D'une approche modulaire à une approche orientée composant pour le développement de systèmes autonomes : Défis et principes

Matthieu Gallien†, Fahmi Gargouri†, Imen Kahloul*, Moez Krichen*, Thanh-Hung Nguyen†
Saddek Bensalem†, Félix Ingrand*

*LAAS/CNRS, Université Toulouse
†Verimag Laboratory, Université Grenoble I, CNRS

***Abstract*— Les systèmes autonomes sont des systèmes complexes qui reposent sur la coopération/interaction entre différents composants logiciels. Ces logiciels sont de types assez variés, avec des contraintes de fonctionnement temporelles assez diverses. L'ensemble doit cependant respecter les spécifications du concepteur et garantir des contraintes de sûreté. Pour y parvenir, nous proposons l'utilisation d'une approche orientée composants basée sur l'outil BIP, pour, dans un premier temps, le développement de la couche fonctionnelle de systèmes autonomes (robots, satellites, etc.), et plus tard, envisager d'étendre cette méthodologie vers la couche décisionnelle. Cette approche vient naturellement compléter l'approche modulaire de GenoM jusqu'alors utilisée pour développer la couche fonctionnelle de l'architecture LAAS. Elle permet de mettre en place des systèmes robustes et sûrs en produisant un contrôleur d'exécution correct par construction, et en fournissant un modèle formel qui peut être utilisé avec divers outils de vérification et de validation.**

I. Introduction

L'ingénierie système repose sur l'idée qu'un système complexe est construit en assemblant des composants. D'où le principe de la conception orientée composant qui prévoit que les grands systèmes sont conçus à partir de systèmes plus simples. Cette approche confère, entre autres, des avantages de réutilisation, d'analyse et validation modulaires, reconfigurabilité, contrôlabilité, etc.

Les systèmes autonomes sont des systèmes complexes qui reposent sur la coopération/interaction entre différents composants logiciels. Ces logiciels sont de types assez variés, avec des contraintes de fonctionnement temporelles assez diverses. L'ensemble doit cependant respecter les spécifications du concepteur et garantir des contraintes de sûreté.

Une des principales limites de l'état de l'art actuel est l'absence d'un paradigme (modèle) unifié pour la description et l'analyse des flux d'information entre les composants. Un tel paradigme permettrait aux concepteurs et développeurs système de formuler des solutions basées sur des concepts tangibles, bien fondés et organisés plutôt que d'utiliser des mécanismes de coordination dispersés tels que les sémaphores, les moniteurs, l'envoi de messages, les appels à distance, les protocoles, etc.

Les concepteurs de systèmes complexes, tels que les robots autonomes, ont besoin de techniques d'analyse évolutives pour garantir des propriétés essentielles (temporelles et de sûreté). Pour faire face à la complexité de cette tâche, ces techniques sont appliquées à une description orientée composant du système dont les propriétés globales sont satisfaites par construction ou peuvent être déduites des propriétés de ses composants. De surcroît, la description componentisée fournit une base pour la reconfiguration et l'évolutivité.

Dans cet article, nous proposons l'utilisation d'une approche orientée composants basée sur l'outil BIP [2], pour le développement de la couche fonctionnelle de systèmes autonomes (robots, satellites, etc.) en prenant comme exemple la couche fonctionnelle de l'architecture du LAAS[1]. Au delà de cette première étape, nous envisageons de propager cette méthodologie vers la couche décisionnelle.

Cette approche vient naturellement compléter l'approche modulaire de GenoM jusqu'alors utilisée pour développer la couche fonctionnelle de l'architecture LAAS. Elle permet de mettre en place des systèmes robustes et sûrs en produisant un contrôleur d'exécution correct par construction, et en fournissant un model formel qui peut être utilisé avec divers outils de vérification et de validation.

Dans la suite, nous allons commencer par une présentation sommaire de l'architecture robotique déployée au LAAS (dans la Section II) et de l'outil de développement de systèmes temps réel BIP conçu à VERIMAG (dans la Section III). Ensuite, nous allons présenter le principe d'intégration GenoM / BIP qui fera l'objet de la Section IV, et enfin, vérifier ces résultats dans la Section V.

II. L'Architecture du LAAS

A. Présentation

Une architecture spécialement dédiée au développement de systèmes pour robots autonomes a été développée au LAAS. Il s'agit d'une architecture générale (cf. Figure 1) permettant de décomposer un système en trois niveaux ayant des propriétés temporelles et des représentations différentes. Ces niveaux sont organisés comme suit :

Le niveau décisionnel : À ce niveau, sont déployés des outils de supervision, en l'occurrence open-PRS [7], et de planification, comme le planificateur symbolique IXTET

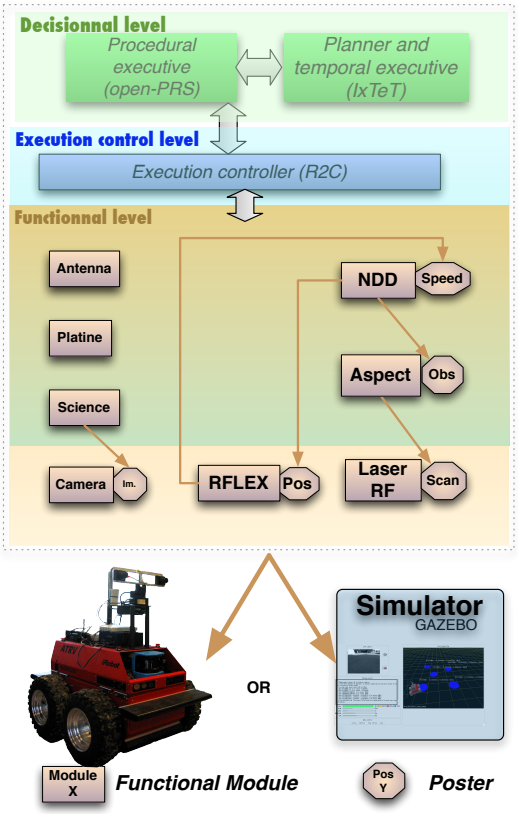

Fig. 1. Instance de l'architecture LAAS pour le robot DALA.

qui inclut depuis [8] le contrôle temporel de l'exécution de plan. Ces composants permettent de produire un plan d'action et d'en superviser l'exécution tout en restant réceptifs et réactifs aux événements émanants de la couche inférieure.

Le niveau fonctionnel : Inclut les fonctions sensorimotrices, les fonctions de traitement ainsi que les boucles de contrôle du robot. Chacune d'elles étant encapsulées dans un module contrôlable et communicatif généré par GenoM (pour générateur de modules) [5]. Chaque module offre des services activables via des requêtes par la couche décisionnelle et utilise des posters pour stocker les données produites à l'usage des autres modules ou de la couche décisionnelle.

Le contrôle d'exécution : Représente l'interface entre les deux niveaux précédents. Il s'assure du respect des contraintes de sûreté en garantissant la cohérence du système. Dans les dernières années, le contrôleur utilisé était le R2C [9]. Il est à noter que ce contrôleur garantit la bonne exécution des services des modules fonctionnels et l'interaction entre ces derniers, mais n'assure en aucun cas le contrôle d'exécution interne du module lui même.

B. GenoM

Tous les modules GenoM sont construits selon un modèle générique, décrit dans Figure 2, qui sera instancié selon la fonctionnalité en question du robot. Chaque *module* propose un ensemble de *services* actionnables par une requête client (ordre émanent de la couche décisionnelle).

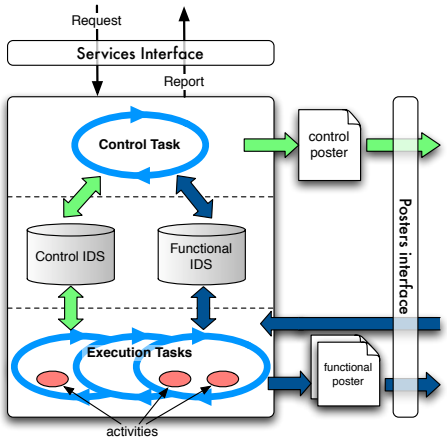

Fig. 2. Schéma générique d'un module GenoM.

Cette dernière sera interceptée par la *tâche de contrôle* qui, selon le type de requête, va effectuer une lecture écriture de données stockées en interne, ou lancer une activité dans la *tâche d'exécution* correspondante. Une *tâche d'exécution* contient une ou plusieurs *activités*, dont une activité permanente qui s'exécute pendant toute la durée de vie du module. Elle est particulièrement utile pour les modules qui veulent toujours faire un traitement au début de chaque cycle de la *tâche d'exécution* avant de passer aux autres activités à exécuter.

Figure 3 décrit le comportement d'une *activité* inspiré du cycle de vie d'un thread. Initialement dans l'état ETHER, indiquant ainsi une absence d'activité du service, l'automate passe à l'état START à la réception de la requête (avec la transition `"request"`). Si aucune incompatibilité ou problème de paramétrage ne sont notés, l'automate démarre l'exécution du codel relatif à l'activité en passant à l'état EXEC (`"started"`), état pendant lequel il alterne avec l'état interne IDLE afin de permettre l'exécution des autres activités de la *tâche d'exécution*.

Si, lors de sa fin nominale, l'activité s'exécute correctement, elle bascule dans l'état END avec envoi du bilan de succès (`"OK"`). Sinon, en cas de problème détecté pendant l'exécution, l'activité bascule de l'état EXEC à l'état FAIL en libérant les ressources, accompagné d'un bilan expliquant les causes du problème. Il est aussi possible que l'activité soit interrompue à tout moment (sous demande explicite du client ou lors du lancement d'un service incompatible), auquel cas, l'automate bascule dans l'état INTER, libère les ressources et envoie un bilan final d'interruption (`"interrupted"`).

Dans ces trois cas de fin d'activité (succès, échec ou interruption), l'automate retourne ensuite à l'état ETHER.

Il est à noter que, dans l'automate : schéma générique d'une activité, le contrôle (effectué par une tierce entité) intervient à différents moments. Par exemple, lors du lancement (de

ETHER à START) ou lors de l'interruption (de EXEC à INTER). Ce chevauchement peut prêter à confusion. Dans cet article, nous proposons de séparer, à partir du niveau le plus fin (à savoir le service d'un module), le contrôle de l'exécution. Ce fera l'objet de la Section IV-A.

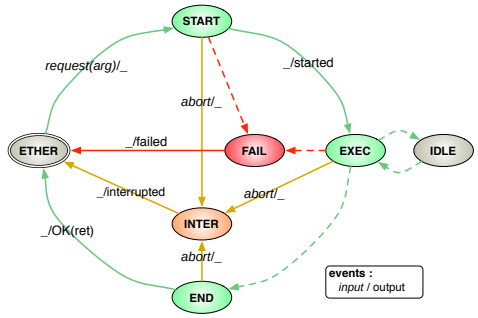

Fig. 3. Execution automaton of an activity.

1) Exemple illustratif de couche fonctionnelle: La méthodologie GenoM veut que chaque fonctionnalité du robot soit encapsulée dans un module. Toutefois, certaines fonctionnalités, telle la navigation, sont assez complexes au point d'être assurées par un ensemble de modules.
Comme exemple d'exécution, nous nous somme intéressés à DALA, un iRobot ATRV, et plus précisément aux modules impliqués dans la navigation tel qu'illustré plus haut dans Figure 1.

- Le module *sick* : Il s'agit du module qui gère le capteur laser. Il produit entre autres des mesures cartésiennes relatives au repère du robot ou au repère d'origine correspondant aux segments perçus.
- Le module *aspect* : Il utilise le poster de *sick* pour dresser une carte des obstacles avoisinant le robot.
- Le module *ndd* : Ce module se charge d'assurer la navigation en évitant les obstacles pour atteindre une position but. Pour ce faire, il récupère la position actuelle dans le poster Pos de *pom* ainsi que la carte d'obstacle le poster Obs d'*aspect* . En sortie, il fournit le poster Speed contenant une consigne en vitesse.
- Le module *rflex* : C'est le module qui assure la locomotion du robot en asservissant les roues sur la consigne en vitesse produite par *ndd* . En sortie, il donne une indication de position relative au déplacement du robot (position odométrique améliorée par la mesure du gyroscope).

III. L'OUTIL DE VERIMAG : BIP

BIP [2] (Behavior, Interaction and Priorities) est une plateforme logicielle pour la modélisation de composants temps réel hétérogènes. BIP considère le modèle d'un composant comme une superposition de trois couches. La couche inférieure modélise le comportement (B pour behavior) du composant sous forme d'un ensemble de transitions. Au dessus, s'ajoute la description des interactions (I) entre ces transitions via des connecteurs. Enfin, la couche supérieure correspond à un ensemble de règles de priorités (P) traduisant la politique d'ordonnancement des interactions.

Ce découpage(séparation) confère l'avantage de séparation entre le comportement proprement dit du composant, et la structure globale du système.
L'idée de base de BIP est qu'un système complexe n'est autre que la composition de sous-systèmes plus simples. Dans ce même ordre d'idées, BIP permet la composition hiérarchisée des composants, et ce en partant des composants dits atomiques qu'on assemble par des connecteurs et des priorités sur ces derniers.

Les éléments de base de BIP relatifs aux différentes couches précédemment évoquées sont les suivants :

A. Composant atomique et "Behavior"

Un composant atomique (exemple : Figure 4) contient un ensemble d'états de contrôle et des transitions entre eux. Ces dernières définissent le comportement du composant et sont franchissables par des ports utilisés pour la synchronisation inter composants. Il est possible de définir une garde (condition) sur la transition (x>0 dans l'exemple) ainsi qu'un traitement (du code C/C++ comme y := f(x)) à faire si la transition est franchie.

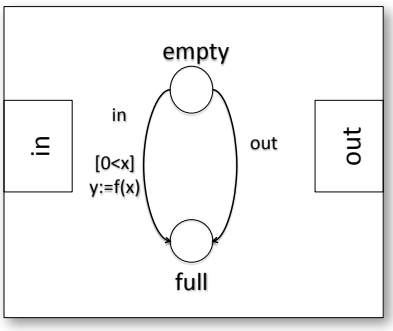

Fig. 4. Exemple de composant atomique

B. Connecteurs et "Interaction"

Si l'on considère les composants (atomiques ou composés) comme des blocs, les connecteurs permettent de les assembler afin d'obtenir un système entier. Un connecteur est un ensemble de ports des composants pouvant interagir. Une interaction (d'un connecteur) est, quant à elle, un sous ensemble des ports de ce premier. On peut la considérer comme une instance du connecteur. Un mécanisme de typage des ports est utilisé essentiellement pour distinguer deux types de synchronisation comme illustré dans Figure 5. À savoir, la synchronisation forte dite *rendez-vous* (cf. Figure 5-a), et la synchronisation faible dite *broadcast* (cf. Figure 5-b) identifiée par un triangle sur le port complet.

C. Politique d'ordonnancement et "Priority"

Dans le cas où, sur un port, plusieurs connecteurs sont définis et qu'il se trouve que plus qu'une interaction est possible, ce conflit est traité par la définition de priorités affectées aux connecteurs. Ainsi, il est possible de filtrer les interactions à exécuter parmi celles possibles.

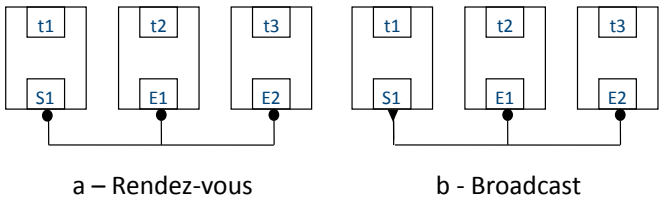

Fig. 5. les connecteurs et les deux types de synchronisations.

IV. INTÉGRATION GenoM / BIP

A. Componentisation de la partie générique et structurelle d'un module GenoM

Nous avons vu, dans la Section II-B, que la couche fonctionnelle de l'architecture du LAAS respecte une approche modulaire telle qu'un module correspond au schéma générique décrit dans Figure 2. Dans cette approche de componentisation, nous présentons dans Figure 6 le modèle BIP d'un module GenoM, suivant ce formalisme de décomposition de la couche fonctionnelle :

Functional level : := (Module)+
Module : := (Service)+ . (Execution Task)+ . (Poster)+
Service : := (Service Controler) . (Activity)
Execution Task : := (Timer)+ . (Scheduler Activity)

Où le "+" veut dire que le composant contient au moins une sinon plusieurs occurrences du sous-composant, et le "." traduit la composition de plusieurs sous-composants.

Par ailleurs, le *Scheduleur Activity* va ordonnancer le lancement des *Activities* des services relatifs à la *task Execution* en question. Dans un *service*, nous avons voulu séparer le contrôle, de l'exécution en les gérant dans des sous composants séparés (cf Figure 7) appelés respectivement *Service Controler* (à gauche de la figure) et *Activity* (à sa droite).

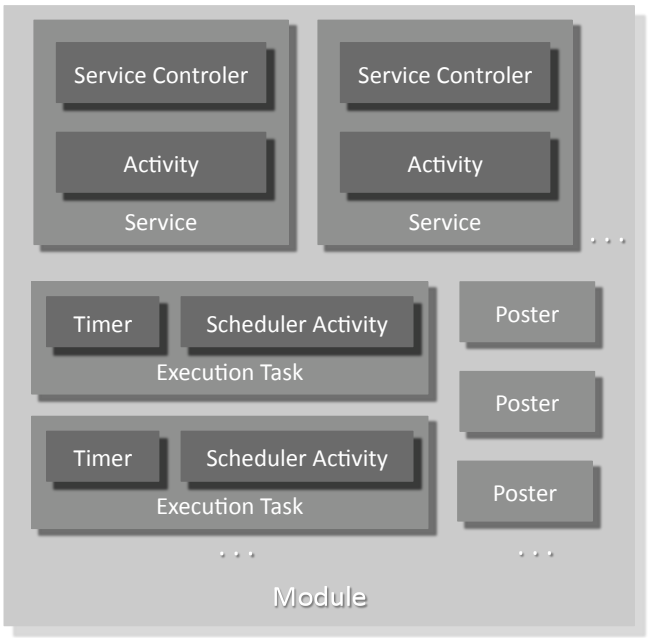

Fig. 6. La componentisation d'un module GenoM

Nous avons essayé de faire en sorte que le modèle BIP de service soit le plus fidèle possible à l'automate générique de GenoM. Cette démarche de dissociation nous permet d'appliquer la nouvelle structure et la lier à celle de GenoM, entre autres, via les bibliothèques de codels existants (cf. II-B).

Il est à noter que les transitions "control", "error", "start", "end", "fail" et "inter" du *Service Controler* s'accompagnent d'une modification de la variable interne "status" et d'informations sur l'état d'exécution de l'activité.

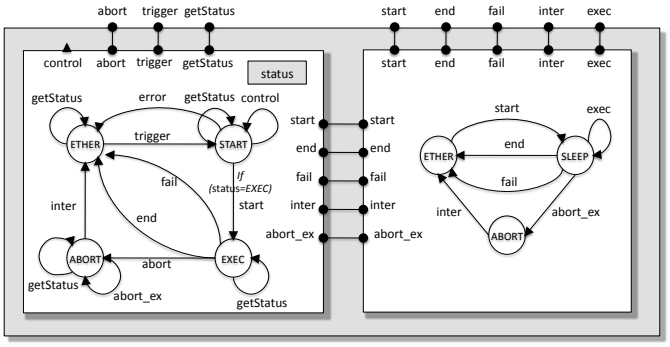

Fig. 7. Modèle BIP d'un service GenoM.

B. Componentisation de NDD

Nous avons modélisé en BIP le module *ndd* responsable de la navigation. Comme le montre Figure 8, il contient une tâche d'exécution qui gère cinq services dont un service *Permanent* et des posters (représentés par un composant générique *poster*). La tâche d'exécution (cf Figure 9) se réveille périodiquement synchronisée par un sous composant Timer. À chaque période, elle déclenche le service Permanent et elle permet aux services de s'exécuter grâce à leurs interactions.

Grâce au formalisme BIP, il est possible de modéliser des relations assez complexes à définir. À titre d'exemple, le déclenchement du service Stop provoque l'interruption du déplacement qui se traduit par le `"abort"` du service `GoTo` si ce dernier est en court d'exécution. Cette opération correspond à connecteur de type broadcast entre la tâche d'exécution et le port (notation port :service) trigger :Stop, et un autre connecteur de type broadcast entre ce premier connecteur et le port abort :Goto.

Par ailleurs, une autre propriété doit toujours être vérifiée, à savoir qu'un déplacement ne doit se faire que s'il est précédé par un `SetParams` et un `SetSpeed` qui initialisent correctement le module, d'où le connecteur entre trigger :GoTo, getStatus :SetParams et getStatus :SetSpeed.

À partir de ce modèle, la chaîne d'outil de BIP génère le code que "BIP Engine" pourra exécuter. Ce code contient des appels aux codels contenus dans des bibliothèques initialement développées pour GenoM afin d'exécuter les activités du robot.

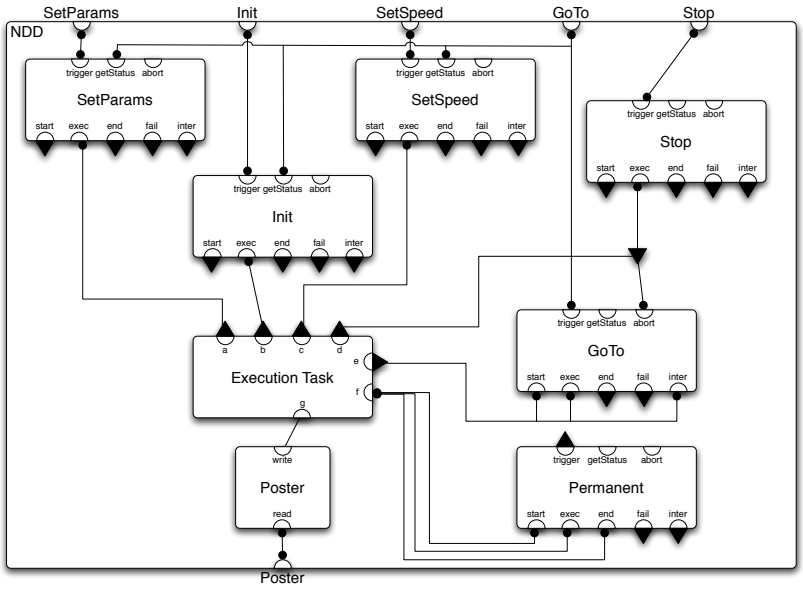

Fig. 8. Modèle BIP de *ndd* .

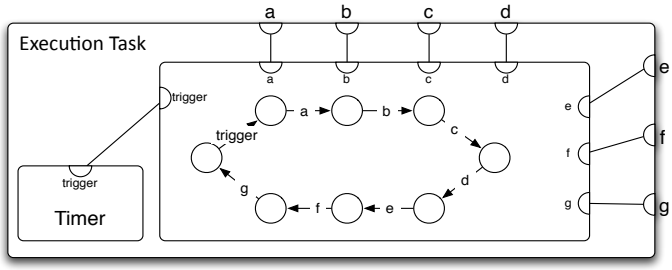

Fig. 9. Modèle BIP de l'Execution Task.

Le code généré pour *ndd* a été intégré et exécuté dans l'environnement du robot DALA du LAAS. Il convient de souligner que cette simulation, ayant pour particularité l'hétérogénéité de la couche fonctionnelle (des modules GenoM et BIP), met en évidence la compatibilité GenoM/BIP ainsi que la compatibilité module BIP/couche décisionnelle. Cela renforce l'objectif de l'approche qui cherche s'adjoindre et compléter l'approche GenoM.

V. Vérification

Les Outils de BIP, ainsi que d'autres outils de vérification génériques, permettent, outre l'exploration exhaustive de l'espace des états du système, la détection des deadlocks potentiels et la vérification de certaines propriétés dans le modèle du robot.

A. Absence de blocage (Deadlock freedom)

Il s'agit d'une propriété d'exactitude essentielle car elle caractérise la capacité du système à exécuter certaines activités au cours de sa durée de vie. Les outils de BIP permettent la détection de blocages potentiels par analyse statique des connecteurs dans le modèle BIP [6]. Cette méthode consiste à utiliser les invariants générés automatiquement pour prouver la non-satisfiabilité des prédicats caractérisant les blocages globaux. Deux types d'invariants sont générés : (i) Les invariants de composants qui sur-approximent l'ensemble d'états atteignables de ces derniers, et (ii) les invariants d'interaction qui sont les contraintes sur les états des composants liées aux interactions.

Les invariants d'interaction sont obtenus par le calcul des "traps", des abstractions des états finis du système vérifié. La méthode est implémentée dans l'outil D-Finder[3], qui prend comme entrées des programmes BIP et applique les stratégies prouvées pour éliminer des blocages potentiels par calcul d'invariants de plus en plus forts. Au cour de la vérification du système modélisé par cet outil, aucun blocage n'a été trouvé, d'où ce système est deadlock-free.

B. Propriétés de sûreté

Une propriété de sûreté garantit qu'un événement non prévu n'aura jamais lieu. Pour ce faire, nous utilisons des méthodes basées sur "le model checking". Ce dernier consiste à construire un modèle fini du système analysé et à en vérifier les propriétés souhaitées que l'on souhaite. La vérification

s'applique sur une exploration complète ou partielle du modèle. Les avantages principaux de cette technique sont :

- La possibilité de rendre l'exploration du modèle automatique.
- La facilité de production de contre-exemples, lorsque la propriété est violée.

Le modèle considéré, dans le cas d'utilisation d'un outil de "model checking", est un système de transitions étiquetées (STE). Il existe plusieurs outils utilisant cette technique de vérification. Dans cette section, nous présentons les résultats obtenus suite à l'utilisons 3 de ces outils pour vérifier une propriété donnée.

1) Vérification avec l'outil Aldébaran: Aldébaran [4] est un outil qui vérifie les systèmes communicants représentés sous forme de systèmes de transitions étiquetées (STE). Il permet de minimiser et de comparer ces systèmes par rapport à des relations d'équivalences (exp., bisimulation, équivalence d'observabilité, etc.).

Dans notre cas, nous utilisons la fonction de minimisation. Cette dernière permet de générer le plus petit graphe équivalent à un STE par rapport à la relation d'équivalence de sûreté (safety) [10]. Le graphe minimisé obtenu sera utilisé pour vérifier manuellement des propriétés que le système original doit satisfaire.

Dans la suite, nous considérons une propriété de fraîcheur de données entre les modules *ndd* et *aspect* . Ce dernier génère périodiquement une carte locale décrivant les obstacles dans le voisinage du robot. Cette carte est produite par le service *ASPECTFromPosterConfig* (AFPC) de *aspect* et est stockée dans son poster *Obs* . Ensuite elle est lue depuis ce poster par *ndd* pour être utilisée par le service Goto.

La propriété de fraîcheur entre *ndd* et *aspect* consiste à imposer le fait que la carte produite par *aspect* soit suffisamment "fraîche" avant d'être lue par le module *ndd* . Nous imposons donc la contrainte temporelle suivante :

$$date\_lecture \text{ - } date\_ecriture \leq a$$

où "a" est un paramètre que l'on a choisi (arbitrairement) égale à 2 unités de temps système (tick). Cette contrainte consiste à fixer un délai maximal de 2 ticks pour lire la dernière carte écrite.

La Figure 10 montre le modèle adopté pour la vérification de cette propriété. Notre objectif est de l'imposer par construction en fixant les périodes de chaque module dans le Timer de sa tâche d'exécution. La synchronisation entre les deux modules est réalisée par le connecteur entre les deux Timers. L'action de lecture est chargée lors de l'exécution du service *Goto*.

Les périodes des modules GenoM *aspect* et *ndd* sont respectivement 4 et 10 ticks, soit 40 et 100 ms. La vérification avec Aldébaran n'est pas possible. En effet, l'outil ne peut pas générer le graphe lorsque le modèle est assez grand. Nous avons choisi donc d'utiliser les observateurs pour vérifier la propriété.

2) Vérification en utilisant les observateurs: Pour vérifier une propriété *P* dans un système représenté sous forme de graphe, on peut construire un observateur ("Observer") pour

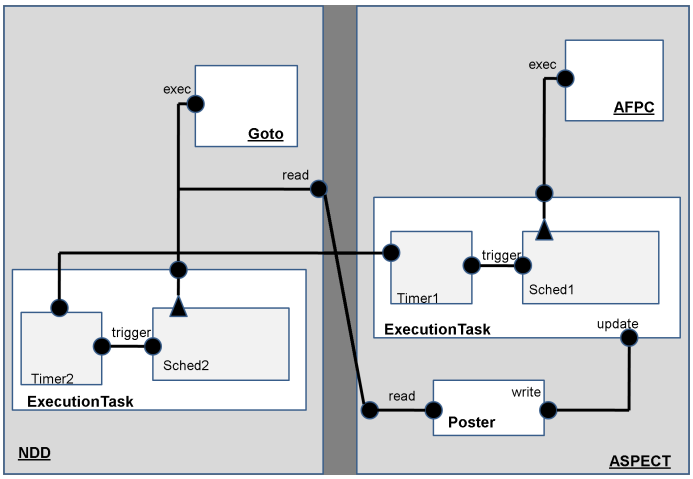

Fig. 10. Le modèle à vérifier.

cette propriété. Cet observateur est représenté sous la forme d'un automate qui contrôle le comportement du système et signale une erreur si *P* est violée. La Figure 11 montre l'observateur que nous avons utilisé pour vérifier notre propriété de fraîcheur. Il contient une variable c qui représente le compteur déclenché après une écriture. Si une action de lecture se fait lorsque la variable c a déjà dépassé la valeur 2, alors l'observateur va passer dans son état "ERROR" qui signifie que la propriété est violée. Durant l'exploration des états globaux du système, si un état global contenant l'état "ERROR" de l'observateur est atteint, alors la propriété n'est pas satisfaite.

Le résultat de l'application de cette méthode sur notre modèle est négatif. Le graphe d'exploration de notre système contient l'état "ERROR". L'observateur a donc détecté une violation de la propriété. Cette violation aurait pu être vu autrement. En effet, avec les périodes fixées pour chaque modules, nous remarquons bien qu'il est possible qu'il s'écouler jusqu'à 39 ms entre une écriture et la lecture suivante. Or, si la contrainte est au maximum 20 ms, il est donc probable que la propriété soit violée.

Une éventuelle solution de ce problème consiste à choisir soigneusement les périodes des deux modules.

VI. Conclusion

L'essor de la robotique de service, et le déploiement de robots d'exploration coûteux et loin de l'homme sont deux moteurs qui poussent la robotique autonome vers une plus grande sûreté, en particulier, de l'ensemble des logiciels. Jusqu'à ce jour, les robots autonomes sont développés avec des méthodologies et des approches "classiques" essentiellement inspirées du "génie logiciel" (modularisation, hiérarchisation, organisation en couches). Toutefois, la complexité des fonctions mises en oeuvre sur ces robots, et l'utilisation d'approches décisionnelles en font des objets logiciels particulièrement difficiles à valider ou à vérifier. Nous proposons une approche originale de développement de ces logiciels qui

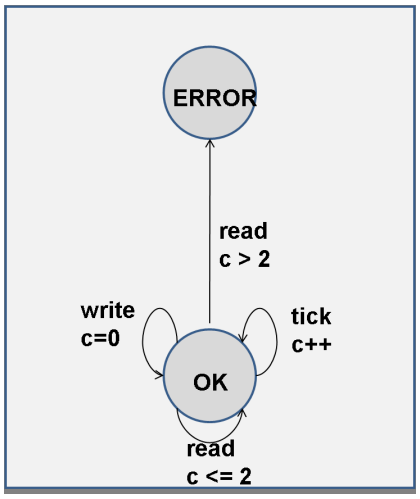

Fig. 11. Observateur.

s'appuie sur une architecture (LAAS) et des outils préexistants de la robotique (GenoM) en y adjoignant une approche de modélisation par componentisation (BIP) qui permet la synthèse de contrôleurs sûrs par construction, et la vérification de propriétés critiques pour le déploiement de systèmes temps réel. Nous avons montré comment des modules fonctionnels d'un robot mobile peuvent être modélisés en BIP à travers un processus évolutif à partir des modules existants. On obtient ainsi une couche fonctionnelle dont le contrôleur garantit des propriétés fortes, liées à l'application elle-même (séquence d'initialisation correcte, évitement d'interactions dangereuses, etc). De plus, le modèle obtenu peut être vérifié avec divers outils afin de garantir des propriétés critiques (absence de blocage fatal, propriétés temporelles, etc). Notre objectif est de poursuivre, vers les couches décisionnelles, le déploiement de cette approche afin de bénéficier de contrôleurs et d'outils formels sur l'ensemble des logiciels du robot autonome.

REFERENCES

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, 'An architecture for autonomy', *IJRR, Special Issue on Integrated Architectures for Robot Control and Programming*, **17**(4), (1998).

[2] A. Basu, M. Bozga, and J. Sifakis, 'Modeling heterogeneous real-time components in BIP', in *SEFM*, Pune, India, (2006).

[3] S. Bensalem, M. Bozga, T.H. Nguyen, and J. Sifakis, 'Compositional verification for component-based systems and application', Technical Report TR-2008-8, Verimag, (2008).

[4] M. Bozga, J-C. Fernandez, A. Kerbrat, and L.Mounier, 'Protocol verification with the aldebaran toolset', in *Software Tools for Technology Transfer 1*, pp. 166–183, (1997).

[5] S. Fleury, M. Herrb, and R. Chatila, 'GenoM : A tool for the specification and the implementation of operating modules in a distributed robot architecture', in *IROS*, Grenoble, France, (1997).

[6] G. Goessler and J. Sifakis, 'Component-based construction of deadlock-free systems', in *FSTTCS*, Bombay, India, (2003).

[7] F.F. Ingrand, R. Chatila, R. Alami, and F. Robert, 'PRS : A High Level Supervision and Control Language for Autonomous Mobile Robots', in *IEEE International Conference on Robotics and Automation*, Mineapolis, USA, (1996).

[8] S. Lemai, *IxTeT-eXeC : planning, plan repair and execution control with time and resource management*, Ph.D. dissertation, Institut National Polytechnique de Toulouse, 2004.

[9] F. Py and F. Ingrand, 'Dependable execution control for autonomous robots', in *IROS*, Sendai, Japan, (2004).

[10] C. Rodriguez, *Spécification et validation de systèmes en XESAR*, Ph.D. dissertation, Institut National Polytechnique de Grenoble, 1988.