Rapport de projet

Formal Software Engineering

GOHIER Brian, FAGNIEZ Florian et RULLIER Noémie
28 novembre 2013

Table des matières

1 Introduction

L'objectif de ce projet fut de manipuler des spécifications formelles, prouver des propriétés, produire une implémentation et expérimenter. L'ensemble des expérimentations formelles a été conduit avec les réseaux de Petri temporels et paramétré à l'aide de l'outil Romeo. L'implémentation répartie sera programmée à l'aide de Java/RMI.

2 Spécification formelle

2.1 Question 1 : modéliser à l'aide d'un réseau de Pétri le comportement du réveil

Pour rappel voici les spécifications formelles du projet concernant le réveil :

Le réveil On considère la modélisation du comportement d'un réveil simplifié. Ce réveil possède un seul buton pour « l'armement » ou le « désarmement »/ Lorsqu'il est mis en position « armé », le réveil sonnera 8 heures plus tard, si il n'a pas été désarmé. Il est initialement désarmé et peut-être armé à tout moment.

Nous obtenons le réseau de Pétri suivant :

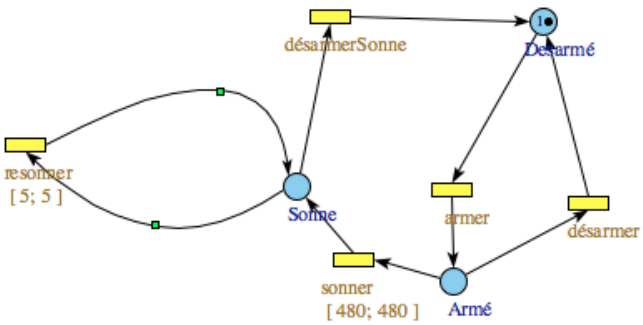


FIGURE 1 – Système réveil

2.2 Question 2 : modéliser à l'aide d'un réseau de Pétri le comportement de l'humain

Pour rappel voici les spécifications formelles du projet concernant l'humain :

L'humain On considère aussi un humain « utilisateur » de ce réveil. Dans cette fonction, il peut être « endormi » ou « réveillé ». Initiallement réveillé, il décide après avoir veillé entre **14 et 23 heures** d'« armer » le réveil et il s'endort. Il peut s'éveiller soit spontanément aprés avoir dormi entre **1 et 12 heures** soit en entendant la sonnerie. Dans cet état intermédiaire, il peut décider de désarmer le réveil immédiatement ou de s'endormir.

Nous obtenons le réseau de Pétri suivant :

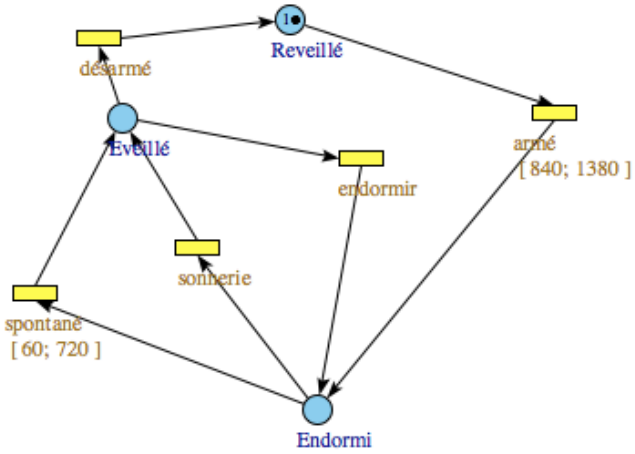


FIGURE 2 – Système

2.3 Question 3 : modéliser à l'aide d'un réseau de Pétri le comportement du système

Pour rappel voici les spécifications formelles du projet concernant le système :

Le système On considère que le réveil et l'humain sont situés dans des lieux différents éloignés et que l'intéraction s'effectue à travers l'échange de messages sur Internet. Les demandes d'armement et de désarmement, ainsi que la sonnerie sont véhiculés par messagerie. On fait l'hypothèse que les messages mettent moins de 10 minutes pour arriver à destination

Nous obtenons le réseau de Pétri suivant :

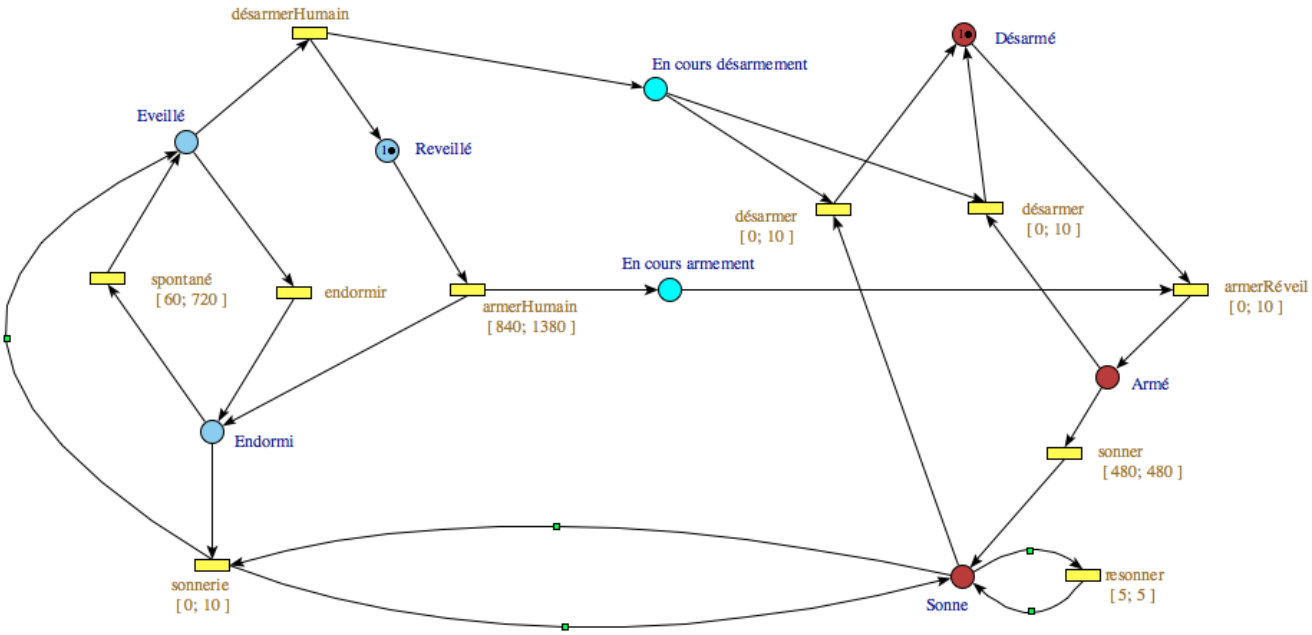


FIGURE 3 – Système complet

3 Analyse de l'espace d'état (non temporisée)

3.1 Question 4 : prouver que votre modèle est non-borné si c'est le cas

Au départ notre modèle était effectivement non-borné : lors de la génération du fichier d'espace d'état, nous obtenions un fichier d'une taille anormalement grande. En effet en regardant de plus près ce fichier, nous pouvions remarquer que nous avions un nombre élevé d'états dont certains contenaient une place qui pouvait contenir plusieurs jetons.

Pour résoudre cela, nous avons créé une place complémentaire qui force ces places à ne recevoir/-créer qu'un seul et même jeton. Si la place qu'on veut borner a un jeton, la place complémentaire n'a pas de jeton et inversement.

3.2 Question 5 : Construire et dessiner l'espace d'états du système

L'espace d'états du système à été crée sans contraintes temporelles. On peut remarquer que nous avons 8 états et 0 ou 1 seul jeton dans chaque place.

Ce modèle est ainsi borné, ce qui implique que notre modèle possédant les contraintes temporelles est également borné.

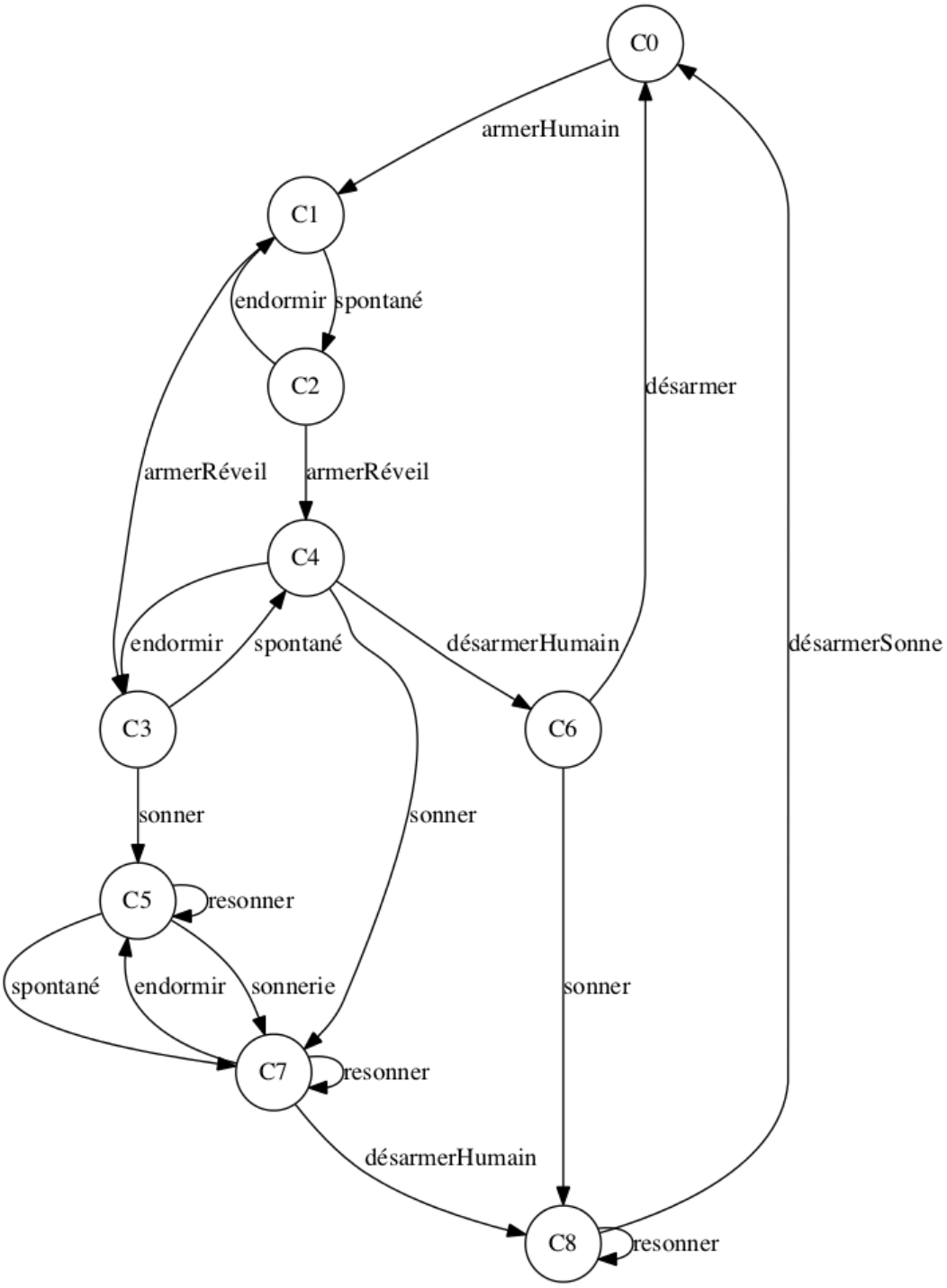


FIGURE 4 – Espace d'état du système sans contraintes de temps

Pour construire le graphe sans contraintes de temps, nous allons sur le menu Romeo, nous selectionnons PN puis Marking Graph.

Si on désire construire le graphe avec contraintes de temps, nous allons dans le menu Romeo, T-TPN puis state-class graph (cf.question 8 pour le graphe).

3.3 Question 6 : y-a-t-il des situations de blocages ? Peut-on toujours revenir à l'état initial ?

Pour savoir s'il existe une situation de blocage, nous utilisons la propriété :
$EF[0, inf]deadlock$ dans l'outil *CHECK* de Romeo. Lorsque nous vérifions la propriété, il nous est dit qu'aucune situation de blocage n'existe.

Afin de savoir si nous pouvions toujours revenir à l'état initial, nous allons définir deux configurations différentes :
- l'état initial (réveillé/désarmé)
- un état autre que l'état initial (Un état endormi/désarmé/en cours d'armement)

Nous en ressortons la formule suivante :
$AF[0, inf]((M(2) = 1 and M(4) = 1 and M(7) = 1) => (M(1) = 1 and M(4) = 1))$
Grâce à cette formule, nous vérifions qu'à partir de l'état suivant l'état initial, on peut toujours revenir à celui-ci.

3.4 Question 7 : est-on sûr que la sonnerie sur le réveil correspond bien à la dernière demande d'armement de l'humain ? Sinon, tracer un chronogramme mettant en évidence le problème

Afin d'évaluer ce problème, nous avons utilisé la formule suivante :
$AG[0, inf](M(6) = 1 => M(5) = 0)$,
On vérifie ainsi que lorsque le réveil est dans l'état sonne, il ne peut pas être en cours d'armement. Donc la sonnerie correspond bien à la dernière demande d'armement de l'humain.
La réponse sortie à la suite de cette propriété nous renvoie **vrai** dans notre modèle.

4 Analyse temporisée. Espace d'état symbolique

4.1 Question 8 : construire et dessiner le graphe symbolique des états. Constater que certains comportements ne sont plus possibles du fait des contraintes temporelles

Nous avons pu générer le .dot correspondant au graphe, cependant (étant donné la lourdeur du fichier) nous n'arrivons pas à l'ouvrir.

4.2 Question 9 : en utilisant la logique temporelle TPN-TCTL offerte par Romeo, essayer de trouver par essais successifs la durée possible pendant laquelle l'humain reste endormi

Nous nous sommes rendus compte que lorsque notre jeton passait dans l'état « Sonne » par la transition « Sonner », nous réinitilisions sans cesse la transition sonnerie et son cycle temporel.
Avec la formule $(M(2) = 1) - - > [1000, inf](M(3) = 1)$ qui permettait de savoir si l'humain est obligé d'être éveillé à partir de 1000 unités de temps. L'ordinateur nous retournait la valeur false. Notre modèle ne permettait donc pas de connaître la durée possible pendant laquelle l'humain reste endormi et nous allons donc calculer la durée au bout de laquelle il est obligé de s'éveiller.
$(M(2) = 1) - - > [0, 719](M(3) = 1)$: renvoie false $(M(2) = 1) - - > [0, 720](M(3) = 1)$: renvoie true.
Nous nous sommes rendu compte que pour que le résultat soit toujours **true**, la formule devait être de la forme : $(M(2) = 1) - - > [x, y(avec y = x + 720)](M(3) = 1)$ On remarque que cette période correspond à l'interval durant lequel l'humain doit se réveiller spontanément (ici 720 unités de temps max).

4.3 Question 10 : reprendre la question 7 à la lumière de la question 9

$(M(6) = 1) - - > [0, inf](M(5) = 0)$

Cette formule nous permet de nous assurer que lorsque le réveil sonne, il ne peut pas être en l'état « armé » quelque soit « l'état » de l'humain (à savoir endormi ou réveillé).

5 Analyse paramétrée

5.1 Question 11 : faire l'expérience consistant à calibrer les moments où le réveil peut sonner de sorte que la période d'endormissement permette bien le réveil spontané spécifié entre 1 et 12h

Dans notre schéma, il y a deux manières de se réveiller :
- Spontanément
- Ou à l'aide du réveil.

Notre humain peut se réveiller spontanément quand le réveil est armé mais qu'il n'a pas encore sonné.

Nous avons cependant remarqué une erreur dans notre modèle : nous ne pouvons distinguer par quelle transition(sonnerie ou spontanée) l'humain passe à son éveil lorsque le réveil sonne. Afin de pouvoir calibrer le réveil selon les contraintes imposées par la question, il faudrait créer des places supplémentaires permettant de détecter le réveil de l'humain soit par une place « Spontannée » soit par une place « Sonnerie »

Ces places supplémentaires nous permettraient ainsi de vérifier notre propriété de la manière suivante :
- La transition permettant au réveil de sonner (transition **sonner**) aurait comme paramètre [x,y]
- Dans **Check-property** il faudrait vérifier que l'humain se réveille spontanément sans l'aide du réveil
- Ou alors : le réveil sonne, mais le temps que la sonnerie parvienne à l'humain (à cause des conditions de temps asynchrones), l'humain peut également se réveiller de façon spontannée.

5.2 Question 12 : choisir des valeurs pour ces paramètres respectant les contraintes trouvées à la question 11. Refaire alors une analyse non paramétrée en construisant et dessinant l'espace d'états

Cette question étant la suite indirecte de la question 11, nous ne pouvons y répondre. Mais nous allons apporter des éléments de réponse en suivant la méthodologie que nous avons appliquée à la question 11.

Parmi les résultats obtenus à la question précédente, on choisit des résultats qui respectent les contraintes trouvées et on dessine l'espace d'état à l'aide de **Romeo** et de ses outils.

5.3 Question 13 : peut-on conclure que la spécification est maintenant correcte ?

On ne peut pas conclure sur le fait que la specification soit correcte. On peut simplement en déduire qu'elle est plus complète, mais pas correcte.

6 Implémentation

Nous avons implémenté le problème en **Java** avec comme protocole de communication le **RMI**. Tout notre code est documenté et renseigné dans une JavaDoc présente dans le dossier *doc* de notre projet Java (n'hésitez pas à vous y réferer pour comprendre le fonctionnement du code). Nous avons également fourni les tests permettant de vérifier le bon fonctionnement de nos objets et une interface graphique permettant de visionner plus facilement le réveil et de pouvoir intéragir plus facilement en tant qu'humain grâce à des actions accessibles par de simple cliques sur des boutons.

Pour les contraintes de temps nous avons utilisé des objets Date (*java.util.Date*). Afin de faciliter les tests manuels pour vérifier le bon fonctionnement de l'application, nous avons défini la plupart

des contraintes de manière statique, avec des attributs **static final** dans les classes concernées. Chacun de ces attributs possède un attribut de test lui correspondant, qui permet de réduire les temps d'attente.

Exemple : l'humain doit veiller 14 heures avant de pouvoir armer son réveil (soit 840 minutes), l'interface de l'humain possède alors un attribut *armMinimum* fixé à 840, ainsi qu'un attribut *TEST_ armMinimum* fixé à 15 (en secondes).

Pour pouvoir utiliser les valeurs de test, il suffit de définir la variable globale *TESTING* à vrai (**true**). Cette variable est déclarée dans la classe complémentaire **complementary.MiamMiam**.

6.1 Question 14 : écrire et tester indépendamment le code exécutant les automates du réveil et de l'humain. Les transitions pourront être déclenchées interactivement et on pourra utiliser un service d'horloge

Nous avons implémenté les deux automates indépendamment l'un de l'autre. C'est à dire que le réveil peut très bien fonctionner seul, et l'humain également. En terme de conception nous avons fait comme suit :

Pour l'humain :
- **Interface :** human/*HumanInterface*.java
- **Classe :** human/Human.java

Pour le réveil :
- **Interface :** alarmClock/*AlarmClockInterface*.java
- **Classe :** alarmClock/AlarmClock.java

Pour tester ces deux modules séparément, nous avons contruit des classes de test en **JUnit** pour chacun des automates.

Pour l'humain :
- tests/HumanTest.java

Pour le réveil :
- tests/AlarmClockTest.java

Les deux classes de test nous permettent de vérifier que chaque passage d'un état à un autre se fait correctement, et que lorsqu'un de ceux-ci n'est pas autorisé il est bien stoppé.

6.2 Question 15 : programmer la communication entre les deux automates de la question précédente en utilisant les services de communications

Pour la communication entre les deux processus nous avons donc utilisé **RMI**. Pour faire communiquer une action de l'un à l'autre nous avons utilisé une classe intermédiaire permettant de faire passer les messages suivant les transitions.

Le temps de transition d'un message varie entre 0 et 10 minutes, nous avons donc implémenté une classe héritant de **Thread** qui va effectuer une action donnée à condition qu'un contrat donné soit rempli à un temps aléatoire entre 0 et 600 (secondes). Chaque type de message hérite donc de cette classe avec sa propre action et son propre contrat de transmission.

La classe complémentaire joue également le rôle de régulateur des demandes, c'est à dire qu'elle possède un jeton unique qui va permettre d'autoriser ou non une demande. Si une demande est déjà en cours cela va permettre d'attendre la fin de celle-ci avant d'en effectuer une nouvelle.

6.3 Question 16 : essayer de reproduire les situations que nous avons analysées précédemment au niveau du modèle

6.3.1 Lancement de l'application

Pour tester les situations analysées précédemment il suffit de lancer les instances des automates. Pour ne pas attendre trop de temps (réelles contraintes de temps) il faut définir la variable *TESTING* à vrai (**true**) dans la classe **complementary.MiamMiam**

Chaque automate possède une classe principale permettant de créer une instance de chaque objet.

Pour l'humain :

– **Main :** human/main/HumanLauncher.java

Pour le réveil :

– **Main :** alarmClock/main/AlarmClockLauncher.java

Il faut d'abord lancer l'instance de réveil pour déployer celui-ci, puis l'instance de l'humain pour se connecter dessus. Une fois le réveil lancé vous pourrez constater qu'il affiche l'heure actuelle (de la machine) et que celle-ci s'actualise chaque seconde. Deux témoins sont présents sur le réveil, un pour signaler quand il est armé, avec la date où il doit sonner, et un pour signaler qu'il est en train de sonner (en état de sonnerie). Vous pouvez avoir un aperçu de ce que vous devriez avoir au lancement du réveil sur la **Figure 5 – Lancement du réveil**.

Vous pouvez alors lancer l'humain. Une fenêtre avec des boutons apparaîtra. Au lancement tous les boutons sont désactivés, car il doit veiller avant de pouvoir armer le réveil (apperçu sur la **Figure 8 – Lancement du client**).

Une fois que l'humain aura veillé assez longtemps il pourra armer le réveil (apperçu sur la **Figure 9 – Client prêt**). En mode *test*, il est possible de choisir un heure de sonnerie grâce à une fenêtre (apperçu sur la **Figure 11 – Sélection d'une heure de sonnerie**).

Lorsque l'humain a armé le réveil, un message (classe *messages.ArmMessage*) est en cours de transit vers la classe complémentaire (*complementary.MiamMiam*) qui s'occupera de le redistribuer au réveil. Quand le réveil receptionne ce message il s'arme (apperçu sur la **Figure 6 – Réveil armé**). L'humain quant à lui va passer à l'état **endormi** (*asleep*), dans lequel il doit attendre au moins 1 heure (10 secondes en mode test) avant de pouvoir se réveiller par lui-même (apperçu sur la **Figure 10 – Client armé**). Son interface affiche alors la prochaine heure de sonnerie.

Si l'heure de sonnerie est atteinte, le réveil va se mettre en mode sonnerie (apperçu sur la **Figure 7 – Réveil en train de sonner**), et va transmettre le message de sonnerie (classe *messages.RingMessage*) à l'humain. Ce dernier pourra alors soit se rendormir, soit désactiver le réveil. Dans le cas où il souhaite désarmer le réveil, il va envoyer une demande de désarmement (classe *messages.DisarmMessage*). Ainsi le cycle sera terminé.

6.3.2 Figures annexes de l'interface graphique

Le réveil

FIGURE 5 – Lancement du réveil

FIGURE 6 – Réveil armé

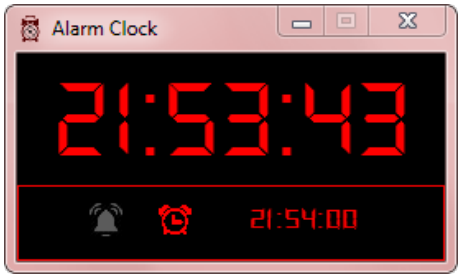


FIGURE 7 – Réveil en train de sonner

L'humain

FIGURE 8 – Lancement du client

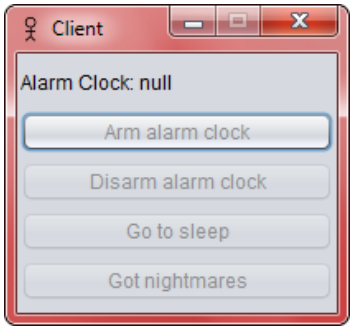


FIGURE 9 – Client prêt

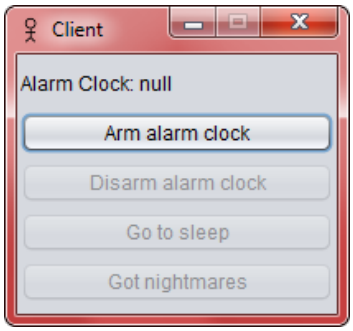
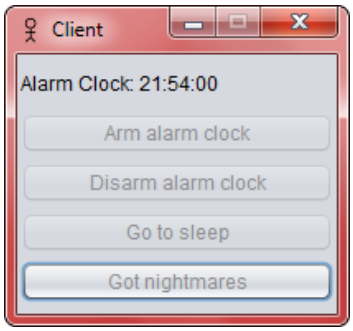
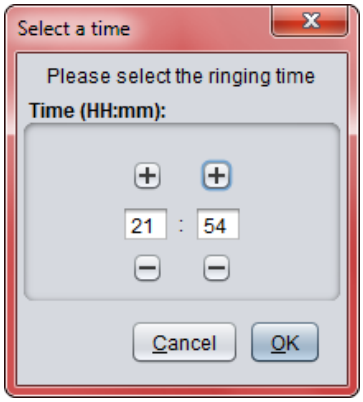


FIGURE 10 – Client armé



Sélection de l'heure de sonnerie

FIGURE 11 – Sélection d'une heure de sonnerie



7 Eléments de conclusion

Ce projet nous a permis de manipuler des outils tels que Romeo et nous rendre compte des différentes contraintes (temporelles et techniques) qui accompagnent un projet. On se rend ainsi compte que l'on souhaite sans cesse améliorer notre modèle et le rendre « parfait ». Le souci étant que la perfection n'existe pas, nous sommes obligés de faire des compromis sur certains choix techniques et trouver un moyen de rendre nos spécifications « correctes »et le plus complet possible.