Contribution au développement d'une plateforme de reconnaissance d'activités

Omar Abdalla

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

centre de recherche SOPHIA ANTIPOLIS - MÉDITERRANÉE

RAPPORT FINAL DU STAGE

Contribution au développement d'une platforme de reconnaissance d'activités

Tuteur : Mme Hélène RENARD-FERRY, hrenard@polytech.unice.fr

Encadrants : Mme Annie RESSOUCHE, Annie.Ressouche@sophia.inria.fr

M. Daniel GAFFE, Daniel.Gaffe@unice.fr

Par : Omar ABDALLA, o-abdalla@hotmail.com
M2 IFI IAM délocalisé à l'UFE

Le 01/09/2014

Remerciement

Ce stage s'est déroulé, durant cinq mois, au sein de l'équipe STARS (Spatio-Temporal Activity Recognition Systems) à l'INRIA Sophia Antipolis – Méditerranée. Dans un premier temps, j'insiste à remercier mes encadrants Mme Annie RESSOUCHE et Monsieur Daniel GAFFE pour leurs conseils qui m'ont aidé à effectuer ce travail. J'insiste aussi à les remercier pour leur disponibilité tout au long du stage pour répondre à toutes mes questions. Cela m'a beaucoup aidé à avancer quand j'ai rencontré des problèmes.

Dans un deuxième temps, je souhaite remercier mon tuteur Mme Hélène RENARD-FERRY pour son soutien et l'attention attribuée à mon travail et aussi pour ses efforts faits pour m'envoyer des retours sur les rapports.

Je tiens aussi à remercier tous les membres du jury, présents lors de ma soutenance, pour leur attention et tout intérêt qu'ils ont porté sur ma présentation et pour la discussion sur mon travail.

Dans un dernier temps, je souhaite remercier l'Office Méditerranéen de la Jeunesse et l'INRIA pour leur financement durant ma période de mobilité. En particulier, je tiens à exprimer ma gratitude à Mme Jane DESPLANQUES, l'assistante de l'équipe STARS, qui m'a beaucoup aidé lors de toutes démarches administratives pour faciliter les choses.

Résumé

De nos jours, la recherche dans le domaine de reconnaissance d'activité n'a pas cessé d'évoluer. De nombreuses approches ont été étudiées précédemment afin d'observer et d'identifier les activités effectués par des êtres humains, des animaux, ou des véhicules. Un challenge important est de générer automatiquement des moteurs de reconnaissance d'activités à partir de patterns génériques spécifiés par un utilisateur.

Dans ce travail, nous montrons tout d'abord l'intérêt de l'approche synchrone pour adresser ce problème, puis nous étudions des langages synchrones, notamment ESTREL et Light Esterel (LE). Ensuite, le projet CLEM est expliqué ainsi que ses fonctionnalités et son rôle dans le travail du stage. Par la suite, le projet SAM, le générateur de moteurs de reconnaissance de scénario est introduit ; son dévéloppement est illustrer l'objectif du stage. Ensuite, une transcription automatique des scénarios exprimés avec un langage dédié, SAM-L, en langage synchrone est proposée en montrant bien les démarches suivies pour arriver à cette solution. Finalement, des vérifications sont faites pour prouver formellement que certains comportements du scénario sont ceux attendus.

Abstract

Nowadays, the research in the field of activity recognition is consistently evolving. Several approaches have already been studied to observe and detect the activities taking place by human beings, or animals, or vehicles. An important challenge is to automatically generate activity recognition engines from generic patterns specified by the user.

In this work, we will show first of all, the reason why the synchronous approach is most suitable for this problem. Then, we will discuss the synchronous programming languages, such as ESTEREL, and Light Esterel (LE). Afterwards, the CLEM project is explained, as well as its functionalities, and its role in the work of this internship. Following CLEM, the project SAM, the activity recognition engine is introduced; its development is the main goal of this internship. Then, an automatic translation of scenarios, written in SAM-L, to synchronous language is proposed, while showing all the steps taken to get this solution. Finally, verifications are made to formally prove that certain behaviors of the scenarios are those that are to be expected.

Table des matières

Table des figures

I. Introduction

1. Contexte de l'équipe STARS :

Le stage se déroule au sein de l'équipe STARS (Spatio-Temporal Activity Recognition Systems) à l'INRIA Sophia Antipolis – Méditerranée. L'équipe STARS travaille sur la conception de systèmes cognitifs pour la reconnaissance d'activités **[1]**. Plus précisément, STARS s'intéresse à l'interprétation sémantique en temps réel des scènes dynamiques observées par des caméras vidéo et d'autres capteurs. Ainsi, STARS étudie des activités spatio-temporelles sur le long terme effectuées par des êtres humains, des animaux ou des véhicules. Le problème majeur de l'interprétation sémantique de scènes dynamiques, est l'écart important entre l'interprétation subjective des données et les mesures objectives fournies par les capteurs.

Pour résoudre ce problème, STARS propose de nouvelles techniques dans les domaines de la vision et des systèmes cognitifs pour la détection d'objets physiques, la reconnaissance et l'apprentissage d'activités, la conception et l'évaluation des systèmes. Le travail scientifique des chercheurs de STARS se retrouve dans le domaine de la vidéosurveillance sur les aspects sécurité ou encore dans l'observation médicale des personnes âgées à domicile ou des patients atteints de la maladie d'Alzheimer.

L'équipe STARS travaille sur deux axes de recherche :

1. L'interprétation de scènes pour la reconnaissance d'activités : l'interprétation de scènes a pour objectif de résoudre le problème complet d'interprétation allant de l'analyse bas-niveau du signal jusqu'à la description sémantique du contenu de la scène observée par les capteurs (cameras vidéos et éventuellement capteurs d'environnement). STARS travaille plus particulièrement en perception, interprétation et apprentissage.

2. L'architecture logicielle pour la reconnaissance d'activités : STARS développe une plateforme logicielle appelée SUP. Cette plateforme est modulable pour gagner en généricité. L'objectif est d'assurer la réutilisabilité, l'extensibilité, la sûreté de fonctionnement et la maintenance logicielle. STARS développe également des techniques de compilation qui permettent de générer automatiquement des moteurs de reconnaissances synchrones.

2. Contexte du travail de stage :

Les systèmes de reconnaissance d'activités détectent en temps réel certains évènements qui ont lieu dans leur environnement puis les interprètent pour renvoyer un rapport sur la situation actuelle. Autrement dit, ils réagissent aux évènements d'entrées en produisant des évènements de sortie, on peut ainsi les considérer comme des « systèmes réactifs » **[3]**.

Des modèles de scénarios sont utilisés pour indiquer clairement le comportement du système et sa réaction envers les évènements détectés par les différents capteurs. L'équipe STARS travaille sur la conception d'un module de reconnaissance appelé SAM **[8]**. Il va servir à générer des moteurs de reconnaissances et permet de vérifier des modèles de scénarios. Ces modèles de scénario sont des patterns décrivant comment des évènements primitifs provenant du contexte s'enchaînent.

II. État de l'art

1. Systèmes réactifs et temps réels

Le terme « système réactif » est utilisé couramment pour désigner les systèmes interagissant en permanence avec leurs environnements et pour les distinguer des systèmes transformationnels. Un système transformationnel effectue des traitements des données initiales et engendrent des résultats qui sont indépendants de l'environnement comme par exemple un compilateur **[2]**.

Dans la programmation synchrone, nous pouvons distinguer encore plus précisément les systèmes réactifs des systèmes interactifs. Ces deux systèmes communiquent en permanence avec leur environnement. Cela dit, un système interactif tient compte des stimuli avec son propre rythme. Il est capable de se synchroniser avec son environnement et le faire attendre, comme pour le cas des processus concurrents dans les systèmes d'exploitation ou dans les systèmes de gestion de base de données. Alors que le rythme de l'évolution d'un système réactif est imposé par son environnement. Ce type de système engendre des réactions adaptées aux stimuli externes, et ces réactions modifient l'environnement selon le rythme de ces stimuli **[3]**.

La figure 1 **[4]** illustre un système réactif où les réactions « Si » engendrent à l'aide des fonctions « Fi » des évènements « Ei ». Ces fonctions « Fi » sont soumises chacune d'entre elles à des contraintes temporelles imposées par l'environnement.

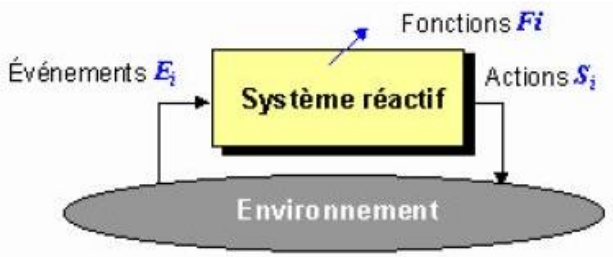

Figure 1: Système réactif

Au contraire des systèmes interactifs, les systèmes réactifs sont toujours conçus pour être déterministe, c'est-à-dire que pour une même séquence d'entrée, ils produisent la même séquence de sortie. De plus, un système peut être à la fois réactif et temps-réel, ces deux caractéristiques vont souvent de pair, mais pas nécessairement **[3]**. De tels systèmes se trouvent fréquemment en commande et contrôle. Nous les retrouvons maintenant aussi dans les domaines de la robotique, des surveillances médicales, les systèmes embarqués, et les systèmes de navigation d'avion. Les caractéristiques d'un système temps réel sont les suivantes :

- **Contraintes temporelles :** le système doit respecter des contraintes temporelles prédéfinies pour pouvoir réagir dans une durée de temps suffisamment courte et dans tous les cas inférieur à une borne imposée. Il existe plusieurs types, un système strict avec une durée précise pour le temps de réponse à ne pas dépasser pour éviter la défaillance du système. Il existe aussi un système souple, moins exigent qui se permet de ne pas respecter le temps de réponse, cela causera une dégradation dans sa performance mais sans conséquences dramatiques.
- **Parallélisme** : le système comporte différents sous-systèmes, fonctionnant en parallèle, et qui sont en interaction. Le comportement global peut facilement devenir incohérent si on ne met pas en place un mécanisme adapté de synchronisation.

- **Prévisibilité des comportements** : le système doit nous permettre de prévoir son comportement lors des situations critiques. Il doit donc respecter toutes les contraintes imposées précédemment, dans n'importe quelle condition de fonctionnement (robustesse). Pour les cas des systèmes réactifs et temps réel, on cherche à avoir aussi un comportement déterministe.
- **Sûreté de fonctionnement**: le système doit toujours être disponible et doit pouvoir réagir à tout moment. Il doit assurer en permanence certaines fonctions critiques quel que soit l'environnement.

2. Programmation des systèmes réactifs

2.1 Approches Classiques

Dans notre système, l'ordinateur matérialise souvent le contrôleur et cela explique effectivement l'importance de la conception des systèmes réactifs. Plusieurs approches existent pour résoudre ce problème **[3]**.

Une solution classique consiste à modéliser les systèmes par des automates déterministes. Cela garantit un comportement déterministe mais cette solution reste limitée par le modèle « plat » de l'automate. Il est très difficile d'exprimer le parallélisme et la hiérarchie des systèmes et cela engendre une croissance du nombre d'état. De plus, toute modification dans les spécifications change totalement la structure de l'automate ; d'où la difficulté de maintenance du système.

Une autre solution est de considérer des tâches séquentielles qui communiquent entre elles et qui sont gérées et synchronisées par un noyau. Cette solution est très courante mais a aussi ses limites. On ne peut pas directement exprimer des contraintes temporelles dans le langage. De plus, le programmeur a besoin de mécanismes de synchronisation comme les sémaphores dont la mise-en-œuvre n'est pas toujours facile. Finalement, l'asynchronisme rend l'analyse et la mise au point des programmes très difficiles.

Nous pouvons donc en conclure que programmer des systèmes réactifs avec des solutions classiques est très difficile à maîtriser à cause des aspects temporels et de leur parallélisme intrinsèque.

2.2 Approche Synchrone

D'où l'idée d'aborder ce problème avec une approche qui prend en compte les spécificités des systèmes réactifs en développant des langages plus spécialisés. Nous voulons que les systèmes réactifs soient déterministes en produisant toujours la même réaction quand le même stimulus se répète. Cet aspect a inspiré la création des langages qui offrent une conception modulaire et parallèle et assurent aussi le déterminisme et l'aspect temps-réel de l'implémentation mono-bloc.

Dans le système réactif ci-dessous, chaque évènement est pris en compte et génère une sortie. La durée de prise en compte n'a pas d'importance pourvu que la sortie se fasse avant l'arrivée de l'évènement suivant. D'où l'idée du paradigme synchrone d'attacher cette sortie à son évènement déclencheur.

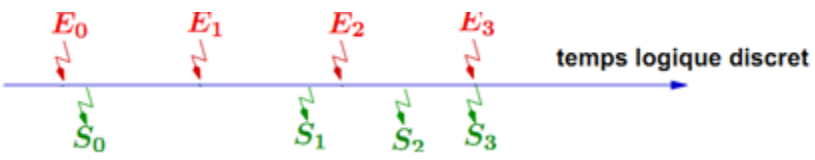

Figure 2: Différentes durées de prise en compte d'événements

En synchrone, le système réagit instantanément aux évènements externes dans leurs ordres d'arrivée (instants). Ces derniers sont des signaux d'entrée aux quels le système répond par des signaux de sortie dans un ordre chronologique qui suit le flux d'évènements d'entrée. Nous nous intéressons donc à l'enchainement des évènements indépendamment du temps physique (voir figure 3) [5].

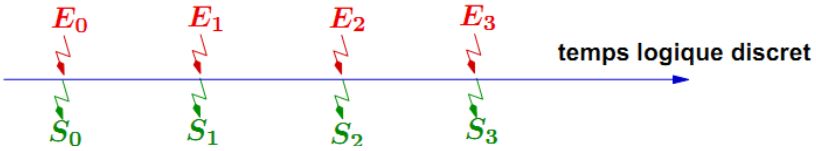

Figure 3: Flux d'événements en temps logique

De plus, tous les processus dans le système synchrone doivent être au courant de l'arrivée des évènements à n'importe quel instant logique ; leur arrivée ainsi que la réaction qu'elle a causée sont diffusées à toutes les parties de système. Les processus parallèles partagent donc les mêmes instants pour que le parallélisme des réactions devienne déterministe. Ces réactions sont atomiques non-interruptibles car elles ont lieu dans le même instant. La figure 4[5] montre un exemple d'une réaction simple « S » à une entrée « E » à l'aide de la fonction de sortie « $S_i = f(M_i, E_i)$ » et celle de transition « $M_{i+1} = g(M_i, E_i)$ ». La réaction dépend donc à la fois de l'état actuel de la mémoire interne « M » et de signal d'entrée « E ». Le fait qu'un évènement et sa réaction ont lieu dans le même instant logique peut poser un problème. En effet, le compilateur doit s'assurer qu'il n'y ait pas de cycle de causalité et détecter toutes les incohérences. Il compile le code sous forme d'automate d'état fini (machine de Mealy) dont le temps d'exécution total est borné. D'où la possibilité de valider l'hypothèse synchrone dans la totalité du système.

```
Système(E, S)
   mémoire M
   M := M0
   boucle
      attendre(E)
      S = f(M, E)
      M = g(M, E)
      écrire(S)
   fin boucle
```

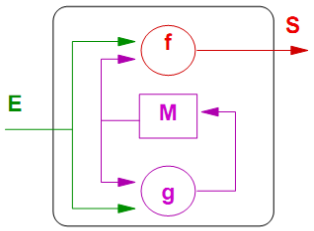

Figure 4: Modèle sous forme d'automates d'état fini ou automate de Mealy

3. Les langages synchrones:

Les langages synchrones manipulent tous des signaux typés (entrée, sortie, locaux) ainsi que des actions qui modifient le programme. Un programme synchrone exprime les contraintes que nous souhaitons imposer entre les signaux. Ces programmes peuvent être écrits en langages impératifs comme ESTEREL et SYNCCHARTS qui utilisent des structures de contrôle pour exprimer de telles contraintes. Ces dernières peuvent être aussi exprimées avec des langages déclaratifs comme LUSTRE et SIGNAL par des systèmes d'équations booléens [6].

3.1 ESTEREL

Les langages synchrones se différencient par leur mode d'expression : flot de données, textuel ou graphique, etc… On va s'intéresser plus particulièrement au langage ESTEREL qui a été développé par le Centre de Mathématiques Appliquées (CMA) à l'Ecole des Mines de Paris et par l'INRIA à Sophia Antipolis en 1983 **[7]**.

ESTEREL est un langage impératif qui exprime les instructions d'exécution de manière séquentielle pour la machine. Il est adapté à la programmation de contrôleurs, de protocoles de communication, et à la manipulation de données (types prédéfinis ou définis par l'utilisateur et variables locales, etc…). Ce langage s'exprime clairement et offre les aspects de parallélisme et de mécanismes de trappe et de préemption. Il permet aussi de manipuler le temps logique avec des instructions comme « await S» qui attend la prochaine arrivée d'un signal S et « halt » qui interrompe l'exécution.

La figure 5 ci-dessous est un exemple de programme écrit en ESTEREL **[2]** :

```
1  module Speedometer:
2  input Second, Meter;
3  output Speed : integer in
4    loop
5      var Distance := 0 : integer in
6        do
7          every Meter do
8            Distance := Distance+1
9          end every
10       upto Second;
11       emit Speed(Distance)
12     end var
13   end loop
14 end module
```

(a)

```
1  module SpeedSupervisor:
2  input Second, Meter;
3  output TooFast in
4    signal Speed : integer in
5    [ run Speedometer
6    ||
7      every Speed do
8        if ?Speed > MaxSpeed
9        then emit TooFast
10       end if
11     end every
12   ]
13   end signal
14 end module
```

(b)

Figure 5: Exemple de modules en ESTEREL

La figure 5 a) est un programme qui décrit le comportement d'un capteur de vitesse sous forme d'un module en ESTEREL. Ce module observe 2 signaux d'entrée « Second » et « Meter » (ligne 2) présents que lorsque la voiture se déplace pendant une seconde et un mètre respectivement. Le module réagit en émettant un signal de sortie « Speed » (ligne 3) qui détermine la valeur de la vitesse de déplacement en m/s de type entier. Nous pouvons remarquer que tous les évènements d'entrée sont considérés comme des évènements temporels. Nous appelons ce phénomène « le temps multi-forme ».

Le corps de ce module est la boucle (ligne 4 à 13) qui initialise une variable locale « Distance » à zéro (ligne 5) de type entier. Cette variable va s'incrémenter à chaque arrivée d'un signal « Meter » (ligne 7 à 9) pour calculer la distance parcourue par la voiture. Cette itération s'arrête quand le signal « Second » arrive (ligne 6 à 10 avec « do…upto Second »). Ensuite le système émet le signal « Speed » à chaque arrivée du signal « Second » (ligne 11).

La figure 5 b) est un programme qui décrit le comportement d'un contrôleur de vitesse sous forme d'un module en ESTEREL aussi. Ce module fait appel au module précédent de capteur de vitesse. Ce module observe les 2 mêmes signaux d'entrée « Second » et « Meter » (ligne 2). Il réagit en émettant un signal de sortie « TooFast » (ligne 3) lorsque la vitesse actuelle est supérieure à la borne « MaxSpeed ».

Un signal local « Speed » (ligne 4) est utilisé pour transmettre la valeur de la vitesse actuelle qui résulte de l'exécution du module Speedmometer (ligne 5). Ce dernier est exécuté en parallèle (ligne 6)

avec un processus de comparaison de la dernière valeur du signal « Speed » avec celle de « MaxSpeed ». Nous utilisons l'opérateur « ? » avant le signal « Speed » pour nous permettre d'obtenir sa dernière valeur. Ce processus est appelé à chaque transmission de signal « Speed » (ligne 7 à 11 avec « every Speed do »). Le système réagit alors selon le résultat de cette comparaison.

2.1 Light Esterel (LE)

LE est un langage impératif réactif synchrone qui est beaucoup inspiré d'ESTEREL. Il n'introduit pas de nouvelles notions pour la manipulation de temps logique, mais permet de regrouper plusieurs styles de spécifications pour la conception des applications. De plus, LE ajoute la compilation modulaire qui manquait en ESTEREL : il résout le problème de taille du code compilé et permet une compilation séparée des sous-modules.

Ce langage définit des modules. L'interface d'un module déclare l'ensemble des évènements d'entrée qu'il écoute, ainsi que l'ensemble d'évènements de sortie qu'il émet en réaction. Le langage LE suit la philosophie de développement des logiciels dirigés par les modèles qui est maintenant bien connu comme un moyen de gérer la complexité, pour atteindre un taux élevé de réutilisation et réduire considérablement l'effort de développement.

nothing	ne fait rien
emit S	le signal S est immédiatement présent dans l'environnement
present S { P1} else { P2}	si S est présent $P1$ est exécuté sinon $P2$
$P_1 \gg P_2$	P_1 est exécuté puis P_2
$P_1 \| P_2$	parallèle synchrone : les exécutions de P_1 et P_2 sont commencées simultanément et l'instruction termine quand ces deux exécutions sont terminées
abort P when S	P est exécuté normalement jusqu'à l'instant où S est présent
loop {P}	exécute P et recommence dès que cette exécution est terminée
local S {P}	la portée de S est restreinte à P
run M	appel du module M
pause	stoppe jusqu'à la prochaine réaction
wait S	attend la prochaine réaction où S est présent

Figure 6: Opérateurs simples du langage LE

LE regroupe plusieurs styles de conception des applications (voir figure 7) **[9]**. Ce langage permet de spécifier des contrôleurs grâce à un format graphique d'automates hiérarchiques inspirés des SYNCCHARTS (fig. 7B). Il se différencie d'ESTEREL par la prise en compte directe du format automate. En effet, SYNCCHARTS nécessite une réécriture préalable vers ESTEREL qui est coûteuse en efficacité. Il permet aussi la conception avec un langage textuel d'une application dirigée par évènements (fig. 7A). Finalement, les applications flots de données peuvent être exprimées par des systèmes d'équations proches de LUSTRE (fig. 7C).

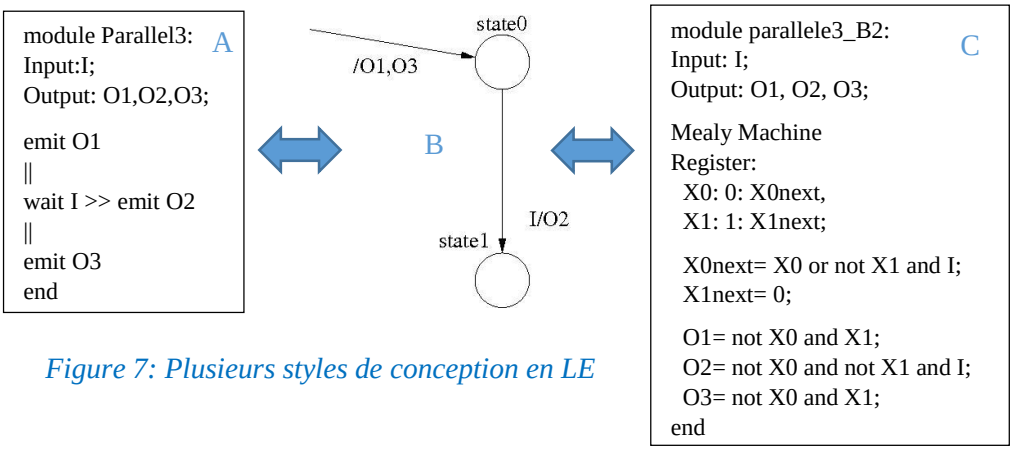

Figure 7: Plusieurs styles de conception en LE

Les figures 7A, 7B et 7C montrent trois styles différents pour concevoir la même application en LE. Cette application exécute en parallèle des actions synchrones d'émission des signaux O1 et O3 au premier instant logique. Ensuite, le module reste bloqué jusqu'à l'arrivée du signal I qui engendre en réaction l'émission de O2 et la terminaison du module. À ce stade, toutes nouvelles arrivées de I n'auront plus aucun effet sur le système qui restera dans son état final.

La figure 8 est le modèle classique équivalent aux trois formulations de la figure 7. State2 est l'état initial puis après un instant logique, on émet O1 et O3 en sortie puis au deuxième instant logique, on reste bloqué dans le state 0 tant qu'il y'ait pas de signal I qui arrive. Dès son arrivée, on passe au State1 qui est notre état final.

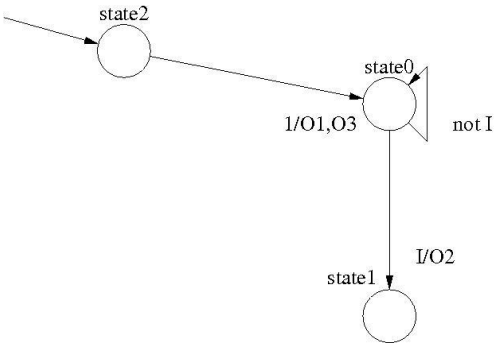

Figure 8: Automate classique équivalent

4. Le projet CLEM

4.1 Introduction

En collaboration avec l'équipe LEAT (Laboratoire d'Electronique, Antennes et Télécommunications), l'équipe STARS a conçu une boîte à outils appelée CLEM **[9]** autour du langage synchrone LE (mentionné précédemment). Celle-ci permet de compiler de façon modulaire les programmes LE, de les vérifier et de générer du code pour différents cibles logicielles ou matérielles.

4.2 Fonctionnalités de CLEM

Le fonctionnement de CLEM suit quatre phases principales illustrées dans la figure 9 :

4.2.1 Conception :

Il s'agit de la phase dédiée à la description des systèmes réactifs synchrones en utilisant LE comme nous avons vu précédemment avec les différents moyens de conception possibles (graphiques, textuels et impératifs déclaratifs).

4.2.2 Compilation :

4.2.2.1 La sémantique sous forme de systèmes d'équations :

Pour pouvoir compiler de manière modulaire les programmes LE, on implémente les règles de sa sémantique équationnelle. LE étant un langage synchrone, ses programmes sont réactifs et engendrent des évènements de sortie en fonction des évènements d'entrée. Nous pouvons voir un évènement en tant qu'un signal qui nous apporte une information sur son état, autrement dit, son « statut » **[9]**.

4.2.2.2 Algèbre quadri-valuée :

Cette sémantique génère un système d'équation quadri-valuées pour chaque programme calculant le statut des signaux de sortie et locaux en fonction des statuts des signaux d'entrée. L'ensemble de statut ξ (ξ = {⊥, 0, 1, ⊤}) est défini pour nous permettre, d'une manière très fine, d'analyser l'information portée par l'évènement, contrairement à l'approche simple booléenne.

Les statuts 0 (respectivement 1) signifient que le signal est absent (respectivement présent). Ensuite, le statut « ⊥ » ou bottom signifie que le signal est indéterminé. Finalement, le statut « ⊤ » (ou erreur ou top) signifie que le signal est incohérent, par exemple, il possède des statuts de présence et d'absence en même temps dans des parties différentes de l'application **[9]**.

4.2.2.3 Equivalence entre l'algèbre quadri-valuée et l'approche booléenne :

Après avoir associé un ξ système d'équations à chaque programme, nous passons à la transformation du système d'équations quadri-valuées en un système d'équations booléennes :

B : ξ → IB x IB , avec :

- B (⊥) = (0, 0),
- B (0) = (1, 0),
- B (1) = (1, 1),
- B (⊤) = (0, 1).

Cet encodage nous permet de passer de chaque ξ équation en deux équations booléennes.

Cette sémantique permet alors la compilation modulaire séparée de programme sur les cibles matérielles / logicielles (code C, code VHDL, SystemC, code de synthèse FPGA, etc...). Chaque module est compilé en format LEC (Light Esterel Compilation). L'avantage de ce format LEC est qu'il permet de représenter clairement des équations booléennes (réprésentant des équations quadri-valuées) qui sont ordonnées par un algorithme d'ordonnancement inspiré de la méthode CPH (Critical Path Method) **[9]**.

En effet, l'algorithme garde assez d'informations sur la causalité des signaux entre eux pour évaluer le système d'équation. Les équations sont ensuite ordonnancées en fonction des dépendances des variables du système.

4.2.3 Simulation :

Il s'agit de la simulation des programmes LE déjà compilés au format LEC. Cette phase de simulation permet d'afficher graphiquement le comportement des signaux et leurs valeurs. CLEM utilise un simulateur graphique d'automate implicite au format blif appelé « blif_simul ».

4.2.4 Finalisation :

La sémantique quadri-valuée (même avec une représentation booléenne) impose de projeter le modèle en booléen lors de la génération de code cible. Ce processus se nomme la finalisation. Dans cette étape, tous les signaux avec un état bottom sont considérés comme absents.

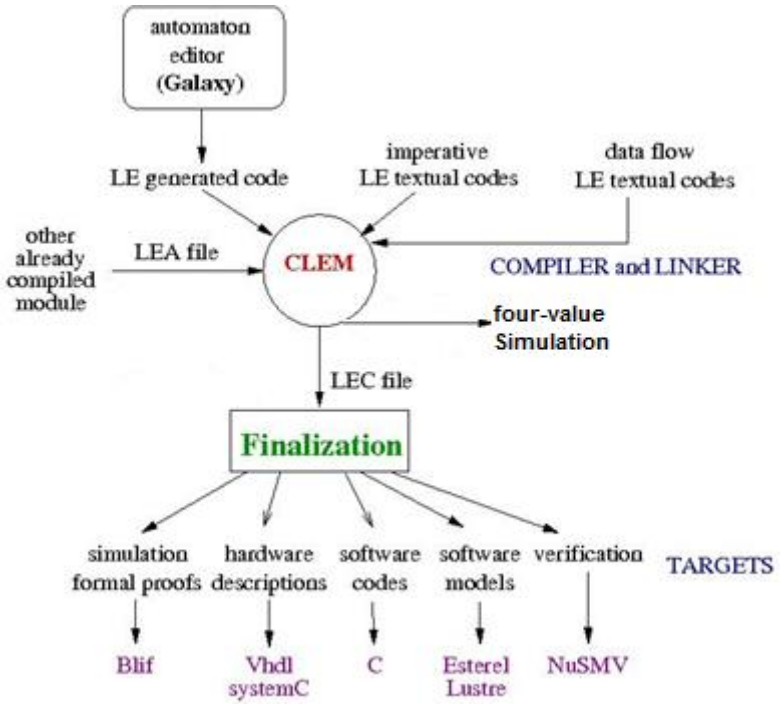

Figure 9: CLEM

5. SAM - le générateur de reconnaissance de scénario

5.1 Introduction

Dans son domaine de recherche de surveillance et reconnaissance, STARS a besoin de systèmes de reconnaissance d'activités automatiques pour reconnaître les activités observées. Ces systèmes détectent en temps réel certains évènements qui ont lieu dans leur environnement puis les interprètent pour renvoyer un rapport sur la situation actuelle. Autrement dit, ils réagissent aux évènements d'entrées en produisant des évènements de sortie, on peut ainsi les considérer comme des « systèmes réactifs » (voir chapitre précédent). Ceci explique alors le grand intérêt que ce travail porte sur l'approche synchrone.

Dans ce contexte, STARS propose un langage concret pour décrire ces activités sous formes d'un groupe de modèles de scénarios avec des contraintes temporelles entre eux. Ce langage permet aux utilisateurs de décrire leur propre modèle de scénario. Pour reconnaître ces modèles, nous considérons la description des activités en tant que systèmes synchrones réactifs. Dans l'avenir, SAM va générer un système de reconnaissance pour chaque modèle de scénario. Ce sera un outil de génération de moteurs de reconnaissance d'activités principalement qui permet la simulation et la vérification. Nous souhaitons également vérifier les propriétés de sûreté de fonctionnement avec des outils de model-checking : c'est un des objectifs du stage. Cette dernière caractéristique résultera de l'approche synchrone utilisée.

5.2 Langage de scénario SAM-L

Le langage de SAM s'appelle « SAM-L », il sert à décrire des modèles de scénario spatio-temporel, facile à utiliser pour différentes applications **[8]**. S'appuyant sur une approche théorique, l'équipe STARS a défini une sémantique décrivant le comportement de programmes de SAM-L. Les modèles de scénario sont décrits avec SAM-L. Le paradigme synchrone permet de compiler les programmes en automates à état fini, en utilisant les règles d'une sémantique comportementale. S'appuyant sur la sémantique comportementale de SAM-L, nous associons à chaque modèle de scénario une machine d'état que nous allons représenter sous la forme d'automate de Mealy et qui entrera dans la chaine de compilation CLEM.

Ces modèles de scénario sont des patterns décrivant comment des évènements primitifs provenant du contexte s'enchaînent. Ils sont décrits à l'aide des opérateurs de SAM-L comme par exemple : « before », « overlap », « during », « meets » (c'est-à-dire respectivement, avant, simultanément, pendant, juxtapose). Il existe aussi des primitives : « beginning of » et « termination of » qui aident à gérer l'instant du début ou de la fin du scénario. Il y'a aussi « duration » et « delay » qui sont utiles pour exprimer les contraintes temporelles imposées sur l'évolution du scénario comme sa durée et des délais.

Un scénario composé décrit en SAM-L se compose généralement de quatre parties :

1. **Components :** les composants du scénario (qui sont eux-mêmes des sous-scénarios).
2. **Relations :** les relations temporelles entre composants.
3. **Forbidden**: les relations temporelles interdites qui font interrompre la reconnaissance d'activités.
4. **Constraints :** un ensemble de contraintes spatio-temporelles. Ces contraintes doivent être vraies à tout instant, alors que les relations temporelles expriment des contraintes sur l'évolution temporelle du scénario plus généralement.

Un scénario primitif se compose d'un évènement et éventuellement des contraintes à respecter.

La figure 10 est un exemple d'un scénario utilisé dans le domaine de la surveillance, particulièrement des personnes âgées vivant seules, pour s'assurer de leur santé durant la journée. Il s'agit du scénario d'utilisation d'une microonde écrit en SAM-L.

Tout d'abord, nous identifions des scénarios primitifs et un scénario composé qui est le scénario principal. Cela introduit donc une hiérarchie dans la description d'un scénario en SAM-L. Chaque scénario parmi ces scénarios primitifs fait appel à un évènement cité dans la partie « **Data** ». De plus, dans notre exemple, ils se composent seulement de la partie « **Constraint** », alors que notre scénario principal se compose des trois parties « **Component** », « **Relation** » et « **Constraint** ».

Nous identifions aussi, la partie « **Data** » qui déclare les types « Person » et « Equipement » et les évènements liés à la caméra « close_to » et « far_from » ainsi que ceux liés au capteur de pression placés sur le microonde « is_used » et « is_not_used ».

Pour définir le scénario principal UsingMicrowave, nous utilisons donc les scénarios primitifs comme staysAt, microwaveUsed, microwaveNotUsed et farFrom qui correspondent chacun à un évènement respectif. En effet, les scénarios primitifs sont les composants des scénarios composés comme nous pouvons le voir dans la partie « **Component** ». Ensuite, les relations temporelles existantes entre ces composants sont définies dans la partie « **Relation** ». Finalement, nous retrouvons la constante « threshold » qui était définie dans la partie « **Data** », qui aide à définir une contrainte temporelle sur la durée d'utilisation du microonde « m_used » dans la partie « **Constraints** » du scénario principal.

```
;; Using Microwave Scenario

Data:
Type: Person, Equipment

Function: is_microwave(Person) : boolean;

Event:  far_from(Person, Equipment);
        close_to(Person, Equipment);
        is_used(Equipment);
        is_not_used(Equipment);

Constant: threshold: integer;
end Data

primitive scenario staysAt(person:Person; eq:
Equipment) {
close_to(person, eq)
Constraint: is_microwave(eq);
}

primitive scenario microwaveUsed(eq: Equipment) {
is_used(eq)
Constraint: is_microwave(eq);
}

primitive scenario microwaveNotUsed(eq:
Equipment) {
is_not_used(eq)
Constraint: is_microwave(eq);
}

primitive scenario farFrom(person: Person; eq:
Equipment) {
far_from(person, eq)
Constraint: is_microwave(eq);
}

main scenario usingMicrowave (
person: Person; eq: Equipment)
{
   Component:
    p_stay :   staysAt (person, eq);
    m_used : microwaveUsed (eq);
    m_not_used :  microwaveNotUsed (eq);
    p_far :   farFrom (person, eq);

   Relation:
    m_used during p_stays;
    m_used before m_not_used;
    m_not_used before p_far;

   Constraint:
    duration(m_used) >= threshold;
    is_microwave(eq);
}
```

Figure 10: Exemple d'un scénario (Utilisation d'une micro-onde)

III. Travail effectué

1. Objectif du stage

Le travail demandé durant ce stage de 5 mois est d'étudier la sémantique SAM-L. Puis, de proposer une transcription automatique des scénarios SAM-L en langage synchrone. L'objectif est de d'exprimer la sémantique de SAM-L en LE. De cette manière, les moteurs de reconnaissance seront une implémentation directe de l'automate synchrone modèle du programme LE associé au scénario. Pour faciliter cette implémentation, nous allons nous appuyer sur CLEM qui répond aux besoins de SAM comme vérifier des modèles de scénarios ainsi que générer des moteurs de reconnaissances (voir figure 11). D'où la nécessité d'un passage de la sémantique synchrone SAM-L en LE pour bénéficier de CLEM et de son environnement de développement.

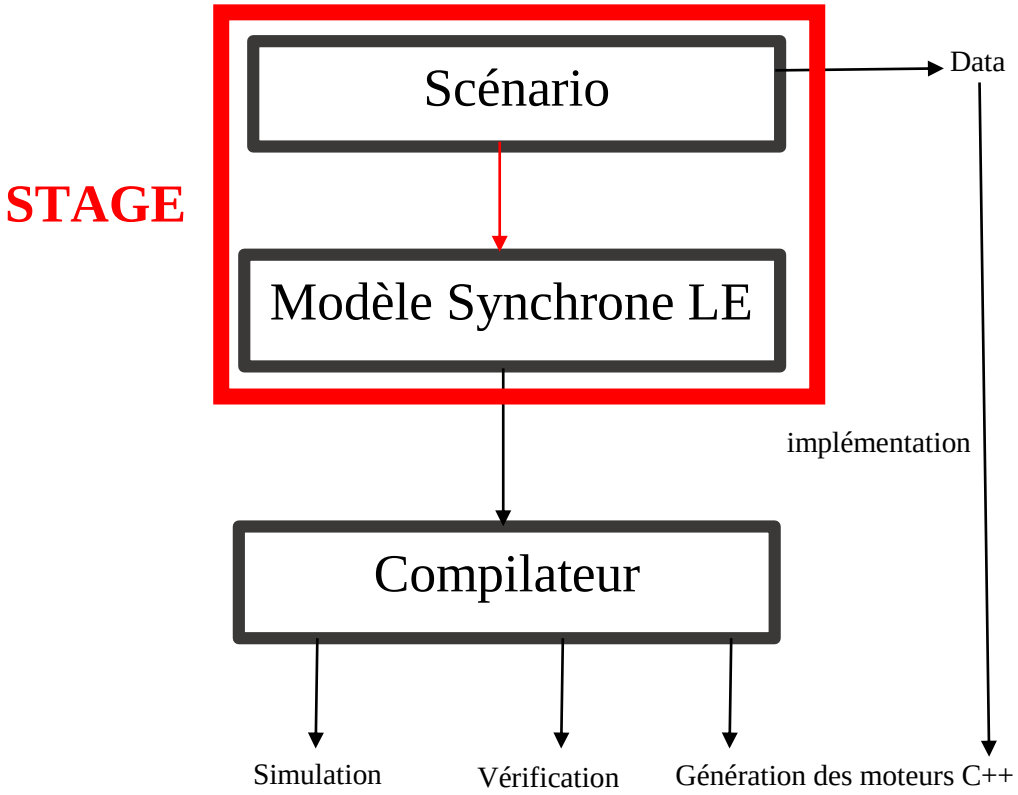

Figure 11: Objectif du stage

En effet, le modèle d'un scénario est un automate de Mealy exprimé en LE qui va calculer deux signaux « beginning » et « termination ». Ces signaux serviront de points de repère, pour le début et la fin de scénario. Ce modèle permettra de concevoir le moteur de reconnaissance de façon naturelle, comme implémentation de l'automate.

2. Passage de SAM-L en LE :

Afin d'exprimer la sémantique de SAM-L en LE, nous allons, dans un premier temps, étudier un exemple complexe : « bankAttack » et nous appuierons sur cet exemple pour en déduire une traduction des opérateurs de SAM-L en LE. Cette traduction nous permettra de calculer la sémantique de tout scénario.

2.1 Scénario « bankAttack » en SAM- L

```
;; Bank Attack Scenario
Data:
Type: Person, BackCounter, Branch, Entrance,
        FrontCounter, Safe, SmokingArea;

Function: is_any(Person) : boolean;

Event: in_back_counter(Person);
      in_branch(Person);
      in_entrance(Person);
      in_front_counter(Person);
      in_safe(Person);
      in_smoking_area(Person);
end Data

primitive scenario
insideBackCounter
(person:Person) {
    in_back_counter(person)
}

primitive scenario insideBranch(person:Person)
{
    in_branch(person)
}

primitive scenario insideEntrance(person:Person)
{
    in_entrance(person)
}

primitive scenario insideFrontCounter
(person:Person)
{
    in_front_counter(person)
}

primitive scenario insideSafe(person:Person)
{
    in_safe(person)
}

primitive scenario insideSmokingArea
(person:Person)
{     in_smoking_area(person)              }
```

```
scenario changeZone (person: Person)
{
  Component:
    pers_in_entrance : insideEntrance(person);
    pers_in_front_counter :
insideFrontCounter(person);

  Relation:
    pers_in_entrance before pers_in_front_counter;

  Constraint:
    delay(termination of pers_in_entrance,
    beginning of pers_in_front_counter) < 5;
}

main scenario bankAttack (
back_counter : BackCounter;
branch : Branch; entrance : Entrance;
 front_counter : FrontCounter; safe : Safe;
smoking_area : SmokingArea;
p1 : Person; p2 : Person; p3: Person)
{
   Component:
    cas_at_pos :   insideBackCounter(p1);
    cas_at_smoke : insideSmokingArea(p1);
    cas_at_safe :  insideSafe(p1);
    rob_enters :   changeZone(p2);
    rob_at_safe:   insideSafe(p2);
    any_in_branch :insideBranch(p3);

   Forbidden: any_in_branch;

   Relation:
    rob_enters during cas_in_place;
    cas_in_place before cas_at_safe;
    rob_enters before cas_at_safe;
    rob_at_safe during cas_at_safe;
    any_in_branch during rob_at_safe;
    cas_in_place : cas_at_pos | cas_at_smoke;

   Constraint:
    delay (beginning of cas_at_safe,
    beginning of rob_at_safe) < 8;
    is_any(p3);
   }
```

Ce scénario reconnait les activités de trois personnes, un employé qui travaille à la caisse dans une banque « p1 », un voleur potentiel de cette banque « p2 », et une troisième personne « p3 ». Concernant l'employé, le scénario reconnait les évènements suivants : travailler derrière son bureau, aller à l'espace fumeur, et aller à la salle des coffres forts. De plus, il reconnait quand le voleur entre dans la banque et également quand il va à la salle des coffres forts (sachant que seul l'employé p1 n'est autorisé à y accéder). Finalement, si une troisième personne entre dans l'agence, le scénario n'est pas reconnu.

2.2 Décomposition et démarche suivie

Notre but est de construire un module LE qui représente la sémantique du scénario bankAttack. Le but de cette sémantique est de calculer le statut des deux signaux de sortie « beginning » et « termination » vrais respectivement quand le scénario commence et quand il termine.

Nous cherchons donc à traduire les scénarios en modules LE. Nous allons tout d'abord commencer par décomposer le scénario décrit en SAM-L selon ses sous-scénarios (figure 12).

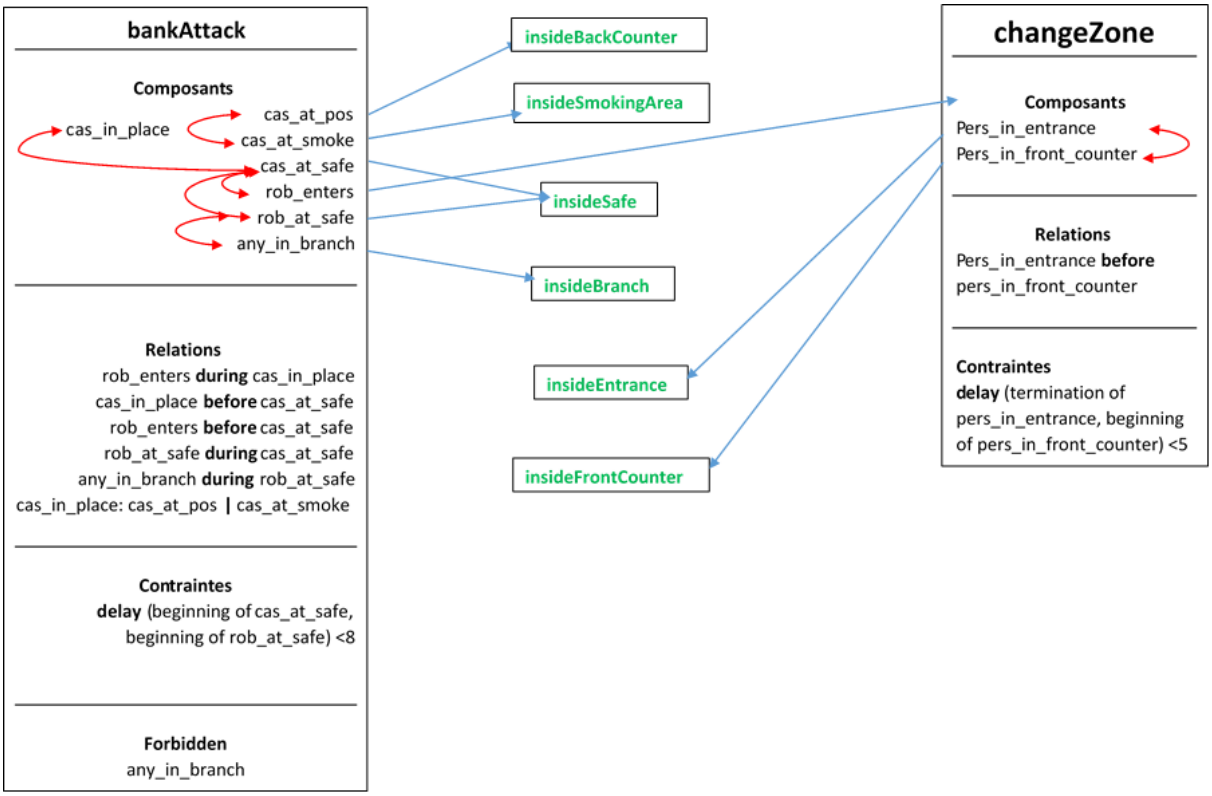

Figure 12: Décomposition du scénario bankAttack décrit en SAM-L

On distingue alors la hiérarchie dans la description de scénario. Nous avons effectivement des scénarios primitifs en vert et les deux scénarios composés bankAttack et changeZone. Nous retrouvons les quatre parties importantes de scénario composé.

Nous pouvons voir que les composants des scénarios primitifs font appel à des scénarios primitifs (les flèches bleues). De plus, ces composants ont des relations entre eux et donc communiquent entre eux (les flèches rouges).

On peut commencer par en déduire cette règle générale de passage de SAM-L en LE.

Règle 1 :
Tout scénario s'exprime en LE dans un module à part qu'on peut appeler grâce à l'opérateur

Nous venons de voir que les évènements des scénarios primitifs sont les six évènements définis dans la partie « **Data** » du scénario décrit en SAM-L. Or, le scénario changeZone est composé de deux scénario primitifs, et le scénario principal bankAttack est composé de changeZone lui-même et les quatre autres scénarios primitifs.

La première approche pour définir les signaux d'entrée de module LE était d'utiliser alors les évènements définis dans la partie « **Data** ». Ensuite, concernant les signaux de sortie, nous allons émettre les signaux « beginning » et « termination » génériques qui signaleront le début et la fin du scénario.

Nous pouvons alors déduire ces deux règles de traductions :

Règle 2 :
Le module associé à un scénario aura comme signaux d'entrée : les évènements de la partie « Event » dans « **Data** » du scénario décrit en SAM-L.

Règle 3 :
Le module associé à un scénario aura comme signaux de sortie : « beginning » et « termination ».

En appliquant ces deux règles à notre exemple, on obtient le code LE suivant :

```
module bankAttack:

Input: in_back_counter_p1, in_smoking_area_p1, in_entrance_p2, in_front_counter_p2, in_safe_p,
       in_branch_p3;

Output: beginning, termination,
```

2.2.1 Scénarios primitifs

Nous pouvons voir dans notre exemple, dans la partie « **Data** » du scénario en SAM-L, que nous avons six évènements. Or, nous avons également six scénarios primitifs, qui ne contiennent qu'un évènement chacun. Nous devons donc trouver un moyen pour les écouter dans notre module LE. Nous

avons décidé de concevoir un automate LE, qu'on a appelé « Event », avec l'aide de l'éditeur graphique Galaxy (voir figure 13).

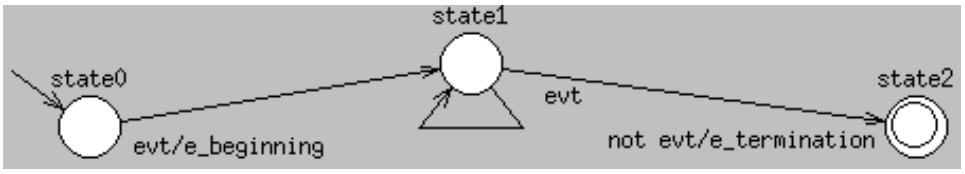

Figure 13: Automate LE Event avec Galaxy

En effet, cet automate sert à déclencher le début de l'évènement en le signalant aux autres relations/évènements. L'évènement en question, une fois démarré, se terminera lorsque le signal d'entrée « evt » ne sera plus présent. Ensuite, l'automate arrivera à l'état final et signalera la fin de l'évènement. Il faut donc toujours garder le signal « evt » émis tant que l'évènement ne s'est pas terminé durant le scénario.

Nous pouvons en déduire la règle suivante :

Règle 4 :
Le module associé à un scénario primitif, est l'automate event. Il sera utilisé avec un opérateur « run » qui fera aussi le renommage de l'évènement.

L'interface se compose d'un seul signal d'entrée et de deux signaux de sortie, comme vu dans le tableau suivant (figure 13) :

NOM	TYPE	SIGNIFICATION
evt	Entrée	Déclenche l'automate
e_beginning	Sortie	Prévient tout autre automate sensible à cet évènement qu'il a commencé
e_termination	Sortie	Prévient tout autre automate sensible à cet évènement qu'il a terminé

Figure 14: Tableau des signaux de l'automate Event

Nous allons donc appeler cet automate avec l'opérateur « Run » en LE pour chaque scénario primitif et les mettre tous en parallèle. En appliquant la règle 1 à notre exemple, on obtient le code LE suivant :

```
Run: ".": event: event;

local cas_at_safe_b, cas_at_safe_t, cas_at_smoke_b, cas_at_smoke_t, any_in_branch_b, any_in_branch_t
 {
    run event [in_back_counter_p1 \evt, cas_at_pos_b\e_beginning, cas_at_pos_t\e_termination]
   ||
    run event [in_smoking_area_p1 \evt, cas_at_smoke_b\e_beginning, cas_at_smoke_t\e_termination]
   ||
…
```

2.2.2 Scénario composés

Nous passons maintenant aux scénarios composés. Nous en avons deux dans notre exemple : le scénario changeZone et le scénario principal bankAttack.

2.2.2.1. Component

Nous allons détailler la composition des scénarios primitifs dans cette partie. En effet, Le scénario changeZone est composé des scénarios primitifs : « insideEntrance(Person) » et « insideFrontCounter(Person) ». Ensuite, le scénario principal bankAttack est composé du scénario changeZone ainsi que le reste des scénarios primitifs.

a) Composants issus d'autres scénario composés

Pour faciliter l'appel aux composants d'un autre scénario composé, nous allons donc traduire les scénarios composés en un module à part chacun. Nous pouvons ainsi faire appel à chaque scénario composé dans un autre scénario grâce à l'opérateur « Run » en cas de besoin. On en déduit la règle suivante :

Nous appliquons la règle 1 à notre exemple, pour appeler le module dédié au scénario changeZone dans le module bankAttack, et on obtient le code LE suivant :

```
Run: ".": changeZone: changeZone;

local changeZone_beginning, changeZone_termination {

run changeZone[changeZone_beginning\beginning, changeZone_termination\termination]
```

b) Composants issus des scénarios primitifs

Ensuite, concernant les composants des scénarios composés, qui sont issus de scénario primitifs, la première approche était d'émettre tous les évènements dans leur bon ordre grâce aux opérateurs LE : « emit » et « >> » (séquence) d'une manière textuelle.

Cela dit, on remarque que ces composants sont présents dans plusieurs relations à la fois. Nous allons détailler ce point dans la partie suivante.

Il fallait donc, modifier cette solution pour obtenir un passage beaucoup plus générique qui pourra s'appliquer aux autres modèles de scénarios sans soucis. Nous allons appliquer la philosophie de conception en LE nous permettant d'utiliser plusieurs styles de conception au sien du même module. D'où l'idée d'ajouter l'automate Event qui a été expliqué en détails précédemment (figures 12 et 13).

Cela nous confirme l'importance de la règle 1 lors de la distinction des composants au sein du même de scénario composé.

Cela dit, nous remarquons dans l'exemple, qu'il existe deux composants différents du scénario principal qui font appel au même scénario primitif : « cas_at_safe » et « rob_at_safe » font appel à « insideSafe(Person) » avec des arguments différents de Person : p1 et p2. On revient sur le comportement du scénario, avec la distinction entre la présence de l'employé et celle du voleur dans la salle de coffres forts au niveau de la reconnaissance d'activité. Nous sommes obligés de reformuler la règle 2, qui prétendait que les évènements dans la partie **« Data »** étaient suffisants pour déduire tous les évènements du scénario comme le suivant :

Règle 2 : (modifiée)
Le module associé à un scénario aura comme signaux d'entrée : le résultat d'un matching effectué entre évènement déclaré dans « **Data** » et évènement écouté par le scénario.

En appliquant ces deux règles à notre exemple, on obtient le code LE suivant :

```
module bankAttack:

Input: in_back_counter_p, in_smoking_area_p, in_entrance_p, in_front_counter_p, in_safe_p1,
       in_safe_p2, in_branch_p;
```

2.2.2.2. Relation

Cette partie concerne les relations temporelles entre les composants des scénarios et la relation « OU exclusif ».

a) Relation OU exclusif

Nous remarquons que parmi les relations du scénario principal bankAttack, nous avons une relation avec un composant qui n'a pas été défini dans la partie « **Component** » : « cas_in_place before cas_at_safe ». Or, « cas_in_place » est défini plus tard, dans la partie « **Relation** » ainsi :
« cas_in_place: cas_at_pos | cas_at_smoke ». Cela veut dire que ce composant peut être soit l'un soit l'autre composant (« | » signifie OU exclusif). Cela nous amène à la règle 5 :

Règle 5:
Une relation OU exclusif entre deux composants est exprimée en LE de la manière suivante :

- L'utilisation de l'opérateur « wait » pour attendre les deux composants
- L'utilisation de l'opérateur « present » pour tester la présence du premier composant et démarrer le premier évènement et « else » pour démarrer le deuxième si le premier n'est pas présent.

Nous appliquons cette règle à notre exemple pour obtenir le code LE suivant dans le module bankAttack :

```
wait in_back_counter_p1 or in_smoking_area_p1
   >> emit cas_in_place_start
   >> present in_back_counter_p1
      { run event [in_back_counter_p1\evt, cas_in_place_b\e_beginning, cas_in_place_t\e_termination] }
       else
          run event [in_smoking_area_p1\evt, cas_in_place_b\e_beginning, cas_in_place_t\e_termination]
```

En effet, nous attendons un des deux évènements et nous testons la présence de l'évènement d'arrivé avec l'opérateur « present » pour savoir s'il s'agit du premier composant ou du second. Dans les deux cas, on exécute l'automate Event avec le nom du composant approprié.

b) Relations temporelles

Les deux relations temporelles, les plus utilisés, qui nous intéressent dans SAM-L sont « Before » et « During ». Leurs comportements sont définis dans les figures 15 et 16 respectivement.

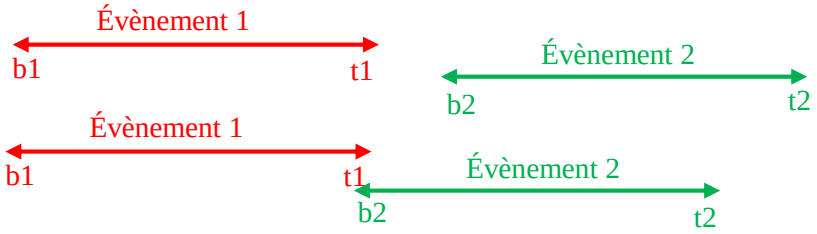

Figure 15: Comportement de la relation before

Comme la figure 15 montre, la contrainte temporelle pour la relation « before » est que le premier évènement doit se terminer avant OU dans le même instant logique que le début du deuxième évènement.

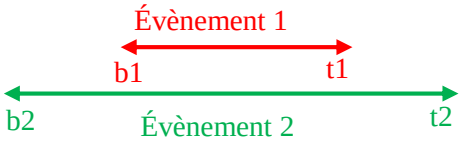

Figure 16: Comportement de la relation during

Les contraintes temporelles pour la relation « during », comme on voit dans la figure 15, sont :

- le premier évènement doit commencer après le début du deuxième évènement.
- Ce premier évènement doit aussi se terminer avant que le deuxième se termine.

La première démarche était de traduire ces relations sous forme d'automate qui décrit le comportement souhaité avec Galaxy (voir figure 17 et 19).

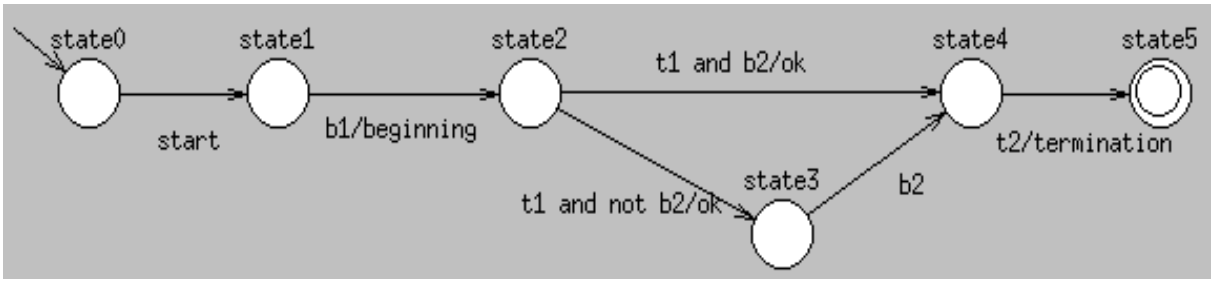

Figure 17: Automate LE de la relation BEFORE avec Galaxy

L'interface se compose de cinq signaux d'entrée et trois signaux de sortie pour décrire « évènement 1 before évènement 2 ». L'enchainement démarre lors de l'émission du signal « start » puis attend le signal « b1 » pour déclencher « beginning » aux autres automates/relations dans le scénario. Ensuite, on attend « t1 » pour déclencher « ok » et il se peut que « b2 » arrive au même instant logique que « t1 », mais cela n'est pas obligatoire. Finalement, la relation se termine en déclenchant « termination » dès l'arrivée de « t2 ». Ces signaux sont listés et expliqués plus précisément dans le tableau suivant (figure 18) :

NOM	TYPE	SIGNIFICATION
Start	Entrée	Déclenche l'automate
b1	Entrée	Début du premier évènement
beginning	Sortie	Prévient autres relations/automates que le premier évènement a été commencé
t1	Entrée	Fin du premier évènement
Ok	Entrée	Prévient autres relations/automates que le premier évènement a été terminé
b2	Sortie	Début du deuxième évènement

t2	Entrée	Fin du deuxième évènement
termination	Sortie	Prévient autres relations/automates que le deuxième évènement a été terminé et cela marque la fin de la relation

Figure 18: Tableau des signaux de l'automate Before

De même pour décrire « évènement 1 during évènement 2 », on utilise les mêmes signaux d'entrée et de sortie que pour l'automate de « before ». Voici l'automate de during (figure 19) :

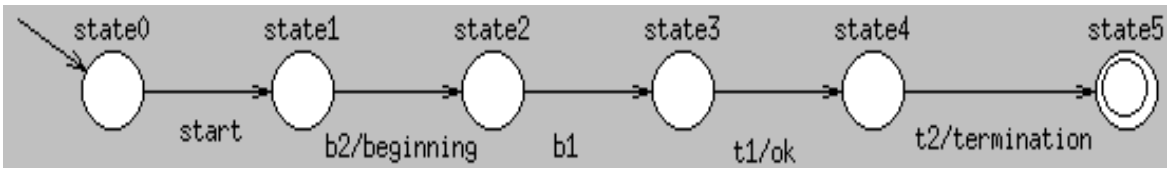

Figure 19: Automate LE de la relation DURING avec Galaxy

L'interface se compose aussi de cinq signaux d'entrée et trois signaux de sortie pour décrire « évènement 1 during évènement 2 ». L'enchainement démarre lors de l'émission du signal « start » puis attend le signal « b2 » pour déclencher « beginning » aux autres automates/relations dans le scénario. Ensuite, on attend « b1 » puis « t1 » dans cet ordre-là, pour déclencher « ok ». Finalement, la relation se termine en déclenchant « termination » dès l'arrivée de « t2 ». Ces signaux sont listés et expliqués plus précisément dans le tableau suivant (figure 20) :

NOM	TYPE	SIGNIFICATION
start	Entrée	Déclenche l'automate
b2	Entrée	Début du deuxième évènement
beginning	Sortie	Prévient autres relations/automates que le deuxième évènement a été commencé
b1	Entrée	Début du premier évènement
t1	Entrée	Fin du premier évènement
Ok	Sortie	Prévient autres relations/automates que le premier évènement a été terminé
t2	Entrée	Fin du deuxième évènement

termination	Sortie	Prévient autres relations/automates que le deuxième évènement a été terminé et cela marque la fin de la relation

Figure 20: Tableau des signaux de l'automate During

On remarque aussi la présence des signaux de sorties (beginning et termination) communes dans ces trois automates. Ces signaux aident beaucoup à l'intégration des relations ou sous-scénarios ou évènements dans le scénario. Ils sont donc utilisés en tant que des points de repère pour repérer le début et la fin de chaque relation ou évènement qui est nécessaire lors de la composition de scénario.

Dans un premier temps, nous avons utilisé ces deux automates pour traduire les relations temporelles « before » et « during ». Cela dit, nous devions émettre les signaux nécessaires pour le démarrage et terminaison de ces automates.

Cela dit, on remarque que vu la répétition de ces relations, et la présence du même composant dans plusieurs relations, nous étions amené à trouver une autre approche.

Règle 6 :
Chaque relation temporelle est exprimée par un « Run » de son automate qui renomme les évènements en fonction des signaux beginning et termination de ses arguments. Ces derniers sont émis par le « run » des modules associés aux composants.

Nous appliquons cette règle à notre exemple pour obtenir le code LE suivant.

```
module bankAttack:
Input: in_back_counter_p1, in_smoking_area_p1, in_entrance_p2, in_front_counter_p2, in_safe_p1,
       in_safe_p2, in_branch_p3;
Output: beginning, termination,
Run:
".": during: during;
".": before: before;
".": changeZone: changeZone;
".": event: event;
local
cas_in_place_start, cas_in_place_b, cas_in_place_t, cas_in_place_ok,
cas_at_safe_b, cas_at_safe_t, cas_at_safe_beginning,
rob_at_safe_beginning, rob_at_safe_b, rob_at_safe_t, rob_at_safe_ok,
any_in_branch_b, any_in_branch_t, any_in_branch_ok,
changeZone_beginning, changeZone_termination, changeZone_ok,
r4_beginning, r1_termination, r2_termination, r3_termination, r4_termination
{
wait in_back_counter_p or in_smoking_area_p
   >> emit cas_in_place_start
   >> present in_back_counter_p
      { run event [in_back_counter_p\evt, cas_in_place_b\e_beginning, cas_in_place_t\e_termination] }
      else
          run event [in_smoking_area_p\evt, cas_in_place_b\e_beginning, cas_in_place_t\e_termination]
   ||
```

Les modules appelés

Relation OU exclusif

Appel aux modules associé au sous-scénario changeZone, et à l'automate « Event » avec renommage de signaux d'interface

```
run changeZone[changeZone_beginning\beginning, changeZone_termination\termination]
||
run event [in_safe_p1\evt, cas_at_safe_b\e_beginning, cas_at_safe_t\e_termination]
||
run event [in_safe_p2\evt, rob_at_safe_b\e_beginning, rob_at_safe_t\e_termination]
||
run event [in_branch_p3\evt, any_in_branch_b\e_beginning, any_in_branch_t\e_termination]
||
;; R1: e2 during e1 - rob_enters DURING cas_in_place
run during[cas_in_place_start\start, cas_in_place_b\b2,  changeZone_beginning\b1,
changeZone_termination\t1, changeZone_ok\ok, cas_in_place_t\t2,  r1_termination\termination]
||
;; R2: e1 before e3 -  cas_in_place BEFORE cas_at_safe
run before[cas_in_place_start\start, cas_in_place_b\b1, cas_in_place_t\t1, cas_in_place_ok\ok,
cas_at_safe_b\b2, cas_at_safe_t\t2, r2_termination\termination]
||
;; R3: e2 before e3 -  rob_enters BEFORE cas_at_safe
run before[cas_in_place_start\start, changeZone_beginning\b1, changeZone_termination\t1,
changeZone_ok\ok, cas_at_safe_b\b2, cas_at_safe_t\t2, r3_termination\termination]
||
;; R4: e4 during e3 -  rob_at_safe DURING cas_at_safe;
run during[cas_in_place_ok\start, cas_at_safe_b\b2, cas_at_safe_beginning\beginning,
rob_at_safe_b\b1, rob_at_safe_t\t1, rob_at_safe_ok\ok, cas_at_safe_t\t2, r4_termination\termination]
```

Nous pouvons voir l'appel de l'automate Event avec un renommage de signaux d'interface pour tenir compte de l'évènement associé. De même pour l'appel de « Before » et « During »

2.2.2.3. Constraint

Cette partie dans SAM-L est utilisée pour exprimer un ensemble de contraintes spatio-temporelles. Ces contraintes doivent être vraies à tout instant, alors que les relations temporelles vues précédemment (during / before) expriment des contraintes sur l'évolution temporelle du scénario plus généralement.

Dans l'exemple du scénario bankAttack, la contrainte est « delay (beginning of cas_at_safe, beginning of rob_at_safe) < 8 ». On impose que le délai entre le début du composant « cas_at_safe » et le début du « robt_at_safe » soit strictement inférieur à 8. Il s'agit bien de 8 instants logiques.

Cela se traduit donc en LE grâce à un compteur binaire. Il émet SUP tant qu'on n'a pas dépassé le délai et émet INF dès qu'on l'a dépassé. Au fait, on fixe le nombre d'instant grâce aux quatre signaux qui agissent en tant que 4 bits, donc pour 8, on attribue les valeurs 1000 aux quatre signaux dans cet ordre. Ensuite, on fait appel à ce compteur dans un double « weak abort » dans le module bankAttack comme on voit dans la figure 21.

```
weak abort{
        weak abort{
                wait cas_at_safe_b
              >> run inc4_cmp_cte8 [INF8\INF]
                }   when INF8 >> emit failure
      }when rob_at_safe_b
```

Figure 21: Passage de SAM-L en LE pour les délais

En effet, on émet un signal « failure » au cas où le signal INF8 est déclenché avant que le signal du composant « rob_at_safe_b » est émis. Cela voudrait dire qu'on a dépassé le délai maximum. Sinon, on n'émet rien.

On en déduit la règle 7 :

Règle 7 :
Le module associé au délai en SAM-L est le compteur approprié, intégré au sein d'une double préemption avec l'opérateur « weak abort ».

2.2.2.4. Forbidden

Cette partie dans SAM-L est utilisée pour exprimer l'ensemble des évènements interdits qui font interrompre la reconnaissance d'activité et donc doit interrompre le scénario. Les relations « forbidden » sont traduites de façon similaire aux relations temporelles.

Dans l'exemple du scénario bankAttack, il y'a un seul évènement interdit « in_branch (Person) » qui fait appel au scénario primitif « insideBranch(Person) », utilisé dans le scénario bankAttack par le composant « any_in_branch ». Ce dernier, se trouve dans une relation « during » avec un autre composant « rob_at_safe ».

Pour tenir compte des « forbidden », nous devons modifier la règle 3 pour ajouter un signal de sortie « failure » :

Règle 3 : (modifiée)
Le module associé à un scénario aura comme signaux de sortie : « beginning », « termination » et « failure ».

On émet donc « failure » quand la relation de l'évènement interdît se termine.

On remarque ensuite que cela n'arrête pas le scénario global, donc le comportement souhaité n'est pas le bon. Il faut donc ajouter une préemption sur tout le scénario pour arrêter la reconnaissance et émettre « failure » en cas d'émission d'un signal local «failure_forbidden ». Ce dernier est émis lorsque l'évènement interdit a lieu.

Pour un modèle général avec un ensemble d'évènements interdits (et donc pas un seul), on émettra un failure_forbidden_1, failure_forbidden_2, etc… Sinon, dans le cas où, cela nous intéresse pas de savoir quel évènement interdit s'est produit, on peut émettre le même signal « failure_forbidden » pour chaque évènement interdit.

Finalement, on s'aperçoit du problème que le scénario ne se termine pas, si on l'utilise dans un autre scénario. En effet, on est obligé de tenir en compte la possibilité qu'aucun évènement interdit n'ait pas lieu. D'où l'intérêt de mettre un « weak abort » sur l'ensemble des évènements interdit en parallèle (voir figure 22).

```
weak abort{
        Evt_Interdit_1 >> emit failure_forbidden_1
    ||  Evt_Interdit_2 >> emit failure_forbidden_2
    || ….
     }  relations_termination
```

Figure 22: Passage de SAM-L en LE pour les évènements interdits

Le signal local « relations_termination » est émis lorsque toutes les relations (during/before) sont terminées (c'est l'application du paradigme synchrone). La figure 22 montre que le scénario se terminera en effet si jamais le signal « relations_termination » a été émis et donc plus besoin d'attendre l'émission des signaux de failure.

3. Vérification du passage de SAM-L en LE :

3.1 Récapitulatif des règles

Règle 1 :
Tout scénario s'exprime en LE dans un module à part qu'on peut appeler grâce à l'opérateur « Run ».

Règle 2 :
Le module associé à un scénario aura comme signaux d'entrée : le résultat d'un matching effectué entre évènement déclaré dans « **Data** » et évènement écouté par le scénario.

Règle 3 :
Le module associé à un scénario aura comme signaux de sortie : « beginning », « termination » et « failure ».

Règle 4 :
Le module associé à un scénario primitif, est l'automate event. Il sera utilisé avec un opérateur « run » qui fera aussi le renommage de l'évènement.

Règle 5:
Une relation OU exclusif entre deux composants est exprimée en LE de la manière suivante :

- L'utilisation de l'opérateur « wait » pour attendre les deux composants
- L'utilisation de l'opérateur « present » pour tester la présence du premier composant et démarrer le premier évènement et « else » pour démarrer le deuxième si le premier n'est pas présent.

Règle 6 :
Chaque relation temporelle est exprimée par un « Run » de son automate qui renomme les évènements en fonction des signaux beginning et termination de ses arguments. Ces derniers sont émis par le « run » des modules associés aux composants.

Règle 7 :
Le module associé au délai en SAM-L est le compteur approprié, intégré au sein d'une double préemption avec l'opérateur « weak abort ».

3.2 Vérification du comportement avec blif_simul

Pour pouvoir vérifier le comportement du scénario et le déroulement des évènements dans le bon ordre, nous allons utiliser l'outil de simulation « blif_Simul » qui est intégré dans CLEM. Ce simulateur nous permet de vérifier le comportement du scénario en émettant les signaux d'entrée et regardant les valeurs des signaux de sortie.

Cela dit, notre scénario qui se compose de plusieurs relations et nombreux évènements et même fait appel à un autre scénario, n'a que trois signaux de sorties pour indiquer le début, fin et a défaillance. Pour observer les signaux « beginning » et « termination » des relations / évènements, il a fallu donc ajouter des signaux de sortie qui sont émis lors de l'émission des signaux de type local qui crée notre scénario global.

Nous rappelons que les relations du scénario bankAttack que nous devons vérifier sont les suivants :

1. rob_enters during cas_in_place;
2. cas_in_place before cas_at_safe;
3. rob_enters before cas_at_safe;
4. rob_at_safe during cas_at_safe;
5. any_in_branch during rob_at_safe; ← en sachant que any_in_branch est un évènement interdit.

Il ne faut pas oublier aussi que : cas_in_place : cas_at_pos | cas_at_smoke (| signifie « ou »)
On en déduit donc que le comportement du scénario bankAttack est comme la figure 23 :

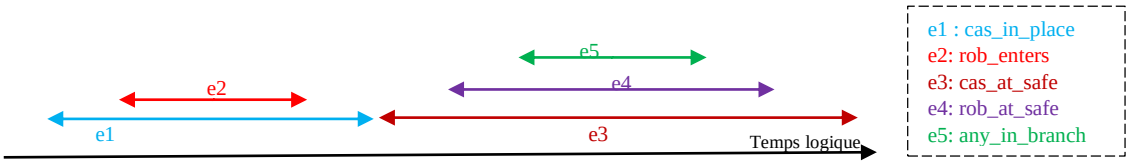

Figure 23: Comportement du scénario bankAttack

Nous remarquons dans la figure 24, tout à gauche, sous la colonne « Signal », les signaux d'entrée de notre module bankAttack, puis la colonne « Sticky » est utilisée pour maintenir un signal émis pour éviter à l'émettre à chaque instant dans la colonne signal. Ensuite, on a les registres et les variables locaux utilisés, on s'intéresser pas trop à cette partie pour notre travail. Finalement, la partie qu'on observe : « Output », tous les signaux de sorties. Les signaux en bleu sont émis, et ceux en jaune ne le sont pas.

3.2.1. Simulation de la terminaison du scénario

Tout d'abord, nous allons simuler la séquence qui devra nous amener à la terminaison du scénario sans avoir eu des failures. L'idée est donc de signaler avec les signaux « beginning » et « termination » le début et la fin de chaque évènement et de chaque relation. Le scénario ne se terminera pas que lorsque toutes les relations (sauf ceux concernant les évènements interdits) auront envoyé leur signal « termination ».

Pour cela, nous allons garder le signal d'entrée « in_back_counter_p1 » émis, il s'agit bien du premier évènement. Au premier instant, nous observons « cas_in_place_start » qui déclenche l'automate de la première relation (rob_enters during cas_in_place). Au deuxième instant, nous avons les signaux « cas_in_place_b » pour marquer le début du premier évènement et « beginning » qui prévient les autres scénarios en parallèle (voir figure 24). Nous aurions eu le même résultat en cas d'émission du signal « in_smoking_area_p1 ».

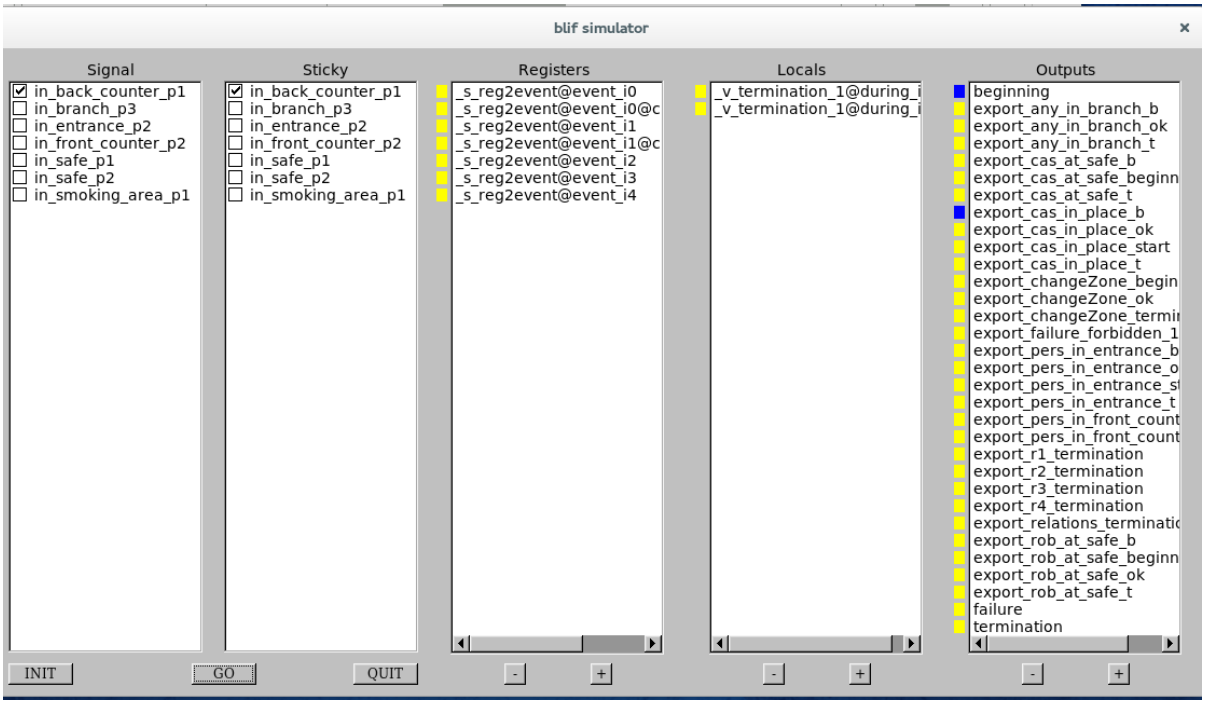

Figure 24: Simulation du scénario BankAttack, 2ème instant logique

L'opérateur « run » nous permet de renommer les signaux d'interfaces, mais nous avons choisi de nommer « beginning » le déclenchement du premier évènement de scénario. De plus, concernant la deuxième relation (cas_in_place before cas_at_safe), son signal beginning concerne le déclenchement du premier évènement du scénario. Nous avons donc aussi choisi de nommer « beginning » pour signaler le début de la première et la deuxième relation ainsi que le début vu qu'ils marquent le début du premier évènement. cas_in_place.

Ensuite, au troisième instant, lorsqu'on maintient le signal d'entrée « in_entrance_p2 », nous avons l'émission du signal «pers_in_entrance_start » et puis, dans le quatrième instant, l'émission de « changeZone_beginning ». Cela marque le début du deuxième évènement, qui est un scénario composite de 2 évènements liée par une relation before : pers_in_entrance before pers_in_front_counter.

Ainsi de suite, nous continuons la simulation en émettant les signaux d'entrées dans le bon ordre pour arriver à l'état final (voir figure 25). Nous pouvons voir en effet l'émission du signal « cas_at_safe_t », qui marque la fin du troisième évènement. Cela marque aussi la fin de la deuxième, troisième et quatrième relation, puisque elles concernent toutes l'évènement « cas_at_safe ». Finalement, nous avons l'émission du signal « relations_termination » qui marque la fin de toutes les relations ainsi que du signal « termination ». Nous pouvons donc conclure que le comportement du scénario est bien celui attendu.

Figure 25: état final de la simiulation de bankAttack

3.2.2. Simulation de l'arrêt de reconnaissance du scénario

Cette fois-ci, nous allons simuler la relation de l'événement interdit « in_branch_p during in_safe_p2 ». Nous allons donc maintenir le signal « in_safe_p2 » ensuite, à l'instant d'après mettre le signal « in_branch_p3 ». Finalement, un instant après, nous allons enlever le signal « in_safe_p2 » pour terminer l'évènement et la relation entre ces deux évènements. Nous pouvons voir dans la figure 26, que les signaux « failure_forbidden_1 » et « failure » sont émis, ainsi que les signaux signalant la fin du quatrième évènement et la relation.

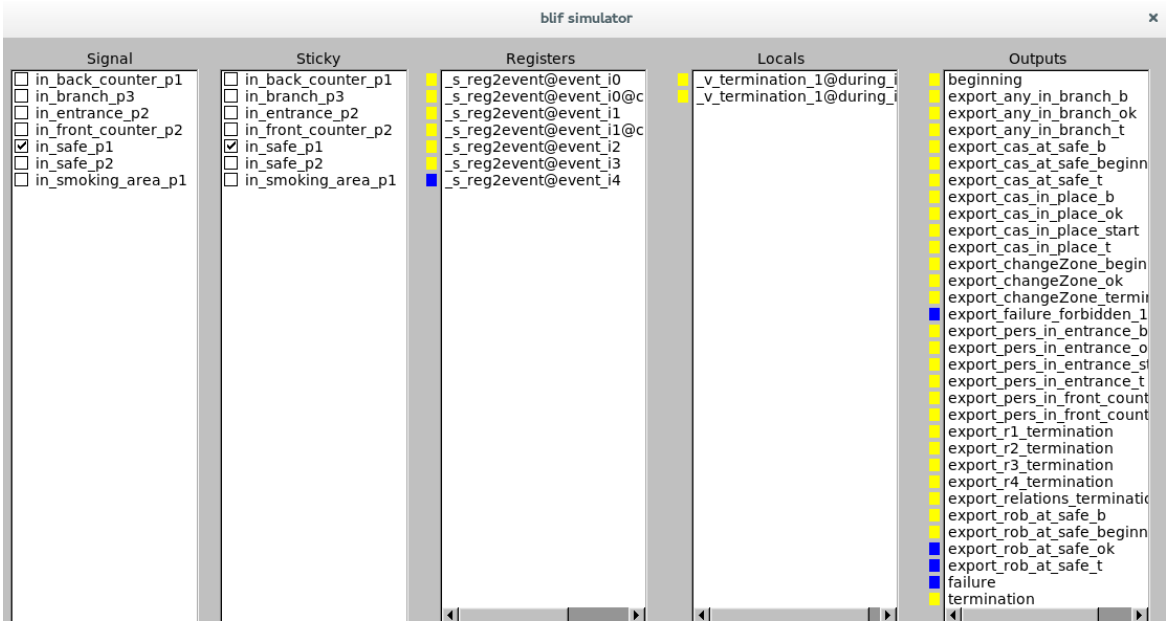

Figure 26: état forbidden_failure - arrêt du scénario bankAttack

Nous avons essayé de continuer la simulation après cet instant, et aucun signal n'est émis car en effet, le scénario s'est arrêté.

3.2.3. Simulation de non-respect des contraintes

Finalement, nous allons vérifier la partie «**constraint** », par exemple, dans le scénario changeZone : delay(termination of pers_in_entrance, beginning of pers_in_front_counter) < 5 . Cela devra afficher un signal de failure mais on ne souhaite pas arrêter le scénario dans ce cas.

Nous allons simplement simuler le scénario changeZone puis nous allons déclencher l'évènement « in_entrance_p » puis le terminer. Ensuite, nous allons dépasser le délai, c'est-à-dire dépasser 5 instants logiques sans commencer l'évènement « in_pfront_counter_p ». Nous pouvons voir en effet l'émission du signal « failure » (voir figure 27). Cela dit, nous pouvons maintenant émettre « in_front_counter_p » et continuer le scénario sans problème.

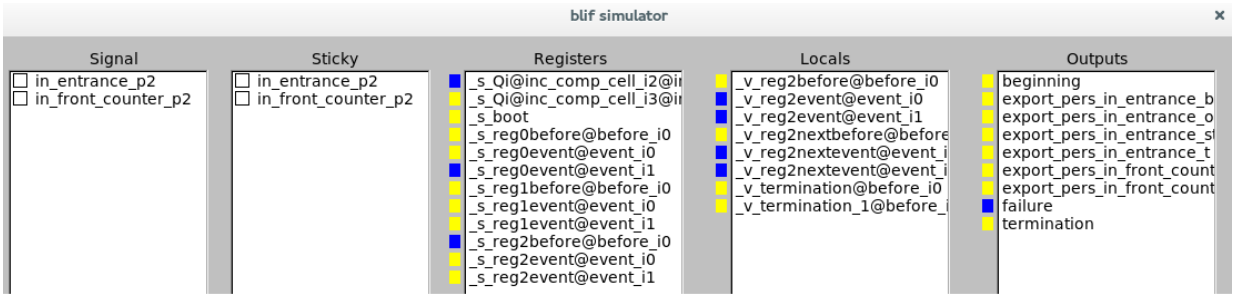

Figure 27: état failure dans la simulation de changeZone

3.3 Vérification formelle avec XEVE

Xeve est un outil de model-checking utilisé pour la vérification des automates à état finis en Esterel qui a été développé par l'INRIA en collaboration avec l'Ecole des Mines/CMA. Cet outil peut être utilisé pour la vérification de scénario LE aussi car son format d'entrée est le fichier blif. Nous avons déjà généré le fichier «bankAttack.blif » de scénario bankAttack lors de la simulation avec « blif_simul ».

Cet outil liste les signaux d'entrée et de sortie du scénario. Il nous permet de choisir si on souhaite donner une valeur (vrai, faux, indéfini) aux signaux d'entrée. De plus, nous pouvons choisir de vérifier si un signal de sortie a été émis (respectivement non-émis), cela colorie le signal choisi en rouge (respectivement bleu). Nous pouvons également choisir de ne pas vérifier le signal (en noir). Ensuite, l'outil nous affiche le résultat et génère un fichier de format « .esi » qui nous illustre la possibilité (s'il y'en a une) d'émission.

Nous souhaitons alors vérifier la possibilité d'émission de signal « relations_termination » (voir figure 28).

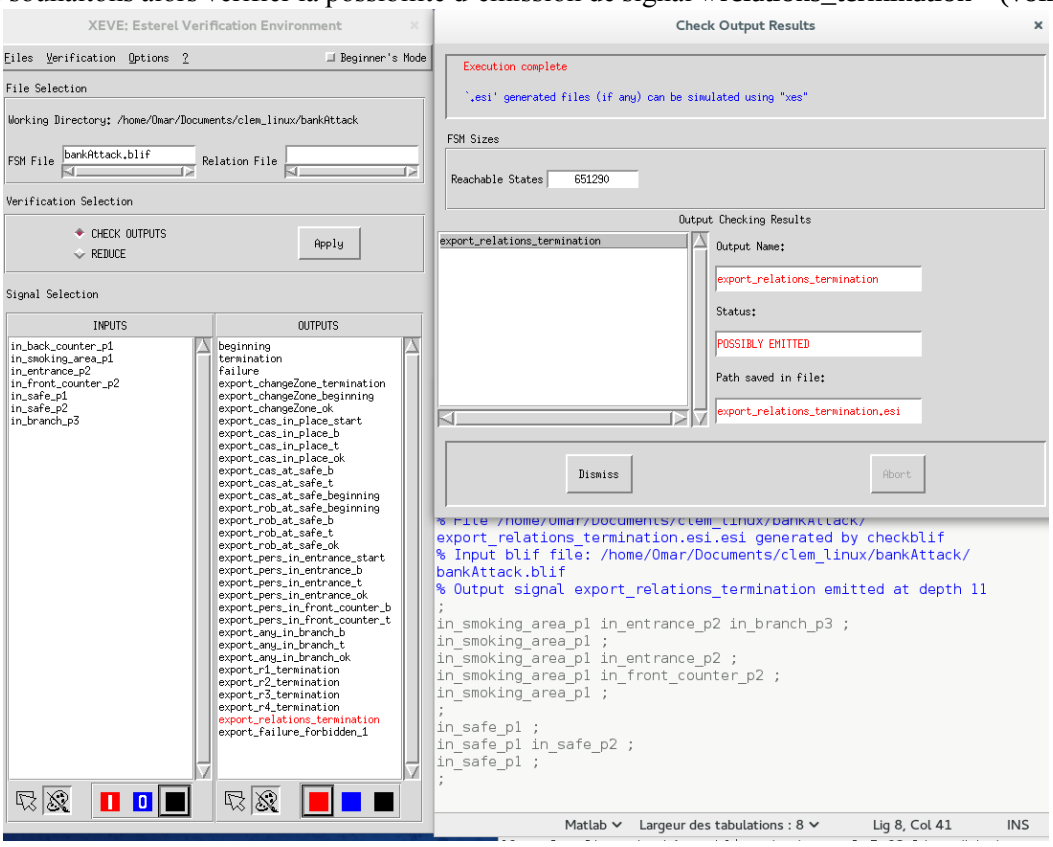

Figure 28: Vérification d'émission du signal « relations_termination »

Effectivement, nous pouvons voir que dans les résultats que le signal est bien émis « Possibly_Emitted ». Ainsi, dans le fichier généré, nous avons l'enchainement des évènements qui aboutit à son émission. Il faut préciser que « ; » signifie qu'on passe à l'instant logique suivant. En effet, cette séquence décrite dans le fichier généré est le déroulement du scénario avec les quatre évènements non-interdits dans le bon ordre, en respectant les relations entre eux.

Ensuite, nous souhaitons vérifier l'émission du signal « failure_forbidden » (voir figure 29).

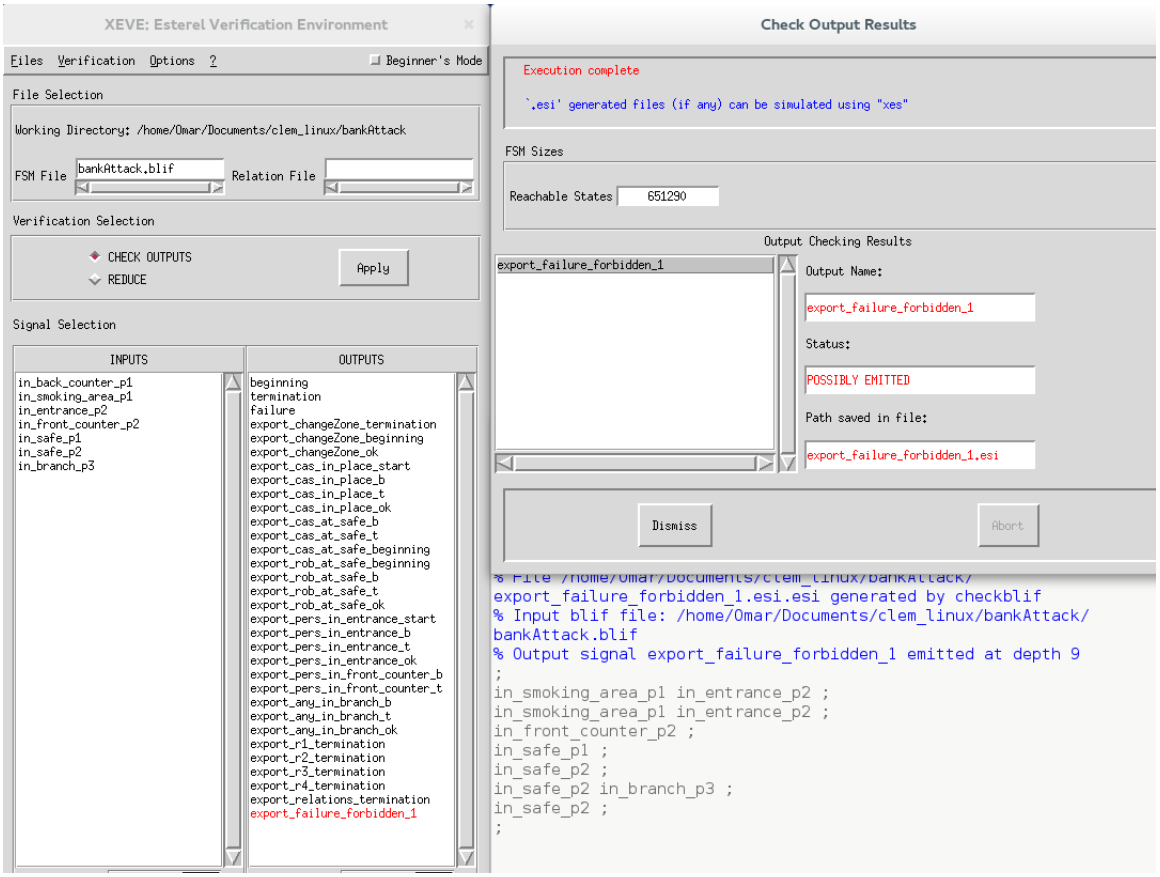

Figure 29: Vérification d'émission du signal « failure_forbidden»

En effet, nous pouvons voir que dans les résultats que le signal est bien émis « Possibly_Emitted ». Ainsi, dans le fichier généré, nous avons l'enchainement des évènements qui aboutit à son émission. La séquence décrite dans le fichier généré est le déroulement de l'évènement interdit « in_branch_p3» et le quatrième évènement « in_safe_p2 » en respectant la relation entre eux. Cela est bien le comportement attendu du scénario.

Finalement, nous souhaitons vérifier l'émission de « failure » lorsqu'on dépasse le délai imposé. Nous avons bien l'émission du signal failure « possibly_Emitted », et puis l'enchainement décrit l'émission de l'évènement « in_entrance_p » et sa terminaison sans émettre « in_front_counter_p » avant le fin du délai maximum.

Nous avons remarqué que dans le fichier généré, la séquence possible pour émettre ce signal failure ne convient pas au comportement de scénario. Nous avons donc du remodifier la traduction de la partie « **Constraint** ».

Ensuite, nous avons refait la vérification avec Xeve (voir figure 30).

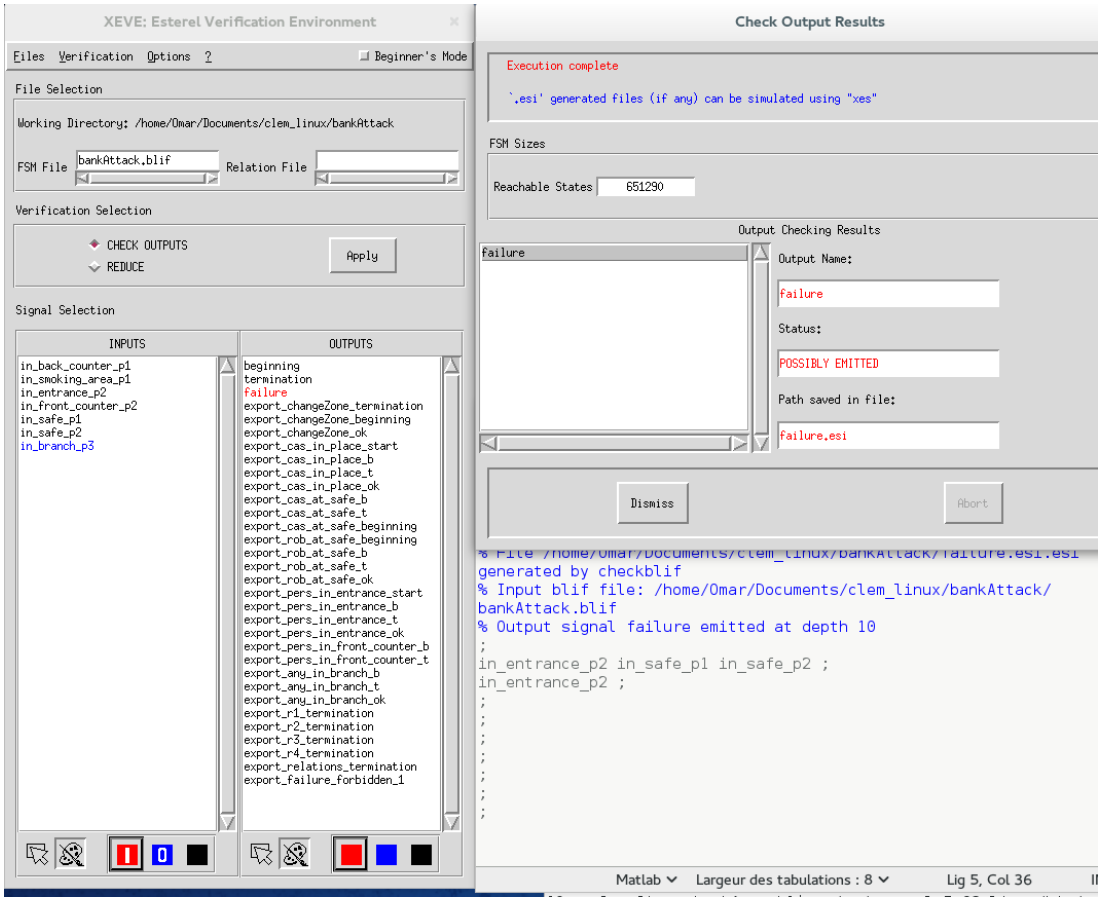

Figure 30: Vérification d'émission du signal « failure»

Effectivement, cette séquence correspond bien au comportement attendu du scénario. Nous avons l'émission de failure, lorsque l'évènement « in_entrance_p2 » est terminé et que le signal « in_front_counter_p2 » n'est toujours pas arrivé pendant la période de délai maximum.

4. Proposition de modification de cette solution :

Nous avons remarqué que cette solution proposée peut encore être améliorée. Nous avons effectué beaucoup d'appels à l'opérateur « run » et tout mis en parallèle grâce à l'opérateur « || ». Cela a entrainé une baisse dans la performance de la compilation de module LE. C'est-à-dire, CLEM prend beaucoup de temps pour compiler et simuler le modèle proposé.

Nous pouvons donc envisager d'implémenter directement dans LE les opérateurs « before », « during », et l'automate « event » pour augmenter la performance.

Finalement, il serait intéressant de modifier le langage de scénario SAM-L pour tenir compte de cette solution proposée. Par exemple, nous avons vu que les scénarios primitifs n'ont pas d'intérêt lors de passage en LE. Il serait donc plus judicieux d'éliminer ces scénarios primitifs et faire appel directement aux évènements dans les scénarios composites non-primitifs. Nous appellerons ces évènements par notre automate (ou opérateur) event en LE.

IV. Conclusion

Ce travail a bien eu lieu durant les cinq mois de stage lors de deuxième semestre de Master 2 Informatique. Il s'est déroulé à l'INRIA de Sophia Antipolis au sein de l'équipe STARS. L'objectif était d'exprimer la sémantique de SAM-L en LE. De cette manière, les moteurs de reconnaissance seront une implémentation directe de l'automate synchrone modèle du programme LE associé au scénario. En effet, ce modèle est bien un automate de Mealy exprimé en LE qui va calculer deux signaux « beginning » et « termination ». Ces signaux serviront de points de repère, pour le début et la fin de scénario.

Nous avons vu, à travers ce rapport, la démarche suivie pour atteindre cet objectif. Tout d'abords, l'étude de langage synchrone LE était nécessaire puis ensuite le langage de scénario SAM-L. Ensuite, nous avons introduit la boîte à outils utilisé CLEM et ses fonctionnalités. Finalement nous avons pu vérifier notre solution grâce au simulateur intégé dans CLEM « blif_simul ». Effectivement, le comportement du scénario est bien celui attendu.

Sur le plan technique et personnel, ce stage m'a aidé à acquérir des compétences dans le domaine de la recherche en informatique. De plus, il m'a permis à mieux comprendre la programmation synchrone et son intérêt dans le domaine de reconnaissance d'activité et surveillance. Finalement, cette expérience professionnelle m'a fait découvrir aussi l'intérêt du travail de chaque personne, au sein de l'équipe, qui porte à l'avancement de la recherche globale.

V. Références

[1] Présentation de l'équipe STARS, http://www.inria.fr/equipes/stars

[2] N. Halbwachs, « Synchronous Programming of Reactive Systems, a tutorial and commented bibliography », Tenth International Conference on Computer-Aided Verification, CAV'98 Vancouver (B.C.), LNCS 1427, Springer Verlag, June 1998.

[3] C. André, « Systèmes Réactifs et Programmation Synchrone », Cours DEA/ESSI3/DESS à l'Université de Nice Sophia Antipolis, Octobre 1998.

[4] Schéma du système réactif, http://www.rennes.supelec.fr/ren/perso/pchlique/tpsreel/sysreac.htm

[5] P. Raymond, « Introduction à l'approche synchrone », Verimag, 2012/2013.

[6] C. André, « Comparaison des styles de programmation de langages synchrones », Rapport de Recherche RR2005-13, ISRN I3S/RR-2005-13-FR, I3S, Sophia Antipolis, Juin 2005.

[7] Esterel Web, http://www-sop.inria.fr/esterel-org/files/

[8] Introduction sur SAM et exemple de SAM-L, http://www-sop.inria.fr/members/Annie.Ressouche/sam.html

[9] D. Gaffé, A. Ressouche. « Compilation modulaire d'un langage synchrone », Technique et Science Informatiques (TSI), no. spécial "Méthodes Formelles", vol.30, p.441-471, Edition Hermes Science, 2011.

VI. Annexe (module bankAttack)

module bankAttack:
Input: in_back_counter_p1, in_smoking_area_p1, in_entrance_p2, in_front_counter_p2, in_safe_p1, in_safe_p2, in_branch_p3;

Output: beginning, termination, failure,

Run:
".": during: during;
".": before: before;
".": changeZone: changeZone;
".": inc4_cmp_cte8: inc4_cmp_cte8;
".": event: event;

local
cas_at_safe_b, cas_at_safe_t, cas_at_safe_beginning,
cas_in_place_start, cas_in_place_b, cas_in_place_t, cas_in_place_ok,
rob_at_safe_beginning, rob_at_safe_b, rob_at_safe_t, rob_at_safe_ok,
any_in_branch_b, any_in_branch_t, any_in_branch_ok,
changeZone_beginning, changeZone_termination, changeZone_ok,
SUP, INF8, failure_forbidden_1, relations_termination,
r4_beginning, r1_termination, r2_termination, r3_termination, r4_termination

```
{
;;e1 - in_back_counter_p1 or in_smoking_area_p1 = cas_in_place
;;e2 - module changeZone (in_entrance_p2 or in_front_counter_p2)
;;e3 - in_safe_p1
;;e4 - in_safe_p2
;;e5 - forbidden event in_branch_p3
weak abort
{
      wait in_back_counter_p1 or in_smoking_area_p1
   >> emit cas_in_place_start
   >> present in_back_counter_p1
       { run event [in_back_counter_p1\evt, cas_in_place_b\e_beginning, cas_in_place_t\e_termination]
}
        else
            run event [in_smoking_area_p1\evt, cas_in_place_b\e_beginning,
cas_in_place_t\e_termination]

   ||
     run changeZone[changeZone_beginning\beginning, changeZone_termination\termination]
   ||
     run event [in_safe_p1\evt, cas_at_safe_b\e_beginning, cas_at_safe_t\e_termination]
   ||
     run event [in_safe_p2\evt, rob_at_safe_b\e_beginning, rob_at_safe_t\e_termination]
   ||
     run event [in_branch_p3\evt, any_in_branch_b\e_beginning, any_in_branch_t\e_termination]
   ||
```

```
   ;; R1: e2 during e1 - rob_enters DURING cas_in_place
   run during[cas_in_place_start\start, cas_in_place_b\b2,  changeZone_beginning\b1,
changeZone_termination\t1, changeZone_ok\ok, cas_in_place_t\t2,
r1_termination\termination]
   ||
   ;; R2: e1 before e3 -  cas_in_place BEFORE cas_at_safe
   run before[cas_in_place_start\start, cas_in_place_b\b1, cas_in_place_t\t1, cas_in_place_ok\ok,
cas_at_safe_b\b2, cas_at_safe_t\t2, r2_termination\termination]
   ||
   ;; R3: e2 before e3 -  rob_enters BEFORE cas_at_safe
   run before[cas_in_place_start\start, changeZone_beginning\b1, changeZone_termination\t1,
changeZone_ok\ok, cas_at_safe_b\b2, cas_at_safe_t\t2, r3_termination\termination]
   ||
   ;; R4: e4 during e3 -  rob_at_safe DURING cas_at_safe;
   run during[cas_in_place_ok\start, cas_at_safe_b\b2, cas_at_safe_beginning\beginning,
rob_at_safe_b\b1, rob_at_safe_t\t1, rob_at_safe_ok\ok, cas_at_safe_t\t2,
r4_termination\termination]
   ||
   ;; delay (beginning of cas_at_safe, beginning of rob_at_safe) < 8
weak abort{
      weak abort{
          wait cas_at_safe_b
              >>   run inc4_cmp_cte8 [INF8\INF]
            } when INF8 >> emit failure
          }when rob_at_safe_b
   ||
   { wait r1_termination || wait r2_termination || wait r3_termination || wait r4_termination }
        >> emit termination
    >> emit relations_termination
   ||
   ;; e5 Forbidden de SAM-L // R5: e5 during e4
    weak abort{
            run during[cas_at_safe_beginning\start, rob_at_safe_b\b2,
rob_at_safe_beginning\beginning, any_in_branch_b\b1, any_in_branch_t\t1,
                    any_in_branch_ok\ok, rob_at_safe_t\t2, failure_forbidden_1\termination]
        } when relations_termination

   }when failure_forbidden_1
    >> emit failure
}
end
```