot qui utilise JRdP est en "free-runing" . Il est très improbable

qu'une erreur soit engendré à cause de cette remarque (car si le conteur déborde , les premiers instances sont fort probablement déjà disparues des sauvegardes des flags. Surtout pour des ordinateurs 64 bit). C'est peut être le même problème de Boeing Dreamliner qui doit être éteinte 2 fois par an.
À retenir : On ne ré-valide pas un flag (de type condition sur statut de service) déjà validé avec la même instance de requête. Sauf dans le cas où "sent" est le statut désiré. Normalement une prise en main de JRdP pourra simplement mettre en évidence ce comportement naturel mais qui m'était compliqué à coder .

2. *Le tirage des transitions valides :* Sur un tour de boucle de JRdP, toutes les transitions valides sont tirés (dans l'ordre d'énumération des transitions). Ce choix permet de révéler au moment de conception du plan de navigation les non déterminismes due à un "divergence en ou" avec des conditions mutuellement non exclusives.

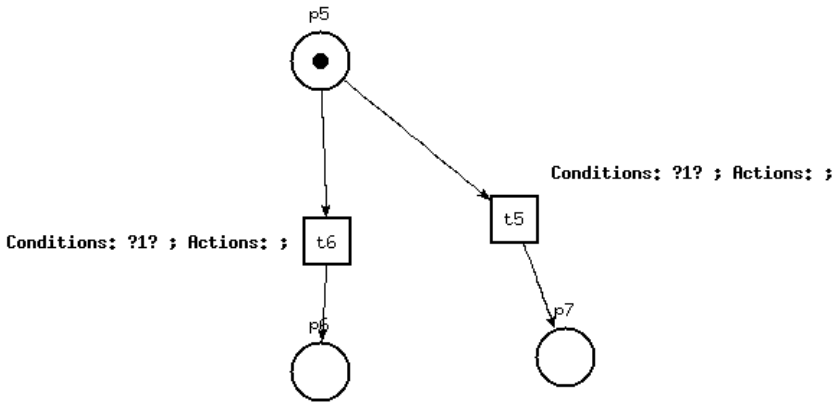

FIGURE 16 – Non determinsme

Le tirage de la transition t5 et t6 (pendant l'exécution) cause l'apparition d'un marquage négatif de la place p5 (puisque on a un seul jeton sur cette place). JRdP considère cette situation un erreur de modélisation de plan de navigation. Il permet de notifier l'utilisateur pendant le runtime en retournant un erreur avant de sortir (exit 1 ;).
C'est intéressant de noter que dans les conventions des Réseaux de Petri, le tirage des transitions dans ce cas de non déterminisme se fait aléatoirement puis on recalcule la nouvelle situation.

1.2.2 Exemple d'utilisation

Dans cet exemple, on utilisera le composant GenoM disponible sur :
http ://robotpkg.openrobots.org/robotpkg/doc/demo-genom3
Nous veillons à garder le plan de navigation le plus simple possible. Le composant demo fournit les services suivants :

1. Fonctions :
   Stop() : Interrompre MoveDistance, GotoPosition.
2. Activités :
   GotoPosition(posRef) : Avancer à la position posRef.

MoveDistance(distRef) : Avancer de la position distRef.
Monitor(monitor) : Superviser la position monitor.

3. Attributs :
   GetSpeed () : Récupérer la vitesse.
   SetSpeed () : Mise à jour de la vitesse.
4. Exceptions :
   INVALID_SPEED : La valeur de la vitesse est invalide.
   TOO_FAR_AWAY : La valeur de la position est invalide.

Le plan de navigation :

Plan nominal : Mettre la vitesse à "FAST", Donnez une consigne de position et superviser une autre. Si la position supervisée est atteint, on modifie la vitesse à "SLOW" pendant 0.1 m puis on la remet à "FAST".

Gestion d'erreur : Les exceptions dans cette exemple dépendent de la vitesse et de la position données. Pour illustrer cette partie du plan on peut donner des entrées non valides dans le plan nominal.
Il est intéressant de noter les types de traitement des erreurs :

- Traitement d'erreur dans la couche fonctionnelle à l'aide de GenoM. Relancer un service avec différents paramètres si il a levé une certaine exception est un traitement qui peut être transparent à la supervision avec JRdP.
- Traitement d'erreur dans la couche décisionnelle avec JRdP. Le traitement est donc spécifié à l'aide des outils de modélisations fournis par JRdP (structure RdP, structures de données ...)
- Traitement d'erreur avec JRdP et avec GenoM. Ce type de traitement est le plus intéressant puisque il permet de représenter le plan de gestion de l'erreur avec JRdP sans entrer dans le détail du traitement qui sera codé avec GenoM (sous forme d'activités ou de fonctions) et appelé avec JRdP.

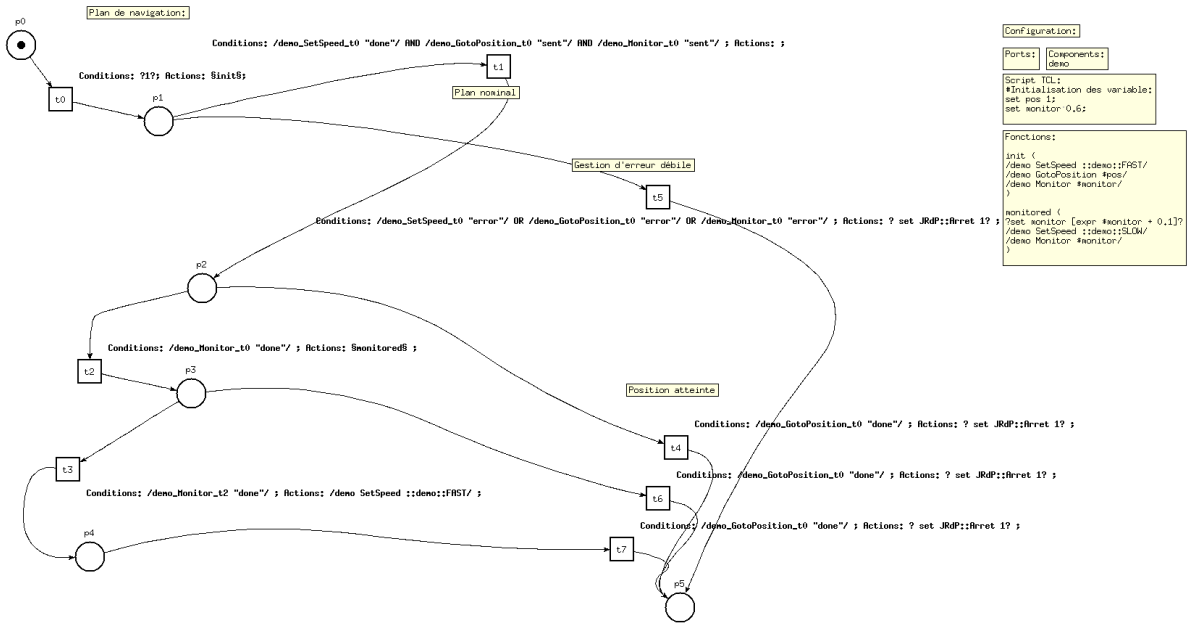

FIGURE 17 – Plan nominal dans l'exemple

Remarque : Dans le plan nominal, l'ensemble des cas possibles sont considérés en reliant les places p2,p3 et p4 à la place p4.
Comme dans la supervision avec les Grafcets, deux alternatives de représentation sont possibles (supervision interprétée ou structurelle). L'utilisateur de JRdP peut choisir le paradigme le plus convenable.
Le fichier source.ndr de l'exemple est disponible dans la dossier doc de JRdP.

1.3 Fonctionnement Interne

1.3.1 Fonctionnement du JRdP

Structures de données : Afin de comprendre l'organigramme de JRdP, il faut assimiler le choix des structures de données fait pendant la conception :

1. Requests_status : c'est un array (au terme de TCL , {clé1 valeur1 clé2 valeur2 ...}). Il permet de sauvegarder l'état de chaque requête lancée sur tirage des transitions. la clé est le nom de la requête (par exemple demo_Move_t0) et la valeur est une liste (premier indice contient le statut , par exemple "done", et le deuxième indice contient l'exception dans le cas de "error" comme statut).
2. Flags : Une liste de taille le nombre des transitions et contenant des arrays. L'array à l'indice numéro de la transition contient les flags associés à cette transition et il prend la forme suivante : requête1 list1 requête2 list2 .... Pour chaque requête on associe une liste contenant le flag (1 ou 0) de la requête, le statut désiré pour la requête, l'exception désiré au cas de "error" et l'identifiant de la dernière instance de requête qui a validé le flag. les listes sont donc de la forme {"flag" "status" "exception" "0" }. Flags se configure à l'aide des fonctions sensibilise_transition_service dans Configuration_RdP.
3. Flags_cond : Une liste de taille le nombre de transitions. Elle permet d'associer la formule logique de la condition à la transition. La formule logique de la condition doit avoir comme opérandes les flags de la transition contenues dans Flags. On prend l'exemple de condition suivant : /demo_Move_t0 "done"/ OR /demo_GetSpeed_t0 "done"/. La formule logique qui doit être sauvegardé dans Flags_cond est de la forme : flag-de-demo_Move_t0 || flag-de-demo_GetSpeed_t0. Flags_cond est généré à partir de la configuration graphique contenue dans le .ndr et à l'aide de Sources_Tcl/Generate_configuration.tcl.
4. Actions_transitions : Une liste de taille le nombre des transitions. Elle contient les actions associés à chaque transition. Elle est configurée à l'aide des fonctions associer_..._transition dans Configuration_RdP.
5. Transitions_sensibilisees : Liste de taille le nombre de transitions et permettant de marquer les transitions sensibilisées (marquage des places pré-transition différent de zéro).
6. Transitions_valides : Liste de taille le nombre de transitions et permettant de marquer les transitions valides (la condition de la transition est valide et donc elle tirable).
7. Conditions : Liste contenant un drapeau représentant la condition de la transition. Si la condition de la transition est valide , le drapeau se met à 1 et vis-versa.
8. M : Une liste de taille le nombre des places et contenant le marquage des places.

Organigramme JRdP : On présente dans cette paragraphe l'organigramme simplifié de JRdP. Des parties de l'organigramme ne seront pas détaillées dans cette paragraphe :

FIGURE 18 – Organigramme de JRdP simplifié

La génération de la configuration et de la définition matricielle seront expliqué dans le paragraphe suivant.L'organigramme de la boucle de JRdP est simplifié comme suit :

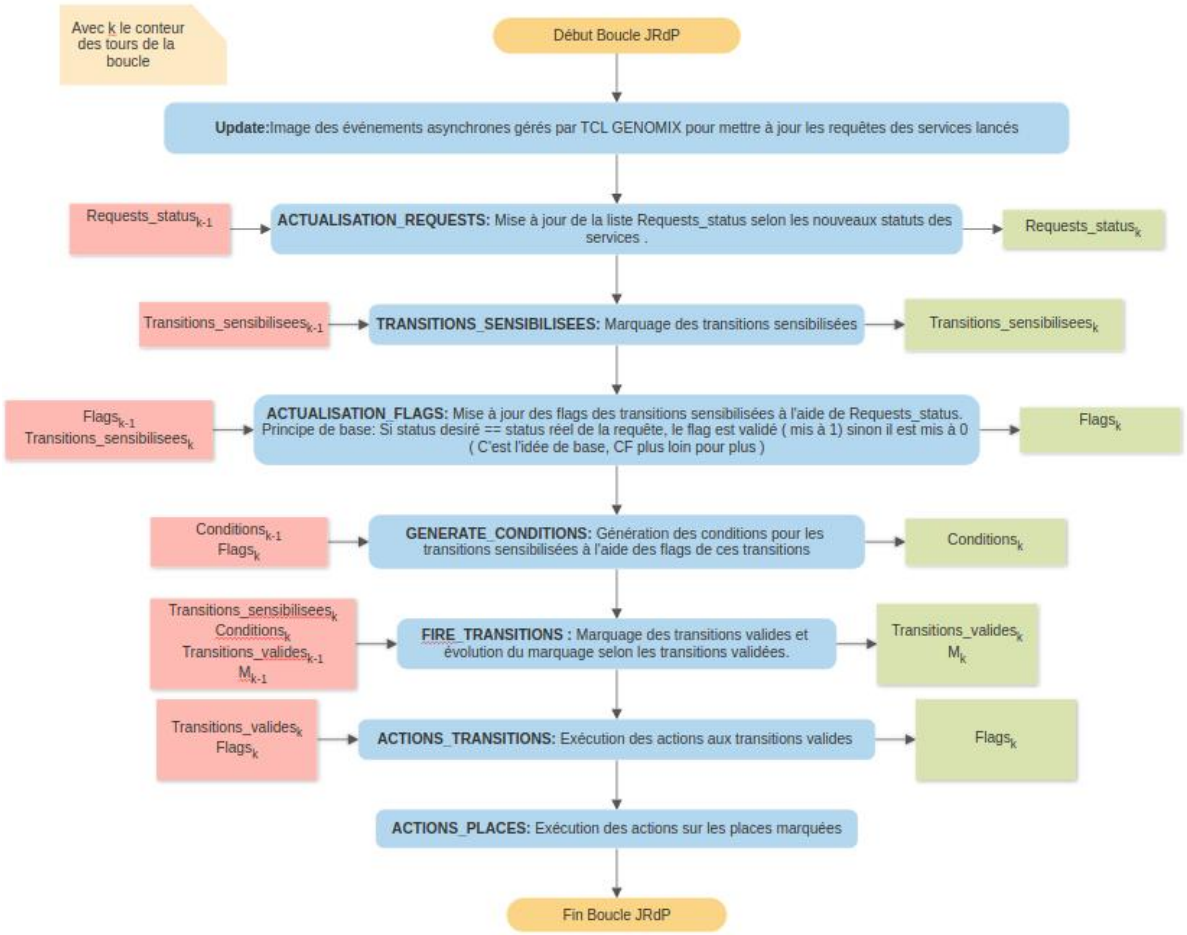

FIGURE 19 – Organigramme de la boucle de JRdP simplifié

1.3.2 Emploi de nd comme interface graphique

En récupérant le fichier source.ndr, La structure du réseau de Petri est configurée en générant la définition matricielle contenue dans Matrices.tcl . Les labels des transitions et les notes sont interprétées pour générer le fichier de configuration Configuration_Rdp.tcl.

Génération de la définition matricielle : Le programme de cette fonctionnalité est moins complexe que celui de la génération de la configuration. Tout curieux est donc redirigé directement vers le code source dans Generate_matrices.tcl.

Génération de la configuration : L'organigramme suivant explique le principe de base de la génération de la configuration à partir du fichier .ndr :

FIGURE 20 – Organigramme de génération de la configuration

Dans l'organigramme qui suit, on présente le fonctionnement du traitement des transitions à partir des labels comme exemple des traitements mentionnés dans l'organigramme qui précède. Le code source commenté reste à disposition des curieux pour toute correction ou débogage :

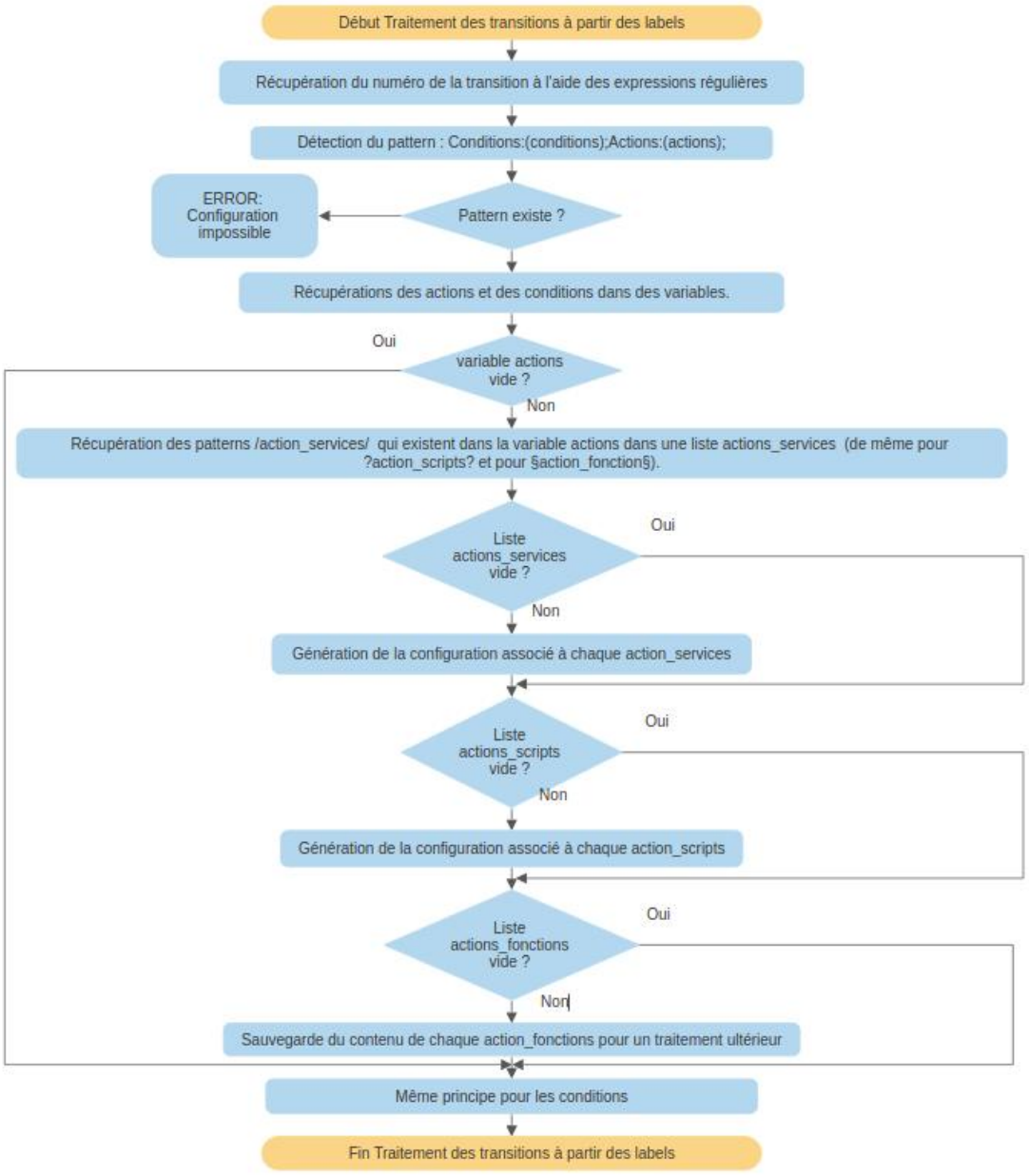

FIGURE 21 – Organigramme de traitement des transitions à partir des labels

C'est toujours délicat d'expliquer le principe d'un programme à longueur relativement moyenne sans entrer dans les détails mais on a essayé d'encapsuler des parties qui peuvent être assimilés avec plus d'aisance à partir du code source directement.

Pilotage du stepper simulator de nd à partir de JRdP : L'objectif est de pouvoir visualiser l'évolution du marquage du RdP de supervision en dynamique pendant la phase de conception. Je tiens à remercier M.Bernard BERTHOMIEU qui a permet à cette fonctionnalité d'exister. Le principe est illustré dans la figure suivante :

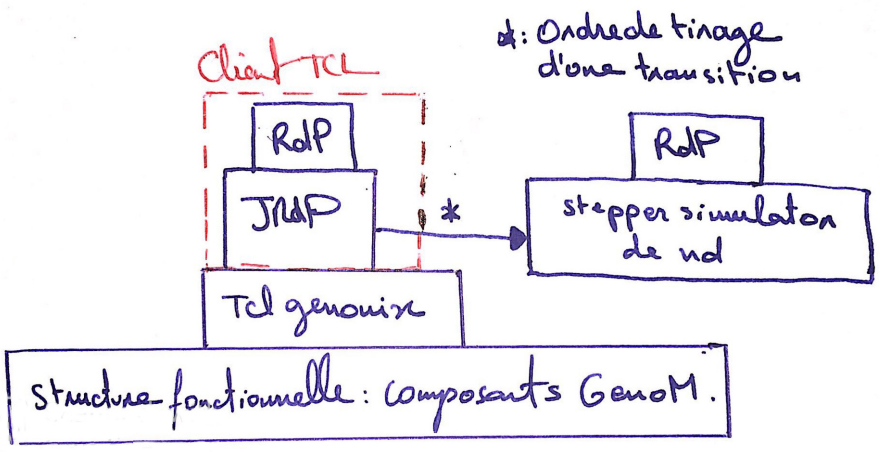

FIGURE 22 – Pilotage nd stepper simulator par JRdP

JRdP exécute le plan de navigation décrit par le RdP enrichi (et l'ensemble RdP JRdP constitue un client tcl qui envoie ses requêtes et reçoit le retours à travers TCL GENOMIX). Donc le marquage du RdP associé à JRdP évolue contrairement à celui du stepper nd. Pour synchroniser les deux RdP, JRdP envoi les ordres de tirage des transitions si ils sont valides à nd à travers un named pipe (pile FIFO). La création du pile fifo de communication est géré par JRdP mais pour que nd tient en compte de cette communication, le fichier .ndr (une version temp) est augmenté par un script TCL (puisque nd est écrit en TCL/TK). L'outil stepper simulator peut récupérer les ordres de tirages de la pile fifo et les exécuter.
Remarque : Dans le cas d'un non déterminisme (CF 1-2 À savoir pour bien utiliser l'outil), nd reçoit les transitions à tirer simultanément dans l'ordre d'énumaration des transitions et tire la première. Il est donc normal que nd montre le message "transition not enabled" et JRdP montre une erreur de non déterminisme.