Stage au LAAS

Gwenn Le Bihan
gwenn.lebihan@etu.inp-n7.fr
ENSEEIHT

4 septembre 2025

Introduction

Ce stage porte sur l'intégration de Nix et NixOS dans les processus de développement et de déploiement logiciel dans le domaine robotique au sein du LAAS. Nix, le *package manager*, et NixOS, l'OS, sont des technologies permettant une reproductibilité, une qualité importante dans le monde de la recherche.

J'ai été aussi amenée à travailler sur la création d'un *plugin* pour Gazebo, un logiciel de simulation robotique, pour l'utiliser avec le *SDK* d'un robot de Unitree.

Table des matières

[1] Domain-Specific Language

1 Étude bibliographique I

1.1 Reproductibilité

1.1.1 État dans le domaine de la programmation

La différence entre une fonction au sens mathématique et une fonction au sens programmatique consiste en le fait que, par des raisons de practicité, on permet aux functionS des langages de programmation d'avoir des *effets de bords*. Ces effets affectent, modifient ou font dépendre la fonction d'un environnement global qui n'est pas explicitement déclaré comme une entrée (un argument) de la fonction en question [1].

Cette liberté permet, par exemple, d'avoir accès à la date et à l'heure courante, interagir avec un système de fichier d'un ordinateur, générer une surface pseudo aléatoire par bruit de Perlin, etc.

Mais, en contrepartie, on perd une équation qui est fondamentale en mathématiques:

$$\forall E, F, \forall f : E \to F, \forall (e_1, e_2) \in E^2, e_1 = e_2 \Rightarrow f(e_1) = f(e_2) \tag{1}$$

En programmation, on peut très facilement construire un f qui ne vérifie pas ceci:

```
from datetime import date

def f(a):
  return date.today().year + a
```

Selon l'année dans laquelle nous sommes, `f`(0) n'a pas la même valeur.

De manière donc très concrète, si cette fonction `f` fait partie du protocole expérimental d'une expérience, cette expérience n'est plus reproductible, et ses résultats sont donc potentiellement non vérifiables, si le papier est soumis le 15 décembre 2025 et la *peer review* effectuée le 2 janvier 2026.

1.1.2 Contenir les effets de bords

En dehors du besoin de vérifiabilité du monde de la recherche, la reproductibilité est une qualité recherchée dans certains domaines de programmation [2]

Il existe donc depuis longtemps des langages de programmation dits *fonctionnels*, qui, de manière plus ou moins stricte, limite les effets de bords. Certains langages font également la distinction entre une fonction *pure*[2] et une fonction classique [3]. Certaines fonctions, plutôt appelées *procédures*, sont uniquement composées d'effet de bord puisqu'elle ne renvoie pas de valeur [4]

1.1.3 État dans le domaine de la robotique

En robotique, pour donner des ordres au matériel, on intéragit beaucoup avec le monde extérieur (ordres et lecture d'état de servo-moteurs, flux vidéo d'une caméra, etc), souvent dans un langage plutôt bas-niveau, pour des questions de performance et de proximité abstractionnelle au matériel

De fait, les langages employés sont communément C, C++ ou Python[3] [5], des langages bien plus impératifs que fonctionnels [6].

[2] sans effets de bord

[3] Il arrive assez communément d'utiliser Python, un langage haut-niveau, mais c'est dans ce cas à but de prototypage, et le code contrôlant les moteurs est écrit dans un langage bas niveau plus appelé par Python par FFI

L'idée de s'affranchir d'effets de bords pour rendre les programmes dans la recherche en robotique reproductibles est donc plus utopique que réaliste.

1.1.4 Environnements de développement

Cependant, ce qui fait un programme n'est pas seulement son code: surtout dans des langages plus anciens sans gestion de dépendance simple, les dépendances (bibliothèques) du programme, ainsi que l'environnement et les étapes de compilation de ce dernier, représentent également une partie considérable de la complexité du programme (par exemple, en C++, on utilise un outil générant des fichiers de configuration pour un autre outil qui à son tour configure le compilateur de C++ [7])

C'est cette partie que Nix, le gestionnaire de paquet, permet d'encapsuler et de rendre reproductible. Dans ce modèle, la compilation (et de manière plus générale la construction, ou *build*) du projet est la fonction que l'on veut rendre pure. L'entrée est le code source, et le résultat de la fonction est un binaire, qui ne doit dépendre que du code source.

$$\forall\ \text{src}, \text{bin}, \forall f \in \text{bin}^{\text{src}}, \forall (P_1, P_2) \in \text{src}^2, P_1 = P_2 \Rightarrow f(P_1) = f(P_2) \tag{2}$$

Ici, P_1 et P_2 sont deux itérations du code source (src) du programme. Si le code source est identique, les binaires résultants de la compilation (f) sont égaux, au sens de l'égalité bit à bit.

On a la proposition (1), avec $E = \text{src}$, l'ensemble des code source possibles pour un langage, et $F = \text{bin}$, l'ensemble des binaires éxécutables

Nix ne peut pas garantir que le programme sera sans effets de bords au *runtime*, mais vise à le garantir au *build-time*.

1.2 Nix, le gestionnaire de paquets pur

1.2.1 Un *DSL*[4] fonctionnel

Une autre caractéristique que l'on trouve souvent dans la famille de langages fonctionnels est l'omniprésence des *expressions*: quasi toute les constructions syntaxiques forment des expressions valides, et peuvent donc servir de valeur

`def g(x, y):`
`  if y == 5:`
`  x = 6`
`  else:`
`  x = 8`
`  return f(x)`
	`let g x y = f (`
	`  if y = 5 then`
	`  6`
	`  else`
	`  8`
	`)`

Python (if et else sont des instructions)	**OCaml** (if et else forment une expression)

Afin de décrire les dépendances d'un programme, l'environnement de compilation, et les étapes pour le compiler (en somme, afin de définir le $f \in \text{bin}^{\text{src}}$), Nix comprend un langage d'expressions [8]. Un fichier `.nix` définit une fonction, que Nix sait exécuter pour compiler le code source.

Expression d'une fonction en Python	En Nix
`lambda f(a): a + 3`	`{ a }: a + 3`

Voici un exemple de définition d'un programme, appelée *dérivation* dans le jargon de Nix:

```
{
 src-odri-masterboard-sdk,
```

[4] Domain-Specific Language

```
  lib,
  stdenv,
  jrl-cmakemodules,
  cmake,
  python3Packages,
  catch2_3,
}:

stdenv.mkDerivation {
  pname = "odri_master_board_sdk";
  version = "1.0.7";

  src = src-odri-masterboard-sdk;

  preConfigure = ''
    cd sdk/master_board_sdk
  '';

  doCheck = true;

  cmakeFlags = [
    (lib.cmakeBool "BUILD_PYTHON_INTERFACE" stdenv.hostPlatform.isLinux)
  ];

  nativeBuildInputs = [
    jrl-cmakemodules
    python3Packages.python
    cmake
  ];

  buildInputs = with python3Packages; [ numpy ];

  nativeCheckInputs = [ catch2_3 ];

  propagatedBuildInputs = with python3Packages; [ boost ];
}
```

La dérivation ici prend en entrée le code source (`src-odri-masterboard-sdk`), ainsi que des dépendances, que ce soit des fonctions relatives à Nix même (comme `stdenv.mkDerivation`) pour simplifier la définition de dérivation, ou des dépendances au programmes, que ce soit pour sa compilation ou pour son exécution (dans ce dernier cas de figures, les dépendances sont inclues ou reliées au binaire final)

1.2.2 Un ecosystème de dépendances

Afin de conserver la reproductibilité même lorsque l'on dépend de libraries tierces, ces dépendances doivent également avoir une compilation reproductible: on déclare donc des dépendances à des *packages* Nix, disponibles sur *Nixpkgs* [9].

Parfois donc, écrire un paquet Nix pour son logiciel demande aussi d'écrire les paquets Nix pour les dépendances de notre projet, si celles-ci n'existent pas encore, et cela récursivement. On peut ensuite soumettre nos paquets afin que d'autres puissent en dépendre sans les réécrire, en contribuant à *Nixpkgs* [10]

Pour ne pas avoir à compiler toutes les dépendances soit-même quand on dépend de `.nix` de *nixpkgs*, il existe un serveur de cache, qui propose des binaires des dépendances, Cachix [11]

1.2.3 Une compilation dans un environnement fixé

Certains aspects de l'environnement dans lequel l'on compile un programme peuvent faire varier le résultat final. Pour éviter cela, Nix limite au maximum les variations d'environnement. Par exemple, la date du système est fixée au 0 UNIX (1er janvier 1990): le programme compilé ne peut pas dépendre de la date à laquelle il a été compilé.

Quand le *sandboxing* est activé, Nix isole également le code source de tout accès au réseau, aux autres fichiers du système (ainsi que d'autres mesures) pour améliorer la reproductibilité [12]

Un complément utile: compiler en CI

Pour aller plus loin, on peut lancer la compilation du paquet Nix en *CI*[6], c'est-à-dire sur un serveur distant au lieu de sur sa propre machine. On s'assure donc que l'état de notre machine de développement personnelle n'influe pas sur la compilation, puisque chaque compilation est lancée dans une machine virtuelle vierge [13].

1.3 NixOS, un système d'exploitation à configuration déclarative

Une fois le programme compilé avec ses dépendances, il est prêt à être transféré sur l'ordinateur ou la carte de contrôle embarquée au robot.

Lorsqu'il y a un ordinateur embarqué, comme par exemple une Raspberry Pi [14], il faut choisir un OS sur lequel faire tourner le programme.

La encore, un OS s'accompagne d'un amas considérable de configuration des différentes parties du système: accès au réseau, drivers,…

Sur les OS Linux classiques tels que Ubuntu ou Debian, cette configuration est parfois stockée dans des fichiers, ou parfois retenue en mémoire, modifiée par l'execution de commandes.

C'est un problème assez récurrent dans Linux de manière générale: d'un coup, le son ne marche plus, on passe ½h sur un forum à copier-coller des commandes dans un terminal, et le problème est réglé… jusqu'à ce qu'il survienne à nouveau après un redémarrage ou une réinstallation.

Ici, NixOS assure que toute modification de la configuration d'un système est *déclarée* (d'où l'adjectif « déclaratif ») dans des fichiers de configurations, également écrit dans des fichiers `.nix` [15].

Ici encore, cela apporte un gain en terme de reproductibilité: l'état de configuration de l'OS sur lequel est déployé le programme du robot est, lui aussi, rendu reproductible.

1.4 Gazebo & Unitree

1.4.1 Contexte

J'ai également été approchée pour travailler sur la création d'un *plugin* pour Gazebo, un logiciel de simulation robotique [16].

Le but était de pouvoir utiliser ce logiciel de simulation open source avec un robot de la companie Unitree, le H1v2 [17], un robot humanoïde tout-usage.

1.4.2 Une base de code partiellement open-source

Une partie du code source de ce SDK n'est pas disponible, et n'est que distribué sous forme de binaires [18]. J'ai donc chercher à comprendre cette partie du code par ingénierie inverse, ce qui ne s'est pas avéré nécéssaire.

Au final, en explorant le code source du plugin pour un autre logiciel de simulation, Mujoco [19], [20], j'ai pu comprendre comment interfacer le SDK avec Gazebo.

[6] Continuous Integration, lit. intégration continue

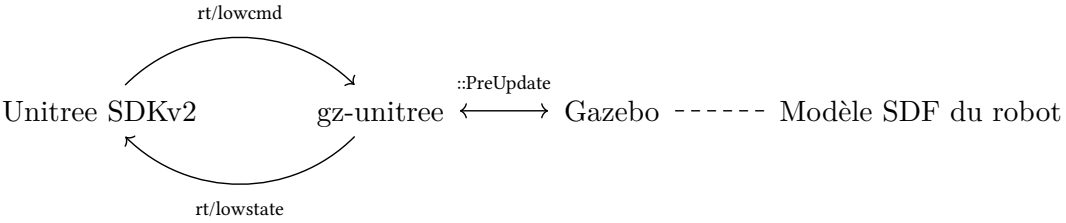
1.4.3 rt/lowstate, rt/lowcmd

Le SDK de Unitree fonctionne via des canaux DDS, une technologie de communication temps-réel bas niveau [21].

Deux de ces canaux donnent accès au contrôle (resp. à l'état) bas-niveau des moteurs (resp. capteurs) du robot: rt/lowcmd (resp. rt/lowstate).

Grâce à l'étude des paquets transmis via Wireshark, j'ai pu débugger les communications entre mon plugin, *gz-unitree*, et le SDK.

Et au final, mon plugin fonctionne, en simulant un robot H1v2 via ces deux canaux:



1.4.4 Des tests end-to-end automatisés

Je souhaitais permettre de tester le code sur simulateur de manière automatique: on push un commit modifiant une politique de contrôle du robot, et, automatiquement, en CI, un test sous simulateur est lancé. On reçoit un artéfact avec une vidéo filmant le test.

Pour faire ceci, il a fallu rendre la fonctionnalité native à Gazebo d'enregistrement vidéo automatisable: elle ne l'est pas nativement, il a donc fallu que je duplique le code du module Gazebo correspondant, afin d'y rajouter de quoi contrôler l'entregistrement vidéo via des *topics* Gazebo.

Il y a aussi un challenge lié au fait que, en CI, il n'y a pas d'interface graphique, ce qui rend le lancement de l'interface graphique de Gazebo impossible. Il faut donc simuler une interface graphique avec *XVFB*, un serveur X virtuel [22].

1.4.5 Packaging sous Nix

Le packaging sous Nix de *gz-unitree* est en cours, mais se heurte à quelques problèmes liés à l'état du packaging Nix de Gazebo lui-même: gazebo est packagé dans un *overlay* tierce, *gazebo-sim-overlay* [23], qui n'a pas mis à jour une des bibliothèques de Gaazebo depuis plus d'un an [24]

2 Journal de bord

2.1 19-23 Mai

- Mise en place de l'environnement de développement
- Documentation sur Nix le langage [8]
- Découverte de la description d'une dérivation [25] et d'un flake
- Découverte de l'infrastructure autour de nixpkgs (github, la CI, Hydra [26]…)
- Packaging en flake et CI basique (nix build) de open-dynamic-robot-initiative/{interface_controls,master-board} [27], [28]
- Début du travail de packaging de open-dynamic-robot-initiative/robot_properties_solo
- Migration de robot_properties_solo vers uv [29]

- Début du packaging de xacro sur NixOS/nixpkgs [30]
- Création d'un JSON Schema [31] pour des fichiers de configuration de robot_properties_solo et mise en place d'une CI pour les valider [32]
- Recherche autour d'une potentielle validation au runtime en C++ des fichiers de config par le JSON Schema
- Découverte des overlays Nix

2.2 26-28 Mai

- Continuation du travail précédent

2.3 2-6 Juin

- Début de recherches sur l'installation de NixOS sur Raspberry Pi [14] 400 et 5
- Flash du firmware master-board sur un testbench
- Test du packaging de odri_control_interface [27] avec les scripts de démos à l'aide d'un testbench
- Début de recherches sur la création d'un plugin Gazebo [16] communiquant avec la couche bas niveau du SDK2 [33] d'Unitree afin de simuler du code pour le robot H1 [34] dans Gazebo

2.4 9-13 Juin

- Progrès sur l'accès à la couche bas niveau du SDK2 [33]
  - Analyse via Wireshark des paquets
  - Analyse du code source du plugin Mujoco [19] fourni par Unitree

2.5 16-20 Juin

- Réussite de l'accès à la couche bas niveau du SDK2 via les définitions IDL [35] fournies par Unitree
- Documentation sur le système de plugins de Gazebo [16]
- Début de travail sur le bridge Gazebo/unitree: gz-unitree
  - Implémentation de la communication DDS [21] entre un binaire d'exemple d'utilisation du SDK2 et le plugin Gazebo

2.6 23-27 Juin

- Construction du *lowstate* à envoyer au SDK2 depuis *gz-unitree*:
- Utilisation du modèle SDF [36] du robot H1-2 [17] au lieu de H1 [34], ajout d'un sol au monde du SDF

2.7 30 Juin - 4 Juillet

- Continuation du travail: essais pour rajouter un capteur IMU sur le robot, essais pour faire fonctionner l'auto-collision

2.8 7-11 Juillet

- Capteur IMU rajouté
- Ajout du tick (temps) de simulation
- Essais d'utilisation de gz-unitree avec les politiques RL[7] de Gepetto

2.9 14-18 Juillet

- Tentatives d'amélioration des performances pour améliorer le RTF: passage de 10% à 15%
  - Parallélisation de l'envoi des messages des DDS dans un thread différent du principal
  - Optimisations classiques
- Ecriture d'une recette *Just* [37] pour configurer l'environnement de développement, sur Arch Linux ou Ubuntu
- Reproduction des résultats sur un OS et une machine différente

2.10 21-25 Juillet

- Évaluation de gazebo-sim-overlay [23] comme solution pour un packaging Nix
- Recherche sur un mode headless de gazebo suite à des erreurs de QT sous devshell Nix

2.11 28 Juillet - 1 août

- Recherche autour de l'utilisation de Gazebo dans des environnements CI/CD [38], en particulier pour capturer une simulation en vidéo

Bibliographie

[1] Brian Lonsdorf, « Professor Frisby's Mostly Adequate Guide to Functional Programming », 2015, *Github*. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://github.com/MostlyAdequate/mostly-adequate-guide/blob/master/ch03.md

[2] « Reproducible Builds ». Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://reproducible-builds.org/

[3] Fortran 2015 Committee Draft (J3/17-007r2), *ISO/IEC JTC 1/SC 22/WG5/N2137*. International Organization for Standardisation, 2017, p. 336-338. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://wg5-fortran.org/N2101-N2150/N2137.pdf

[4] « Relationship Between Routines, Functions, and Procedures », 13 janvier 2025, *IBM*. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://www.ibm.com/docs/en/informix-servers/15.0.0?topic=statement-relationship-between-routines-functions-procedures

[5] « Different Types of Robot Programming Languages », 2015, *Plant Automation Technology*. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://www.plantautomation-technology.com/articles/different-types-of-robot-programming-languages

[6] « Imperative programming: Overview of the oldest programming paradigm », 21 mai 2021, *IONOS*. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://www.ionos.com/digitalguide/websites/web-development/imperative-programming/

[7] Bill Hoffman et Kenneth Martin, *The Architecture of Open Source Applications (Volume 1) CMake*. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://aosabook.org/en/v1/cmake.html

[7] Reinforcement Learning

[8] Consulté le: 19 mai 2025. [En ligne]. Disponible sur: https://nix.dev/manual/nix/2.17/language/

[9] Consulté le: 3 septembre 2025. [En ligne]. Disponible sur: https://search.nixos.org/packages

[10] NixOS Wiki Authors, « Nixpkgs/Contributing ». Consulté le: 3 septembre 2025. [En ligne]. Disponible sur: https://wiki.nixos.org/wiki/Nixpkgs/Contributing

[11] « Cachix — Nix binary cache hosting ». Consulté le: 3 septembre 2025. [En ligne]. Disponible sur: https://www.cachix.org/

[12] « Nix (package manager) — Sandboxing ». Consulté le: 3 septembre 2025. [En ligne]. Disponible sur: https://wiki.nixos.org/wiki/Nix_(package_manager)#Internals

[13] « GitHub-hosted runners », *Github*. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://docs.github.com/en/actions/concepts/runners/github-hosted-runners

[14] Consulté le: 6 juin 2025. [En ligne]. Disponible sur: https://www.raspberrypi.com/

[15] Fernando Borretti, « NixOS for the Impatient », 7 mai 2023. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://borretti.me/article/nixos-for-the-impatient

[16] Consulté le: 6 juin 2025. [En ligne]. Disponible sur: https://gazebosim.org/

[17] Consulté le: 30 juin 2025. [En ligne]. Disponible sur: https://www.unitree.com/h1/

[18] « unitreerobotics/unitree_sdk2, main branch, lib/x86_64 ». Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://github.com/unitreerobotics/unitree_sdk2/tree/c8a71e281093593f4dcc7bceb3b3b529ff0e36b4/lib/x86_64

[19] Consulté le: 16 juin 2025. [En ligne]. Disponible sur: https://mujoco.org/

[20] Unitree Robotics, « Unitree mujoco ». Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://github.com/unitreerobotics/unitree_mujoco

[21] « DDS Interoperability Wire Protocol Specification Version 2.5 ». Consulté le: 24 juin 2025. [En ligne]. Disponible sur: https://www.omg.org/spec/DDSI-RTPS/

[22] I. David P. Wiggins The Open Group, « XVFB », *X.org*. Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://www.x.org/archive/X11R7.7/doc/man/man1/Xvfb.1.xhtml

[23] Consulté le: 21 juillet 2025. [En ligne]. Disponible sur: https://github.com/muellerbernd/gazebo-sim-overlay

[24] « update gz-msgs · Issue #2 · muellerbernd/gazebo-sim-overlay ». Consulté le: 4 septembre 2025. [En ligne]. Disponible sur: https://github.com/muellerbernd/gazebo-sim-overlay/issues/2

[25] Consulté le: 19 mai 2025. [En ligne]. Disponible sur: https://nix.dev/manual/nix/2.17/language/derivations

[26] Consulté le: 21 mai 2025. [En ligne]. Disponible sur: https://hydra.nixos.org/

[27] Consulté le: 20 mai 2025. [En ligne]. Disponible sur: https://github.com/open-dynamic-robot-initiative/odri_controls_interface

[28] Consulté le: 20 mai 2025. [En ligne]. Disponible sur: https://github.com/open-dynamic-robot-initiative/master-board

[29] Consulté le: 22 mai 2025. [En ligne]. Disponible sur: https://docs.astral.sh/uv/

[30] Consulté le: 22 mai 2025. [En ligne]. Disponible sur: https://github.com/NixOS/nixpkgs/pull/409754

[31] Consulté le: 23 mai 2025. [En ligne]. Disponible sur: https://json-schema.org/draft/2020-12/json-schema-core

[32] Consulté le: 21 mai 2025. [En ligne]. Disponible sur: https://github.com/open-dynamic-robot-initiative/robot_properties_solo

[33] Consulté le: 6 juin 2025. [En ligne]. Disponible sur: https://github.com/unitreerobotics/unitree_sdk2

[34] Consulté le: 6 juin 2025. [En ligne]. Disponible sur: https://www.unitree.com/h1/

[35] « Interface Definition Language Specification Version 4.2 ». Consulté le: 18 juin 2025. [En ligne]. Disponible sur: https://www.omg.org/spec/IDL/4.2/About-IDL

[36] « SDFormat Specification ». Consulté le: 30 juin 2025. [En ligne]. Disponible sur: http://sdformat.org/spec

[37] Consulté le: 21 juillet 2025. [En ligne]. Disponible sur: https://just.systems/

[38] Fiorella Zampetti, Vittoria Nardone, et Massimiliano Di Penta, « Problems and Solutions in Applying Continuous Integration and Delivery to 20 Open-Source Cyber-Physical Systems ». doi: http://dx.doi.org/10.1145/3524842.3527948.