G^{en}oM3:
Building middleware-independent robotic components

Comparaison de middleware:
YARP, MS Robotics Dev Studio, URBI, OpenRTM-aist, ROS

Pablo Rauzy

15 février 2011

Table des matières

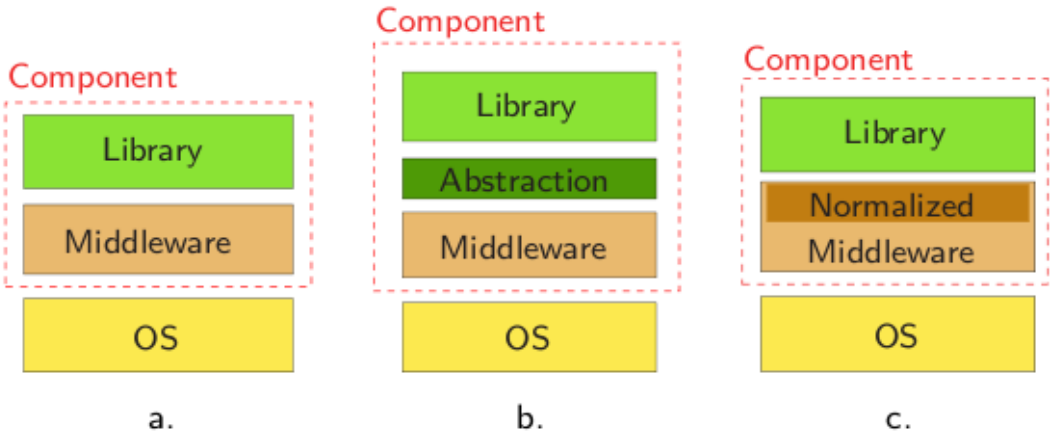

FIGURE 1 – Architecture classique de composants

1 G^{en}oM3 : Building middleware-independent robotic components

1.1 Introduction

Les robots autonome sont des systèmes très complexes avec un nombre importants de composants et donc de logciels différents. Non seulement il y a des tâches très variées à accomplir mais elles le sont sous différentes contraintes de temps (temps réel, synchrone...). En plus de cela tout ces composant doivent intéragir et communiquer, d'où l'appellation de "système complexe".

Pour maitriser cette complexité, l'approche d'architecture "par composant" à fait ses preuves. Une particularité de la robotique est le côté intrinsèquement dynamique de l'ensemble de composant qui communiquent et doivent se synchoniser. Cette dernière tâche est effectuer par le middleware.

Dans l'article, "middleware" est défini comme le composant logiciel en charge de la communication et de la syncrhonisation ente composants ainsi que de l'accès au système d'exploitation ainsi qu'au différents pilotes. Il existe beaucoup de middleware et le choix du middleware influence énormément le design général du composant, au point que si le middleware doit changer, c'est parfois jusqu'à tout le composant qu'il faut redévelopper.

Afin d'augmenter la réutilisabilité du code, et donc d'avancé l'industrialisation de la robotique, le papier que je présente présente une méthode d'ingénérie logiciel implémenté dans un framework de développement de composant qui se veut autant que possible indépendants du middleware : G^{en}oM3.

L'idée principale est de séparer la partie algorithmique (la logique du composant) de son intégration au middleware de manière à ce que le middleware puisse être interchangeable tout en profitant pleinement de leurs capacités spécifiques.

1.2 Être indépendant du middleware : approches existantes

Une solution évidente pour s'abstraire d'un composant logiciel est de développer une couche de compatibilité par dessus le logiciel que l'on veut remplaçable (fig. 1b). Le problème est que c'est une tâche ingrate à faire à la main et que ce genre de couche logiciel occasione presque sûrement des pertes de performance.

Une solution plus élégante est la standardisation (fig. 1c). Haha. Non seulement c'est un rêve qui n'a que très rarement un echo commercial, mais en plus c'est particulièrement difficile à mettre en place dans le monde de la robotique où tout est designé de manière très spécifique, pour un objectif en partculier. Imposer une norme dans ce monde là reviendrait un prendre un plus petit dénominateur commun des fonctionnalités nécessaires aux différents types de composants, et développer un composant portables serait beaucoup trop limitant.

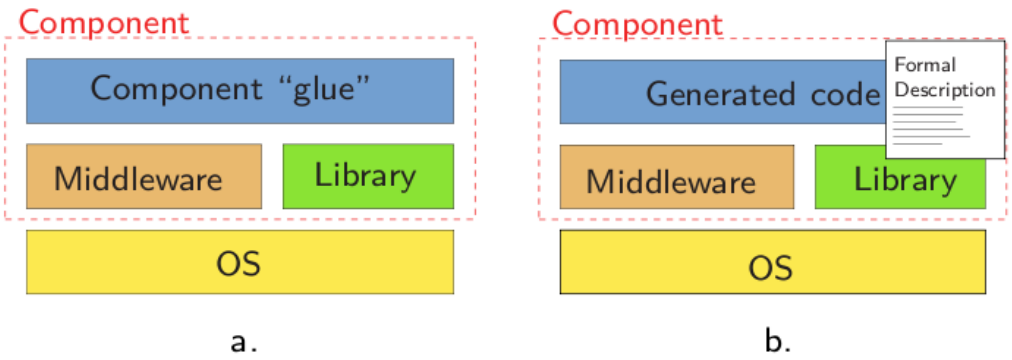

FIGURE 2 – L'appoche utilisé par G^{en}oM3

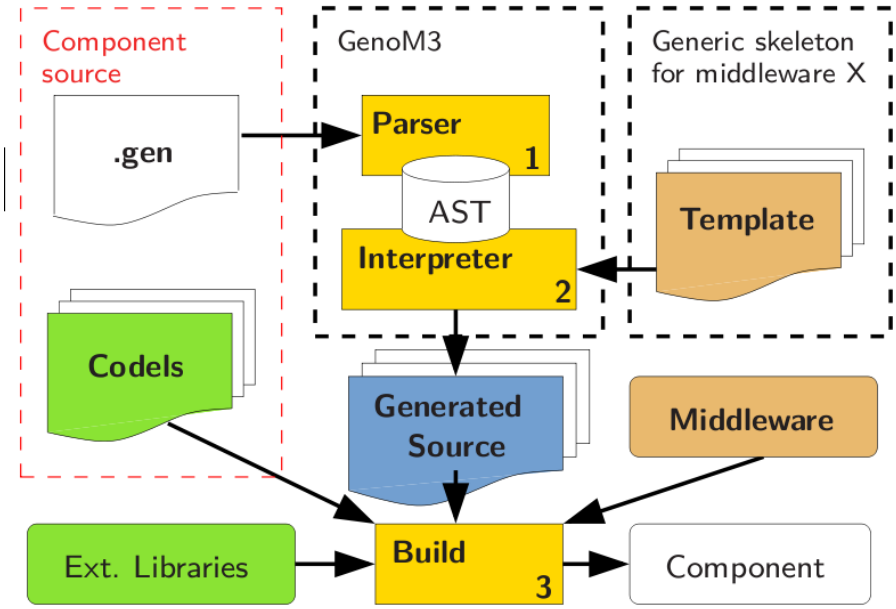

FIGURE 3 – Aperçu général du fonctionnement de G^{en}oM3

Une autre solution serait de rendre les différents middlewares interopérables. De cette manière un composant serait toujours très dépendemment lié à son middleware mais au moins des composants reposants sur différents middlewares pourraient quand même communiquer. Le problème est que cette interopérabilité ne peut se mettre en place que par la création de couche logiciel ayant les même défauts que la première solution proposées : pertes drastique de performance (par exemple le possible besoin de réencoder les données systématiquement).

1.3 G^{en}oM3

Afin de se sortir des problèmes présentés par l'approche classique dont on vient de parler, l'article propose de séparer la logique et le middleware complètement en ayant une glue qui vient se rajouter par dessus pour faire le lien entre les deux parties (fig. 2a). De cette façon, l'indépendance vis à vis du middleware est entièrement concentrée dans la glue. G^{en}oM3propose en fait de générer automatiquement le code de la glue, à partir d'une description formel du composant (fig. 2b).

La figure 3 propose un aperçu du fonctionnement général de G^{en}oM3.

Les modèles de composants ("component templates") contiennent tout le code qui n'appartient pas à la logique du composant. Seul un modèle est requis pour un middleware donné et peut-être réutilisé partout où ce middleware l'est. En revanche il est possible d'en avoir plusieurs et cela permet au coût d'une simple recompilation de tester des architectures alternatives (avec et sans threads par exemple). Ces modèles peuvent être développés dans n'importe quel langage de programmation en l'utilisant pleinenment, du moment qu'il est compatible avec le reste du composant.

Le fichier de description du composant contient toutes les informations nécessaires à la description du composasant. GenoM3construit une représentation de ces données dans un format compatible avec le langage de templating. Ces données comprennent :

- La définition des stuctures de données utiliser dans l'interface.
- Les méta-données comment les langages utilisés, les structures de données internes...
- Les informations sur les tâches : fonctions de la logique qui doivent s'exécuter de manière périodiques. Ces informations contiennent la période et le temps d'exécution.
- Les informations sur les services : contient des informations sur les entrées/sorties, les tâches en jeux et leur relations, les possible exceptions.
- Les "ports" : connexion des données entre les différentes tâches et services.
- Les évennements disponibles dans l'interface.

Les codels ("code elements") sont les fonctions qui constituent la logique du composant. Elles ne sont pas en charge de gérer leurs entrées et sorties (cette partie là est gérée dans la glue générée à partir des informations dans la description du composant) et ne s'occupe que de la partie algorithmique.

1.4 Conclusion

L'article parle ensuite d'exemples d'utilisations concrète de GenoM3. Il conclue enfin sur des possibles utilisation du framework pour faire de la comparaison de différent middleware de manière plus objective que précédemment possible. De manière générale, l'approche proposée par l'article est très pertinante pour la recherche et développement de robots et le prototypage de ces derniers.

2 Comparaisons de middleware :

Dans cette sections nous allons comparer différents middleware : YARP, MS Robotics Dev Studio, URBI, OpenRTM-aist et ROS.

2.1 YARP

YARP se concentre sur la communication entre composants. Il a été design par et pour des chercheurs et est codé en C++. Il faut coder les composants en C++ pour l'utiliser.

Un composant YARP peut être compatible avec ROS.

`http://eris.liralab.it/yarp/`

2.2 Microsoft Robotics Developer Studio

Celui-ci est le seul de la liste qui n'est pas libre, et qui ne tourne que sur un seul système d'exploitation (Windows). Il s'utilise en C# ou via VPL ("visual programming language"), une interface graphique à base de bloque tout fait que l'on connecte pour programmer le robot.

C'est une IDE complet, de l'éditeur de code au packaging prêt à l'emploi pour toute une série de robot vendu dans le commerce, en passant par un simulateur en 3D du robot.

http://www.microsoft.com/robotics/

2.3 URBI

URBI a aussi une approche par composant. Les composants sont des UObject codés en C++ et sont orchestrés par un programme écrit en UrbiScript. UrbiScript est un langage spécifiquement fait pour cette tâche (DSL, "Domain Specific Language"). Il donne "gratuitement" la possibilité d'avoir du parallélisme, de la synchronisation, de la programmation évenementielle, et permet un réglage avancé du flot de contrôle grâce à l'utilisation de tags.

Un composant URBI peut être compatibale avec ROS.

http://www.gostai.com/products/urbi/

2.4 OpenRTM-aist

Il est aussi orienté composants, mais beaucoup des articles et de la documentation sont en japonnais, j'ai donc eu plus de mal à avoir un bon aperçu de ce middleware. Je sais cependant que c'est un prototype d'implémentation du standard RTC ("Robotic Technology Component") développé par l'OMG ("Object Management Group") pour essayer de standardiser les middlewares pour la robotique. C'est basé sur un standard plus général appeler CORBA ("Common Object Request Broker Architecture").

C'est programmable en C++, Java ou Python, et ça peut aussi être compatible avec ROS.

http://www.openrtm.org/

2.5 ROS

ROS est un système d'exploitation complet (pas seulement un middleware). Du coup son utilisation permet un contrôl complet même du plus bas niveau. Il a une notion de "nodes" similaires à la notion de "composants" des autres middleware. Il peut se programmer en C++ ou Python.

C'est visiblement la référence puisque les autres se ventent d'être compatible avec.

http://www.willowgarage.com/pages/software/ros-platform

2.6 Conclusion

D'après tout ce que je j'ai pu lire sur ces middleware, j'ai vraiment l'impression qur ROS est la référence dans le domaine pour l'instant. Mais chacun à clairement des avantages pour des tâches spécifiques et c'est finalement pour le mieux que les développeurs de robots puissent avoir le choix.