Rapport –

Question 1 : Opérateur Button

Objectif

L'opérateur **Button** doit mémoriser l'appui d'un bouton :

- lorsque `press` passe de `false` à `true`, la sortie `req` devient `true`,
- ensuite `req` reste à `true` même après relâchement,
- l'implémentation actuelle ne prévoit pas encore de remise à zéro (reset).

Méthode

La solution a été séparée en deux parties :

1. **Détection de front montant**
    - Utilisation des blocs `previous`, `Init` et `not` pour détecter le passage de `press` de `false` à `true`.
    - La sortie de cette logique correspond au signal `rise`.
2. **Mémorisation**
    - L'idée est de définir une variable locale `latched` qui conserve son état :
    - `latched = (false -> pre(latched)) OR rise;`
    - `req = latched;`
    - Dans le schéma, cette mémoire a été réalisée avec un deuxième `previous` + `Init` + un `or`.
    - Pour pallier la difficulté de définir une variable locale en mode *Write/Read* dans la version Étudiante de SCADE, une étiquette `latchedWrite` a été introduite pour représenter la sortie du `or`.
    - Cela provoque des messages d'erreurs lors du check (variable non définie / unused flow), mais l'intention fonctionnelle est respectée.

Résultat attendu (simulation)

- Au démarrage (`press = false`) → `req = false`.
- Si `press = true` pendant un cycle → `req` devient `true`.
- Lorsque `press` revient à `false` → `req` reste `true` (latch).

Limitation rencontrée

La version Étudiante ne permet pas de gérer correctement la définition et l'utilisation d'une variable locale unique (`latched`). Pour contourner, la sortie du `or` a été nommée `latchedWrite` et réutilisée ailleurs. Cela explique les erreurs signalées par l'outil lors du check, mais le comportement fonctionnel attendu est conforme à l'énoncé.

Conclusion

L'opérateur **Button** a été modélisé pour mémoriser un appui. Le schéma est fonctionnel en simulation malgré les limitations de la version Étudiante qui empêchent une gestion propre des variables locales. Les erreurs relevées seront mentionnées comme conséquence de ce contournement technique.

Question 2 : Machine à états de la porte

Objectif

Spécifier le comportement d'une porte au moyen d'une **machine à états** qui renseigne en sortie son statut (`doorStatus`).
Les statuts possibles sont :

- `Closed` (fermée)
- `Opening` (en cours d'ouverture)
- `Open` (ouverte)
- `Closing` (en cours de fermeture)

Méthode

1. Création d'un type énuméré `DoorStatus` avec les quatre valeurs ci-dessus.
2. Création d'un **Node Operator** `Door` avec :
    - Entrées :
        - `openDoor : bool` (commande d'ouverture)
        - `closeDoor : bool` (commande de fermeture)
    - Sortie :
        - `doorStatus : DoorStatus`
3. Conception d'une **machine à états** contenant les quatre états `Closed`, `Opening`, `Open`, `Closing`.
    - L'état initial est `Closed`.
    - Transitions définies :
        - `Closed → Opening` si `openDoor = true`
        - `Opening → Open` (transition immédiate dans la version simplifiée)
        - `Open → Closing` si `closeDoor = true`
        - `Closing → Closed` (transition immédiate dans la version simplifiée)
4. Dans la **version Étudiante de SCADE**, l'éditeur n'affiche pas directement l'onglet *Actions/Activity*.
    - Pour contourner cette limitation, l'assignation de la sortie `doorStatus` a été placée dans une **note** à l'intérieur de chaque état.
    - Exemple :
        - Dans l'état `Closed` : `doorStatus := Closed;`
        - Dans l'état `Opening` : `doorStatus := Opening;`
        - etc.

Résultat attendu (simulation)

- Au démarrage, `doorStatus = Closed`.
- Lorsqu'une commande `openDoor = true` est donnée, la machine passe par `Opening`, puis `Open` (`doorStatus = Opening` puis `Open`).
- Lorsqu'une commande `closeDoor = true` est donnée, la machine passe par `Closing`, puis `Closed` (`doorStatus = Closing` puis `Closed`).

Conclusion

La machine à états de la porte a été réalisée avec quatre états correspondant aux différents statuts.
L'utilisation des **notes dans chaque état** pour assigner `doorStatus` permet de pallier les limites de la version Étudiante de SCADE, tout en obtenant le comportement attendu.

Rapport – Question 3 : Machine à états de la passerelle

Objectif

Spécifier le comportement de la passerelle au moyen d'une **machine à états** donnant en sortie son statut (`bridgeStatus`).
Les statuts possibles sont : `Retracted`, `Deploying`, `Deployed`, `Retracting`.

Méthode

1. Définition du type énuméré `BridgeStatus`.
2. Création d'un Node Operator `Bridge` avec deux entrées (`deployBridge`, `retractBridge`) et une sortie (`bridgeStatus`).
3. Conception d'une machine à états avec 4 états :
    - État initial : `Retracted`.
    - Transitions :
        - `Retracted → Deploying` si `deployBridge = true`.
        - `Deploying → Deployed` (simplifié).
        - `Deployed → Retracting` si `retractBridge = true`.
        - `Retracting → Retracted` (simplifié).
4. Assignation de la sortie `bridgeStatus` par une **note** dans chaque état (solution de contournement car l'onglet *Actions/Activity* n'est pas disponible dans la version Étudiante).

Résultat attendu (simulation)

- Au démarrage : `bridgeStatus = Retracted`.

- Si `deployBridge = true` → passage par `Deploying` puis `Deployed`.
- Si `retractBridge = true` → passage par `Retracting` puis `Retracted`.

Conclusion

La machine à états de la passerelle a été réalisée de façon analogue à celle de la porte, avec des états adaptés. L'utilisation de notes pour assigner `bridgeStatus` dans chaque état permet d'obtenir un modèle fonctionnel malgré les limitations de la version Étudiante de SCADE.

Question 4

Objectif

Assurer la **synchronisation** entre la porte et la passerelle afin de garantir la sécurité des passagers.

Solutions mises en place dans le contrôleur

1. **Ouverture de porte conditionnée** : la porte ne peut s'ouvrir que si la passerelle est déjà déployée (`bridgeStatus = Deployed`).
2. **Blocage du mouvement de la passerelle** : la passerelle ne peut ni se déployer ni se rétracter lorsque la porte est ouverte ou en train de s'ouvrir.
3. **Fermeture de porte** : elle peut se produire indépendamment de la passerelle, dès que l'ordre est donné.
4. **Rétraction de passerelle** : autorisée uniquement si la porte n'est pas ouverte.

Rôle du contrôleur

Le contrôleur agit comme un **filtre logique** :

- il reçoit les demandes brutes (`openDoorCmd`, `closeDoorCmd`, `deployBridgeCmd`, `retractBridgeCmd`) ainsi que les statuts (`doorStatus`, `bridgeStatus`),
- il décide s'il est **autorisé ou non** de transmettre la commande aux automates de la porte et de la passerelle.

Conclusion

Le contrôleur garantit que la passerelle est toujours déployée avant l'ouverture de la porte et qu'elle reste immobile tant que la porte n'est pas refermée. La logique de décision repose donc sur les statuts de la porte et de la passerelle en plus des commandes.

Rapport – Question 5 : Commandes d'ouverture de porte et de déploiement de passerelle en station

Objectif

Spécifier l'envoi des commandes d'ouverture de la porte et de déploiement de la passerelle lorsque la rame est en station et qu'une demande a été effectuée par un passager (porte ou passerelle).
La commande doit être maintenue tant que l'action n'est pas réalisée (porte effectivement ouverte ou passerelle effectivement déployée).

Méthode

1. **Entrées considérées** :
   - `inStation : bool` (position de la rame),
   - `reqDoor : bool` (demande de porte),
   - `reqBridge : bool` (demande de passerelle),
   - `doorStatus : DoorStatus` (état courant de la porte),
   - `bridgeStatus : BridgeStatus` (état courant de la passerelle).
2. **Sorties générées** :
   - `deployBridge : bool` (commande de déploiement),
   - `openDoor : bool` (commande d'ouverture),
   - `resetBridgeReq : bool` (réinitialisation de la demande passerelle),
   - `resetDoorReq : bool` (réinitialisation de la demande porte).
3. **Implémentation graphique** dans un *Node Operator* :
   - **Déploiement de passerelle** :
   - `deployBridge = inStation`
   - `AND (reqDoor OR reqBridge)`
   - `AND NOT( (bridgeStatus = Deployed) OR (bridgeStatus = Deploying) )`
   - **Ouverture de porte** :
   - `openDoor = inStation`
   - `AND (reqDoor OR reqBridge)`
   - `AND (bridgeStatus = Deployed)`
   - `AND NOT( (doorStatus = Open) OR (doorStatus = Opening) )`
   - **Resets** :
   - `resetBridgeReq = (bridgeStatus = Deployed)`
   - `resetDoorReq   = (doorStatus = Open)`
   - Pour réaliser les comparaisons, des **variables locales initialisées à des valeurs énumérées** (`vDeployed`, `vDeploying`, `vOpen`, `vOpening`) ont été utilisées comme contournement, la version Étudiante ne permettant pas de créer directement des constantes d'énumération.

Résultat attendu en simulation

- **Déploiement** : quand la rame est en station et qu'un passager demande porte ou passerelle, la passerelle commence à se déployer. La commande `deployBridge` reste active jusqu'à ce que `bridgeStatus = Deployed`.
- **Ouverture** : dès que la passerelle est déployée, la commande `openDoor` devient active. Elle reste active tant que `doorStatus` n'est pas `Open`.
- **Resets** : une fois la passerelle déployée (`bridgeStatus = Deployed`), la demande associée est réinitialisée ; de même pour la porte quand elle est ouverte (`doorStatus = Open`).

Conclusion

Le contrôleur en station garantit une séquence correcte :

1. Déploiement de la passerelle,
2. puis ouverture de la porte.
   Les commandes sont maintenues jusqu'à exécution complète et les demandes passagers sont réinitialisées automatiquement.

Question 6 : Fermeture de porte et rétraction de passerelle lors du départ immédiat

Objectif

Spécifier l'envoi des commandes de **fermeture de la porte** et de **rétraction de la passerelle** lorsqu'un départ immédiat est déclenché, afin d'assurer la sécurité avant le mouvement de la rame.

Méthode

1. **Entrées considérées** :
   - `immDeparture : bool` (signal indiquant le départ immédiat),
   - `doorStatus : DoorStatus` (état de la porte),
   - `bridgeStatus : BridgeStatus` (état de la passerelle).
2. **Sorties générées** :

- `closeDoor : bool` (commande de fermeture),
- `retractBridge : bool` (commande de rétraction).

3. **Logique de décision** :
   - **Fermeture de porte** :
   - `closeDoor = immDeparture AND NOT(doorStatus = Closed)`

     La commande est envoyée tant que la porte n'est pas encore fermée.

   - **Rétraction de passerelle** :
   - `retractBridge = immDeparture AND NOT(bridgeStatus = Retracted)`

     La commande est envoyée tant que la passerelle n'est pas encore rétractée.

4. Pour réaliser ces comparaisons, des variables locales initialisées à des valeurs énumérées (`vClosed`, `vRetracted`) sont utilisées, comme pour les questions précédentes, en remplacement de constantes (indisponibles dans la version Étudiante).

Résultat attendu en simulation

- Si `immDeparture = true` :
  - et que la porte est ouverte (`Open` ou `Opening`), la commande `closeDoor` est activée jusqu'à ce que `doorStatus = Closed`.
  - et que la passerelle est déployée (`Deployed` ou `Deploying`), la commande `retractBridge` est activée jusqu'à ce que `bridgeStatus = Retracted`.
- Une fois les deux équipements sécurisés, les commandes retombent automatiquement à `false`.

Conclusion

Le contrôleur en cas de départ immédiat garantit la fermeture de la porte et la rétraction de la passerelle si nécessaire. Les commandes sont maintenues jusqu'à exécution complète, ce qui assure une séquence sécurisée avant la mise en mouvement de la rame.