Integration Continue de Modèle

You

September 1, 2023

Contents

1 Introduction

Lors de travaux commun sur des modèles, plusieurs problématique se lèvent et notamment sur l'intégration continue de ceux-ci dans un projet.
Ce rapport fait une liste non exhaustive des problématiques pouvant être rencontrés, ainsi que des solutions pouvant être mis en place pour s'adapter à celle-ci.

Les exemples utilisés seront principalement en *Tina*, ou en *xmi* (Représentation des modèles utilisés par Ecore sur Éclipse). Le gestionnaire de version utilisé sera Git et l'hébergeur GitHub.

1.1 L'intégration continue

L'intégration continue (CI) est une approche clé dans le domaine du développement logiciel moderne. Elle vise à améliorer l'efficacité, la qualité et la collaboration au sein des équipes de développement en automatisant le processus d'intégration et de validation du code source. L'idée fondamentale derrière l'intégration continue est de fusionner fréquemment les changements de code dans un référentiel centralisé, suivi d'une série d'opérations automatisées, telles que la compilation, les tests et le déploiement, pour détecter rapidement et résoudre les problèmes éventuels.

Au cœur de l'intégration continue se trouvent les systèmes d'intégration et de déploiement continus (CI/CD). Les outils de CI/CD automatisent l'exécution des tests unitaires, des tests d'intégration et des tests de déploiement pour s'assurer que le code nouvellement intégré n'affecte pas négativement le fonctionnement global de l'application. Si un problème est détecté, l'équipe de développement est immédiatement alertée, ce qui lui permet de réagir rapidement et de corriger les erreurs avant qu'elles ne perturbent le projet.

L'intégration continue présente de nombreux avantages, tels que la réduction des risques associés aux mises à jour de logiciels, l'amélioration de la collaboration au sein des équipes de développement, l'accélération du processus de développement et l'amélioration de la qualité globale du code. En outre, l'automatisation des tâches répétitives libère les développeurs des tâches fastidieuses, leur permettant de se concentrer davantage sur l'innovation et l'amélioration des fonctionnalités.

1.1.1 Représentation de l'intégration continue (CI/CD)

Commençons par essayer de comprendre l'intégration continue, ainsi que ses principales fonctionnalités.

Comme dit précédemment, l'intégration continue comporte 3 aspect: la gestion de version, les tests, et le déploiement.

La gestion de version est l'aspect qui comprend la mise en commun des différentes versions des contributeurs d'un projet. Pour cela, chaque utilisateur doit pouvoir accéder au travail commun, ajouter des versions, récupérer des versions, etc ...

Les tests sont la pour vérifier que toutes les versions mises en communes corresponde à des normes définis par le projet. Cela peut être des vérification de build, d'exécution de code, de convention d'écriture, et autres. Il peut effectuer différentes actions en fonction du résultat du test.

Après ça, vient la phase de déploiement, si le code vérifie les test nécessaire, le projet peut être déployé sur l'environnement de test ou sur l'environnement de production.

1.1.2 Méta modèle de l'intégration continue

1.1.3 Ontologie de l'intégration continue

1.2 Fonctionnalité offerte par Git et GitHub concernant l'intégration continue

1.2.1 Fichier config du dépot Git

Le fichier config (se trouvant dans ./.git/ d'un dépot git) permet de stocker les paramètres et les options qui contrôlent le comportement de Git pour ce projet particulier. Il offre plusieurs sections, permettant chacune de paramétrer différent comportement du dépot. Les principales sections et options comprennent [core] pour les configurations de base de Git, [remote "nom_distant"] pour les informations sur les dépôts distants, [branch "nom_branche"] pour les configurations de branches spécifiques, [user] pour les informations sur l'utilisateur du dépôt, [alias] pour définir des raccourcis de commandes, [merge] pour configurer les stratégies de fusion, [pull] pour spécifier le comportement des commandes git pull, [push] pour configurer les commandes git push, [submodule] pour les paramètres de sous-modules, et [receive] pour les hooks de serveur pour le dépôt distant.

Celle qui nous intéresse le plus pour l'intégration continue sont les suivantes : [merge], [pull], [push] Voici différentes options, (non-exhaustive) que l'on peut paramétrer:

[merge]

- tool: Spécifie l'outil de fusion externe à utiliser lorsqu'un conflit survient.

- ff: Active ou désactive les fusions "Fast-Forward" lors des opérations de fusion.
- commit: Détermine si Git doit automatiquement créer un commit de fusion lorsqu'une fusion est effectuée.
- log: Active ou désactive l'affichage du log après une fusion.
- driver: Permet de spécifier un programme externe personnalisé pour gérer les fusions automatiques. Ceci est utilisé pour les fusions personnalisées pour les fichiers binaires ou les formats spécifiques.

[pull]

- rebase: Détermine si git pull doit utiliser le rebase plutôt que la fusion pour récupérer les changements du dépôt distant.
- ff: Active ou désactive les fusions "Fast-Forward" lors d'un git pull.
- default: Spécifie la branche distante à utiliser par défaut pour récupérer les modifications.

[push]

- default: Détermine le comportement par défaut de git push. Les valeurs possibles sont nothing, matching, simple, current, et upstream.
- followTags: Si activée (valeur : true), git push poussera également les balises (tags) associées aux commits poussés.
- defaultCurrent: Si activée (valeur : true), git push sans argument poussera la branche courante vers son homologue sur le dépôt distant.
- defaultUpstream: Si activée (valeur : true), git push sans argument poussera vers la branche de suivi (upstream) de la branche courante, si elle est configurée.
- matching: Active ou désactive la vérification des branches correspondantes lors de la poussée.
- current: Active ou désactive la vérification de la branche courante lors de la poussée.
- simple: Active ou désactive la vérification de la branche courante lors de la poussée, mais refuse les poussées qui créeraient de nouvelles branches sur le dépôt distant.

L'option driver de la section merge sera étudié plus en détails dans la section 2.

1.2.2 Les Hooks Git

Les hooks sont des scripts écrit par l'utilisateur, et qui sont exécuter lors de l'utilisation de commande git. Chaque hook est déclenché à un moment précis lors du Workflow de l'intégration continue. Un hook est toujours facultatif, il n'est pas nécessaire d'en avoir un. Voici un schéma qui résume les différentes actions de git ainsi que l'ordre d'exécution des hooks:

Pour connaître les arguments, et les retours de chaque hook, il faut se référencer au site de Git. L'exemple du pre-commit sera abordé lors de la section 2.1.2.

1.2.3 Les GitHub Action

GitHub actions est un procédé donné par GitHub afin de définir des WorkFlows afin de build, tester ou de déployer le projet. Une action est déclenché par un évenement sur le dépot GitHub, comme une pulll request, ... Le workflow permet de faire tourner en séquentielle et en parallèle différente tâche, s'éxécutant dans une machine virtuelle.

Pour créer un Workflow, il faut créer un fichier d'extension *yml* dans le dossier .github/workflows/ .

Un fichier *yml* ce présente comme ceci:

```
name: learn-github-actions
run-name: ${{ github.actor }} is learning GitHub Actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
```

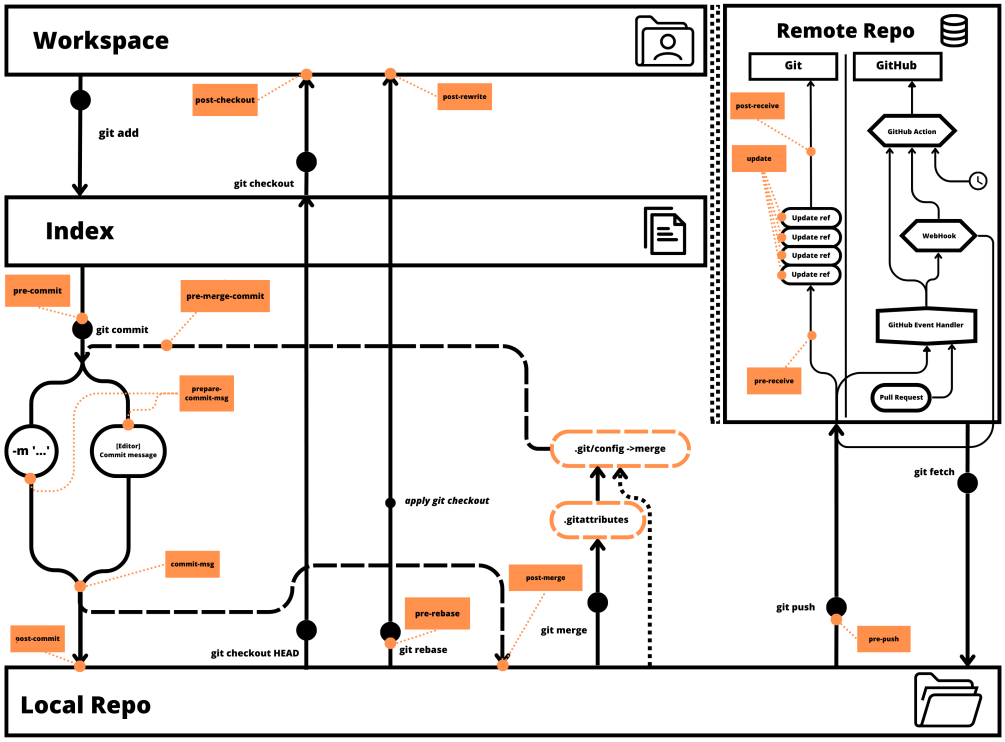

Figure 1: Schéma du workflow de Git

```
  with:
    node-version: '14'
- run: npm install -g bats
- run: bats -v
```

Pour des informations en détails sur comment définir un Workflow: voici le site officiel présentant ceux-ci.

1.2.4 Les GitHub WebHooks

Un webhook sur GitHub est un mécanisme qui permet à GitHub d'envoyer des notifications automatiques aux applications externes en réponse à des événements spécifiques sur le dépôt GitHub. Lorsque vous configurez un webhook pour un dépôt, GitHub enverra une requête HTTP POST à l'URL spécifiée chaque fois que l'événement déclencheur se produit. Ces événements déclencheurs peuvent être des actions telles que la création d'un nouveau commit, l'ouverture d'une nouvelle pull request, la création d'une nouvelle branche, la fusion d'une pull request, etc.

Pour faire un WebHook, il est donc nécessaire d'avoir un serveur gérant la réception de la requêtes HTML. On peut soit écrire le serveur nous mêmes, et donc avoir un contrôle total sur ce qui est fait, soit passer par des services comme CircleCI, Jenkins...

La section 2.2.1 donne un exemple de configuration d'un WebHook réagissant à une pull request.

2 Fusion de versions d'un modèle simple

Durant un projet qui utilise un gestionnaire de version, il est souvent nécessaire de faire des fusions d'un même modèle mais ayant chacun subit des modifications différentes. Il est alors nécessaire de combiner ces changements afin d'obtenir le modèle comprenant toutes les améliorations faites pour chaque versions. C'est ce qu'on appelle un *merge*.
Par défaut un *merge* est fait par une fusion à trois points: *l'ancêtre* (common ancestor), *la base* (into), et *la distante* (in)

Par défaut, la fusion compare le texte différent entre l'ancêtre et la base puis entre la distante et l'ancêtre, et combine les différences trouvé dans la version fusionné.
Si cela marche plutôt bien pour des fichiers texte comme du code, cela ne convient pas pour la représentation de modèle. En

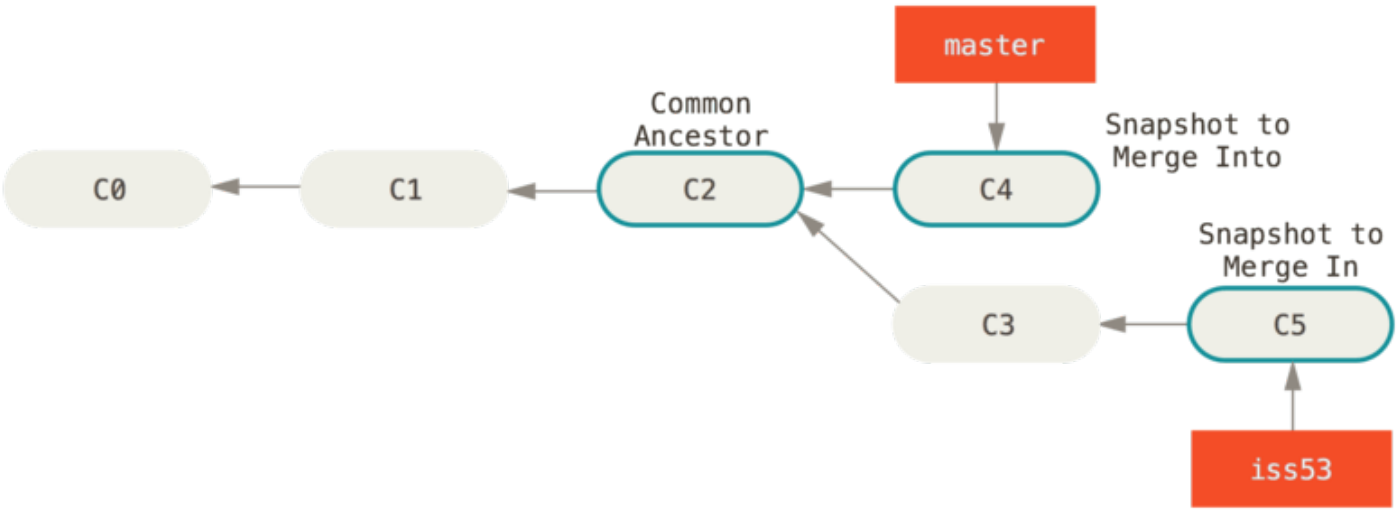

Figure 2: Schéma des fichiers utilisés lors d'une fusion
https://git-scm.com

effet, le fichier n'est pas forcément représenté dans un fichier texte (il peut par exemple être binaire), et leurs représentation n'est pas normalisé: un même modèle, qui peut souvent être perçu comme un graphe, ne possède pas d'ordre sur les éléments qu'ils contient, mais seulement des liens entre eux. Ainsi, un même modèle, écrit dans un même langage peut avoir plusieurs fichiers différent le décrivant.

Il est donc nécessaire de décrire soit même comment fusionner un modèle.

2.1 Gestion en local

Le gestionnaire de version Git permet de travailler en local, avant de partager aux autres le travail effectué. Chaque utilisateur peut donc avoir ses propres branches locales, et peut vouloir les fusionner.

Dans cette sous partie, nous allons voir comment un utilisateur peut configurer son propre *merge* ainsi que normaliser les modèles.

2.1.1 Configuration d'un merge

Un contributeur, travaillant en local avec Git veut donc pouvoir fusionner des branches. Git offre la possibilité, de modifier le comportement du merge utilisé pour fusionner les fichier lorsque on tape la commande **git merge *branch***.

Pour cela, il faut d'abord créer le script de fusion. Celui-ci prend en argument trois fichier: *l'ancêtre* (common ancestor), *la base* (into), et *la distante* (in). Il écrit ensuite dans le fichier servant de *base* le résultat de la fusion.

Un exemple d'un script simple de fusion, en *.sh*, qui ajoute le contenu de l'ancêtre et du fichier distant à celui de la base:

```
echo "Ma fusion"
$1 >> $2
$3 >> $2
```

Une fois le script écrit, il faut définir à Git comment et quand l'utiliser. Pour définir comment utiliser le merge, il faut ajouter une section dans le fichier *.git/config* depuis la racine du dépôt.

Une section pour définir le merge se présente comme ceci:

```
... (sections précédentes)
[merge "nom-du-merge"]
    driver = path/vers/le/script $O $A %B
```

```
... (sections suivantes)
```

nom-du-merge est à choisir selon les préférences, et le path est à commencer à partir de la racine du projet (la où se trouve le dossier *.git*).
Il est donc possible de définir plusieurs merge, chacun avec son script, en ajoutant d'autre section avec des noms différents.

Enfin, il faut définir quand utiliser le merge. Dans le fichier (il peut-être nécessaire de le créer) *.gitattributes*, se trouvant à la racine du projet, écrivez sur une ligne, les fichiers, ou les expressions régulières décrivant les fichiers, ainsi que le merge à utiliser pour ceux-ci.
Par exemple:

```
*.mrg merge=nom-du-merge
```

Ici, notre merge *nom-du-merge* sera utilisé pour tous les fichiers dont l'extension est *mrg*.

Tout ceci étant local, et utilisant le dossier *.git* qui n'est pas possible de diffuser par git, il est conseillé de diffuser un script qui va copier les fichiers utiles depuis un dossier que l'on partage dans le projet, et qui va les copier dans le *.git* local de chaque utilisateur.

```
cp .gitconfig/config .git/config
git config merge.nom_du_driver.driver .gitconfig/merger
chmod u+x .gitconfig/merger
cp .gitconfig/.gitattributes ./gitattributes
```

2.1.2 Hook

La fusion pouvant tout de même échouer, il est parfois nécessaire de vérifier manuellement comment fusionner les fichiers, et de corriger manuellement les conflits. Cependant, la représentation des modèles dans les fichiers est rarement sérialisé. Par exemple,*ecore* du framework Eclipse Modeling écrit les éléments d'un modèles dans l'ordre d'ajout lors de la création de celui-ci.

Pour ce modèle:

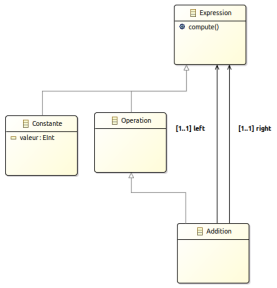

Figure 3: Méta-modèle d'une expression

Ecore peut le représenter de plusieurs manières:

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="demo" nsURI="http://www.example.org/demo" nsPrefix="demo">
  <eClassifiers xsi:type="ecore:EClass" name="Constante" eSuperTypes="#//Expression">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="valeur"
    eType="ecore:EDataType␣http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Operation" eSuperTypes="#//Expression"/>
  <eClassifiers xsi:type="ecore:EClass" name="Addition" eSuperTypes="#//Operation">
    <eStructuralFeatures xsi:type="ecore:EReference" name="right" lowerBound="1" eType="#//Expression"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="left" lowerBound="1" eType="#//Expression"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Expression">
    <eOperations name="compute"/>
  </eClassifiers>
</ecore:EPackage>
```

ou par exemple:

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="demo" nsURI="http://www.example.org/demo" nsPrefix="demo">
    <eClassifiers xsi:type="ecore:EClass" name="Expression">
    <eOperations name="compute"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Constante" eSuperTypes="#//Expression">
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="valeur"
        eType="ecore:EDataType␣http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Operation" eSuperTypes="#//Expression"/>
  <eClassifiers xsi:type="ecore:EClass" name="Addition" eSuperTypes="#//Operation">
    <eStructuralFeatures name="left" xsi:type="ecore:EReference"  lowerBound="1" eType="#//Expression"/>
    <eStructuralFeatures name="right" xsi:type="ecore:EReference"  lowerBound="1" eType="#//Expression"/>
  </eClassifiers>

</ecore:EPackage>
```

Pour de gros modèles, il devient vite difficile de comparer manuellement deux modèles en texte. Il est donc intéressant de normaliser l'écriture de ceux-ci dans les fichiers. Par exemple, le modèle précédant deviendrait:

```
<ecore:EPackage
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    xmi:version="2.0"
    name="demo"
    nsURI="http://www.example.org/demo"
    nsPrefix="demo">
    <eClassifiers
        xsi:type="ecore:EClass"
        name="Constante"
        eSuperTypes="#//Expression">
        <eStructuralFeatures
            xsi:type="ecore:EAttribute"
            name="valeur"
            eType="ecore:EDataType␣http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    </eClassifiers>
    <eClassifiers
        xsi:type="ecore:EClass"
        name="Operation"
        eSuperTypes="#//Expression"/>
    <eClassifiers
        xsi:type="ecore:EClass"
        name="Addition"
        eSuperTypes="#//Operation">
        <eStructuralFeatures
            xsi:type="ecore:EReference"
            name="left"
            lowerBound="1"
            eType="#//Expression"/>
        <eStructuralFeatures
            xsi:type="ecore:EReference"
            name="right"
            lowerBound="1"
            eType="#//Expression"/>
    </eClassifiers>
    <eClassifiers
        xsi:type="ecore:EClass"
        name="Expression">
        <eOperations
            name="compute"/>
```

```
    </eClassifiers>
</ecore:EPackage>
```

Ici, chaque élément est trié par ordre alphabétique, par rapport à sa balise, ses attributs, et ses enfants.
Le code utilisé pour trier le fichier est donné en annexe.
Cela permet une facilité à lire le fichiers, et à le comparer avec une autre version de lui.
Afin d'être certain que chaque version du fichier dans chaque commit de git soit bien normalisée, on peut créer un **hook** de **pre-commit**. A chaque fois que l'utilisateur commit en local, il va exécuter le script. Ce script ce place dans le dossier *.git/hooks*. Il doit se nommer *pre-commit* et être exécutable (*chmod u+x pre-commit*).
Par exemple:

```
#!/bin/sh

# An example pre-commit hook script to run xmlsort.py on each file in the commit.
# Called by "git commit" before creating a commit.

# Function to execute xmlsort.py on a file
xmlsort_file() {
    python3 .git/hooks/xmlsort.py -v "$1"
}

pwd
# Iterate over the staged files
for file in $(git diff --name-only --cached); do
    # Check if the file is an XML file
    if [ "$(file␣-b␣--mime-type␣"$file")" = "text/xml" ]; then
        # Execute xmlsort.py on the file
        xmlsort_file "$file"
        # Add the modified file back to the staging area
        git add "$file"
    fi
done

exit 0
```

Il va exécuter sur chaque fichier commit et de format xml le script défini précédemment qui normalise le fichier avant de commit. Cela permet de maintenir une cohérence pour chaque fichier, et de ne travailler qu'avec des fichiers normalisé.
Il existe d'autres **hook** pouvant être utile pour la gestion des fichiers:

- pre-push
- pre-rebase
- pre-receive
- ...

Pour plus de détails sur les différents hooks disponible, il est possible de consulter le site de Git

2.2 Fusion à distance, sur GitHub

Ce qu'on viens de voir précédemment permet de travailler correctement en local. Cependant, lorsqu'on partage ce qu'on vient d'écrire aux autres membres du projet, on passe sur une branche distance. Cette branche peut être stocké sur GitHub ou GitLab par exemple. Nous avons travaillé principalement avec GitHub, qui est plus utilisé que GitLab, mieux documenté, et possède énormément de fonctionnalité pour l'intégration continue.

2.2.1 Merge personnalisé sur GitHub

Lors qu'on fait une pull-request sur GitHub, le merge utilisé ne peut pas être changé. On peut changer la stratégie utiliser par GitHub pour la façon d'effectuer le merge avec certains paramètre, mais cela ne permet pas de le personnaliser, et ne convient pas a la fusion de modèle.

Une des possibilités est de faire des **WebHook**. Comme les hooks vu précédemment, ils ne sont effectué que lorsque certaine action sont faites. Un WebHook est une requête HTTP (POST) envoyé à un serveur. Il faut donc un serveur allumé à tout instant permettant de réceptionner les requêtes. Suite à ces requêtes, qui contiennent des infos sur les actions effectuées, le serveur peut réagir et faire des modification via l'API GitHub.

Pour créer un **WebHook**, il faut définir à qu'elle adresse est *l'endpoint* (l'URL où envoyer la requête) sur GitHub. Il est aussi possible d'ajouter un Secret, permettant d'assurer des connexions sécurisé entre GitHub et notre serveur.

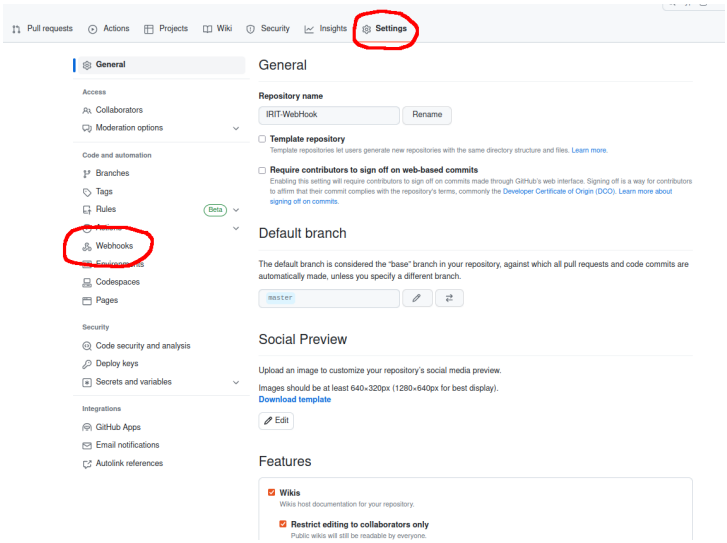

Figure 4: Location des paramètres de gestion des WebHooks

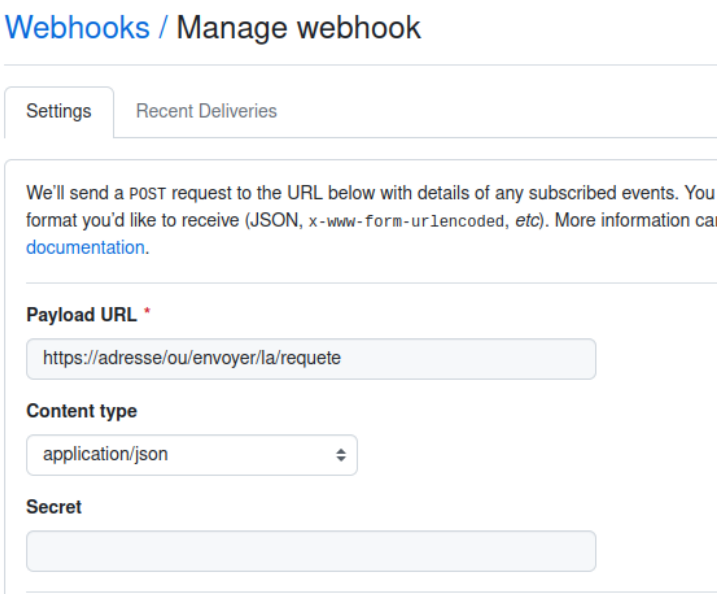

Figure 5: Définition de l'URL pour le WebHook

Il faut ensuite définir quand une requête doit être envoyé.

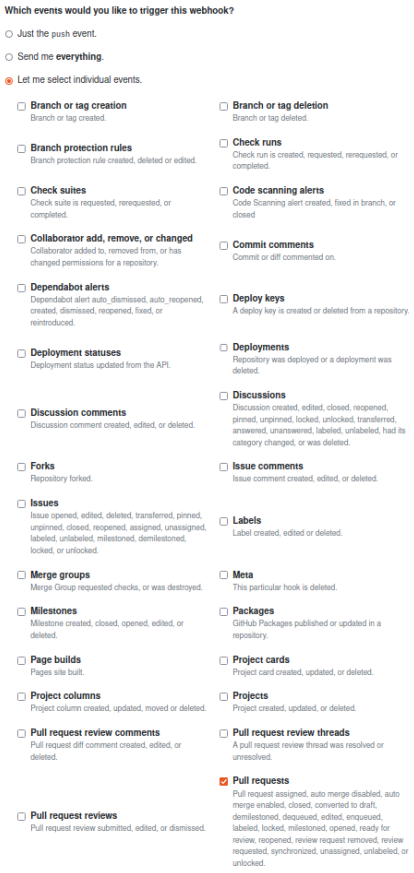

Figure 6: Définition des actions déclencheur d'une requête

2.3 Règles et limites de la fusion des modèles

2.3.1 L'exemple de Tina

La fusion est effectuée en utilisant le format .net avec 2 fichiers Tina (le dernier nœud d'ancêtre commun n'est pas utilisé pour l'instant). Voici les étapes de la fusion et l'ensemble des règles utilisées pour décider du résultat des processus de mise en correspondance et de fusion.

Analyse : Chaque fichier est analysé à la recherche de transitions, de lieux et de l'en-tête. Dans les lignes de transition, les arcs sont extraits et si les lieux associés ne sont pas définis par des lignes de lieux dans le fichier, ils seront créés. Une fois le processus terminé, le fichier est entièrement décrit par un objet *Model* contenant tous les objets de transitions et de lieux.

Appariement : Nous voulons ensuite trouver les éléments communs aux deux modèles. En fait, il s'agit d'éléments qui font partie de la même structure de base ou d'éléments qui ont été créés pour remplir le même rôle et qui seront donc fusionnés:

- Les éléments sont appariés à l'aide d'un solveur qui recherche l'ensemble des paires où la somme des similitudes est maximale. La similarité est un taux entre 0 et 1 qui est évalué différemment selon l'élément (voir ci-dessous). Les paires dont la similarité est inférieure au seuil défini (*EQ_THRESHOLD*) sont rejetées.
- Les places sont mises en correspondance sur la base de leur *nom* et de leur *label* en extrayant les mots du langage naturel qui les constituent puis en utilisant des dictionnaires de synonymes et de l'analyse sémantique.
- Les transitions peuvent être appariées sur la base de leur *nom* et de leur *label* mais aussi de leur structure en utilisant le ratio des objets *arc* qu'elles partagent, maintenant que les lieux ont été appariés (voir la comparaison des arcs ci-dessous). Pour toutes les parties de l'attribut temporel (accolades ouvertes ou fermées, temps de début et fin), les valeurs par défaut sont écrasées et les autres cas sont considérés comme des collisions.
- En ce qui concerne les objets *arc*, la similarité est celle de l'objet *place* associé. Comme pour les transitions, si une paire d'arcs comparés a des attributs *weight* ou *kind* en conflit, la valeur de similarité minimale (0) est attribuée, ce qui permet d'écarter toute correspondance possible pendant l'appariement. Toute différence sur ces attributs est considérée comme conflictuelle, la seule exception étant le cas d'une *weight* par défaut, qui est écrasé.

Fusion : Une fois que les objets ont été mis en correspondance, ils sont fusionnés l'un avec l'autre et leurs attributs sont unifiés:

- Lorsqu'il existe des équivalences (c'est-à-dire que *name* et *label* sont considérés comme équivalents puisqu'ils ont été appariés), l'avantage est donné au destinataire de la fusion pour écraser la valeur de l'autre, afin de conserver la cohérence. Toutes les valeurs par défaut sont écrasées comme décrit ci-dessus.
- Une fois la fusion terminée, l'objet *Model* destinataire est le résultat de la fusion. Toutes les informations sont écrites dans un fichier merged.txt.

Règles qui ont émergé de ce travail sur Tina:

- Les noms (transitions et lieux) doivent être uniques dans le champ d'application du fichier.
- Les noms ne doivent pas contenir les caractères suivants :

```
NORMAL_ARC = "*"
TEST_ARC = " ?"
INHIBITEUR_ARC = "?-"
STOPWATCH_ARC = " !"
STOPWATCH_INHIBITOR_ARC = "!-"

LABEL_SEPARATION = " :"
DEFAULT_T1 = "w"
PL_SEPARATION = "->"
```

- Les noms ne doivent pas être exactement les marqueurs pour les éléments : "pl", "tr", "an", ...
- Il est recommandé de donner des noms et des étiquettes significatifs mais simples pour une meilleure correspondance des mots.

3 Représentation d'un modèle complexe

Jusqu'à maintenant, un modèle est représenté par un unique fichier, écrit dans un unique langage. En réalité, un modèle est bien plus complexe, et est souvent composé de sous-modèle. Chaque sous-modèle étant lui-même un modèle, cela crée une arborescence. Chaque modèle représente quelque chose de différent par rapport aux autres, ils correspondent donc souvent à des corps de métiers différents, et ne sont alors pas écrit dans le même langage que les autres. De plus, certains modèles ne peuvent être étudier que en commun avec d'autres, il faut alors créer des connexions entre les différents modèles. La fusion, le stockage et la simulation d'un modèle se complexifie alors.

3.1 Méta-modèle d'un stockage d'un modèle

Nous allons d'abord étudier comment stocker un modèle complexe, ainsi que les informations pour les tester, ou les simuler. Faisons une liste des points à prendre en compte:

- Un modèle complexe possède une racine : le modèle final représentant le total de tout les sous-modèles, qui correspond au M0 (la réalité). On l'appellera le modèle "Global".
- Comme précisé précédemment, un modèle complexe et représenté par un ou plusieurs sous modèles. l y a alors deux types de modèles: ceux qui sont composés de sous-modèle, et ceux qui ne sont pas subdivisés, et où le modèle sera alors directement représenté dans son répertoire. On les nommera respectivement des modèles "Node" et des modèles "Leaf".
- On peut parfois combiner les sous-modèles afin de faciliter la simulation, la visualisation du modèle ou les tests de celui. Une Node peut donc posséder un **moteur de transformation**. La transformation devrait être horizontales. Elle peut cependant être endogènes comme exogènes, et peut produire plusieurs modèles de sortie (Many-to-Many). Pour plus d'information, sur la transformation de modèle, voici le lien Wikipédia sur ceux-ci.
- Un modèle doit pouvoir être testé, et ceux en dehors d'une simulation. La classe Modèle doit donc avoir une banque de test contenant différents test, exécutable sur le modèle.
- Enfin, on veut pouvoir "simuler" un modèle, par exemple en temps réel. Il faut donc un processus de simulation attaché à une Leaf. Pour simuler une Node, il faut effectuer une co-simulation. On doit donc alors simuler chaque Node, grâce à un **Orchestrator**. Pour connecter les différents sous-modèles ensemble, il est nécessaire de définir des points de connections, qui va représenter les entrées/sorties d'une simulation ou d'une co-simulation. Ce sera à l'orchestrator d'implémenter les liens entre ces points de connexion.

On obtient alors un méta-modèle ressemblant à ceci :

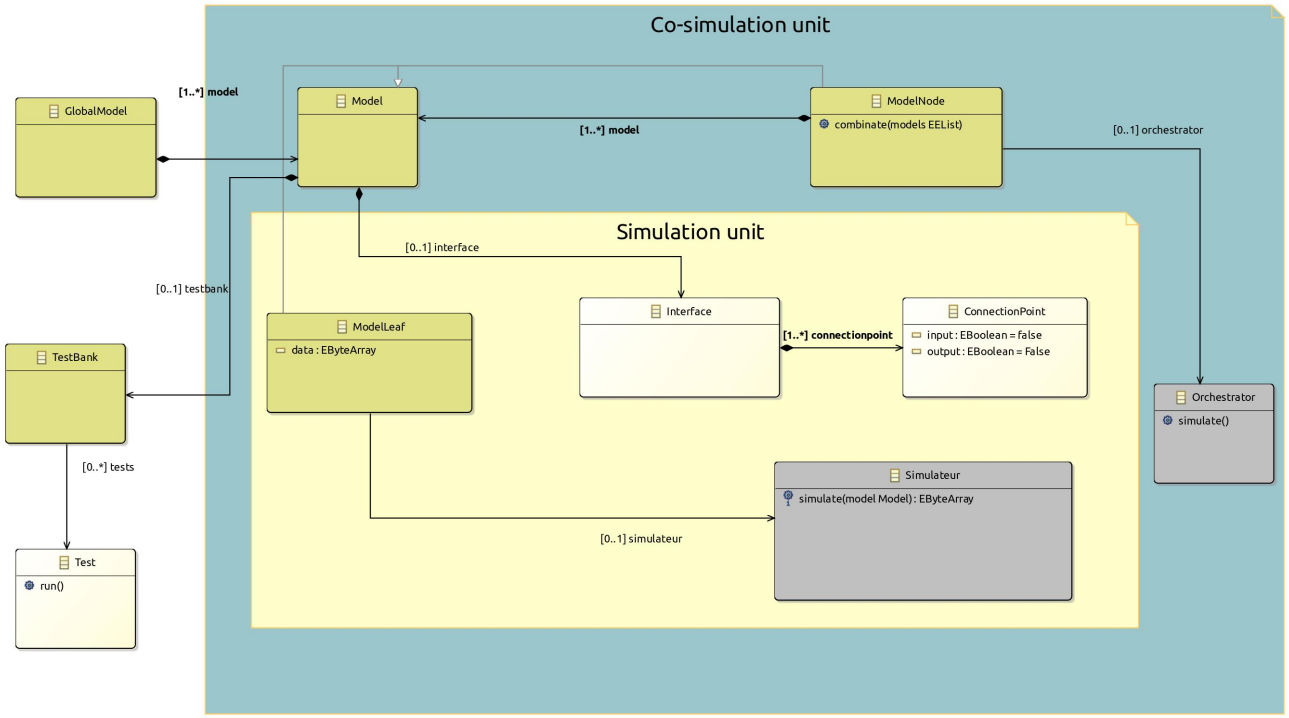

Figure 7: Méta-modèle de l'architecture d'un modèle complexe

3.2 Simulation d'un modèle complexe

Une fois notre modèle représenté correctement, il est souvent nécessaire de pouvoir le tester/simuler afin de vérifier d'éventuelle erreur ainsi que sa conformité par rapport à la réalité. Notre modèle contenant donc des sous-modèles, il est nécessaire de faire de la co-simulation.

3.2.1 Standard FMI

Le **FMI** définit un format standard pour l'échange de modèles et fournit une interface de programmation (API) qui permet aux outils de modélisation et de simulation de communiquer entre eux. Cela permet aux utilisateurs de combiner des modèles provenant de différentes sources, de les interconnecter et de les simuler ensemble.

Les modèles **FMI** sont encapsulés dans ce qu'on appelle des "Functional Mock-up Units" (**FMU**), qui sont des fichiers autonomes contenant toutes les informations nécessaires pour exécuter le modèle. Les **FMU** peuvent être utilisés dans divers environnements de simulation qui prennent en charge le standard **FMI**.

Le **FMI** prend en charge différents types de modèles, y compris les modèles continus, les modèles à événements discrets et les modèles hybrides. Il fournit également des fonctionnalités avancées telles que l'accès aux paramètres et variables du modèle, la manipulation des conditions initiales, l'enregistrement des résultats de simulation, etc.

FMI est une norme bien répandu dans le milieu professionnel: voici une liste d'outils utilisant ce standard.

La co-simulation en utilisant la norme FMI ce passe en plusieurs étapes:

1. **Exporter les modèles en FMUs**. Cela peut être fait à partir d'outil tel que Dymola, Simulink, Catia Systems, ... Le FMU (Functional Mock-up Unit) contient toutes les informations nécessaires pour décrire le modèle de simulation, y compris les équations, les paramètres, les variables d'entrée et de sortie, ainsi que les configurations spécifiques du modèle. Il s'agit d'un package autonome qui encapsule le modèle et permet son intégration dans un environnement de co-simulation compatible avec le standard FMI. Le FMU peut être exporté dans deux versions du standard FMI : FMI 1.0 et FMI 2.0. Dans FMI 1.0, le FMU est généralement un fichier binaire avec une extension ".fmu". Dans FMI 2.0, le FMU peut être un fichier binaire ou un fichier au format XML avec une extension ".fmu". Le format FMU est conçu pour assurer la portabilité des modèles et faciliter leur utilisation dans différents outils de simulation compatibles avec le standard FMI.

2. **Configurer l'environnement de co-simulation**. Il existe plusieurs outils et plateformes logicielles qui prennent en charge le standard FMI et vous permettent de créer un environnement de co-simulation. Certains exemples courants sont Dymola, SimulationX, JModelica.org, OpenModelica, ou encore la bibliothèque FMU de Modelon.

4 Annexe

4.1 Script de tri des fichiers ou dossiers xml

Le script ne trie que les fichiers xml. Ici, un fichier xml est repéré si il contient ¡. Cela peut être améliorer.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from lxml import etree
import glob
import os
import argparse
import xml.dom.minidom



def format_node(node):
    if isinstance(node, str):
        # N ud texte
        if node.strip():
            return [node.strip()]
        else:
            return None
    elif isinstance(node, xml.dom.minidom.Element):
        #  l ément DOM
        lines = []
        lines.append(f"<{node.tagName}")
        for attr_name, attr_value in node.attributes.items():
            lines.append(f'␣␣␣␣{attr_name}="{attr_value}"')


        if len(node.childNodes) > 1:
            lines[len(lines) - 1] += ">"
            child_nodes = [child for child in node.childNodes if format_node(child) is not
            if child_nodes:
                for child in child_nodes:
                    child_lines = format_node(child)
                    lines.extend(f"␣␣␣␣{line}" for line in child_lines)

            lines.append(f"</{node.tagName}>")
        else:
            lines[len(lines) - 1] += "/>"
        return lines

    return None

# Fonction récursive pour trier les éléments et sous-éléments
def trier_elements(element, nsmap):
    element[:] = sorted(
        element,
        key=lambda child: (
            child.tag,
            sorted(child.attrib.items()),
```

```
            child.text,
            [trier_elements(e, nsmap) for e in child]
        )
    )
    # Réattribuer les préfixes de namespace d'origine aux éléments triés
    for child in element:
        child.attrib.update({k: v for k, v in child.attrib.items() if k.startswith("{")})
        trier_elements(child, nsmap)

def trier_file(path):

    # Charger le fichier XML
    tree = etree.parse(path)
    root = tree.getroot()
    nsmap = root.nsmap

    # Trier tous les éléments et sous-éléments
    trier_elements(root, nsmap)

    # Convertir l'arbre lxml en arbre DOM
    xml_string = etree.tostring(root, encoding='utf-8')
    dom = xml.dom.minidom.parseString(xml_string)

    # Générer une version formatée du fichier XML trié
    formatted_lines = format_node(dom.documentElement)
    formatted_xml = "\n".join(formatted_lines)

    # Enregistrer le fichier XML trié et formaté
    with open(path, 'w') as f:
        f.write(formatted_xml)


# Charger chaque fichier XML dans le dossier courant et les sous-dossiers
def trier_doss(path):
    for file in glob.glob(path + '/*'):
        if os.path.isfile(file):

            # Check the data in file to see if it contains some XML
            with open(file, 'r') as f:
                data = f.read()
                if not '<' in data:
                    continue
            trier_file(file)


    # Pour chaque sous-dossier, appeler la fonction trier_doss
    for folder in glob.glob(path + '/*'):
        if os.path.isdir(folder):
            trier_doss(folder)



if __name__ == "__main__":

    parser = argparse.ArgumentParser(description='Trier les éléments et sous-éléments d\'un
    parser.add_argument('file', help='Chemin vers le dossier ou fichier')
```

```
# Option verbose
parser.add_argument('-v', '--verbose', action='store_true', help='Afficher les informat
args = parser.parse_args()
file = args.file
# Appeler la fonction trier_doss avec le dossier courant
if os.path.isfile(file):
    # Check the data in file to see if it contains some XML
    with open(file, 'r') as f:
        data = f.read()
        if '<' in data:
            trier_file(file)
else:
    trier_doss(file)
```

References