UNIVERSITÉ CÔTE D'AZUR

Rapport TER
Semestre Paire du Master 1,
Année Académique 2020/2021

Génération de moniteurs SCADE à partir de scénarios de Safety pour le véhicule autonome

Réalisé par :

Marino Samuele

Laubry Vincent

Encadré par :

Frédéric Mallet

Bouali Amar

Table de matières

Introduction

Le sujet se place dans le contexte du projet *PSPC* (Projets Structurants Pour la Compétitivité), de la région *ADAVEC* (Adaptation automatique du Degré d'Autonomie du Véhicule à son Environnement et au Conducteur), en collaboration avec la *Renault Software Factory*.

Il s'agit de partir de quelques scénarios de sûreté, discutés avec les partenaires industriels du projet, ces scénarios sont capturés sur la forme de diagrammes de séquences temporisés. Les propriétés temporelles et causales décrites par ces scénarios doivent être traduites dans un langage formel non ambigu qui peut être utilisé pour un traitement automatique.

Le langage utilisé est *CCSL* (Clock Constraint Specified Language). Ce langage intermédiaire est utilisé pour l'analyse formelle et sert de base pour la génération de moniteurs écrits en *SCADE* qui doivent être intégrés dans le système *SCADE* pour en valider le comportement.

SCADE (Safety Critical Application Development Environment) est un environnement de développement intégré, basé sur le langage Lustre diffusé par Esterel Technologies, destiné à la conception de systèmes critiques.

Etat de l'art

Le projet *ADAVEC* est un projet collaboratif né en juillet 2020 qui a pour objectif un transfert de responsabilité en toute sécurité. Il est spécialisé donc au changement des modes entre la conduite autonome et la conduite manuelle et vice-versa. Ses partenariats sont :

- La société d'ingénierie logicielle française *AVISTO* ;
- La startup fondée par des ingénieurs du monde de l'innovation et du logiciel automobile *EPICNOC* ;
- La *Renault Software Factory* ;
- L'*Université Nice-Sophia Antipolis.*

Parmi les membres de l'université il y a un contributif de part d'un professeur du département d'Electronique, **Charles André**, qui s'est spécialisé sur les langages synchrones comme *SyncCharts*, l'ancêtre de *SCADE*, et qui est d'autant plus celui qui, avec le professeur **Frédéric Mallet**, a inventé le langage *CCSL* que nous avons appris dans ce projet.

Ils nous ont fourni une librairie *SCADE* de base, créée en 2010, qui fournissait une librairie d'opérateurs qu'il fallait implémenter et adapter là où c'était nécessaire pour l'avancement du projet *ADAVEC* lui-même.

Nous avons aussi eu comme base une librairie de Frederic Mallet en *Eclipse Modeling Tools* (Java), qui consistait en une génération de codes du *CCSL* vers Java, aussi bien qu'une interface de Générateur *SCADE*.

Et pour finir, nous avons eu comme base les scénarios eux-mêmes, conçus par les membres du projet *ADAVEC*, ainsi que les ingénieurs Renault, écrites en Excel.

Nos objectifs pour le projet TER consistaient sur l'étude des processus de génération des moniteurs *SCADE* à partir du *CCSL* et l'implémenter, aussi bien que l'étude des scénarios de Safety du projet *ADAVEC* décrits en Excel, en capturer les stimuli et le transformer d'un langage informel et ambigu en un langage formelle et non ambigu, afin de vérifier la bonne conception et viabilité des scénarios données.

Le chemin de ces scénarios était donc le suivant :

Excel → CCSL → Java → SCADE.

A travers Excel nous devions identifier des relations à écrire en *CCSL*.
Ensuite, il fallait appeler les relations écrites en *CCSL* en utilisant Java, de façon formelle et compréhensible à toute l'équipe.
Finalement, générer du code *SCADE* à partir des scénarios réécrits en Java.

Travail effectué

Le travail effectué dans un point de vue globale peut être décomposé en trois parties.

- Comment pouvoir transformer des scénarios écrits d'une façon ambigüe sur Excel en une façon non ambigüe sur Java, en utilisant le langage CCSL comme intermédiaire entre Excel et Java

- Etudier et adapter la librairie SCADE donné par Mr. Charles André afin de comprendre le fonctionnement des opérateurs pour les horloges

- Générer du code SCADE en partant du code CCSL à l'aide de l'environnement Eclipse Modeling Tools (Java) donné par Mr. Frederic Mallet.

Transformation des scénarios

Nous avons commencé l'étape de la transformation par l'analyse des scénarios données par les ingénieurs en Excel. Ainsi, nous avons reçu 31 scénarios différentes à analyser.

Chaque scénario présente :

- Une **Mode de départ**, qui représente si le véhicule est en mode manuelle ou autonome au moment du début du scénario,
- Un ensemble d'éléments qui donnent le contexte du scénario en question. Parmi les éléments il y a au moins un trigger qui détermine la Mode finale. Parmi les éléments nous y retrouvons :
    - **Weather**, le temps atmosphérique du scénario étudié ;
    - **Road**, le type de route ;
    - **Traffic**, l'intensité du trafic routier ;
    - **Vehicle**, si les capteurs étaient bien fonctionnants ;
    - **Human**, l'état du stress, de santé ou d'attention du conducteur.
- Et une **Mode finale**, qui représente l'état attendu à la fin du scénario.

Notre première approche consistait dans l'énumération de ces scénarios, afin de mieux pouvoir les regrouper et les analyser. Parmi les 31 scénarios, donc :

- 7 étaient en Mode de départ manuelle, mais aucun changement de mode était prévu ;
- 8 étaient en Mode de départ manuelle, avec possibilité de changement de mode ;
- 8 étaient en Mode de départ autonome, mais aucun changement de mode était prévu ;
- 8 étaient en Mode de départ autonome, avec le changement de mode était prévu.

Nous avons bien distingué les scénarios en leur donnant un sigle composé d'une lettre et d'un chiffre pour bien les différencier (A e B pour les modes de départ manuelles, C e D pour les modes de départ autonomes, avec un chiffre allant jusqu'à 8, ou 7 dans les scénarios de type A).

Nous avons pensé de générer un fichier CCSL par scénario, ce qui revient donc à la création de 31 fichiers CCSL. Cependant, nous avons réalisé que nous avions appliqué des relations

systématiquement à chaque scénario, en revenant à un implicite copier-coller l'un de l'autre. Donc nous avons abandonné cette pensée pour chercher une autre plus productive.

Après ultérieure analyse des scénarios, nous avons estimé de ne pas tenir en compte la volonté du conducteur de changer de mode, parce que nous ne pouvons pas intervenir sur la volonté du conducteur lui-même. Nous nous sommes donc focalisées là où le changement de mode était bien prévu. Et c'était là que nous avons dû supposer des choix par ambiguïté des scénarios, parce qu'il y avait bien des scénarios dont, même s'il n'y avait pas aucun changement de mode prévu, tel changement c'est vérifié, et vice-versa.

Nous avons donc ignoré les scénarios dont la mode de départ coïncidait avec la mode finale, et nous nous sommes focalisées là où il y avait bien le changement de mode, tout en gardant les sigles pour bien se référer aux scénarios d'origine.

Nous avons donc formulé des relations qui définissaient le changement de mode lui-même sur CCSL et modifié le code généré en Eclipse d'une façon telle que ces scénarios se révélaient faciles à comprendre par les ingénieurs Renault et à ceux qui ne connaissaient pas le langage CCSL. Cela revient à transformer un morceau de code comme celui-là, écrit en CCSL :

```
Specification DebugMode {
    Clock start finish Trigger ReactionTime end Context
    [
        Let Mode be start + finish
        Exclusion start # finish
        Let Condition be Trigger * Context
        Delay = Trigger $ 1 on ReactionTime
        Precedence Mode < Trigger
        Precedence Trigger < Mode
        Precedence Mode <= Delay
        Precedence Delay <= finish
        Precedence finish <= end
    ]
}
```

En celui-là :

```
changeMode(simple, "Manual", "Automatic", "Drunk", clockList);
```

Ainsi nous pouvons bien voir que le conducteur passe en mode manuelle vers la mode automatique quand il est ivre.

Et donc voici les scénarios où il y avait bien changement de mode :

```
// CHANGE MODES
changeMode(simple, "Manual", "Automatic", "Drunk", clockList);                                                    // Scenario A4
changeMode(simple, "Manual", "MRM", "Unconscious", clockList);                                                    // Scenario A5
changeMode(simple, "Manual", "Automatic", new String[]{"HeavyRain", "NotFocused"}, clockList);                    // Scenario A6
changeMode(simple, "Manual", "MRM", new String[]{"HeavyRain", "HeavyTraffic", "FaultySensors", "LongDriving"}, clockList);  // Scenario B3
changeMode(simple, "Manual", "MRM", new String[]{"CountrySide", "Influenced"}, clockList);                        // Scenario B4
changeMode(simple, "Manual", "Automatic", "Influenced", clockList);                                               // Scenario B5
changeMode(simple, "Manual", "Automatic", new String[]{"HeavyRain", "LongDriving"}, clockList);                   // Scenario B8
changeMode(simple, "Automatic", "MRM", new String[]{"HeavyRain", "Drunk"}, clockList);                            // Scenario C2
changeMode(simple, "Automatic", "Manual", new String[]{"FaultySensors", "Stress"}, clockList);                    // Scenario C3
changeMode(simple, "Automatic", "MRM", "Death", clockList);                                                       // Scenario C4
changeMode(simple, "Automatic", "Manual", "HighwayExit", clockList);                                              // Scenario D1
changeMode(simple, "Automatic", "Manual", new String[]{"HighwayExit", "FaultySensors"}, clockList);               // !  // Scenario D3
changeMode(simple, "Automatic", "Manual", "Inactive", clockList);                                                 // Scenario D4
changeMode(simple, "Automatic", "MRM", "Unconscious", clockList);                                                 // Scenario D5
```

Une fois ces scénarios bien définis, nous nous sommes rendu compte de différentes choses qui encore une fois nous amènent à l'ambiguïté.

Par exemple, le lecteur aura remarqué que les scénarios D1 et D3 se ressemblent particulièrement. Avec D1 nous passons de la mode Automatic vers la mode Manuelle quand il y a une sortie d'autoroute, tandis que dans le scénario D3 nous passons les mêmes modes quand il y a le même évènement et aussi les capteurs abimés. Toutefois, le conducteur passera déjà en mode manuelle dés qu'il s'approchera à la sortie d'autoroute, ce qui montre la tautologie de ce scénario par rapport au précèdent.

Nous ne pouvons pas savoir si les ingénieurs Renault qui ont conçu ces scénarios étaient au courant de cette répétition ou non, mais cela montre qu'il est plus facile de comprendre les scénarios quand ils sont écrits de façon formelle.

Une autre remarque concerne les scénarios où il y a plus d'un élément trigger, comme les scénarios B3 et B4.

Selon le scénario B3, le conducteur devra passer de la mode manuelle vers la mode MRM (Minimal Risk Manouver, arrêt d'émergence) quand il y a forte pluie, trafic intense, capteurs abimés et que le conducteur a conduis depuis longtemps. Toutefois, il s'agit de trop d'éléments à tenir en compte pour un changement de mode.

Nous pourrons donc déduire que ce n'est pas le AND logique l'opération qui lie les différents triggers, et qu'il pourra s'agir donc d'un OR logique. Mais cela pose problèmes pour le scénario B4, parce que là le conducteur devra passer de la mode manuelle vers la mode MRM s'il est influencé ou bien en campagne. Et il n'y aurait pas de raisons particulières pour lesquels le conducteur devra s'arrêter pendant la campagne, mais cela pourrait avoir son sens s'il est dans la campagne AND il est influencé.

Encore une fois, donc, l'ambigüité nous amène à supposer et à faire des choix, des choix que les concepteurs des scénarios n'ont pas imaginé ou que pour eux c'était évidente et ont réputé ne pas le mettre, un choix qui n'est pas favorable pour les informaticiens qui s'occupent des systèmes critiques.

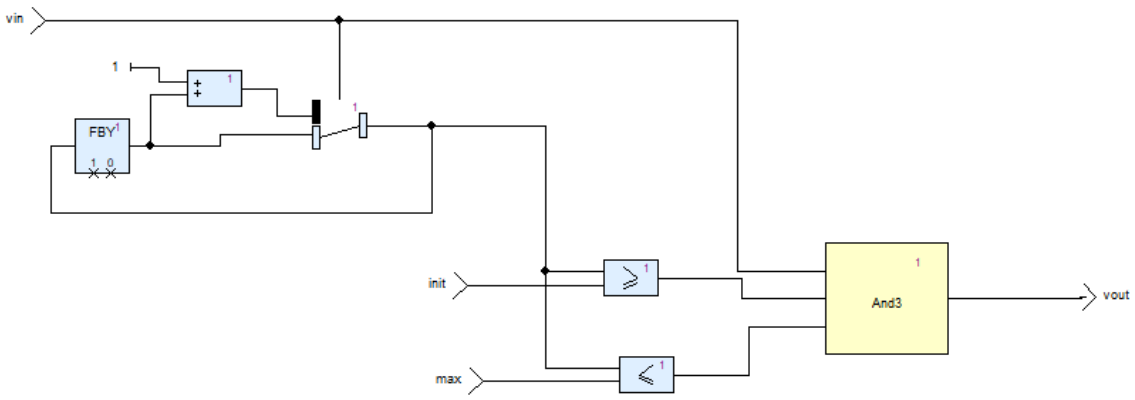
La librairie SCADE

Parmi les premières tâches de ce projet TER, nous avons étudié la librairie de **Charles André**. Nous avons remarqué que la librairie en question a été implémentée le 2010, avec la version 2 du logiciel *SCADE*. De nos jours, nous utilisons la version 6, et c'est la version qu'il nous a été présentée lors du cours de *Safety Critical Systems* du premier semestre de cette année. Par conséquence, il y avait des opérateurs qui ne pouvaient pas être utilisées tels quels pour la version actuelle. D'où la nécessité de notre part de réadapter les operateurs décrits afin de pouvoir être à nouveau utilisées.

Une fois ce travail effectué, nous nous sommes rendu compte qu'il y avait des operateurs qui n'étaient pas implémentées et qu'il fallait donc ajouter. *(Il faut parler du langage CCSL qui était diffèrent en 2010)* Et c'était dans cette occasion que nous nous sommes rendu compte qu'il s'agit d'un projet dont une version SCADE donné par les étudiants ne suffit pas pour en satisfaire tous les besoins.

C'est pour cela qu'il nous a fallu une version complète de ce logiciel, comprise de Licence, à l'aide de Mr. **Amar Bouali** et son équipe d'ingénieurs SCADE. D'autant plus, afin de pouvoir utiliser proprement la version SCADE complète, il était nécessaire une VPN signé Unice. Sans cette VPN, ce n'était pas possible pour nous de sauvegarder nos avancements. Nous n'avons pu obtenir ces deux éléments qu'au dernier mois de notre stage.

Nous avons pu de toute manière implémenter jusqu'à ce que la capacité de SCADE pour les étudiants permît. Notamment, pour certains opérateurs il était nécessaire d'implémenter des operateurs supplémentaires, notamment pour les horloges qui s'activent à partir d'un moment et jusqu'à un autre moment. C'est pour cela que nous avons construit un operateur *InitMax*, dont l'horloge en question avait la même valeur de l'horloge donnée en input à partir d'un certain moment (Init) et jusqu'à un autre certain moment (Max), de la façon suivante :



Concernant les horloges, ils ont trois états possibles :

- PRESENT, pour définir une horloge qui est bien actif à l'instant donnée
- ABSENT, pour définir une horloge qui n'est pas actif à l'instant donnée.
- DEAD, pour définir une horloge qui est ABSENT et ne sera jamais plus PRESENT.

Nous avons traité ces trois états comme s'ils correspondaient au respectifs booléennes, en ajoutant un troisième état pour DEAD. Ainsi, PRESENT est considéré comme TRUE, ABSENT comme FALSE, DEAD comme REALLY FALSE.

Parmi les operateurs (ils seront listés dans la partie Génération du code SCADE) il y a l'Union et l'Intersection, qui peuvent être traitées en façon booléenne respectivement comme un OR et un AND logique. Ainsi, comme dans un OR c'est TRUE l'état absorbant, dans l'Union des horloges, PRESENT est l'état absorbant.

De la même manière, comme dans un AND c'est FALSE l'état absorbant, dans l'Intersection des horloges, l'état absorbant est réservé à l'état DEAD, comme il est plus FALSE de l'état ABSENT lui-même.

De toute manière, même en ayant cette nouvelle conception des horloges avec ce troisième état booléenne, pour ce qui concerne le contexte des véhicules autonomes et notre stage, la mort des horloges n'a pas été appliqué.

Génération du code SCADE

Dans la librairie donnée par Mr. Frédéric Mallet, nous avons eu aussi une Interface Java, appelée *MyScadeGenerator*, qui présentait des méthodes dont le corps n'était pas donné. C'était à nous de remplir ces méthodes d'une façon telle à pouvoir générer un fichier SCADE à l'exécution du programme Java qui l'appelait.

Les méthodes étaient les suivantes :

- AddClock ;
- SubClock ;
- Exclusion ;
- Precedence ;
- Causality ;
- Inf ;
- Sup ;
- Union ;
- Intersection ;
- Minus ;
- Periodic ;
- DelayFor.

Nous avons longuement réfléchi pour le corps de ces méthodes. Après avoir réalisé qu'il y a deux façons pour réaliser un fichier SCADE, une visuelle et une textuelle, nous avons donc adopté l'approche textuelle. Un fichier SCADE textuel en effet présente une syntaxe qui se rapproche du langage Lustre que nous avons étudié avec Mr. Mallet pendant le cours de Programmation Synchrone de ce semestre.

Voici donc comment un fichier SCADE textuel qui représente l'Union entre deux horloges est représenté :

```
L1 = a;
L2 = b;
L3 = Union(L1, L2);
o = L3;
```

Où **a** et **b** sont des horloges, **o** est l'horloge résultante, **L1**, **L2**, et **L3** sont les fils qui lient les horloges (comme dans l'image qui montre l'InitMax) et Union(L1, L2) représente l'action de l'operateur Union sur les deux horloges liées par les fils L1 et L2 (dont l'information des horloges **a** et **b** est passée respectivement).

Afin d'obtenir un fichier similaire, nous avons pensé que cela reviendrait comme écrire dans un fichier. Nous avons donc utilisé un PrintWriter pour écrire ces fichiers de façon automatisée. Nous avons aussi utilisé des compteurs pour les fils, et aussi pour les outputs.

Prenons comme exemple la méthode de l'exclusion. Voici les corps que nous y avons donné :

```
// Method descriptor #8 (Ljava/lang/String;Ljava/lang/String;)V
public void exclusion(java.lang.String arg0, java.lang.String arg1){
    outputfile.println("L" + ++countL + " = " + arg0 + ";");
    outputfile.println("L" + ++countL + " = " + arg1 + ";");
    outputfile.println("L" + ++countL + " = Exclusion(L" + (countL - 2) + ", L" + (countL - 1) + ");");
    outputfile.println("O" + ++countO + " = L" + countL + ";");
}
```

L'objet de type PrintWriter *outputfile* et la méthode *println* font en sorte que dans le fichier dont le nom est donné par *outputfile* soient écrites ces relations qui composeront le fichier lui-même. Le compteur *countL* sera incrémenté avant d'être écrit, et *arg0* et *arg1* correspondront aux horloges définies par le fichier Java qui appelait cette interface. Pour les Exclusions eux-mêmes, on revenait en arrière avec *countL* autant des fois que des arguments de la méthode. Et finalement nous donnons l'output comme dernier *countL*, exactement comme dans un fichier SCADE textuel.

Parmi les méthodes il y en avait certaines qui étaient récurrentes, comme Union et Intersection. En effet, il pourrait très bien avoir une OR et un AND avec plus de deux éléments. Sachant aussi que nous avons appris dans les cours de Programmation Synchrone comment traiter la récursion d'une façon parallèle et plus rapide, notre encadrant nous a fait pression afin d'adopter la même approche aussi dans ces opérateurs. En voici donc ce que nous avons fait :

```
// Method descriptor #14 (Ljava/lang/String;[Ljava/lang/String;)V
public void union(java.lang.String arg0, java.lang.String... arg1){
    int n = arg1.length;
    if (n == 1) {  // Cas terminal: n == 2
        outputfile.println("L" + ++countL + " = " + arg0 + ";");
        outputfile.println("L" + ++countL + " = " + arg1[0] + ";");
        outputfile.println("L" + ++countL + " = Union(L" + (countL - 2) + ", L" + (countL - 1) + ");");
        outputfile.println("O" + ++countO + " = L" + countL + ";");
    } else {
        String[] left = Arrays.copyOfRange(arg1, 0, n/2);
        String[] right = Arrays.copyOfRange(arg1, n/2, n);
        union(arg0, left);
        union(arg0, right);
    }
}
```

(Il faut donner une image encore meilleure)

Nous pouvons voir ainsi le cas d'arrêt par n, la longueur du tableau de String donnée en arg1, correspondant à 1, et nous pouvons voir ici comment la récurrence est effectué de façon parallèle à gauche et à droite du tableau, ce qui permet de résoudre en temps logarithmique la récurrence, au lieu d'en faire un linéaire, dont le temps d'exécution est lui aussi linéaire.

Afin de pouvoir vraiment appeler cette interface, il fallait modifier légèrement la classe Main du program Java généré par CCSL, en ajoutant le morceau qui permettait la liaison entre le fichier Java et l'interface *MyScadeGenerator*. Une fois ce morceau ajouté, il suffit juste d'exécuter le programme Java pour que le fichier SCADE textuel soit généré.

(Donner un exemple simple)

Gestion de projet

Nous avons travaillé sur les trois parties d'une façon parallèle, ce qui signifie que nous avons progressé sur les trois aspects au même temps.

Nous avons travaillé sur les trois parties en permanence en appel Zoom, en choisissant les weekends et particulièrement le jeudi pour se dédier au projet. Nous avons eu également une réunion avec notre encadrant tous les jeudis de 16h à 18h pour lui montrer l'avancement des trois parties, ainsi que recevoir un retour sur ce qui allait bien et ce qui fallait améliorer.

Les appels réguliers ont permis un avancement constant de ce projet, et avec les appels nous avons pu travailler sur le même point, en allant plus vite et plus efficaces sur le même écran.

Le projet lui-même peut être vu et téléchargé par le lien donné dans la section Bibliographie.

Conclusion

Au moment de la rédaction de ce rapport, nous avons bien travaillé sur les trois parties qui concernent la génération du moniteur SCADE.

Nous avons fait en sorte de rendre les scénarios plus lisibles ;
nous avons modifié et ajouté du contenu à la librairie SCADE de Mr. André,
et nous avons aussi généré des fichiers SCADE textuel.

Nous avons donc avancé sur trois projets différents, mais il reste une dernière étape, soit faire la liaison entre ces trois projets afin d'en donner un grand et unique projet, ce qui est notre TER.

Gérer le temps pour se dédier efficacement à ce projet n'a pas du tout été facile, notamment avec les autres projets des différentes matières que nous avons eus.

D'autant plus, avoir reçu tardivement la licence pour SCADE professionnel et la VPN pour en sauvegarder les progresses a porté de la dette à ce projet, que nous avons fait en sorte de palier en avançant sur les autres parties.

Nous espérons porter un projet beaucoup plus stable avant notre passage à l'oral.

Perspectives et réflexions personnelles

Nous nous sommes rendu compte rapidement que ce projet ne correspondait pas à ce que nous pensions de faire. En effet, nous pensions faire un projet entièrement sur SCADE, un peu comme nous avons fait le projet du premier semestre de cette année, qui consistait en l'implémentation d'un ascenseur, faisant attention à ce qu'il se comportait comme prévu, reflexe qu'il est nécessaire d'adopter dés qu'il s'agit de développer des systèmes critiques.

Nous nous sommes retrouvés à donner un contributif sur un grand projet, qui d'autant plus est actuel, en utilisant toutes les connaissances possibles obtenues en Licence.

Et cela a été une expérience exténuante sous certains points de vue, mais bien satisfaisante !

Notre encadrant nous montrait patiemment la bonne voie à adopter sur certaines parties de ce projet, un peu comme s'il y avait une collaboration. En effet, il y a bien eu de la collaboration, vu qu'à ce projet cette université en est un partenariat.

Nous pouvons en conclure que cela sera ce que nous, comme informaticiens du futur, devrons s'attendre lors que nous sortirons de cette université pour mettre en œuvre notre savoir et notre savoir-faire pour travailler.

Nous sommes donc à la fin de ce rapport, en espérant que celui-ci ait plu au lecteur, mais en espérant aussi que celui-là lui ait permis d'en savoir plus sur le projet ADAVEC, aussi bien que notre projet TER lui-même.

MASTER INFORMATIQUE

MARINO Samuele,
LAUBRY Vincent,
M1 Informatique,
Année 2020/2021

Bibliographie

Projet ADAVEC : http://adavec.fr/fr/

CCSL : https://hal.inria.fr/hal-03135428

SyncCharts : https://www.i3s.unice.fr/~andre/SyncCharts/

SCADE: https://www.ansys.com/products/embedded-software/ansys-scade-suite

Lustre: https://www-verimag.imag.fr/The-Lustre-Programming-Language-and?lang=en

Lien GitHub de ce projet : https://github.com/Viince06/TER_SafetyCriticalSystem