CentraleSupélec

Time Constraints of TCAS II
Rapport EI - Groupe 1

Maud Valentin, Juliette Kalflèche, Elisa Faure, Mélanie Augier, Tom Bray, Florentin Labelle et Marc-Antoine Godde

Table des matières

1 Enjeux

10 Septembre 1976, 10 :14 :41 UTC, le vol British Airways Flight 476 et le vol Inex-Adria Flight 550 entrent en collision au dessus de Zagreb, la capitale de la Croatie. Les flux constituent les éléments importants et marquants de la mondialisation. La quantité de flux de marchandises et de personnes s'est considérablement accrue ces dernières années et le transport aérien a joué un rôle crucial dans cette mutation. Ainsi, puisque la mondialisation a pris appui sur le trafic aérien, la nécessité de sécurité est apparue.

Le Traffic Collision Avoidance System (TCAS) s'est aujourd'hui imposé comme étant la solution permettant de répondre à cette nécessité. Cet enseignement d'intégration permet d'étudier l'élaboration et le fonctionnement d'un TCAS d'un point de vue structurel. Au travers de notre travail, nous nous sommes chargé de rendre compte à l'aide d'un modèle des différentes propriétés qu'un TCAS doit satisfaire.

2 Task 1

On cherche à modéliser le comportement de deux avions dans le ciel lorsqu'ils s'approchent l'un de l'autre. Pour modéliser un avion, il convient d'abord de réfléchir aux entrées et sorties des différents composants ainsi que leurs états.

2.1 Description des composants

Les différents composants sont les suivants :

- **Aircraft** : qui fournit les information au sensor.
  - **Input** : une manœuvre du pilote
  - **Output** : l'état envoyé au capteur

- **Sensor** : qui récupère les informations des deux avions et les transmet au threat detection.
  - **Input** : un état envoyé par l'avion
  - **Output** : la position des deux avions

- **Threat Detection** : qui à partir des données du capteur indique si l'avion est en danger.
  - **Input** : la position des deux avions
  - **Output** : annonce le danger au module de résolution

- **Threat Resolution** : qui calcule une solution à la situation de danger et la compare à la solution de l'autre avion.
  - **Input** : l'annonce du danger, un message depuis l'autre avion depuis le canal de communication
  - **Output** : envoie sa solution au canal de communication, envoie le conseil de manœuvre au pilote

- **Pilot** : qui suit les conseils de threat resolution pour sortir l'avion de la zone de danger.
  - **Input** : le conseil de manœuvre
  - **Output** : l'exécution de manœuvre envoyée à l'avion

- **Communication** : qui permet aux deux avions de s'envoyer leur solution à la situation de danger.
  - **Input** : la solution de l'avion
  - **Output** : envoie cette solution à l'autre avion

2.2 Schéma des composants

Voici les automates sans les conditions de garde pour schématiser toutes ces relations entre les états.

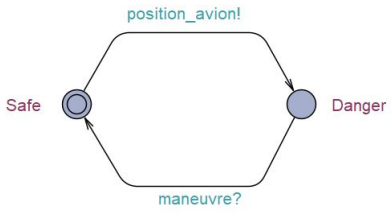

FIGURE 1 – Aircraft model

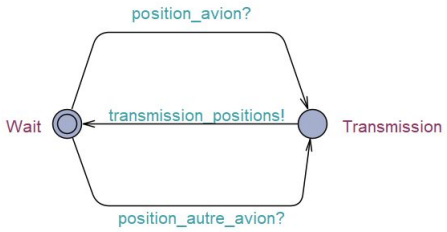

FIGURE 2 – Sensor model

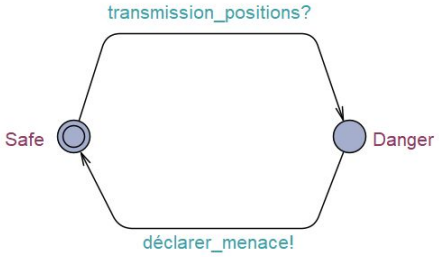

FIGURE 3 – Detection model

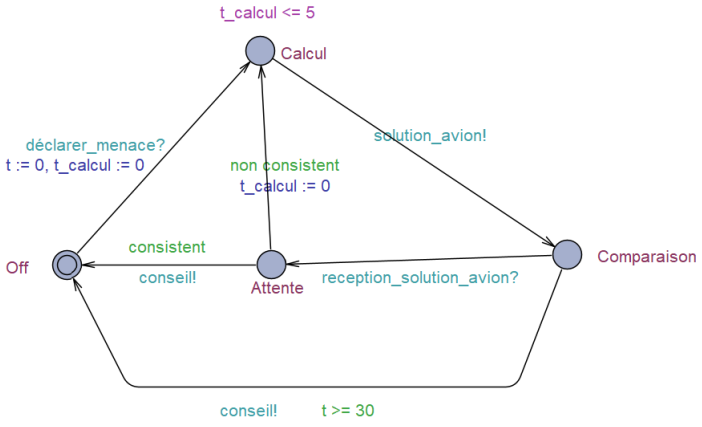

FIGURE 4 – Resolution model

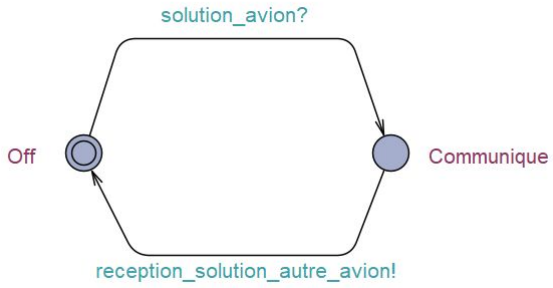

FIGURE 5 – Communication model

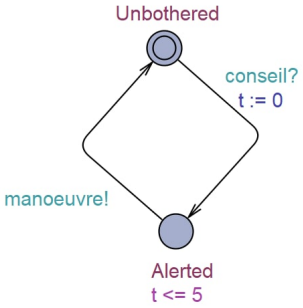

FIGURE 6 – Pilot model

3 Task 2

Ce modèle proche de la réalité ne suffit pas à modéliser logiquement le système. Nous avons donc corrigé certaines relations entre états et certaines relations entre composant, aini qu'ajouté certaines gardes pour garantir les propriétés du système.

3.1 Aircraft model

Dans notre modèle, l'avion se contente d'envoyer sa position au capteur. Il a deux états : Safe et Danger.

Input : manever (depuis le pilote)

Output : danger (vers le capteur)

— Il passe de l'état Safe à Danger en envoyant danger[id] ce qui reset aussi l'horloge t.

— Une fois dans l'état Danger il peut rester dans cet état si t>1, ce qui envoie danger[id] et reset t. Cette horloge permet d'assurer que les transmission ne soient pas instantanées pour éviter les situations Zeno. Il retourne dans l'état Safe lorsqu'il reçoit manever[id] du pilote.

3.2 Sensor model

Dans notre modèle le capteur a deux état possible : un état d'attente et un état de transmission.

Input : danger (depuis l'avion)

Output : emission (vers le détecteur)

— Il est par défaut dans l'état d'attente et lorsque l'avion envoie danger[id] ou le voisin envoie danger[1-id], il passe dans l'état de transmission qui est un état urgent.

— Il quitte cet état en envoyant emission[id] qui va activer le détecteur de danger.

3.3 Detection model

Le module de détection a deux état : On et Off.

Input : emission (depuis le capteur)

Output : detected (vers le module de résolution)

— Il est par défaut dans l'état Off et passe dans l'état On lorsqu'il reçoit emission[id] du capteur.

— L'état On est un état urgent qu'il quitte en envoyant detected[id].

3.4 Resolution model

Le module de résolution est la partie la plus complexe de notre système et a quatre états : Off, Calcul, Wait et Comparaison.

Input : detected (depuis le détecteur), reception (depuis le canal de communication)

Output : solution (vers le canal de communication), advice (vers le pilote)

— L'état initial est l'état Off qu'il quitte pour passer dans Calcul à la réception de detected[id] et qui reset alors les clock t et c.

— L'état Calculation a des gardes $c \leq 5$ *et* $t \leq 30$ car les calculs se font en moins de 5 secondes et le module dans une solution en maximum 30 secondes. Le module qui l'état Calcul en envoyant solution[id] et passe dans l'état Wait.

— L'état Wait n'a pas de garde. Il passe dans l'état Comparaison en recevant reception[id] du module de communication

— L'état Compare n'a pas de garde non plus. Il peut repasser spontanément avec une probabilité d'environ 70 % dans Resolve en envoyant advice[id], il revient aussi dans cette état si $t \geq 30$. Le module peut sinon retourner dans l'état calcul avec comme garde $t < 25$ ce qui reset la clock c.

— L'état Resolve ne peut passer que dans l'état off en envoyant solution[id] et n'a as de garde.

3.5 Pilot model

Le pilote a deux états : Unbothered et Alerted.

Input : advice (depuis le module de résolution)

Output : manever (vers l'avion)

— Il est par défaut dans l'état Unbothered et passe dans l'état Alerted en recevant advice[id] du module de résolution. Cela reset l'horloge t.

— Il peut alors quitter l'état Alerted, qui a pour garde $t \leq 5$ car le pilote doit agir en moins de 5 seconde, en envoyant manever.

3.6 Communication model

Le modèle de communication a deux états : Off et Communique.

Input : solution (depuis le canal de résolution)

Output : reception (vers le canal de résolution)

— Il est par défaut dans l'état Off et l'état Communique est un état urgent.

— Il quitte l'état Urgent pour aller dans Communique en recevant solution[id] et y retourne en envoyant reception[1-id].

3.7 The automata with UPPAAL

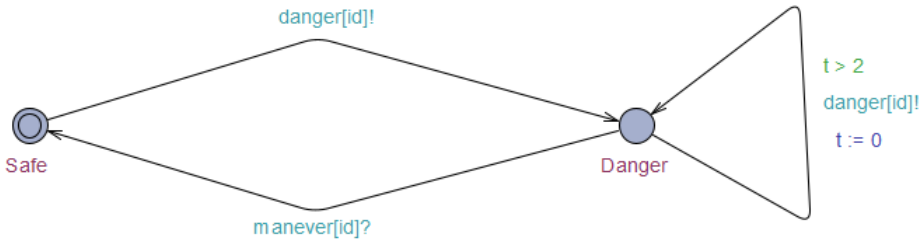

FIGURE 7 – Aircraft model automate

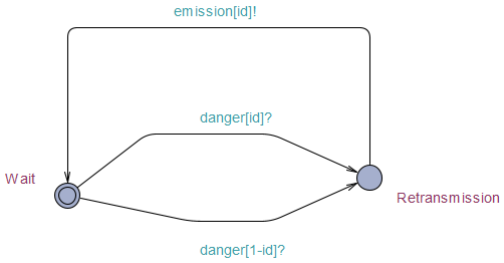

FIGURE 8 – Sensor model automate

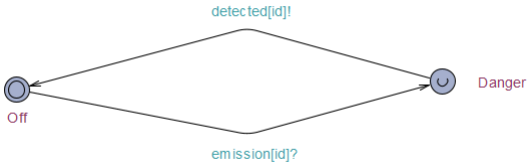

FIGURE 9 – Detection model automate

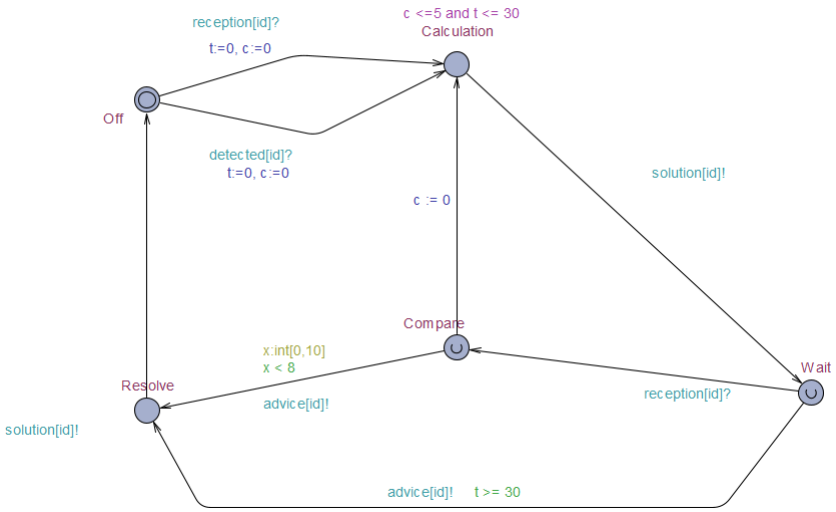

FIGURE 10 – Resolution model automate

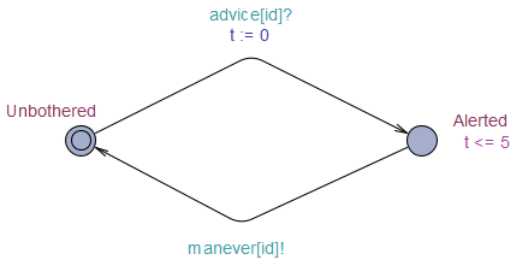

FIGURE 11 – Pilot model automate

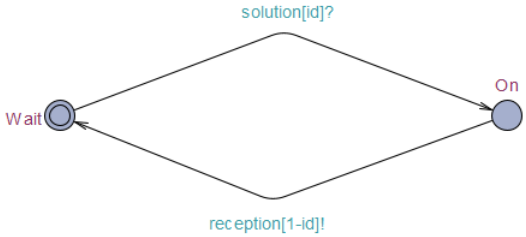

FIGURE 12 – Communication model automate

4 Task 3

4.1 Simplification du précédent modèle

Le premier modèle ne permettait pas de vérifier toutes les propriétés (il y avait par exemple de nombreux deadlocks). La propriété A[] not deadlock n'était en effet pas vérifiée. Nous avons eu du mal à modifier du système afin de la vérifier, à cause des nombreux composants et du fait que le danger n'était pas synchronisé sur les deux avions.

Nous avons donc décidé de changer de modèle en supprimant le composant *sensor* car nous jugions qu'il n'apportait pas quelque chose au modèle et de modifier le modèle *aircraft* en mettant plutôt les deux avions sur le même composant.

4.2 Nouveau système

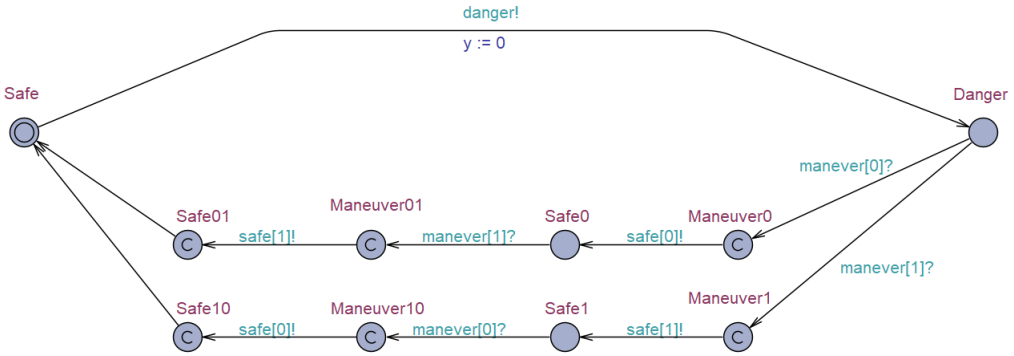

FIGURE 13 – Aircraft model automate

Une horloge globale y a été ajoutée pour vérifier la condition 3 de durée de détection du danger. Le channel danger est en broadcast afin de synchroniser le danger et de ne pas avoir les problèmes de deadlock rencontrés sur notre modèle précédent. Les état intermédiaires pour les *SafeXY* permettent de définir l'état dans lequel chaque avion est safe (deux manières possibles, soit il est le premier à avoir reçu la manoeuvre de sauvetage, soit le second) et d'arrêter l'état de danger avec la channel safe sur le détecteur pour qu'il puisse bien relancer une alerte au prochain danger.

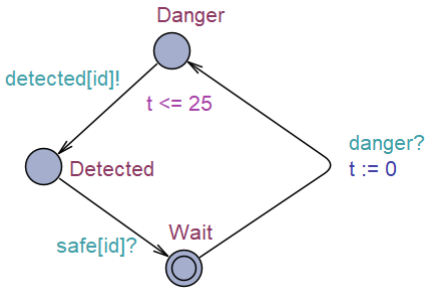

FIGURE 14 – Detection model automate

Nous avons rajouté une horloge locale au détecteur afin de pouvoir vérifier la propriété de durée sur le danger et la channel safe comme expliqué précédemment.

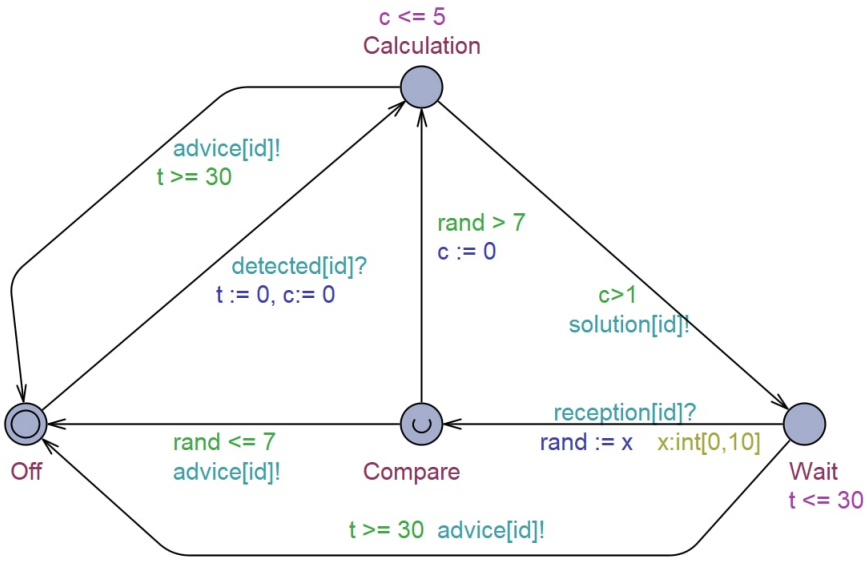

FIGURE 15 – Resolution model automate

Nous avons ajouté un nouveau retour à l'état off depuis Calculation quand la durée de 30 secondes est dépassée et nous avons mis une condition sur le temps de calcul devant être supérieure à 1 (de manière arbitraire) afin de faire avancer le temps.

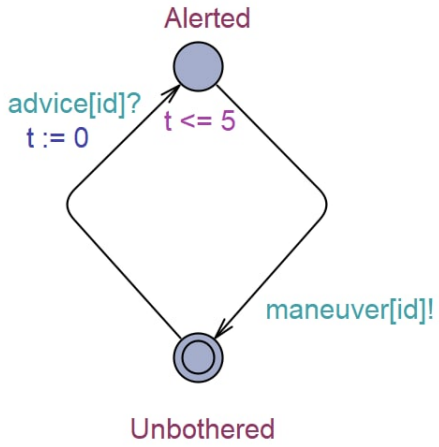

FIGURE 16 – Pilot model automate

Rien n'a été changé sur ce composant.

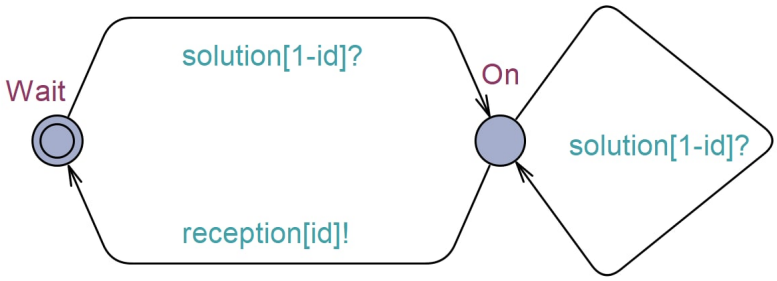

FIGURE 17 – Communication model automate

On a fait une transition de l'état on à lui-même afin que qu'il soit toujours en écoute de l'autre avion, résolvant d'ailleurs ainsi des problèmes de deadlock.

4.3 Vérification des propriétés

1. *Any aircraft is possible in a safe zone.*
   — E<> Sky.Safe1 or Sky.Safe01
   — E<> Sky.Safe0 or Sky.Safe10

2. *Whenever the system enters the dangerous state after thread detection, the corresponding pilot will always implement the resolution advisory to resolve this.*
   — Detector(0).Danger –> Sky.Manoeuvre1 or Sky.Manoeuvre2
   — Detector(1).Danger –> Sky.Manoeuvre1 or Sky.Manoeuvre2

3. *The intruder (the neighboring aircraft enters RA zone) should be detected within 60 seconds.*
   — Sky.Danger –> Pilot(0).Alerted and y<=60
   — Sky.Danger –> Pilot(1).Alerted and y<=60

4. *Any aircraft enters a dangerous zone will always return back to safe zone.*
   — Detector(0).Danger –> (Sky.Safe0 or Sky.Safe10)
   — Detector(1).Danger –> (Sky.Safe1 or Sky.Safe01)

5. *Whenever one aircraft enters a dangerous zone, then the other aircraft will always go into the safe zone.*
   — Detector(0).Danger –> (Sky.Safe1 or Sky.Safe01)
   — Detector(1).Danger –> (Sky.Safe0 or Sky.Safe10)

6. *The system should be deadlock-free.*
   — A[] not deadlock

Le nouveau modèle vérifie toutes les propriétés demandées.

4.4 Extension

Nous sommes le groupe numéro 1 et l'extension que nous devons faire est donc de symétriser notre système.

Néanmoins, pendant notre semaine, nous avons rencontré des soucis avec notre premier modèle pour la vérification des deadlocks et nous avons décidé de synchroniser le danger pour nos deux avions avant de résoudre notre problème, notamment par l'usage d'une channel broadcast pour déclarer le danger.

Nous avons donc fini par symétriser notre système sans que ce soit notre intention initiale.

Ainsi, le modèle présenté à la partie précédente est symétrique.

5 Zeno detection checker

Une situation Zeno arrive lorsqu'un nombre infini de transition arrive dans un temps fini. Ainsi, un comportement Zeno peut apparaître quand le modèle possède une boucle qui peut être parcouru instantanément.

Ainsi, la première étape de détection de Zenoness consiste à repérer les différentes boucles d'exécution. Les horloges présentent sur ces boucles doivent alors vérifier certaines propriétés :

— Être remise à zéro sur au moins une des transitions de cette boucle
— Il existe une garde concernant cette horloge sur une des transitions

Pour étudier cette propriété sur notre modèle, nous allons découper le travail en les différentes tâches suivantes :

1. Parser notre fichier XML grâce à un module Python pour obtenir des listes et des dictionnaires que nous pouvons parcourir grâce à un algorithme
2. A partir de ces dictionnaires et listes, définir toutes les boucles locales, c'est à dire à l'intérieur d'un même composant
3. Regrouper ensuite les boucles de manière globale
4. Vérifier ensuite la propriété de *strongly not-zenoness*

5.1 Parsing

Pour réaliser le parsing, on utilise le module de la librarie standard Python `xml.etree.ElementTree` afin d'en extraire une structure de donnée avec laquelle on peut plus facilement travailler en Python. Plusieurs suppositions sur le modèle ont été prise pour simplifier cette étape. On a en effet considéré que les horloges ne sont pas modifiées dans les fonctions et que les gardes sur les horloges sont nécessairement de la forme $x < / \leq / \geq / > 10$.

5.2 Détection de boucles

Le but est de détecter toutes les boucles du systèmes pour ensuite vérifier la non-zenoness. Pour cela, on procède automate par automate. Pour chaque automate, on part de l'état initial avec toutes les clocks à zéro. On parcourt toutes les transitions possibles depuis cet état (c'est-à-dire dont les clocks vérifient les conditions de garde de la transition et d'invariant de l'état suivant). Si nécessaire les clocks sont mises à jour pour les vérifier.

Ensuite, à chaque état ajouté on regarde si une boucle est formée, sinon on continue d'ajouter des états jusqu'à obtenir une boucle.

5.3 Regroupement des boucles

Le but va être de regrouper les boucles qui ont les mêmes "synchronized actions". On prend en entrée la liste des boucles qui ont été détectées dans la fonction précédente. On crée ensuite une nouvelle liste dans laquelle on mettra les indices des couples de boucle. On va parcourir la liste initiale deux fois pour comparer toutes les boucles entre elles en comparant à chaque fois les "synchronized actions". Si deux boucles ont les mêmes on ajoute à la liste qu'on doit renvoyer (indice-boucle-1, indice-boucle-2).

Quand on sort de ces boucles for imbriquées, on renvoie la liste des couples d'indice que la fonction suivante va utiliser. On a pris en compte ici que les boucles ayant un couple, les boucles seules sont prises en compte directement dans la fonction suivante.

5.4 Vérification de la strong non-Zenoness

Il faut alors ensuite vérifier pour chaque boucle seule qu'elle vérifie les conditions de non-Zenoness. Pour chaque horloge :

— Être remise à zéro sur au moins une des transitions de cette boucle
— Il existe une garde concernant cette horloge sur une des transitions qui garantie un temps d'exécution minimum

Pour les couples de boucles regroupés par synchronisation à l'étape précédente, il suffit de vérifier qu'une des deux vérifie les conditions de non-Zenoness.

5.5 Test de l'algorithme

N'ayant pas eu le temps de réaliser la partie "Détection de boucle", nous n'avons pas pu tester l'intégralité de notre algorithme sur les deux modèles fournis.

6 Organisation du travail

6.1 Organisation générale

1. Tâche 1
   — Réflexion en groupe pour s'approprier le sujet
2. Tâche 2
   — Un groupe sur la modélisation sur UPPAAL
   — Début de rédaction du rapport
3. Tâche 3
   — Un groupe sur la modélisation sur UPPAAL
   — Rédaction du rapport
4. Extension
   — Définition des fonctions python à coder
   — Répartition des 4 fonctions
   — Rédaction du rapport et de la présentation

6.2 Répartition des tâches

— Maud VALENTIN : Modélisaton, écriture du rapport et présentation
— Juliette KALFLECHE : Travail sur le programme de non zenoness, écriture du rapport
— Mélanie AUGIER : Modélisation sur UPPAAL
— Tom BRAY : Modélisation, programme de non zenoness
— Florentin LABELLE : Modélisation sur UPPAAL et programme de non zenosness
— Marc-Antoine GODDE : Ecriture du rapport et présentation
— Elisa FAURE :Modélisation sur UPPAAL et programme de non zenosness

6.3 Avis sur UPPAAL

1. **Prise en main facile** UPPAAL est un logiciel qu'aucun d'entre nous n'avait déjà vraiment utilisé, pourtant, cela ne nous a pas trop gêné pour réaliser le projet. L'interface est plutôt simple et intuitive, ce qui permet rapidement de pouvoir modéliser ce qu'on a en tête.
2. **Pratique pour modéliser des systèmes simples** Nous avons pu modéliser rapidement un premier modèle car les outils sont très simples, il est facile également d'éditer le modèle.
3. **Permet de comprendre les problèmes d'un système simple** Grâce à la trace, on peut suivre le chemin d'un contre-exemple d'une assertion, ce qui permet de comprendre la cause du problème. Quand le système se complexifie, il est néanmoins plus dur de s'y retrouver avec tous les états et

les synchronisations pour comprendre ce qui "bloque" (par exemple au niveau des deadlocks).

4. **Une option collaborative serait la bienvenue** Aujourd'hui, on peut travailler en groupe facilement sur Latex, Powerpoint afin de pouvoir voir les modifications des autres. Il y a également Gitlab afin d'écrire du code et de pouvoir le montrer facilement à son équipe. Ici, nous avons perdu du temps à harmoniser le nom des variables lorsque nous travaillions chacun sur un composant différent. Nous avons fini par nous mettre plusieurs derrière un même ordinateur, ce qui a réduit notre efficacité.
5. **Difficulté à installer le logiciel sur certains ordinateurs** Nous sommes 3 sur 7 membres à ne pas avoir pu ouvrir UPPAAL sur nos ordinateurs du fait d'incompatibilité de processeur, ce qui a réduit notre efficacité.

7 Conclusion

Pour conclure, cette semaine a été un très bon exercice pour appronfondir UPPAAL. Nous avons pu découvrir comment modéliser un système, comment faire abstraction afin de le simplifier ainsi que tout une démarche de vérification (qui a été aussi longue que la partie modélisation). Nous sommes plutôt fiers de ce que nous avons réalisé en une semaine, même si nous avons conscience de tout ce qu'il nous reste à apprendre et que nous pouvons encore aller plus loin, que ce soit en terme de complexité ou d'étude Zeno.