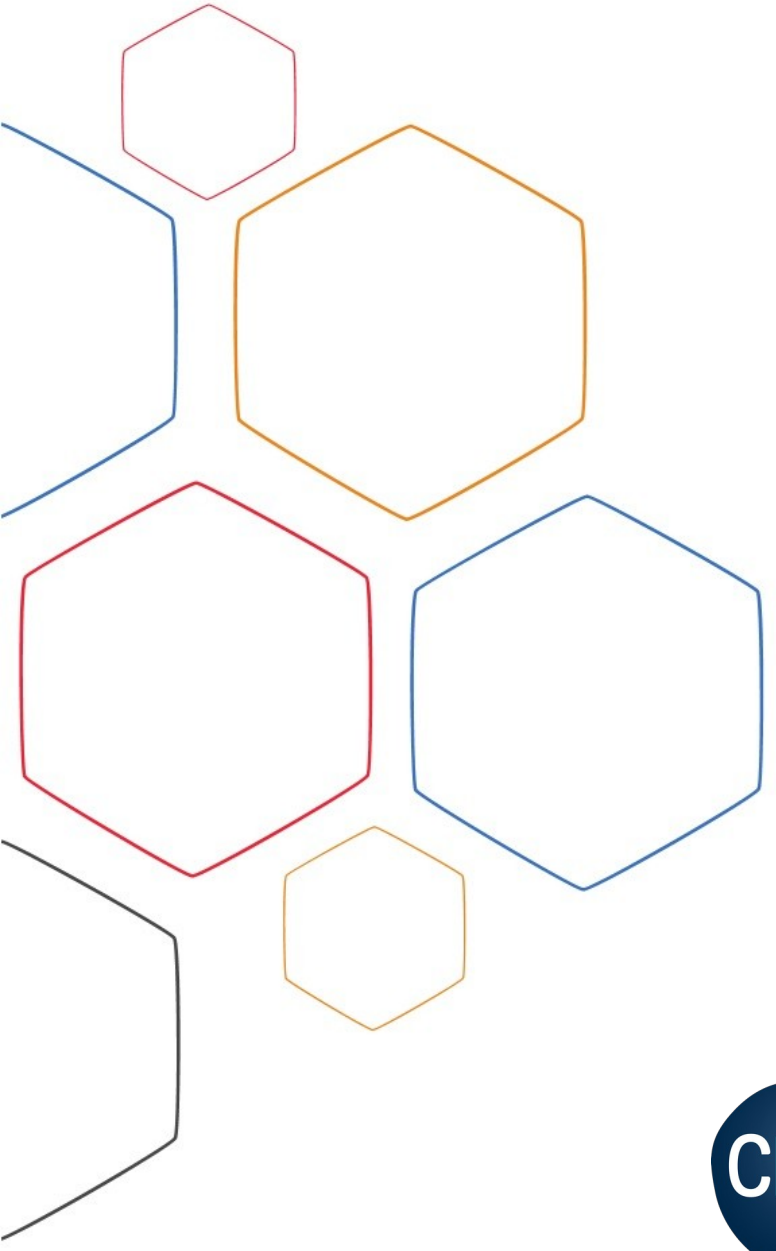


L’enfer de la gestion des dépendances

Retour d’expérience et réflexions

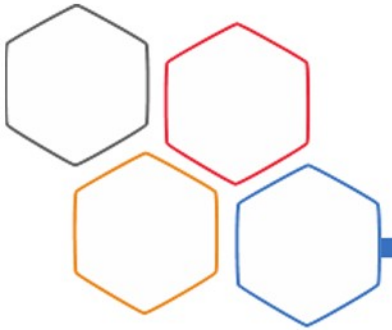
Robin Passama, Ingénieur de recherche CNRS, LIRMM

École Technologique 2RM, Strasbourg, 16 mai 2023

Contexte

- Développement des systèmes logiciels des robots (en recherche)
  - Partagent beaucoup de codes communs (bibliothèques)
  - Évoluent continuellement
  - Doivent s'adapter à des contextes changeant : OS, middleware

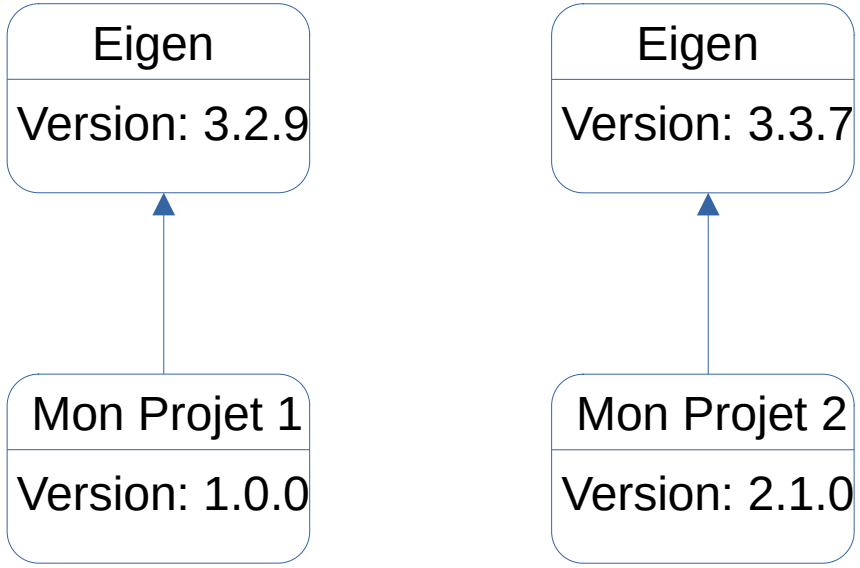
- Approche suivie au LIRMM : **petits projets** VS gros projets
  - Plus facile à maintenir, à faire évoluer, à stabiliser leur API
  - Meilleure « isolation » du travail des développeurs limite :
    - La multiplication de branches devenant souvent « non mergeables » (perte de code in fine)
    - Les effets de bords après merge (introduction de bugs)
  - MAIS : **nécessite une gestion « pointue » des dépendances**



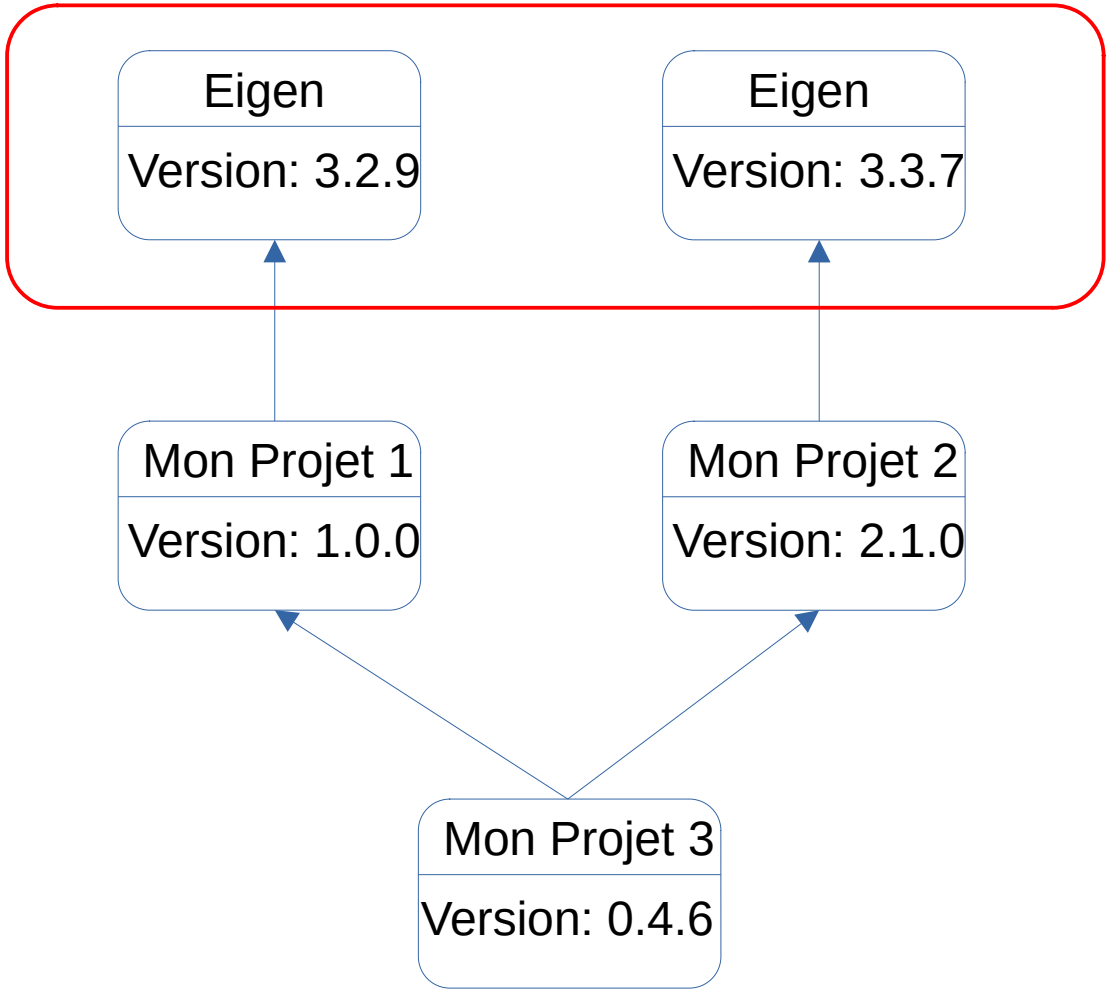
Plan

- **Gestion des versions**
- Expression des contraintes de version
- Résolution automatique des versions
- PID
- Réflexion

Gestion des versions

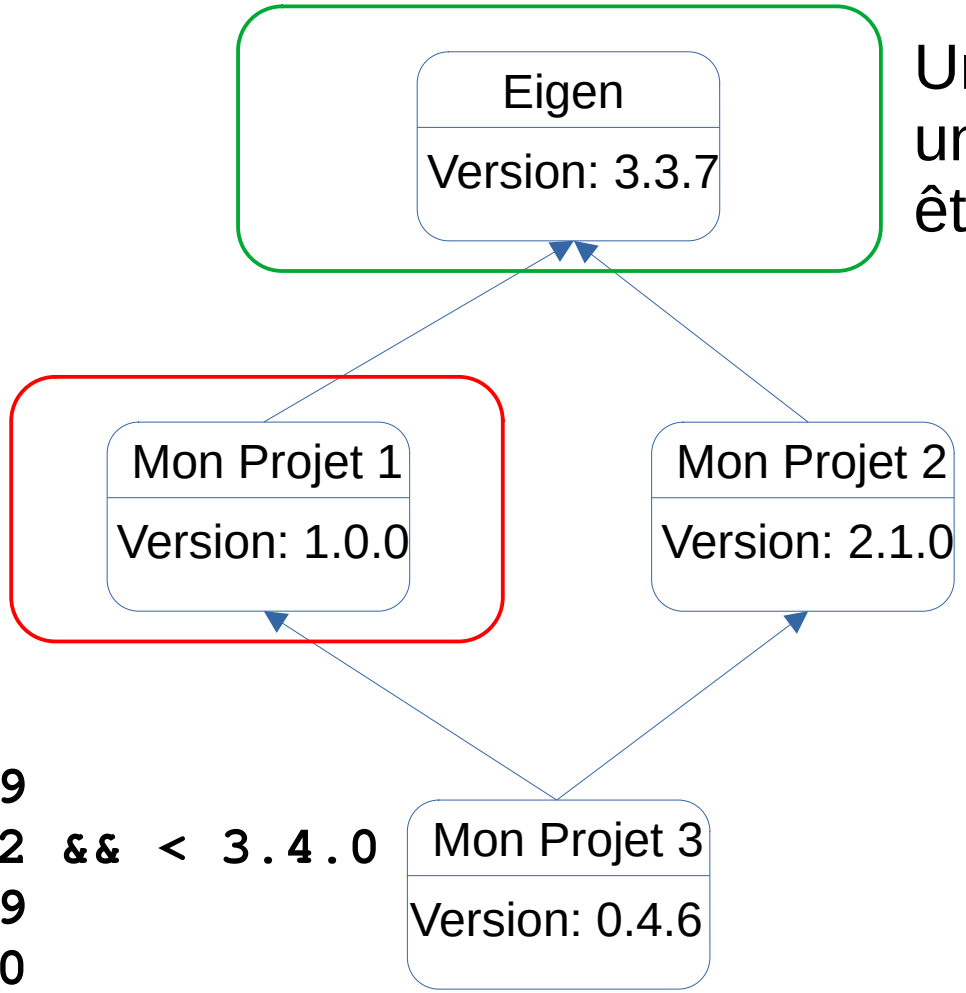


Gestion des versions



Les versions ne sont pas les mêmes !!!

Gestion des versions



Une version unique doit être trouvée

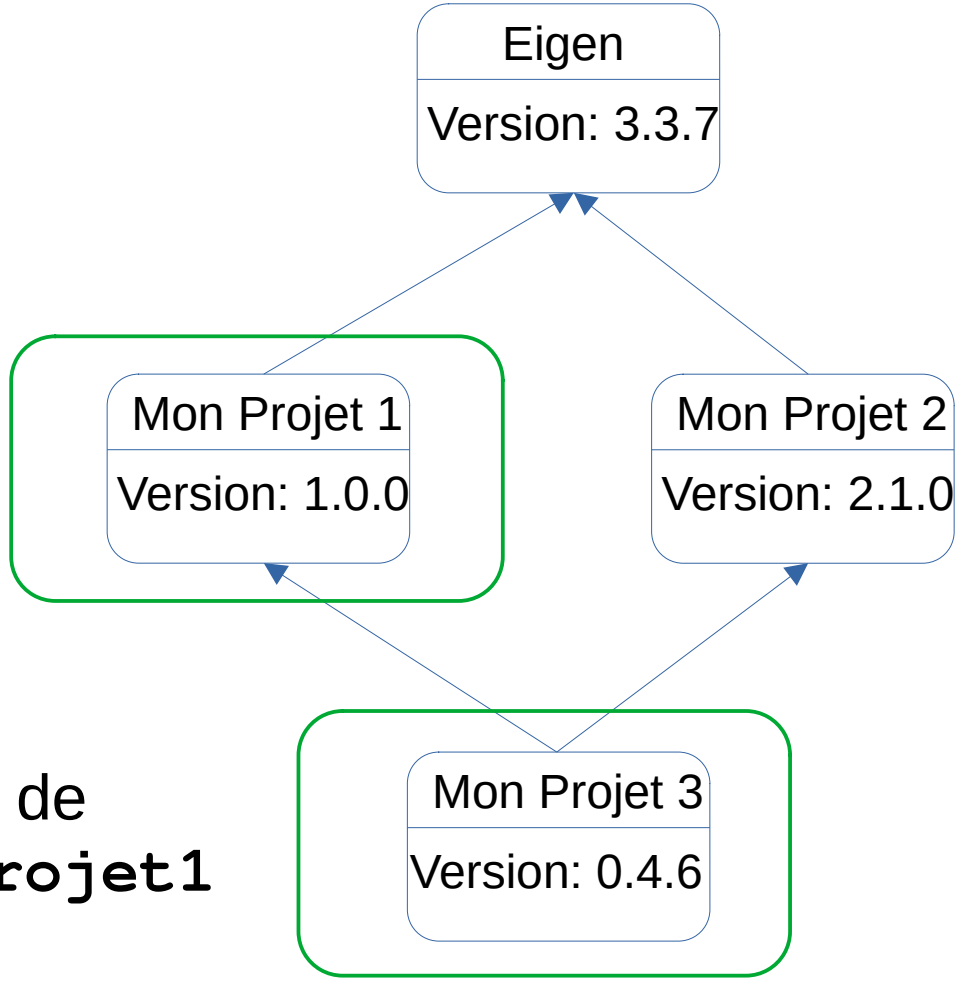
Contrainte de version sur Eigen autorise t'elle d'adapter la version utilisée ?

Exemples :

- **OK : Eigen ≥ 3.2.9**
- **OK : Eigen ≥ 3.2.2 && < 3.4.0**
- **KO : Eigen = 3.2.9**
- **KO : Eigen < 3.3.0**

Pourquoi s’embêter avec les versions ?

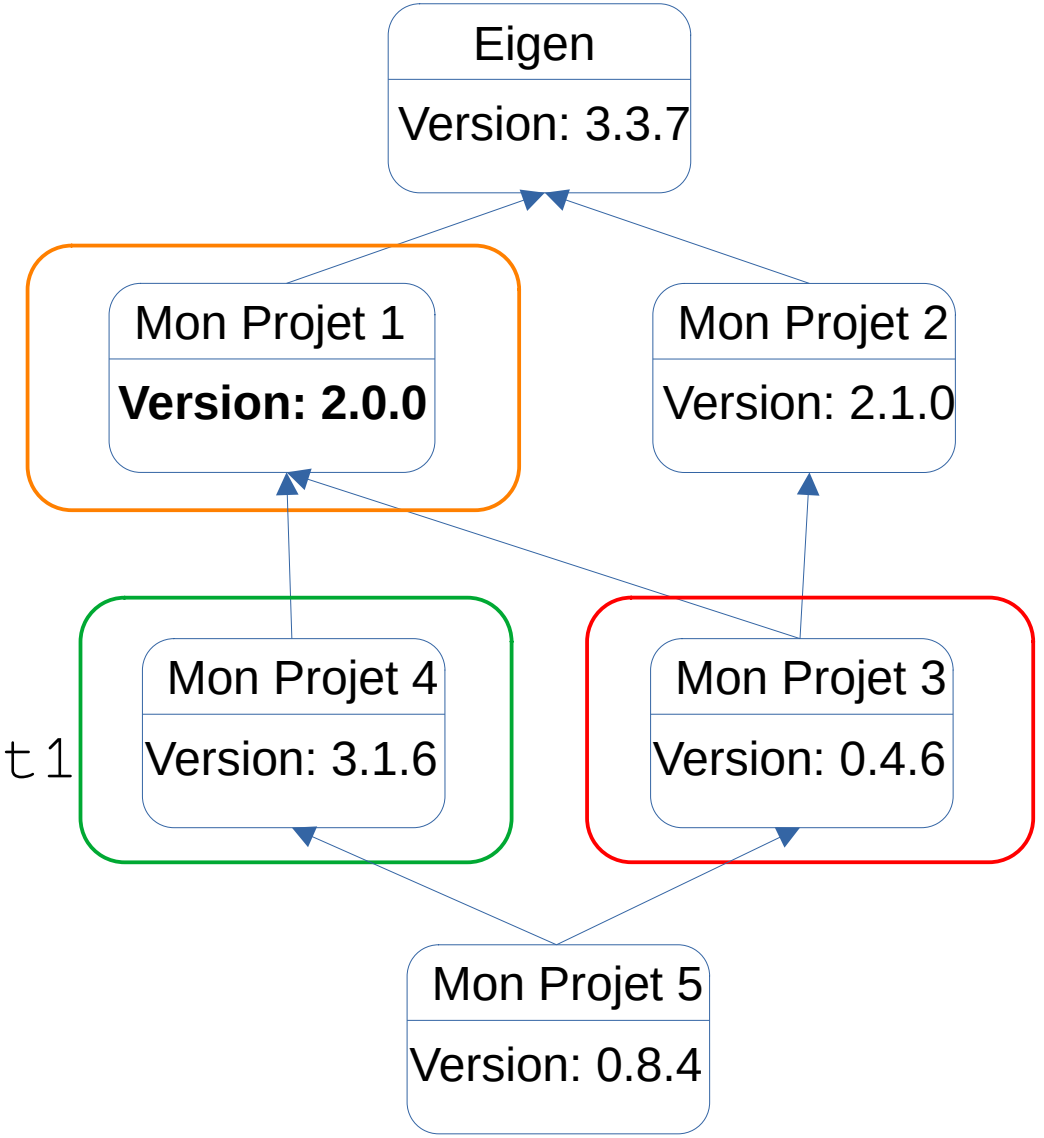
Version 1.0.0 OK !!



Pas de contrainte de version sur **`MonProjet1`**

Pourquoi s'embêter avec les versions ?

Adaptation possible **suivant les contraintes** décrites dans les projets qui l'utilisent



Développé avec l'API version 2 de `MonProjet1`

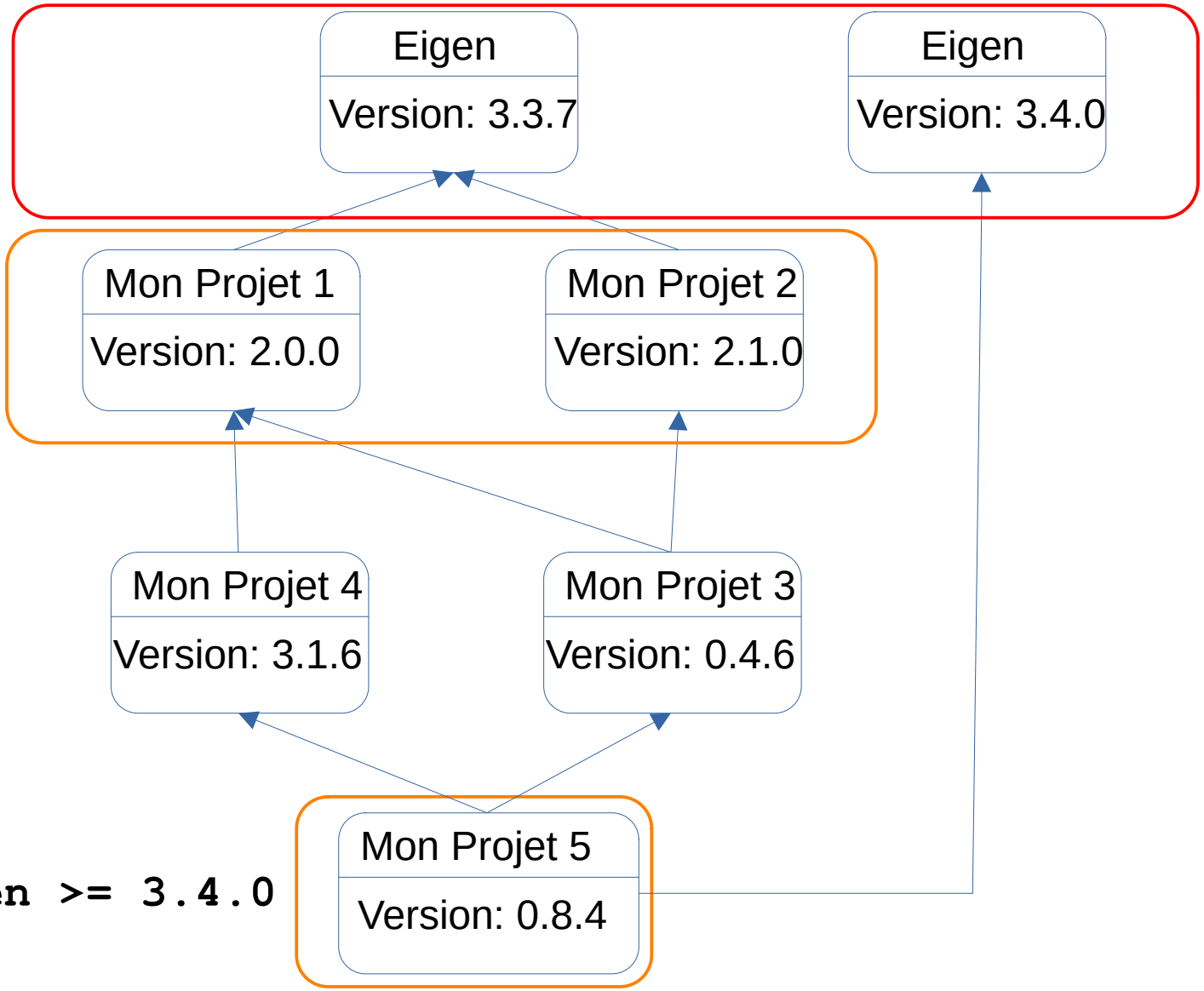
Contrainte :
`MonProjet1 ≥ 2.0.0`

Erreur : code pas prévu pour utiliser l'API version 2 de `MonProjet1`

Propagation des contraintes

Adaptation requise de la version d'Eigen utilisée : `3.4.0`

Peut être impossible !!

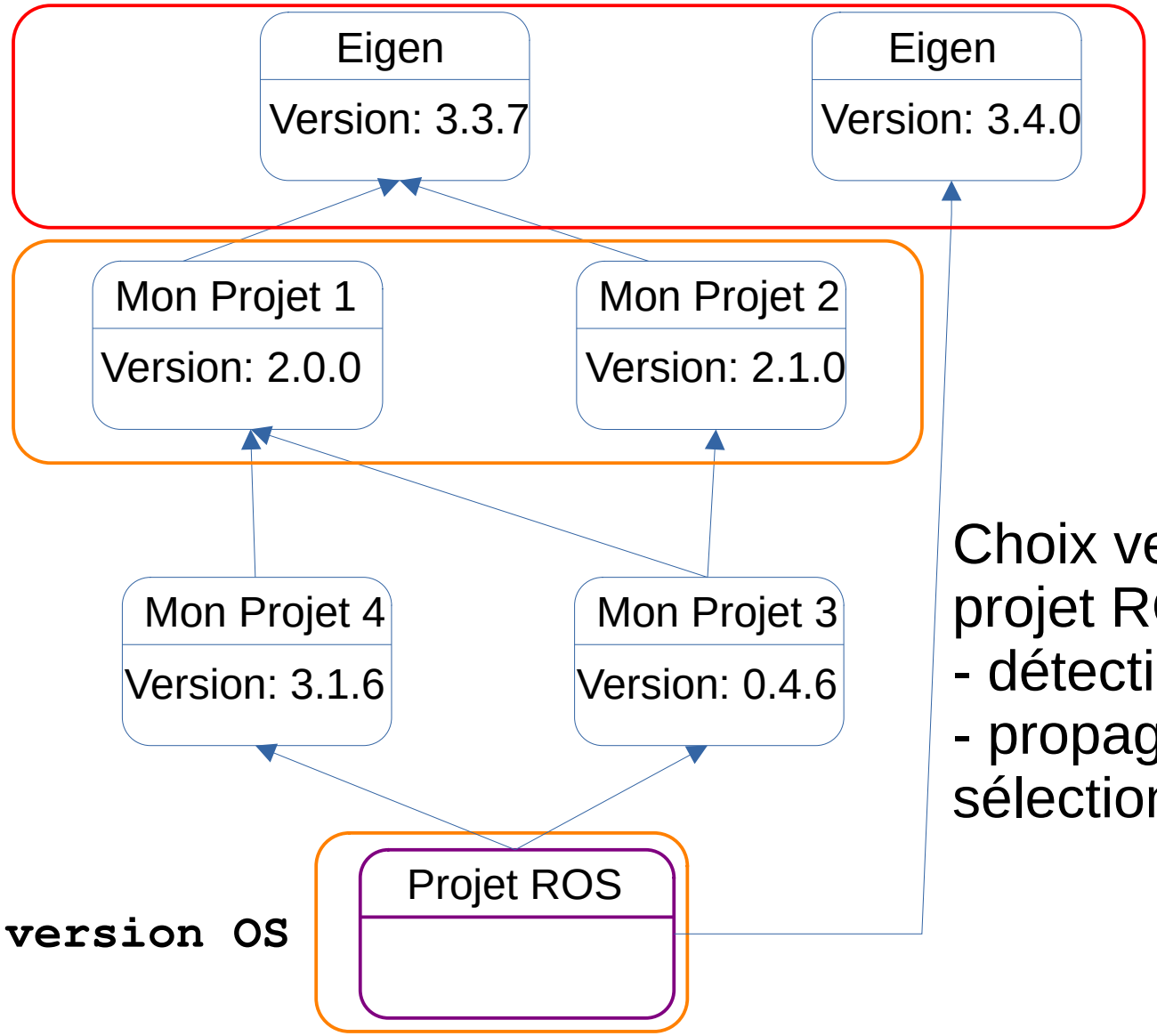


Contrainte : **`Eigen >= 3.4.0`**

Propagation des contraintes

Adaptation requise de la version d'Eigen utilisée : `3.4.0`

Peut être impossible !!



Choix version dans le projet ROS.
- détection version OS
- propagation version sélectionnée

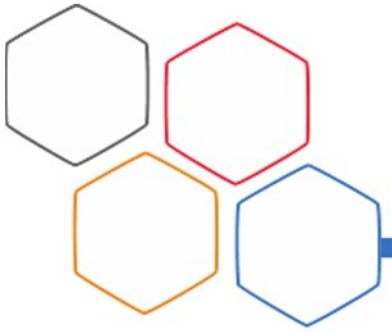
Contrainte : **`Eigen == version OS`**

Gestion des versions

- Définition explicite des contraintes : **nécessaire**
  - Permet une automatisation du processus de résolution des dépendances
  - On peut savoir **s'il n'y a pas de solution et comprendre le problème**

- Adaptation à des contraintes externes : **nécessaire**
  - Permet d'adapter les projets à des contraintes imposées (par l'OS) : **utile pour intégrer nos projets dans ROS**

- Outil de résolution automatique : **nécessaire**
  - Devient trop pénible à gérer « à la main » sur de gros projets



Plan

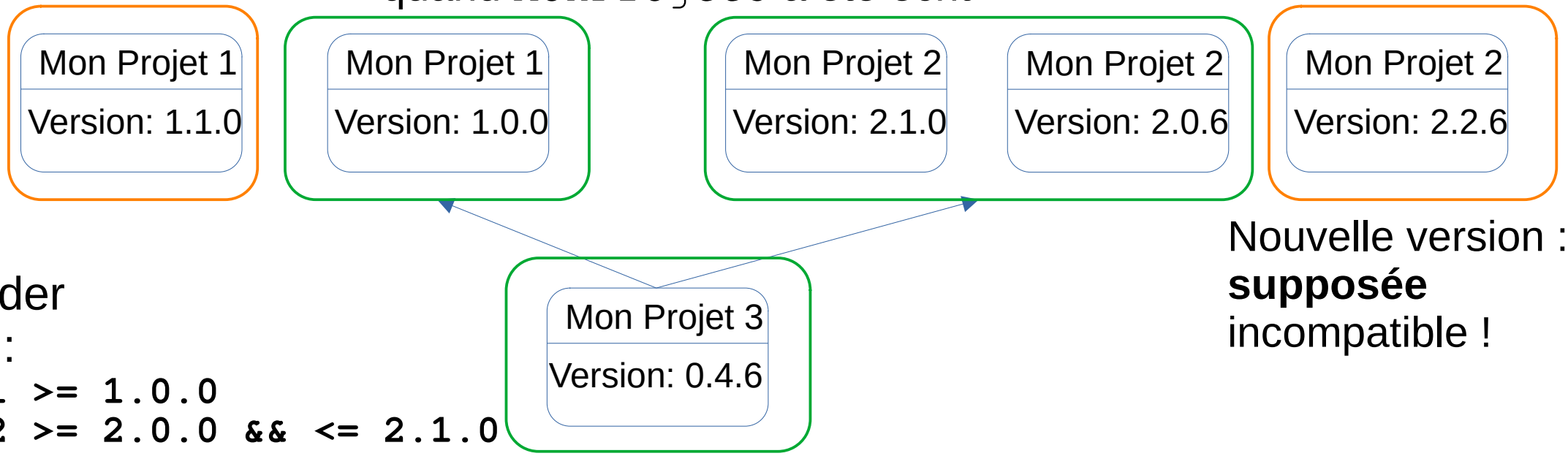
- Gestion des versions
- **Expression des contraintes de version**
- Résolution automatique des versions
- PID
- Réflexion

Expression des contraintes de version

- Problèmes :
  - On ne connaît pas les versions futures d'une dépendance
  - On ne connaît pas la **compatibilité entre les API** des versions

Nouvelle version :
supposée
compatible !

Dernières versions connues
quand `MonProjet3` a été écrit



Nouvelle version :
supposée
incompatible !

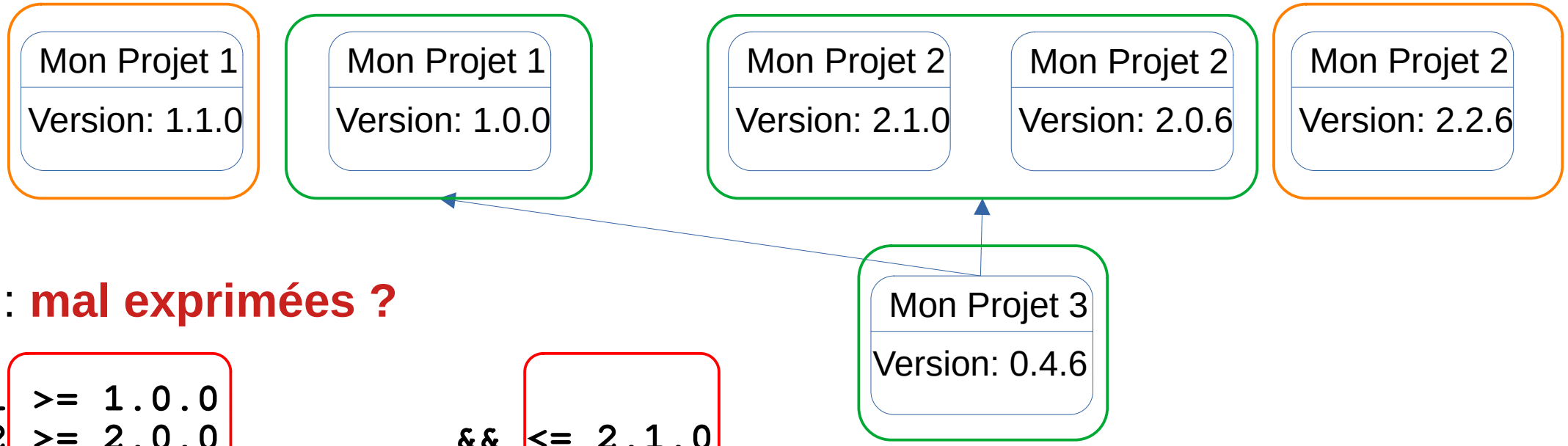
Projet à builder
Contraintes :

```
MonProjet1 >= 1.0.0
MonProjet2 >= 2.0.0 && <= 2.1.0
```

Expression des contraintes de version

Nouvelle version : en fait incompatible !

Nouvelle version : en fait compatible !



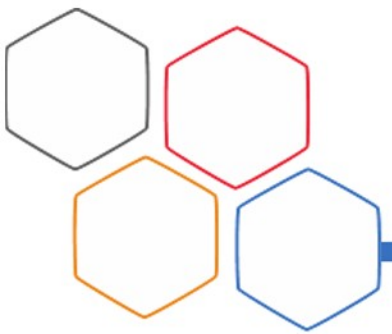
Contraintes : **mal exprimées ?**

```
MonProjet1 >= 1.0.0
MonProjet2 >= 2.0.0
```

```
&& <= 2.1.0
```

Version minimale :
- approximation : version initialement utilisée pour écrire le code de `MonProjet3`

Version maximale:
- approximation : version la plus vieille avec laquelle le code de `MonProjet3` a été testé



Expression des contraintes de version

- Contraintes sur les versions des dépendances
  - **Pas suffisant** pour bien gérer les évolutions de version des dépendances
  - Un mécanisme permettant « d'anticiper le futur » est nécessaire : il doit permettre de savoir dans quelle mesure une version est compatible avec les version existantes.

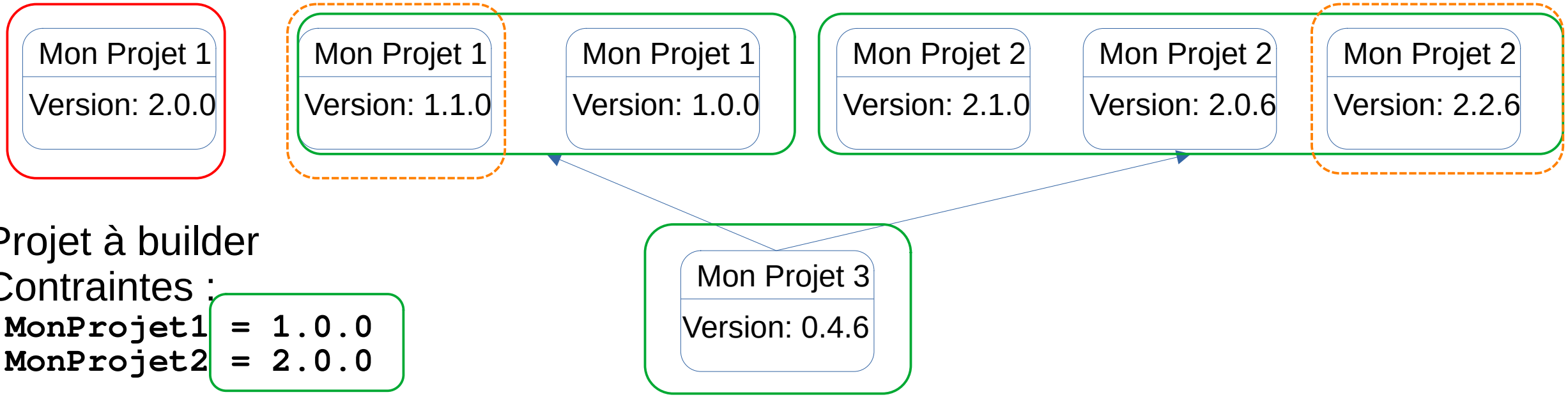
- Principe simple : le « semantic versioning »
  - Façon de numéroter les versions permettant de déduire leurs (in)compatibilités
  - Format de version : `MAJOR.MINOR.PATCH`
    - `MAJOR` : changement implique interface (e.g. API) incompatible
    - `MINOR` : changement implique interface compatible avec précédentes MINOR
    - `PATCH` : changement implique interface identique avec même minor

Expression des contraintes de version

Nouvelle version : incompatible !

Nouvelle version : compatible !

Nouvelle version : compatible !



Projet à builder
Contraintes :

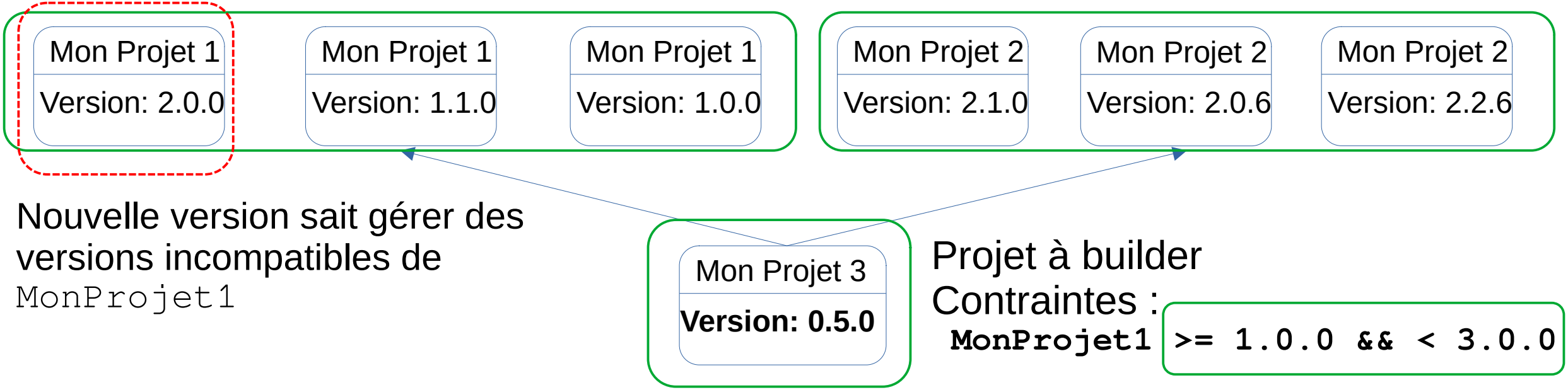
```
MonProjet1 = 1.0.0
MonProjet2 = 2.0.0
```

Plus besoin de gérer des intervalles dont il est bien difficile de donner une **borne supérieure** !!

Expression des contraintes de version

- Intervalles toujours utiles:
  - Lorsqu'un projet sait s'adapter à des versions incompatibles de ses dépendances !!!!

version incompatible avec 1.X



Nouvelle version sait gérer des versions incompatibles de `MonProjet1`

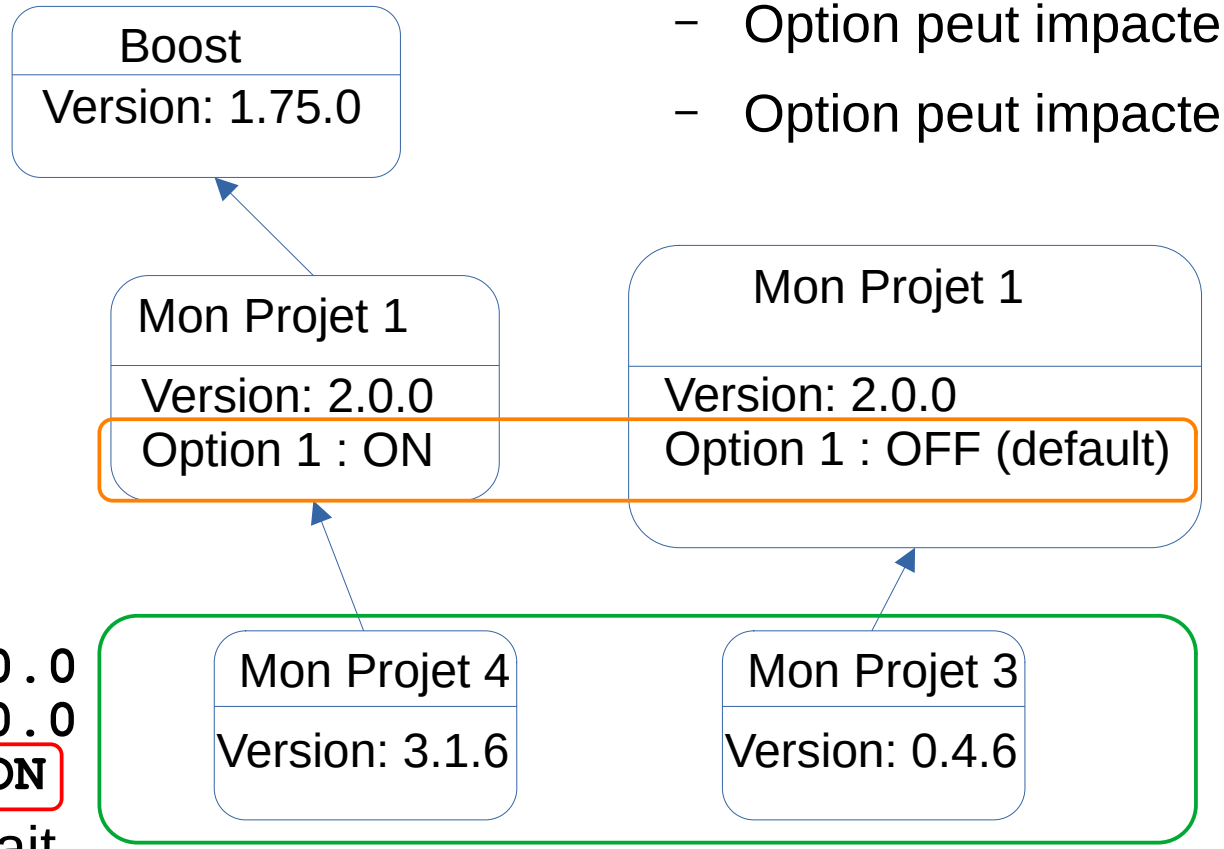
Projet à builder
Contraintes :

`MonProjet1` **`>= 1.0.0 && < 3.0.0`**

Gestion d'intervalles de versions incompatibles

Expression des contraintes de version

- Gestion des options
  - Option peut impacter les dépendances !!!
  - Option peut impacter les composants fournis !!



Options différentes sur la même version !!

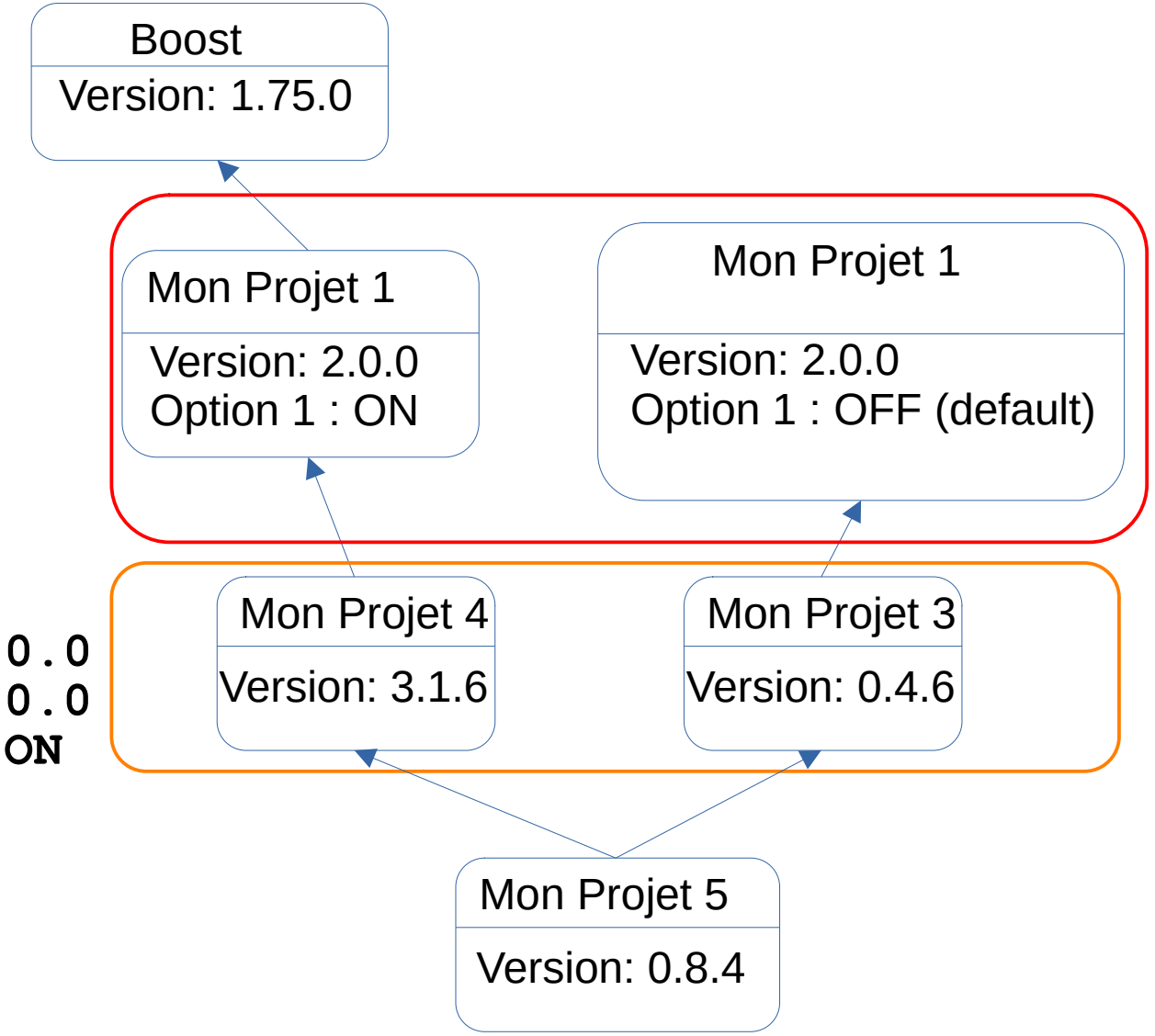
Projet à builder
Contraintes :

```
MonProjet1 >= 1.0.0
           && < 3.0.0
   && Option1 == ON
```

Choix d'une option fait partie de la contrainte

OK car les projets sont isolés

Expression des contraintes de version



Comment choisir ???

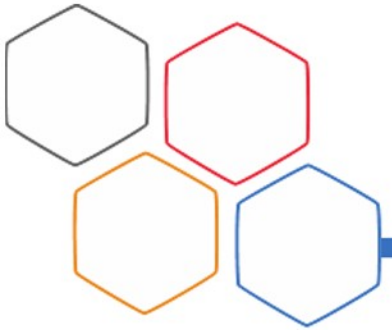
Projet à builder
Contraintes :

```
MonProjet1 >= 1.0.0
           && < 3.0.0
  && Option1 == ON
```

Projets sont incompatibles à cause de leur dépendance à `MonProjet1`

Expression des contraintes de version

- Options
  - Complexité énorme
  - Peut rapidement rendre la résolution **sans solution**
  - Hypothèses pour « résoudre » le problème :
    - Activation d’une option ne rend pas le projet incompatible (ne fait que rajouter des fonctionnalités)
    - Pas de variante (option avec exclusion mutuelle) sinon toujours incompatibilité
- Approche recommandée :
  - Option utilisées pour des aspects **purement internes** (optimisation du code, compilation CUDA, etc.)
  - Option **automatiques** en fonction de la plateforme cible (langage / compilateur disponible, OS, hardware)
  - Remplacer les options d’un projet par de **nouveaux projets**

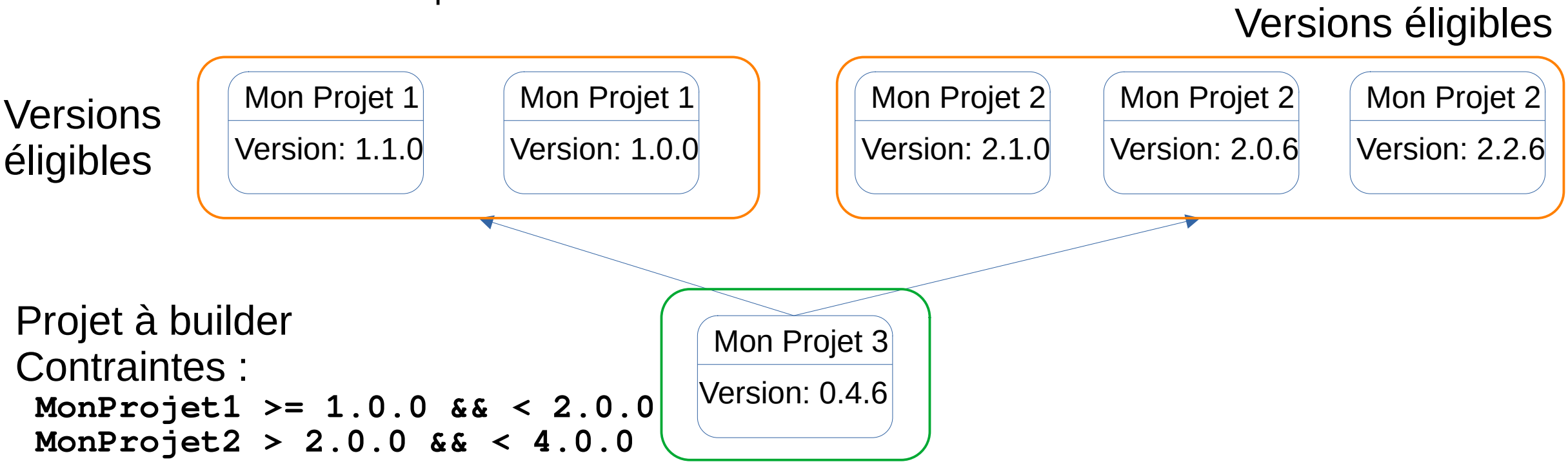


Plan

- Gestion des versions
- Expression des contraintes de version
- **Résolution automatique des versions**
- PID
- Réflexion

Résolution automatique des dépendances

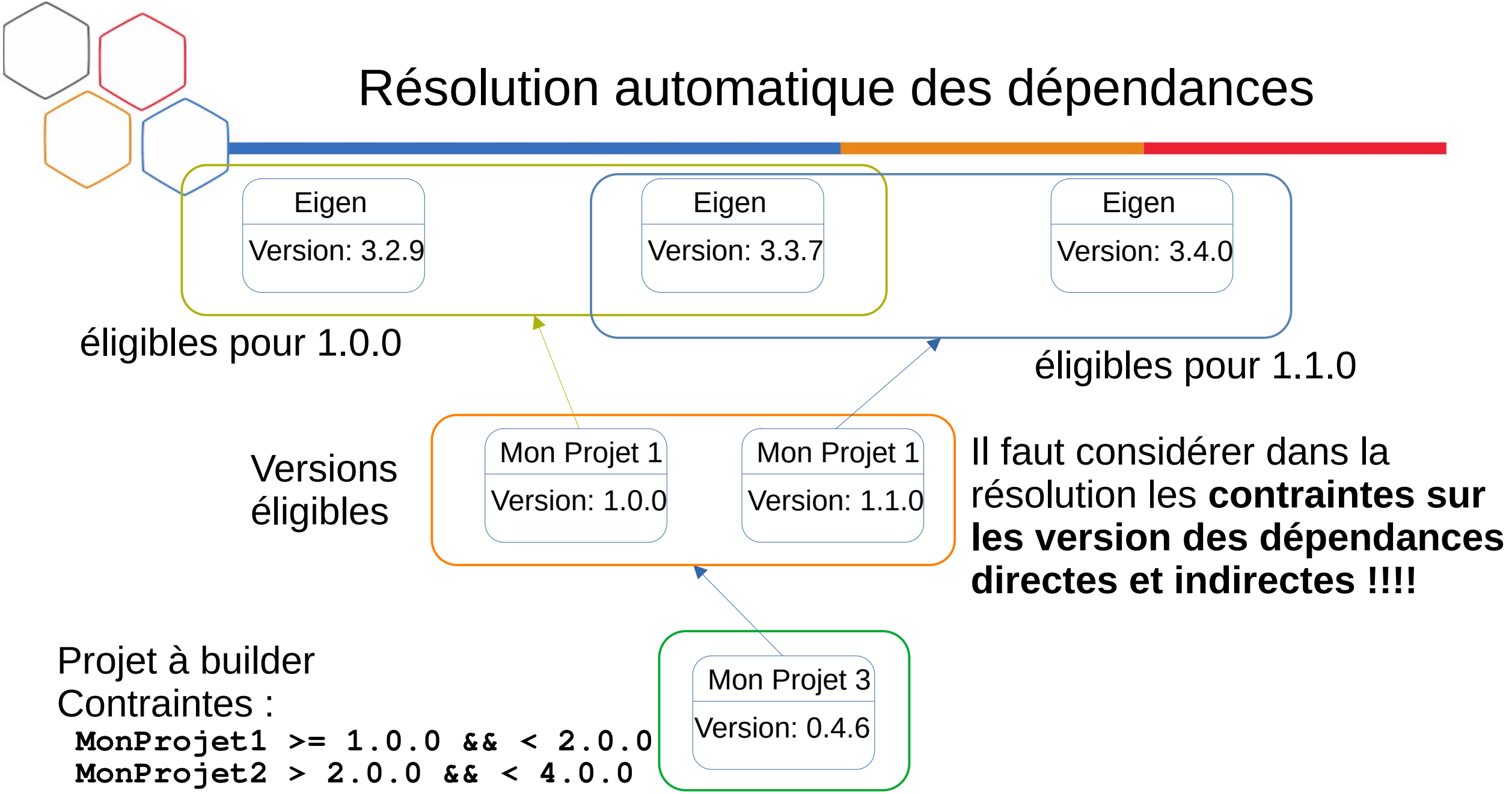
- Problèmes
  - Où trouver ces versions ?
  - Comment et où les installer ?
  - Laquelle choisir ?



Projet à builder
Contraintes :

```
MonProjet1 >= 1.0.0 && < 2.0.0
MonProjet2 > 2.0.0 && < 4.0.0
```

Résolution automatique des dépendances



Il faut considérer dans la résolution les **contraintes sur les version des dépendances directes et indirectes !!!!**

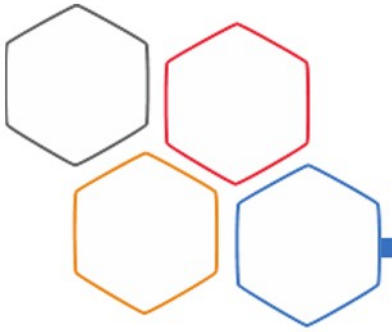
Projet à builder
Contraintes :

```
MonProjet1 >= 1.0.0 && < 2.0.0
MonProjet2 > 2.0.0 && < 4.0.0
```

Résolution automatique des dépendances

- **Déploiement** automatisé : **nécessaire**
  - Autrement il faudrait que l'utilisateur indique à la main chaque version disponible
  - Permet d'élargir la base des versions disponibles pour chaque projet (augmente les chances de trouver une solution)
  - => Résolutions des dépendances liée au **packaging**

- Déploiement depuis les sources ou des binaires : **préférable**
  - Source : permet d'avoir accès à toutes les versions sans restriction
  - Binaire : plus contraint MAIS évite la recompilation (particulièrement utile en CI)



Plan

- Gestion des versions
- Expression des contraintes de version
- Résolution automatique des versions
- **PID**
- Réflexion

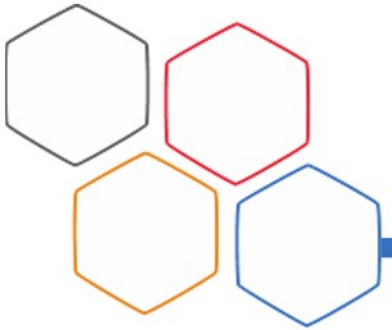
PID - Package Integral Development

- Méthodologie « maison » de gestion du développement
  - Met en œuvre une solution pour répondre aux problèmes posés par la gestion des dépendances.
  - Projets « standardisés » qui suivent tous la même logique
  - Automatisation complète du déploiement et de l'adaptation des dépendances utilisées par les projets.
    - Actuellement : API CMake
    - Désavantages : pas toujours pratique, pas optimal niveau temps d'exécution
  - Wrappers pour gérer des projets externes (e.g. eigen, boost, pinocchio, etc.)
    - Implémentent les recettes pour **builder des versions** ou **déployer les packages systèmes** d'un projet.
    - ~ 200 wrappers disponibles

PID - Package Integral Development

- Limitations
  - Pas de gestion des **options de projets** dans la résolution des dépendances
    - Option peut impacter l'API, les dépendances, etc.
  - Il faut **wrapper les dépendances externes** (projets pas développés avec PID)
    - Travail long et pénible pour les projets avec beaucoup de dépendances (e.g. opencv, VTK) engendrant beaucoup de perte de temps
    - Pas de base existante à par celle que l'on a créé
  - Diffusion de nos codes pas « intuitive »
    - tierces parties doivent utiliser notre modèle de déploiement et de résolution des dépendances

➡ Vers un meilleur système de gestion des dépendances et du packaging



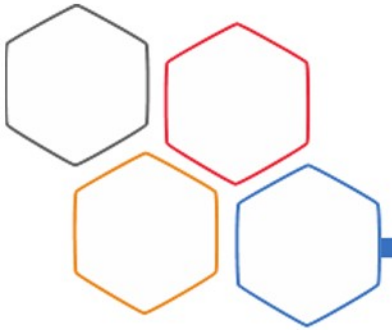
Plan

- Gestion des versions
- Expression des contraintes de version
- Résolution automatique des versions
- PID
- **Réflexion**

Réflexion

- Un **modèle commun de gestion de dépendances** pour la communauté
  - Impose certaines **règles communes**
  - Suffisamment flexible pour s'intégrer aux processus de développement de tout le monde avec un minimum d'efforts
  - Doit reposer sur des « outils mainstream » qui permettent de gérer la totalité du problème

- Avantages
  - Diffusion de nos projets facilitée
  - Plus facile d'intégrer des codes tiers
    - Pas besoin de faire des « wrappers » de projets tiers s'ils sont déjà fait « nativement » par les créateurs des projets



Réflexion

- Quels outils possible ?
  - Proposition : Conan 2.X
    - Solution libre
    - Résolution des dépendances et du packaging
    - Multi plates-formes / multi build system
    - Mainstream pour le C/C++ (mais utilisable pour tout)
    - Customisation possible (API Python)
    - Énorme base de recettes disponibles (pas besoin de les écrire)
    - Gestion du packaging binaire (cache local, serveur de stockage des recettes et binaires, intégré à Gitlab)
    - Gestion des options (variantes) dans la résolution.
    - Gestion du semantic versioning et des contraintes de version

Réflexion

- Quels outils possible ? Autre possibilités ...
  - VCPKG
    - Moins complet que Conan, plus simple d'utilisation, assez similaire au final
    - Aussi très populaire (énorme base de projets disponibles)
  - Conda
    - Très différent de Conan, gestion des dépendances se fait sur une logique de « distribution » (moins adaptable, choix des versions plus « statique », obligé de faire des choix sur les options)
    - Ultra populaire (énorme base de projets disponibles)
    - Environnement virtuel intégré (évite de polluer l'OS si on veut installer des dépendances « système » alternatives)

Conlusion

- La gestion des dépendances c’est compliqué
- La communauté bénéficierait énormément d’avoir un modèle commun pour gérer les dépendances
  - Projet TIRREX, Robotex 2.0, PEPR Organic Robotics : partage des codes et des plates-formes au niveau national !!
- Pas d’outils parfait mais Conan semble le plus complet
  - Nécessite de s’accorder sur des règles communes de description des recettes
  - Gestion des dépendances systèmes à standardiser :
    - automatiser l’utilisation des package manager système,
    - utilisation d’un environnement virtuel pour éviter de polluer le système ?

Merci pour votre attention