Conception et Implantation de Système Fondé sur les Composants. Vers une Unification des Paradigmes Génie Logiciel et Système.

Marc Poulhiès

Université de Grenoble

Conception et Implantation de Système Fondé sur les Composants. Vers une Unification des Paradigmes Génie Logiciel et Système.

THÈSE

présentée et soutenue publiquement le 05/03/2010

pour l'obtention du

Doctorat de l'université de Grenoble

(spécialité informatique)

par

Marc Poulhiès

sous la direction de Joseph Sifakis

Composition du jury

Président :	Jean-Bernard Stefani
Rapporteurs :	Gilles Muller Lionel Seinturier
Examinateurs :	Jacques Pulou Joseph Sifakis Claude le Pape-Guardeux

Mis en page avec la classe thloria.

Résumé

Cette thèse a été co-encadrée par le laboratoire MAPS/AMS de France Telecom R&D (aujourd'hui MAPS/SHINE) et le laboratoire VERIMAG.

Le développement de logiciels pour les systèmes embarqués présente de nombreux défis. Cette thèse s'intéresse à ceux posés par les interactions entre les trois phases de conception (les développeurs construisent à partir de spécifications un modèle du système exprimé dans un langage de conception i.e. de programmation), d'implantation (le modèle précédent est compilé en un exécutable du système qui est finalement déployé sur les plateformes réelles) et de validation (un ensemble de techniques sont mises en œuvre pour vérifier que le système implanté est correct vis-à-vis des spécifications).

Pour cela nous nous intéressons aux caractéristiques du langage de conception et aux techniques de compilation de ce langage. Celles-ci permettent d'obtenir dans notre approche à la fois l'implantation du système et un modèle du système implanté. L'analyse de ce modèle permet la validation de l'implantation, ce qui suppose que ce modèle représente fidèlement le système implanté.

Nous proposons la construction d'un langage de conception basé sur l'utilisation de composants logiciels prédéfinis dont le comportement dynamique est supposé connu. Nous illustrons cette démarche par la réalisation d'un prototype complet d'un langage appelé Buzz, inspiré des modèles de programmation à acteurs dans lequel les composants logiciels utilisés sont des composants Think accompagnés de leur modèle comportemental opérationnel constitué d'un composant BIP.

Le compilateur associé à Buzz que nous avons développé à partir du compilateur Think existant (Nuptse) génère simultanément une architecture à composants Think pour l'implantation et un modèle à composants BIP à des fins d'analyse. Nous évaluons Buzz à travers deux expériences.

La première présente le développement de bout en bout d'un logiciel pour un exemple académique sur lequel nous démontrons la pertinence des choix techniques. Think nous permet un support d'implantation complet (compilation, optimisation, déploiement) et BIP rend possible la vérification d'un ensemble de propriétés dynamiques du système.

La deuxième expérience consiste à porter une application réelle de protocole radio utilisée dans des réseaux de capteurs et développée de manière classique, vers Buzz. Cette expérience démontre l'effectivité de notre proposition tant en termes de langage de programmation (l'expressivité de Buzz structure et simplifie le code original) qu'en termes d'outils de compilation et de vérification.

Abstract

This PhD thesis was co-supervised by the MAPS/AMS laboratory of France Telecom R&D (now MAPS/SHINE) and the VERIMAG laboratory.

Software development for embedded systems has many challenges. In this thesis, we address those related to the interactions between the three following phases of the software development process : the design phase (developers build a model of a system based on its specifications, using a design language *i.e.* a programming language), the implementation phase (the model previously built is compiled into an executable of the system which is then deployed on the target platforms) and the validation phase (a set of techniques are used to verify that the system implementation is correct with respect to the specifications).

To achieve this goal, we study the design language characteristics and compilation techniques. In our approach, they allow the creation of both an implementation of the system and a model for this implementation. Provided that this model faithfully corresponds to the implementation, the analysis of the model can validate the implemented system.

We propose a process for building a language based on predefined software components for which the dynamic behavior is supposed to be given. We illustrate this process with a prototype language, called Buzz, inspired by the actors programming model. Buzz uses Think components enriched with their operational behavioral model in the form of BIP components.

We developed a compiler for Buzz by extending the current Think compiler (Nuptse). It generates both an architecture of Think components for the implementation and a BIP model for the analysis. We evaluate Buzz through two experiments.

The first experiment details the end-to-end software development for an academical example on which we show the soundness of our technical choices. In particular, Think provides implementation support (compilation, optimization, deployment) and BIP allows us to verify the system's dynamic properties.

The second experiment focuses on porting to Buzz an application conventionally developed for a sensor network radio protocol. This experiment underlines the efficiency of our proposal both in terms of programming language (the result is more structured and simpler than the original code) and in terms of compilation tools and verifications.

Remerciements

Je tiens tout d'abord à remercier mes encadrants de thèse : Jacques Pulou, d'Orange Labs, et Joseph Sifakis, du laboratoire VERIMAG. Je leur suis reconnaissant de m'avoir encadré tout au long de ces quatre années de thèse.

Je remercie également Jean-Bernard Stephani, Gilles Muller, Lionel Seinturier et Claude le Pape-Guardeux d'avoir accepté de faire partie du jury de cette thèse. Un merci particulier à Lionel pour les conseils qu'il m'a apportés durant ma rédaction de thèse.

J'aimerais remercier tous les membres de l'équipe qui m'a accueillie à Orange Labs. Les échanges réguliers, techniques mais pas seulement, ainsi que la bonne humeur de Jacques ont rendu les trois années passées sur le site de Meylan très profitables et agréables. Je remercie également les membres du laboratoire VERIMAG, qui m'ont apporté leur support, plus particulièrement durant ma rédaction.

Je tiens aussi à remercier les personnes m'ayant apporté leur aide pour la relecture et la finalisation de mes travaux, mes différents colocataires, et Maxime qui a été un stagiaire exemplaire.

En particulier, je remercie Diane qui a toujours été disponible pour m'aider et m'encourager.

Table des matières

Table des figures

Introduction

Les avancées technologiques dans le domaine de l'électronique et du logiciel permettent à la fois une montée en puissance et une miniaturisation des circuits. Ainsi, nous pouvons aujourd'hui avoir dans la poche des appareils dont les capacités surpassent largement les machines qui étaient à la pointe il y a une dizaine d'années. Cette miniaturisation a ouvert la voie à une véritable prolifération de ces appareils, appelés systèmes embarqués. Notre vie quotidienne dépend de plus en plus de ces systèmes : automobile, téléphonie, moyens de paiements, thermostats, compteurs électriques, etc.

D'un point de vue industriel, cette prolifération soulève de nouveaux défis. Le cycle de développement (de la spécification initiale au produit final) doit être de plus en plus court pour affronter la concurrence. Aussi, les nouveaux produits intègrent toujours plus de fonctionnalités, et ce au prix d'une complexité elle aussi croissante. C'est ainsi que nous pouvons maintenant acheter des téléphones portables capables de communiquer par GSM, 3G, Wifi, Bluetooth et USB, de prendre des photos et des films, d'écouter de la musique, de naviguer sur internet, d'assurer un guidage routier par GPS, etc.

Cette augmentation de la complexité des systèmes nécessite des efforts de développements conséquents, qui vont à l'encontre des objectifs industriels de réduction des coûts et des délais de production. De plus, les méthodes de développement employées ne sont pas toujours adaptées à ces nouvelles contraintes et ne permettent plus d'assurer un niveau de qualité suffisant. En pratique, cela se manifeste par des dysfonctionnements qui peuvent aller du simple désagrément, comme le redémarrage d'un téléphone, jusqu'à des accidents graves avec des conséquences humaines, comme le crash d'un avion.

C'est dans ce contexte que de nouvelles méthodes de développements ont été mises au point. Nous nous intéressons dans cette thèse aux méthodes fondées sur l'utilisation de composants. Ces méthodes s'articulent autour de la composition de briques élémentaires (les composants) pour construire un système. L'idée directrice est que chaque composant possède une spécification permettant de s'assurer de sa compatibilité avec d'autres composants. Ces spécifications prennent des formes variables d'une approche à l'autre et peuvent contenir par exemple l'ensemble des services requis et fournis par le composant, son comportement à l'exécution, etc. Ainsi, seuls des composants compatibles peuvent être utilisées ensemble, assurant que le système est correct par construction. Les composants peuvent être arrangés en bibliothèques pour qu'ils puissent être réutilisés, assurant ainsi un gain en temps et en argent pour les développements. Dans le domaine de l'électronique, cela fait maintenant plusieurs années que ces méthodes sont employées avec succès. Dans le domaine du logiciel, les résultats sont encore mitigés, avec des problèmes pour assurer la correction du système assemblé.

Problématique

Cette thèse se place dans le contexte du développement de logiciel fondé sur les composants pour les systèmes embarqués. Nous souhaitons dans nos travaux apporter des éléments pour une meilleure assurance de la correction des systèmes assemblés. Nous introduisons dans les paragraphes suivants les étapes principales employées lors du développement d'un logiciel. Nous présentons aussi la notion de modèle d'exécution, celle-ci ayant un rôle majeur lorsque la dynamique d'un système à l'exécution est étudiée.

Méthodes de développement

Les étapes principales du développement d'un logiciel

Dans ce manuscrit, nous désignons par *système* le logiciel développé. Au cours du développement, le système est représenté sous différentes formes : des représentations abstraites, que nous appelons *modèles*, et une forme exécutable, que nous appelons *exécutable du système* (ou *système exécutable*). Le *modèle* est exprimé dans un *langage de modélisation*. Le système exécutable s'exécute sur une *plate-forme d'exécution*. Celle-ci peut être matérielle, on dit alors que le système s'exécute sur *machine nue* (« bare metal »). Cette plate-forme peut aussi être une combinaison de matériel et de logiciel : c'est le cas lorsque l'exécutable repose sur un système d'exploitation, avec éventuellement une couche d'intergiciel (« middleware ») entre l'exécutable et le système d'exploitation. Pour différencier les parties logicielles et matérielles, on fera référence aux *plate-forme logicielle* et *plate-forme matérielle*. La figure 1 illustre ces différentes combinaisons.

L'utilisation d'un système d'exploitation et d'un intergiciel permet respectivement d'abstraire le matériel et le système d'exploitation. Cela permet également de ne pas introduire la gestion du matériel dans le système en développement et ainsi de le rendre indépendant de ce matériel. Seul un changement de système d'exploitation impose des modifications dans le système développé. L'utilisation d'un intergiciel permet, de manière similaire, de rendre le système indépendant du système d'exploitation : les changements de système d'exploitation ou de matériel n'ont aucune incidence en terme de développement sur le système.

Le développement d'un logiciel repose principalement sur quatre étapes, qui font partie du *cycle de développement* d'un logiciel. Ces étapes sont détaillées dans les paragraphes suivants. On les retrouve dans la plupart des méthodes classiques (méthode de développement en V, en cascade, ...). Ces quatre étapes sont illustrées sur la figure 2.

Expression des besoins. Phase de récolte des besoins des utilisateurs, en terme fonctionnels (fonctions que doit remplir le système) et non fonctionnels (temps de réponse, sécurité, sûreté, ...). Par exemple, un robot automatique de découpe doit être capable de découper des pièces métalliques d'une certaine forme (propriété fonctionnelle). Le robot doit stopper la découpe si la température de la scie dépasse une température maximale ou si une personne pénètre dans un périmètre autour de la zone de découpe (propriétés non fonctionnelles). Le résultat de cette étape est une spécification. Celle-ci peut être purement textuelle (en langage naturel) ou elle peut utiliser des langages spécifiques (comme UML [1]).

Conception du système. La conception d'un système consiste en la création d'un modèle, à partir des spécifications obtenues lors de l'étape précédente, qui est une représentation abstraite

1. « Unified Modeling Language » : http://www.uml.org/

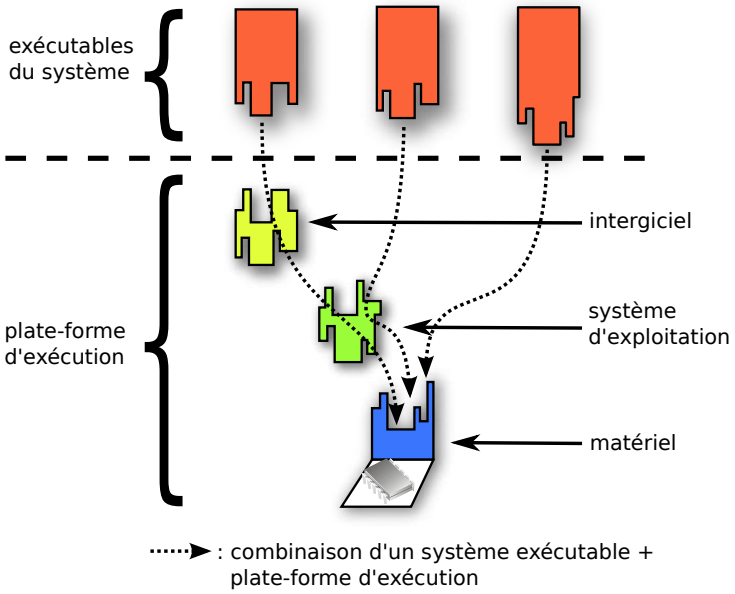


FIGURE 1 – Différentes plate-formes d'exécutions.

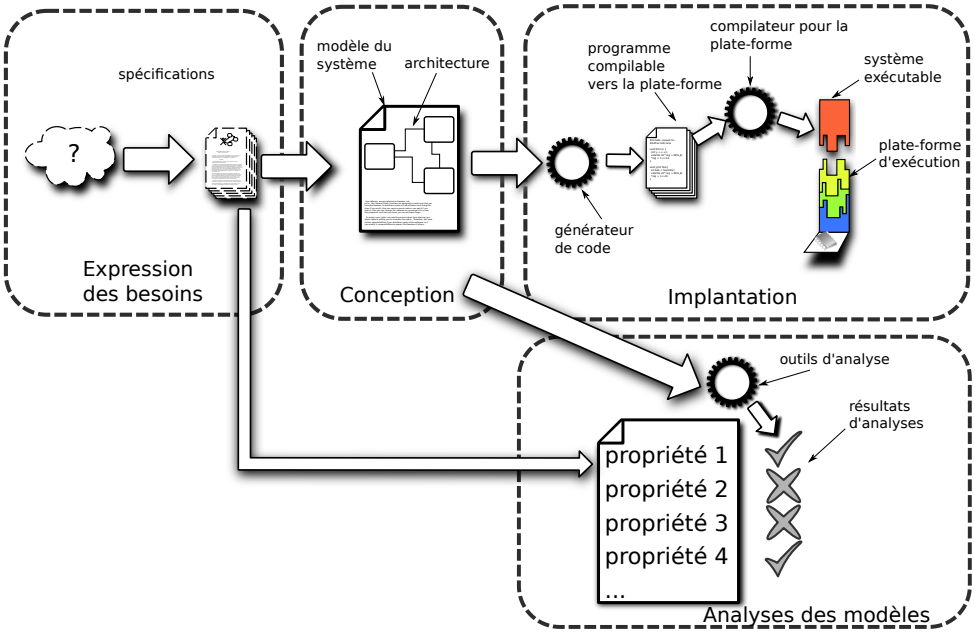


FIGURE 2 – Résumé du cycle de développement.

du système. Il existe de nombreuses solutions pour la création de modèles. Néanmoins, nous nous concentrons dans nos travaux sur les approches à composants, dont nous présentons un sous-ensemble dans la section 1.1. Les modèles issus de cette étapes doivent permettre :

- la structuration du système en « briques élémentaires » manipulables indépendamment les unes des autres. Nous verrons comment la notion de composant peut répondre à cette attente. Une approche classique est la structuration en objets, issue de la programmation orientée objets.
- l'expression du comportement du système à l'exécution. C'est-à-dire que le modèle doit

capturer les propriétés non fonctionnelles du système.

Un modèle du système peut par exemple servir à la structuration du code et des données du système, à la structuration des activités au sein du système. Il peut aussi définir une sémantique pour les communications qui peuvent avoir lieu entre différents éléments du modèle. Nous appellerons ces éléments qui caractérisent un modèle les *primitives* du modèle. Ces primitives sont définies par le langage de modélisation.

Implantation du système. L'implantation est l'étape durant laquelle le système exécutable est construit à partir du modèle du système. Concrètement, le système exécutable est un ensemble de codes exécutables (aussi appelés codes objets) spécifiques à la plate-forme d'exécution. Cette création de code exécutable peut se faire par génération directe à partir du modèle, ou plus couramment par génération de code source dans un langage de programmation pour lequel on dispose d'un compilateur qui générera à son tour du code exécutable (voir la partie droite de la figure 2).

Cette génération de code (exécutable ou source) à partir du modèle peut être totalement automatique, c'est-à-dire que le générateur produit l'intégralité du code à partir du modèle, ou semi-automatique s'il est nécessaire de compléter le code généré. C'est ce processus de génération qui permet de garantir la fidélité entre le modèle du système et le système exécutable (la définition de la fidélité est donnée dans le paragraphe suivant). Il est donc préférable d'automatiser au maximum cette étape d'implantation en déléguant aussi peu que possible la création de code à des développeurs, susceptibles d'introduire des erreurs comme par exemple des erreurs de lecture ou de compréhension du modèle ou des erreurs de programmation.

Techniques d'analyse de modèles. Les techniques d'analyse raisonnent sur le modèle créé pendant la conception et déduisent des propriétés sur le système exécutable, qui est obtenu à la fin de la phase d'implantation. Ceci dans le but de vérifier le respect des spécifications, produites dans la première étape, par le système exécutable. Les propriétés habituellement recherchées sont par exemple :

- l'absence d'interblocage dans le système,
- un temps de réponse du système borné, suite à la réception d'un événement,
- la certitude que le système n'effectura jamais une séquence d'actions donnée (par exemple parce qu'elle est dangereuse).

Les techniques d'analyse peuvent être séparées en techniques de **simulation** et techniques de **vérification**. Ces techniques reposent sur la notion d'*état du système*, qui peut être définie comme une « photo » du modèle correspondant au système à l'exécution. La nature exacte d'un état dépend du modèle et de son niveau d'abstraction. Par exemple, l'état peut être constitué des valeurs de l'ensemble des variables contenues dans le modèle.

À partir d'un état, la **simulation** permet de déterminer l'état atteint lors de l'étape suivante. La notion d'étape est liée au niveau d'abstraction du modèle : elle peut représenter un cycle d'horloge matériel ou bien encore une étape de calcul dans un algorithme. Si le modèle est non déterministe, le simulateur peut avoir à choisir entre plusieurs états suivants possibles. Le résultat de la simulation est donc une séquence d'états correspondant à une exécution du système exécutable parmi l'ensemble des exécutions possibles. Un simulateur n'est pas capable de comparer deux états pour vérifier s'ils sont équivalents ou non : il ne peut pas déterminer si l'ensemble des états que peut prendre le modèle a été parcouru de manière exhaustive ou non. Un simulateur déroule indéfiniment des séquences d'états. Pour cette raison, les résultats de la simulation sont limités : il est possible de vérifier une propriété uniquement pour les états du

modèle que le simulateur à parcouru, mais la simulation ne permet pas de vérifier une propriété pour l'ensemble des états.

Les techniques de **vérification** permettent au contraire de vérifier une propriété pour tous les états du modèle, mais au prix d'une complexité bien plus importante que la simulation. La vérification nécessite d'avoir une relation d'équivalence entre les états du modèle, c'est-à-dire un moyen de déterminer si deux états sont équivalents ou non. Cela permet en particulier le parcours exhaustif de l'ensemble des états (« model checking ») et donc la vérification de propriétés pour toutes les exécutions possibles. Néanmoins, ce parcours exhaustif est limité par la taille de l'espace de l'ensemble des états, qui grandit de manière exponentielle avec la complexité du modèle : les ressources en puissance de calcul et en stockage limitent les capacités d'exploration. Pour limiter cette explosion, il est possible d'utiliser des techniques de réduction du nombre des états. Il existe aussi des méthodes de vérification reposant sur une approche plus *mathématique* basée sur des preuves, mais nous ne les abordons pas dans nos travaux.

Dans les deux cas (simulation et vérification), la pertinence des résultats est fonction de la fidélité entre le modèle et le système exécutable. La **fidélité** peut être définie comme suit :

Une propriété vraie dans le modèle du système est vraie dans le système exécutable.

Un modèle est dit **complet** si en plus la réciproque est vrai :

Une propriété vraie dans le système exécutable est vraie dans le modèle du système.

Il est possible d'implanter le système exécutable de manière à conserver les propriétés du modèle, et donc d'assurer la *fidélité*. Par contre, assurer qu'aucune nouvelle propriété non vérifiable sur le modèle n'est ajoutée au système exécutable pendant sa création, c'est-à-dire assurer la complétude du modèle, est un problème difficile en pratique.

La figure 3 complète la figure 2 pour illustrer ce problème.

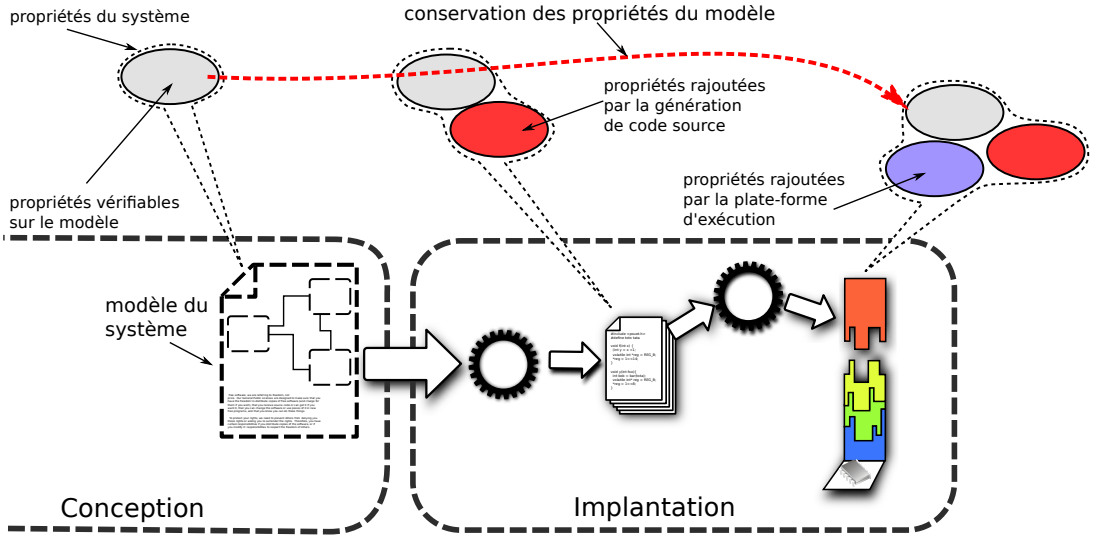


FIGURE 3 – Conservation de la fidélité entre modèle du système et exécutable du système.

Limitations

Le niveau d'abstraction utilisé lors de la construction du modèle a une influence déterminante sur les résultats des analyses ainsi que sur l'automatisation de l'implantation.

De manière générale, un modèle pour lequel il est possible d'obtenir automatiquement une implantation n'est pas adapté aux techniques d'analyses. L'implantation automatique requiert des modèles à un niveau d'abstraction faible (on parle de modèle à *grain fin*). De plus, les langages utilisés (*i.e.* langages de programmation) possèdent rarement une sémantique formelle, et l'histoire montre que les développeurs n'acceptent pas facilement de changer de langage. Ces deux points rendent inapplicables les techniques d'analyses, les modèles étant trop complexes.

Pour permettre les analyses, il faut être capable d'associer au modèle implantable à grain fin un modèle plus abstrait (on parle de modèle à *gros grain*) dont la complexité est plus faible. Cette association présente plusieurs difficultés. La première est qu'il n'est souvent pas possible de l'extraire depuis le modèle à grain fin, par exemple parce que ce dernier ne possède pas de sémantique formelle. Une autre difficulté provient du choix des abstractions faites, qui doivent être suffisantes pour réduire la complexité du modèle tout en permettant la conservation des propriétés qu'on cherche à vérifier. Ces modèles à gros grain sont en contre partie inadaptés à une implantation. En effet, pour un même système, un modèle à gros grain ne couvre potentiellement qu'un sous ensemble des spécifications initiales du système. Une implantation d'un tel modèle, si elle était possible, ne produirait donc pas forcément un système correct.

Composants

Une approche fondée sur *les composants* fournit des outils et/ou des méthodes pour les étapes de *conception*, d'*analyses des modèles* et d'*implantation*. Nous verrons dans le chapitre 1 que de telles approches couvrent de manière plus ou moins complète ces trois étapes. Néanmoins, elles accordent toutes une place centrale à la création d'un modèle durant l'étape de conception. L'unité de structuration de ces modèles est le *composant*[Szy97].

Définition. Les notions de *composant* diffèrent d'une approche à l'autre, mais la notion d'encapsulation est généralement considérée : un composant encapsule un *contenu* qui ne peut interagir avec son environnement (*i.e.* d'autres composants du modèle) que par des points d'accès explicites, souvent appelés *interfaces* ou *ports*. Ces interfaces sont en général orientées pour indiquer si les communications qu'elles portent *rentrent* dans le composant, on parle d'interfaces *serveur*s, ou *sortent* du composant, on parle d'interfaces *cliente*s.

Un modèle est obtenu par l'assemblage de composants. Les interfaces des composants sont liées entre elles par des connexions pour former une *architecture*. Un exemple est donné dans la figure 4.

Motivations. Une des motivations principales de l'utilisation de composants pour le développement de système logiciel est la *réutilisabilité*, qui permet de diminuer les coûts de développement. Elle s'inspire des pratiques courantes utilisées dans l'électronique : un système électronique est obtenu en assemblant des composants préexistants (souvent achetés à une entreprise) sur un circuit électronique. Les connecteurs métalliques concrétisent les *interfaces* et les pistes en cuivre les connexions.

Souvent, les interfaces sont typées, ce qui permet de vérifier qu'un assemblage est correctement typé (correct par construction), c'est-à-dire que les liaisons entre les composants connectent des interfaces compatibles.

Limitations. Une majorité des approches fondées sur les composants se concentre sur leurs aspects purement fonctionnels sans se préoccuper des aspects non fonctionnels. Un composant contient un ensemble de données ainsi que du code, et ses interfaces ne sont que des ensembles

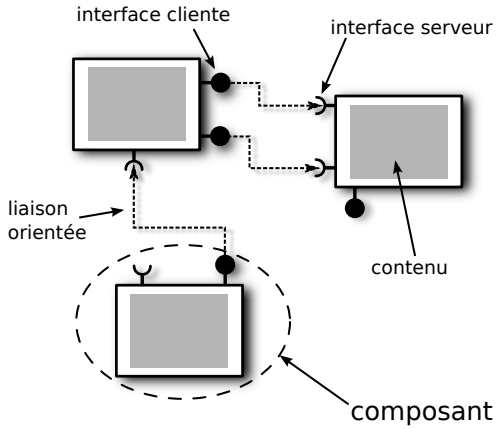


FIGURE 4 – Exemple d'assemblage de composants (architecture).

de signatures de fonctions. Aucun aspect non fonctionnel (*e.g.* temps d'exécution, présence de blocage, sécurité, qualité de service, ...) n'est pris en compte. La compatibilité des composants assemblés est donc limitée à la compatibilité des fonctions des composants et n'assure pas que le système est correct. C. Szyperski évoque cette limitation dans [Szy97, Szy03] lorsqu'il indique qu'un composant doit fournir de manière explicite ses dépendances vis-à-vis de son contexte en plus des dépendances fonctionnelles.

Modèles d'exécution

Un modèle d'exécution est la vue offerte au développeur des aspects dynamiques du système qu'il développe. Cette vue définit principalement la façon dont sont gérées les activités dans le système, c'est-à-dire leurs communications, leurs synchronisations et leur ordonnancement. Nous verrons dans les sections suivantes qu'il existe autant de modèles d'exécution que d'approches. Il est tout de même possible d'isoler deux tendances, que nous présentons dans les paragraphes suivants : les *threads* et les *acteurs*.

Les *threads*. Le modèle à *threads* est le moyen le plus primitif pour exprimer l'utilisation d'activités concurrentes dans un système logiciel. On appelle thread ou *fil d'exécution* l'abstraction d'une exécution séquentielle de code. Ainsi, un système *multi-threads* est un système qui contient plusieurs threads dont les exécutions se font de manière concurrente. Ces threads peuvent communiquer, principalement par partage de mémoire et se synchroniser avec l'utilisation de verrous (sémaphore, mutex, futex, etc). Ils partagent un ensemble de ressources, en général il s'agit des ressources matérielles de la plate-forme d'exécution : processeurs, mémoires, périphériques, etc. Un ordonnanceur gère le partage des ressources entre les différents threads, en se basant sur des critères qui leur sont associés (*e.g.* temps d'exécution, priorité, ...).

Ce modèle est très couramment utilisé car il étend directement le modèle séquentiel simple (qu'on appelle maintenant couramment *mono-thread*), qui reste le modèle le plus utilisé. Il correspond aussi directement à ce qu'offrent les plate-formes matérielles, un thread pouvant être vu comme une abstraction du fonctionnement d'un processeur : à chaque thread correspond un *contexte d'exécution*, c'est-à-dire l'état du processeur (valeurs des registres).

La simplicité apparente des mécanismes de communication et de synchronisation entre les threads s'est révélée être un piège et une des causes principales de défauts logiciels [Lee06]. Les comportements défectueux majoritairement observés sont :

– des interblocages dus à une mauvaise utilisation des verrous ;
– des courses lors d'accès à des données partagées. Ces courses, associées à l'ordonnancement souvent non déterministe des threads, se manifestent par des erreurs « aléatoires » difficiles à localiser et corriger.

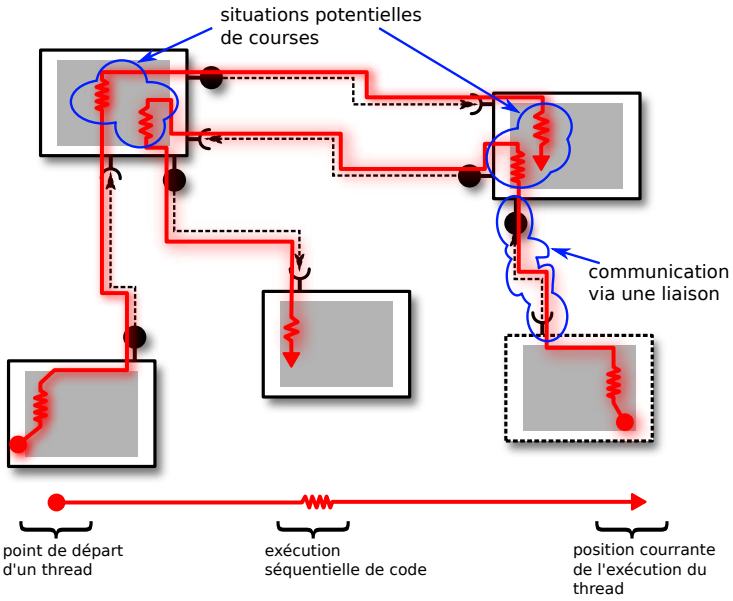


FIGURE 5 – Parcours d'une architecture par des threads.

Les threads sont couramment utilisés dans les approches fondées sur les composants, par exemple dans [MYC]. Ces threads parcourent l'architecture de manière arbitraire, comme illustré par la figure 5. Connaître l'état de chacun des threads pendant l'exécution du système, en particulier la position dans le code exécutable est un problème : il est difficile de déterminer l'état du système à l'exécution et encore plus de le prévoir. Lorsque la connaissance exacte de la localisation de chaque thread est nécessaire, il est possible d'introduire des mécanismes dédiés, avec des guichets à l'entrée et la sortie de chaque composant. Cette situation favorise l'introduction des défauts évoqués précédemment.

Les acteurs. Le modèle à *acteurs* [HBS73] peut être vu comme un mariage entre les composants et les threads. Les threads ne sont pas contraints localement dans le code qu'ils parcourent et peuvent potentiellement traverser tout le code du système. Le modèle à acteurs ajoute une contrainte en imposant le cloisonnement de chaque thread dans un composant, qu'on appelle alors acteur. L'utilisation d'acteurs évite aux développeurs l'utilisation des mécanismes de verrous et de mémoire partagée, et donc évite les problèmes qui leur sont liés. Le code et les données d'un acteur ne peuvent être accédés que par son seul thread, il n'y a pas de course possible. La figure 6 illustre cette notion d'acteur.

Le modèle à acteurs permet une structuration en termes d'activités, en plus de la structuration du code et des données. De manière identique à cette structuration du code et des données, nous sommes persuadés que la prise en compte des activités dans la structuration du système permet une meilleure maîtrise de la maintenance et du cycle de développement en général.

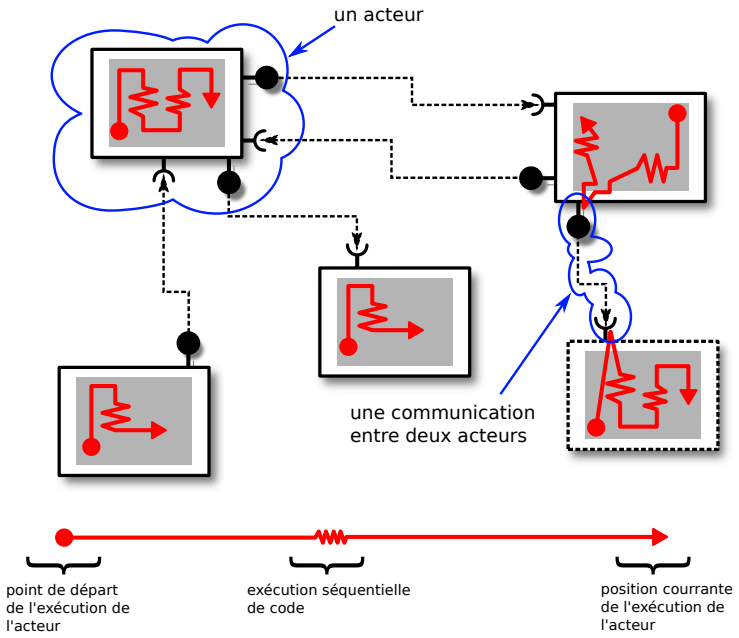


FIGURE 6 – Exécution des acteurs.

Organisation de la thèse

Nous donnons dans cette section le plan suivi dans ce manuscrit.

Première partie, l'état de l'art. Le chapitre 1 présente un ensemble de méthodes et d'outils fondés sur les composants. Cette présentation n'est pas exhaustive, mais balaye le spectre des solutions existantes dans les domaines de l'analyse et l'implantation des systèmes embarqués.

Deuxième partie, la contribution. Cette partie se compose de deux chapitres. Le chapitre 2 introduit une méthode de conception. Cette méthode concerne les étapes de conception, d'analyse et d'implantation. Elle vise à réduire les risques d'introduction de défauts à la conception et à améliorer les capacités d'analyses tout en étant implantable sur des plate-formes d'exécution concrètes. Le chapitre 3 présente un prototype d'implantation de langage et d'outils de la démarche présentée dans le chapitre 2.

Troisième partie, l'évaluation. Le chapitre 4 présente une évaluation du prototype présentée dans la partie précédente. Nous illustrons la mise en œuvre de notre méthode ainsi que des mesures des résultats obtenus à la fois sur un exemple académique et sur une application embarquée sur les nœuds d'un réseau de capteurs.

Conclusion et perspectives. Le dernier chapitre conclut cette thèse et présente des perspectives.

Annexes. Ce manuscrit possède deux annexes. L'annexe A présente des travaux préliminaires à BUZZ consistant en l'utilisation du langage BIP comme langage principal de programmation de système. L'annexe B présente un manuel de l'utilisateur du prototype de compilateur du langage BUZZ présenté dans le chapitre 3.

Première partie

État de l'art

Chapitre 1

État de l'art

[...] the flexibility and power is good, but it does mean that it's also easy to make a mess of it - the old UNIX philosophy of giving people rope, and letting them hang themselves with it if they want to.

Linus Torvalds.

Sommaire

1.1 Approches fondées sur les composants

Nous présentons dans cette section un ensemble d'approches fondées sur les composants. Cet ensemble n'est pas exhaustif mais permet d'illustrer le paysage existant pour le domaine des systèmes embarqués. Toutes les approches présentées dans la suite sont liées au domaine des systèmes embarqués, que ce soit d'un point de vue plus abstrait avec très peu de rapport avec l'implantation, ou au contraire d'un point de vue très proche du matériel. Comme nous le verrons dans la suite de ce document, notre travail peut être vu comme un pont entre ces deux extrêmes. Nous présentons ici des approches qui couvrent autant que possible le spectre compris entre ces extrêmes.

1.1.1 Ptolemy II

Ptolemy II[2] [EJL+03, HLL+03] (P2 dans la suite) est développé par le département de génie électrique et d'informatique (EECS) de l'université de Berkeley, sous la supervision du professeur Edward Lee. Ce projet étudie la modélisation, la simulation et la conception des systèmes concurrents, temps-réel, embarqués. Le projet fournit un ensemble d'outils et de documentations conséquent et complet.

Conception. La conception produit ce que P2 appelle un *modèle*, notion à laquelle nous ferons référence par *modèle-p2* dans la suite, pour éviter la confusion avec le modèle du système qui désigne le modèle global, tel que défini dans l'introduction pour la phase de conception. Ainsi, le modèle du système est un modèle-p2. Un modèle-p2 est une architecture hiérarchique obtenue par assemblage de modèles-p2 ou de *composants* (aussi appelés *acteurs*). Un composant dans P2 possède un contenu (détaillé plus loin) et des *ports*. Ces ports sont reliés entre eux par des liaisons définies par les modèles-p2. P2 ne définit pas de modèle d'exécution, et c'est là une de ses particularités. Chaque modèle-p2 contient en plus de l'architecture de composants/modèle-p2 un *domaine*. Celui-ci définit précisément comment interagissent et s'exécutent les composants et modèles-p2 contenus dans le modèle-p2 parent. C'est ce mécanisme qui permet de faire interagir des composants dont les fonctionnements diffèrent : par exemple des composants flots de données (« dataflow ») et des composants événementiels (« event triggered »). P2 fournit plus d'une dizaine de domaines (dont la liste complète est disponible dans la documentation [LHJ+01]), permettant la conception de systèmes hétérogènes (au niveau du modèle d'exécution).
Le contenu des composants est donné sous la forme d'un programme Java (il est possible d'utiliser d'autres langages). Ces programmes suivent une interface précise :

- ils fournissent chacun une méthode `fire()`, qui sera invoquée pour déclencher l'exécution du composant ;
- chaque port du composant implante une interface Java (appelée `Receiver`) du type `get/set` pour lire/écrire des données sur le port. Ce sont ces méthodes que le programme Java utilise pour communiquer au travers des ports.

Un domaine se décompose en deux parties :

- des classes Java d'implantation de l'interface `Receiver`. C'est ainsi qu'est définie la sémantique des communications entre les composants.
- un *directeur* qui se charge d'invoquer les méthodes `fire()` et de gérer les activités dans les composants (par exemple en plaçant un `Thread` Java dans chaque composant). Il s'agit de l'ordonnanceur.

Un exemple de modèle-p2 est donné dans la figure 1.1.

Analyses. P2 permet la simulation des modèles construits. Hélas, la vérification des modèles n'est pour l'instant pas possible, des travaux sont en cours pour ajouter cette possibilité.

Implantation. P2 permet la génération de code[3] mais cela reste à l'état de prototype. Pour ce faire, chaque composant doit être accompagné d'un modèle (« template ») de code C qui sera complété par le générateur de code. Ce générateur ne supporte actuellement qu'un ensemble restreint de domaines et la question de la récupération de tout le code existant (en Java) ne semble pas abordée. P2 est un outil de conception et permet l'exécution des modèles construits.

2. http://ptolemy.eecs.berkeley.edu/ptolemyII/
3. http://ptolemy.eecs.berkeley.edu/ptolemyII/ptII7.0/ptII7.0.1/ptolemy/codegen/README.html

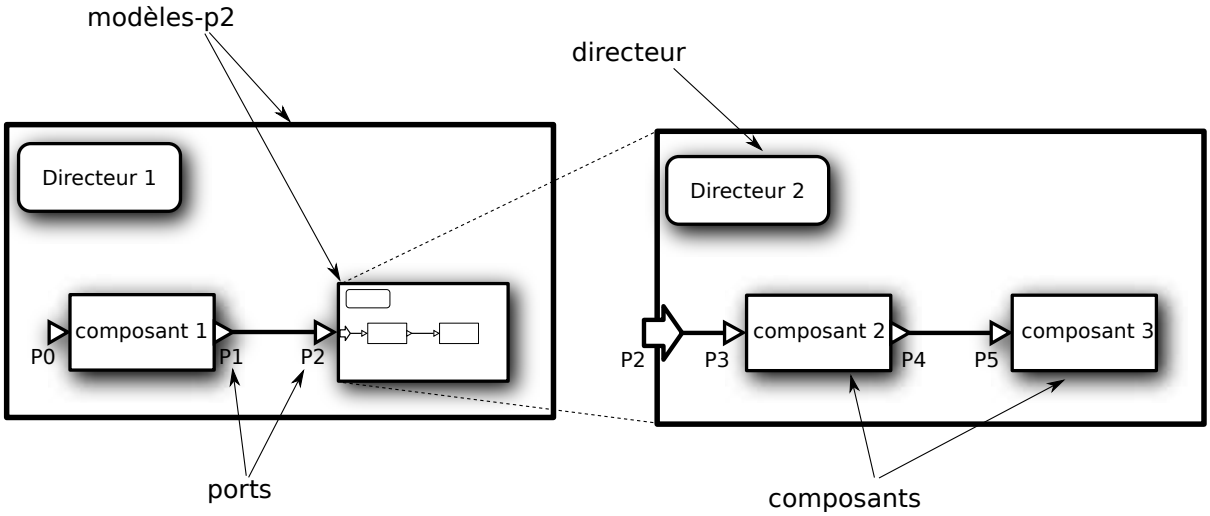


FIGURE 1.1 – Un modèle-p2 dans Ptolemy II.

Leurs compilations pour chargement sur les cibles embarquées n'est semble t il pas un objectif du projet.

1.1.2 AADL

AADL [AAD, FLV03], pour « Architecture Analysis & Description Language », est un langage de description d'architecture standardisé par le SAE (« Society of Automotive Engineers »), une communauté s'intéressant au domaine de l'ingénierie des véhicules. Développé initialement pour le domaine de l'avionique sous le nom de MetaH [Met, Ves00], il aborde maintenant le domaine des systèmes embarqués en général. Autour du langage AADL gravite un ensemble d'outils développés par une communauté composée à la fois d'universités et d'industries. Ces outils sont trop nombreux pour être listés de façon exhaustive, nous en présentons ici une synthèse.

Conception. Le modèle du système issu de la conception en AADL est une architecture de composants. Cette architecture couvre la partie applicative du logiciel du système mais peut aussi couvrir tout ou partie du logiciel habituellement réalisé par la plate-forme d'exécution (système d'exploitation et/ou intergiciel).

Chaque composant AADL appartient à une catégorie. Les composants pour le logiciel applicatif se répartissent parmi cinq catégories : les données, les sous-programmes, les *threads*, les groupes de *threads*, les processus (espace d'adressage virtuel et conteneur de un ou plusieurs *threads*). Les composants pour la plate-forme d'exécution se répartissent parmi quatre catégories : les mémoires, les processeurs, les périphériques et les bus.

Un composant AADL possède des interfaces et un contenu, appelé *implantation* dans la terminologie AADL. Les composants ne peuvent communiquer entre eux qu'au travers de leurs interfaces. Celles-ci sont directionnelles et se composent de ports de données (similaires à de la mémoire partagée), de ports à événements (communications asynchrones avec possibilité de file d'attente), de ports à messages (similaires aux événements avec des données attachées et file d'attente), d'appels synchrones à des composants sous-programmes et d'accès à des données d'autres composants.

Le contenu d'un composant peut prendre les formes suivantes :

– un ensemble de sous-composants avec leurs liaisons (modèle hiérarchique) ;

– le comportement du composant, donné dans un langage de programmation classique, par exemple en Ada95 ou en C ;
– le comportement du composant, donné par un automate [4] [ABS].

À chaque composant il est possible d'associer un ensemble de propriétés. Celles-ci possèdent un nom et une valeur et permettent d'exprimer des aspects non fonctionnels des composants (localisation de code source, temps d'exécution, période d'activation d'une activité, etc).

Les appels à des sous-programmes doivent être déclarés dans le contenu des composants *threads* ou sous-programmes, sous la forme d'une ou plusieurs séquences d'appels, particularité du langage AADL. Les paramètres et valeurs de retour de ces appels sont définis via l'utilisation de connexions (par exemple, un paramètre d'appel est connecté à une donnée locale au composant).

Analyses. Il existe de nombreux travaux concernant l'analyse des systèmes en AADL. [GH08] présente une analyse d'ordonnancement intégrée dans OCARINA [5]. Les résultats de l'outil Bound-T [6], qui calcule les temps d'exécution en pire cas à partir du code objet obtenu après implantation, sont intégrés dans le modèle AADL via des propriétés qui sont utilisées par Cheddar [7] pour effectuer l'analyse à proprement parler.

OSATE [8], un autre outil de référence dans la communauté AADL, permet une analyse de latence des flots de données [FH07]. Elle repose sur la notion de flot présente dans AADL, chaque composant peut spécifier quels sont les flots qui le traversent et quelle latence ce parcours introduit. La figure 1.2 donne la représentation graphique d'un composant définissant deux flots entre ses entrées et ses sorties.

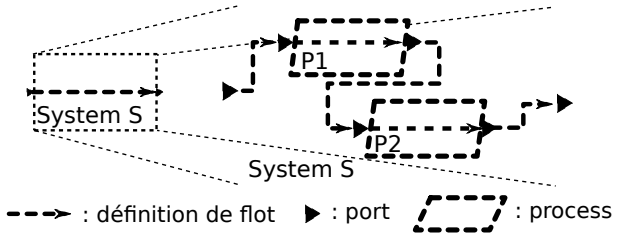


FIGURE 1.2 – Flots de données dans un composant AADL.

L'outil ADeS [9], qui s'intègre avec l'environnement de développement Eclipse [10] et OSATE, permet la simulation des systèmes AADL. Il ne supporte pas l'intégralité du langage et semble relativement limité mais permet tout de même de simuler l'ordonnancement des différents *threads*.

Enfin, il est possible d'analyser un modèle AADL en le traduisant vers un autre langage pour lequel des techniques et outils d'analyses existent. OCARINA permet la traduction vers des réseaux de Pétri. Il existe des traductions vers Fiacre [BBF+08], pour les vérifications concernant la sûreté et la sécurité. [CRBS08] donne une méthode de traduction de AADL vers BIP (voir la section 2.4 pour une description de BIP) et l'illustre en appliquant du « model checking »

4. Cette possibilité a été introduite dans une annexe à la spécification initiale de AADL
5. Suite d'outils OCARINA : http://ocarina.enst.fr
6. Page web de l'outil Bound-T : http://www.tidorum.fi/bound-t/
7. Page web de l'outil Cheddar : http://beru.univ-brest.fr/~singhoff/cheddar/
8. OSATE (« Open Source AADL Tool Environment ») : http://aadl.sei.cmu.edu/aadlinfosite/
9. « Architecture DEscription Simulation », simulateur pour AADL : http://www.axlog.fr/aadl/ades_en.html
10. Page web de l'environnement de développement Eclipse : http://eclipse.org/

pour vérifier l'absence d'interblocage, le respect des échéances des *threads* et la synchronisation entre des composants. Une traduction vers Lustre [JHR+07] offre des possibilités similaires de simulation et de vérification.

Implantation. L'implantation d'un système AADL peut se faire de différentes façons. La plus « classique » consiste à fournir, pour chaque composant *thread* et sous-programme, leur comportement dans un ou plusieurs fichiers d'implantation, donnés dans un langage de programmation, par exemple en C. Le standard définit une API pour interfacer ce code avec le reste du système (manipulation des ports, des données, etc). Ensuite, plusieurs outils sont capables de générer un ensemble de fichiers destinés à être compilés pour des plate-formes cibles. AADL permet l'expression de systèmes s'exécutant directement sur machine nue, mais ce n'est pas la solution adoptée par les outils principaux, qui utilisent une plate-forme d'exécution mixte (système d'exploitation, intergiciel et matériel). L'outil OCARINA permet, en plus du support classique du langage C et Ada, l'intégration de code Lustre, Esterel ou encore Simulink.

L'implantation peut être totalement automatisée (c'est-à-dire sans avoir à fournir de code pour les composants `thread` et sous-programme) dans certaines situations. Par exemple lorsque le contenu des composants est donné sous la forme d'automates.

1.1.3 UML-RT / ROOM

ROOM [SGW94] pour « Real-Time Object Oriented Modeling », a été développé par ObjecTime. Ses concepts ont été intégrés dans l'outil d'IBM Rose Real-Time (RRT dans la suite) en tant que profil UML, appelé UML-RT [Sel98]. Lors de sa publication en 2003, UML 2.0 intégrait à son tour des concepts inspirés de ce profil. Cette section présente UML-RT et les possibilités offertes par l'outil d'IBM.

Conception. Un modèle UML-RT est composé d'un ensemble de *capsules* liées entre elles. Cette notion de capsule correspond à celle d'acteur dans ROOM, et plus généralement à la notion de composant telle que nous l'avons vu dans les sections précédentes.

Une capsule, exporte un ensemble de *ports* qui forme son interface. Ces ports peuvent être de plusieurs types, dont les deux principaux sont :

- les ports terminaux, qui sont directement liés à un automate (unique pour chaque capsule, voir plus loin) qui représente tout ou partie du comportement de la capsule.
- les relais, qui servent de relais entre l'extérieur de la capsule et ses sous-capsules. Dans ce cas, la capsule est « composite ». Il n'y a pas de limite à l'imbrication.

Ces ports représentent l'unique méthode de communication des capsules avec leur entourage. Ils sont reliés entre eux par des connecteurs qui représentent les canaux de communications.

Une capsule peut contenir une description de son comportement sous la forme d'un automate. Cet automate est décrit avec le formalisme classique des « Statecharts » de UML. Historiquement, ROOM définissait un langage nommé « ROOMCharts ». Les transitions de ces automates peuvent être déclenchées par l'arrivée d'un signal par un des port terminaux de la capsule. Une transition peut aussi porter une action (du code dans un langage de programmation), et peut émettre un ensemble de signaux via les ports terminaux de la capsule. Au niveau d'un automate, les transitions sont exécutées de manière « run-to-completion », il n'est pas possible de tirer une nouvelle transition si une transition du même automate est actuellement en cours d'exécution.

Les connecteurs, qui représentent les canaux de communication, sont associés à un protocole qui définit un ensemble de rôles (un pour chaque port impliqué). Ces rôles définissent, au minimum, les ensembles de signaux en entrée et en sortie qui sont permis pour chaque port. Il

est aussi possible d'enrichir le protocole avec un automate définissant les séquences de signaux valides.

Analyses. RRT n'offre que des possibilités de simulation, aucune méthode de vérification n'est offerte. Cette limitation est pointée comme importante par [DS02] qui fait part d'une expérience d'utilisation de RRT pour développer un logiciel embarqué sur des imprimantes. Des travaux tentent de corriger cette limitation en proposant des techniques d'analyses plus puissantes. Hélas, celles-ci ne s'intéressent qu'à des résultats très spécifiques, et ne sont applicables qu'à des cas particuliers de systèmes. Nous pouvons citer [GF96] qui introduit une application de RMA [KRP+93](« Rate Monotonic Analysis ») à un modèle obtenu à partir de ROOM. Cette application nécessite un système dont les activités sont périodiques et la possibilité d'attribuer des priorités à ces activités. [WCkMT] présente des travaux pour l'application de techniques de « model checking » pour vérifier les protocoles utilisés dans les communications entre les composants. Ces travaux semblent cependant être restés dans un état préliminaire.

Implantation. RRT permet la génération de code C++ destiné à fonctionner avec un exécutif nommé « TargetRTS », à rapprocher de « MicroRTS » pour ROOM. En d'autres termes, les implantations issues de RRT reposent sur une plate-forme d'exécution mixte. Les activités du système sont implantées avec des *threads* de la plate-forme d'exécution. RRT permet d'attribuer une ou plusieurs capsules à un *thread*. Il permet aussi la priorisation des capsules à travers des des priorités sur les *threads* attachés aux capsules. En revanche, dans [MKH03], les priorités ne sont pas associées aux capsules, mais aux scénarios. Un scénario peut être vu comme un ensemble d'échanges de messages entre plusieurs capsules. Une même capsule peut être impliquée dans plusieurs scénarios de priorités différentes et la priorité de chaque *thread* n'est alors plus constante comme dans le cas précédent, il est nécessaire de la changer dynamiquement. Ces changement de priorités peuvent alors se révéler coûteux suivant le système d'exploitation utilisé : cela doit être pris en compte si les performances sont un facteur important.

1.1.4 PECOS

PECOS[11] [GCW+02, NAD+02] fournit un environnement pour la spécification, la composition, la configuration et le déploiement de systèmes embarqués construits à partir de composants.

Conception. PECOS repose sur les concepts classiques de *composant*, *port* et *connecteur*. Un composant PECOS possède un contenu, un ensemble de ports et un ensemble de *propriétés*. Trois classes de composants sont définies :

- les composants *passifs*, qui n'ont pas de contexte d'exécution propre. Ils sont invoqués de manière synchrone.
- les composants *actifs*, qui possèdent leur propre contexte d'exécution.
- les composants *événementiels*, similaires à des composants actifs, mais leurs exécutions sont déclenchées par des événements extérieurs (typiquement des interruptions).

Le contenu d'un composant peut être un ensemble de sous-composants dans le cas d'un composant composite, ou être directement le *comportement* du composant, donné dans un langage de programmation. Dans le cas de composant composite, des connecteurs relient les ports des sous-composants.

11. PECOS a été financé par la commission européenne (IST-1999- 20398) et par le gouvernement Suisse (BBW 00.0170). Il regroupait l'université de Bern (UNIBE) et des industriels Allemand (ABB, FZI) et Hollandais (OTI).

Les ports sont des points de partage de données : un port définit un nom de variable, un type et une direction (entrée, sortie, entrée-sortie). Ces données sont les uniques moyens de communication entre les composants.

Le comportement et l'ordonnancement des composants est très proche de ce qui a été présenté pour Ptolemy II. Ptolemy II pousse la séparation entre le comportement et le modèle d'exécution à l'extrême avec la délégation de ce dernier au domaine. L'approche de PECOS est moins radicale, les communications (données partagées) et les activités sont définies par PECOS et il n'est pas possible de les changer. Par contre, l'ordonnancement se fait au niveau de chaque composant composite : un père est responsable de l'ordonnancement des exécutions de ses fils. Cet ordonnancement doit être fourni pour chaque composite par le développeur.

Le comportement d'un composant se présente de manière similaire à celui d'un composant dans Ptolemy II : une procédure, qui est invoquée seulement par le composant parent, peut lire les données sur les ports d'entrées du composant, exécuter du code, et écrire des données sur les ports de sorties du composant. Les connecteurs sont utilisés pour décrire les relations de partage entre les ports.

Comme plusieurs contextes d'exécution sont présents dans le système lorsque des composants actifs et/ou événementiels sont utilisés, des problèmes d'accès concurrents aux données partagées (les ports) existent. Pour s'en prémunir, PECOS possède un mécanisme de synchronisation : chaque composant actif ou événementiel travaille sur une copie des données associées à ses ports. À des moment précis, définis par l'ordonnancement du composant composite, ces données privées sont synchronisées avec les données accessibles par les autres composants. Ces actions de synchronisation s'exécutant exclusivement dans le contexte du composant composite, il n'y a pas de risque de concurrence entre plusieurs composants.

Analyse. Plusieurs travaux s'intéressent à l'analyse des systèmes construits avec PECOS. En particulier, [NAD+02] détaille la façon de vérifier le comportement temporel de ces systèmes. Cette vérification repose sur la modélisation sous la forme de réseau de Petri de l'intégralité du système (comportements des composants, ordonnancements). Il est dans ce cas nécessaire de connaître les valeurs des temps d'exécution en pire cas de chacun des comportements des composants. Il n'est pas clairement précisé si les temps considérés ne concernent que le système développé ou si la plate-forme d'exécution est aussi prise en compte. En effet, les temps d'exécution de la plate-forme ne sont pas forcément négligeables, en particulier en ce qui concerne la gestion des *threads* (le basculement d'un *thread* vers un autre est relativement coûteux). Une autre limitation concerne la construction du modèle qui n'est pas automatique.

Implantation. PECOS permet de créer deux types d'implantation.

Le premier type repose sur la génération de code dans le langage C++ (le comportement des composants doit être donné en C++). Ce code est donné à un compilateur C++ classique. L'exécutable produit repose ensuite sur un intergiciel fourni par PECOS, appelé « Run-Time Environment » ou RTE, et un système d'exploitation, *e.g.* embOS.

Le second type utilise le langage Java (le comportement des composants doit être donné en Java). Le code généré est donné à un compilateur Java. Le résultat s'exécute ensuite sur une plate-forme d'exécution Java. Le plus souvent, il s'agit d'une machine virtuelle s'exécutant sur un système d'exploitation.

Les implantations produites par PECOS sont adaptées aux machines restreintes en terme de compacité de code et de ressources nécessaires (mémoire et puissance de calcul).

1.1.5 TinyOS

TinyOS se définit [12] comme un environnement de conception de systèmes événementiels pour les réseaux de capteurs embarqués. Le projet est porté par l'université de Berkeley et dispose d'une base de code et d'utilisateurs importante. Cet environnement est souvent utilisé comme référence pour l'implantation de systèmes pour les réseaux de capteurs.

Conception. TinyOS repose sur le langage de programmation nesC [GLvB+03]. Ce dernier est un dialecte dérivé du C orienté pour la programmation par composant. On retrouve les notions classiques de *composants*, *interfaces* et *liaisons*. nesC définit deux types de composants : les *modules*, qui contiennent du code nesC (du *comportement*) pour une ou plusieurs interfaces, et les *configurations*, qui contiennent des sous-composants connectés entre eux, par leurs interfaces, via des liaisons. Les interfaces en nesC sont orientées et bidirectionnelles. À chaque type d'interface on associe deux ensembles de méthodes, suivant le sens des appels : les *commandes*, du client vers le serveur, et les *événements*, du serveur vers le client. Ce découpage en événements et en commandes est fortement motivé par l'utilisation du motif appel/réponse (« split-phase »). Une commande représente un appel à un service, dont la réponse est signalée par un événement. Les événements sont aussi utilisés pour représenter les signaux émis par le matériel (interruptions). De manière schématique, les commandes descendent de la couche applicative vers les couches basses réalisant les services du systèmes d'exploitation et les événements remontent les couches dans l'autre sens. Les développeurs de nesC ont fait le choix d'utiliser des piles plutôt que des files pour les mises en attente des événements/commandes. C'est-à-dire qu'un nouvel événement/commande préemptera l'événement/commande en cours d'exécution.

En plus de ces deux types d'appels, nesC possède un mécanisme de tâche. Les tâches s'exécutent en « run-to-completion » et lorsqu'aucune commande ou événement n'est en attente. Les tâches ne peuvent pas se préempter entre elles. L'arrivée d'un événement ou l'invocation d'une commande suspend immédiatement l'exécution d'une tâche. Les tâches sont utilisées pour les étapes de calcul pouvant être réalisées en « tâche de fond ».

Analyses. nesC est un dérivé du langage C. L'intégralité du langage C est accepté pour la programmation du comportement des composants, à l'exception des pointeurs de fonction et de l'allocation dynamique de mémoire. nesC rajoute aussi la notion de bloc de code atomique dont la définition est similaire à celle de moniteur : un seul contexte d'exécution ne peut exécuter ce bloc à la fois.

La compilation est monolithique et prend en compte l'intégralité du système (il n'est pas possible de ne compiler qu'une partie du système par exemple). La rigidité imposée par nesC ainsi que la spécification de blocs de codes atomique permet au compilateur de détecter certaines situations de course.

TinyOS fournit l'outillage nécessaire pour la simulation des systèmes compilés. Cet environnement permet l'évaluation de la consommation d'énergie des nœuds de manière très fine (prise en compte des caractéristiques de la partie matérielle de la plate-forme d'exécution et simulation au niveau des instructions en assembleur).

Plusieurs travaux d'intégration de TinyOS avec d'autres méthodes existent. Viptos [CLZ06] permet la conception et la simulation de systèmes TinyOS depuis l'interface graphique de Ptolemy II. Il met en avant la possibilité de reposer sur d'autres domaines (temps continu, « dataflow », etc) de Ptolemy II pour la modélisation plus précise de certaines partie du système

12. Définition de TinyOS sur la page internet du projet : http://www.tinyos.net/faq.html#SEC-16

(environnement physique, dissipation énergétique, etc). Une traduction dans le langage BIP d'un système TinyOS est aussi possible. Ce travail est détaillé dans ce manuscrit en annexe A.2. Le modèle en BIP obtenu ouvre la voie à des méthodes d'analyses puissantes (*e.g.* « model checking ») et permet aussi la simulation. TinyOS n'inclut pas de méthode de vérification.

Implantation. Une implantation de système TinyOS est un résultat monolithique. La structure en composant n'est qu'une notion de conception, celle-ci disparaît intégralement pendant la compilation. Le compilateur `nesc` est une extension du frontal C de la suite de compilation GCC (« GNU Compiler Collection »), et peut être assimilé à un générateur de code C. Le code exécutable produit s'exécute sur machine nue, c'est-à-dire que l'intégralité du logiciel est décrit par l'architecture développée et compilée.

1.1.6 Fractal et Think

Nous présentons ici Fractal [BCS02, BCL+06] et Think [FSLM02, AHJ+09], qui sont à la base de notre contribution. La présentation donnée dans cette section sera complétée dans la section 2.3 qui donne en particulier plus de détails techniques sur Think.

Fractal est un modèle à composants dont le but est l'administration de systèmes logiciels. Cette administration peut couvrir tout le cycle de vie du logiciel : conception, implantation, déploiement et reconfiguration. Fractal offre pour cela des mécanismes de structuration du code et des données, ainsi que des mécanismes d'introspection et d'intercession. Tous ces mécanismes n'étant pas forcément adaptés selon les cas d'utilisation, Fractal définit un ensemble de niveaux de conformité permettant de doser ce qui doit être utilisé. Le niveau 0 correspondant plus ou moins aux mécanismes offerts par les langages orientés objets tel que Java (strict minimum) alors que le niveau 3.3 indique le support de tous les mécanismes de Fractal (niveau maximum). Fractal n'est pas un modèle dans le sens classique du terme. Ses mécanismes sont présentés de manière générique, sans lien avec aucune application précise. Certains détails, nécessaires à une utilisation concrète du modèle, sont laissés ouverts par Fractal et doivent être spécifiés par ses instances. En ce sens, Fractal est un méta-modèle, ou modèle abstrait. L'écosystème Fractal se compose donc du modèle et de ses nombreuses instances, chacune ayant ses spécificités (systèmes distribués, applications multimédia, systèmes embarqués, ...).Dans la suite de cette section, nous présentons à la fois le modèle Fractal et une de ses instances, Think.

À son origine, Think [FSLM02] (pour « THink Is Not a Kernel ») visait uniquement la création d'application sur machine nue, c'est-à-dire comprenant à la fois l'application et le nécessaire pour piloter le matériel et l'exécution du logiciel (services habituellement rendus par un système d'exploitation). Il s'est révélé être capable d'adresser un ensemble plus large de cibles : de la station de travail possédant déjà un système d'exploitation au micro-contrôleur 8bits nu. L'adhérence du projet au domaine des systèmes d'exploitation réside dans sa bibliothèque de composants nommée Kortex, qui aborde clairement l'implantation de services de systèmes d'exploitation : allocation dynamique de la mémoire, gestionnaire d'interruption, ordonnanceur de tâche, ... Un des fils directeurs de Think est d'utiliser les composants pour modéliser l'intégralité du logiciel, des couches applicatives aux couches les plus basses contenant les pilotes matériels de la plate-forme. Think est avant tout un outil de génie logiciel se focalisant sur la production d'une implantation concrète du système, aucune méthode d'analyse comme la simulation ou le « model checking » n'étant intégrée. Plus de détails sur Think sont donnés dans la section 2.3.

Conception. Un *composant* Fractal, illustré dans la figure 1.3, est constitué d'un *contenu*, d'une *membrane* et d'un ensemble d'*interfaces*. Ces interfaces peuvent être *clientes* dans le cas

de services requis par le composant, *serveurs*, dans le cas de services fournis par le composant, ou de *contrôle* dans le cas de contrôle offert par le composant. Ces interfaces représentent les seuls points d'accès aux composants et implantent ce que FRACTAL appelle des interfaces de langage, autrement dit, un type d'interface. La nature de ces types d'interface n'est pas définie par FRACTAL et doit être spécifiée par les instances de FRACTAL.

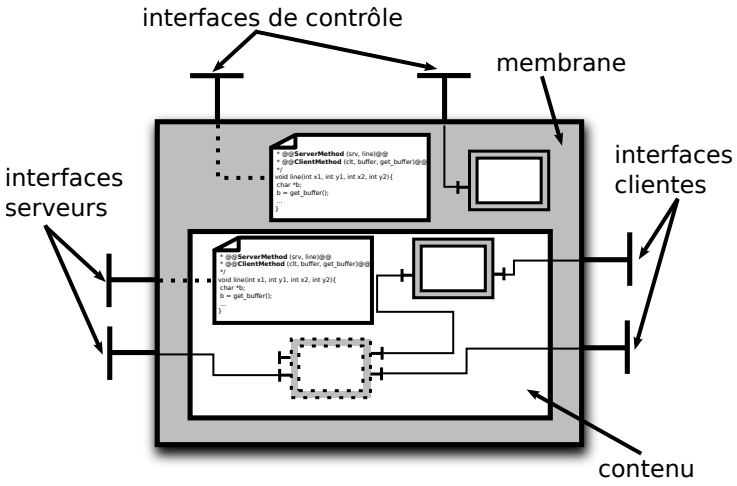


FIGURE 1.3 – Un composant Fractal

Le contenu d'un composant, qui ne concerne que ses aspects fonctionnels, peut être constitué directement du comportement des interfaces serveurs ou de sous-composants (voir la figure 1.3). La membrane (voir aussi la figure 1.3) est similaire au contenu mais ne concerne que les aspects non-fonctionnels (contrôles) du composant (gestion du cycle de vie, services d'introspection et d'intercession, ...). Elle peut contenir elle aussi soit directement le comportement des interfaces de contrôle, soit des sous-composants. Cette distinction entre interfaces fonctionnelles et interfaces de contrôle permet la séparation nette des services rendus par le composant de son administration. La définition de la nature du comportement (pour les interfaces serveurs et de contrôle) est encore une fois à la charge des instances de FRACTAL.

Nous verrons plus tard que THINK cible en particulier des plate-formes d'exécution matérielles, et donc la production d'implantation sur machine nue. Pour remplir cet objectif THINK a choisi l'utilisation du langage C dans plusieurs étapes du développement, y compris pour la définition du comportement des interfaces serveurs et de contrôle (contenu des composants). Pour cette raison, les types des interfaces se présente sous la forme d'un ensemble de signatures de fonctions en langage C.

Les liaisons entre composants, uniques moyens de communication entre eux, se font depuis une interface cliente vers une interface serveur de même type. Aucune sémantique concernant les communications portées par ces liaisons n'est définie par FRACTAL. Toujours poussé par son utilisation du langage C et de sa faible distance avec la plate-forme matériel, THINK ne définit pas à proprement parler de sémantique, mais utilise la sémantique d'appel de fonction du langage C : une invocation de méthode est bloquante et la méthode appelée s'exécute dans le contexte d'exécution de l'appelant.

La hiérarchie joue un rôle important dans FRACTAL. Elle permet au sein d'une même architecture de bénéficier de différents niveaux de détails de manière homogène. L'architecture THINK de la figure 1.4 donne un exemple de cette possibilité d'abstraction et de raffinement. Le plus haut niveau (composant nommé `toplevel`) correspond à l'intégralité du logiciel. Ce composant

contient deux sous-composants :

- `application` qui assure les fonctions attendues du système.
- `lowlevel` qui assure les fonctionnalités « bas niveaux » comme la gestion mémoire ou les communications radio.

À leur tour, ces deux composants contiennent des sous-composants. Les feuilles de cette architecture sont des composants contenant uniquement du comportement.

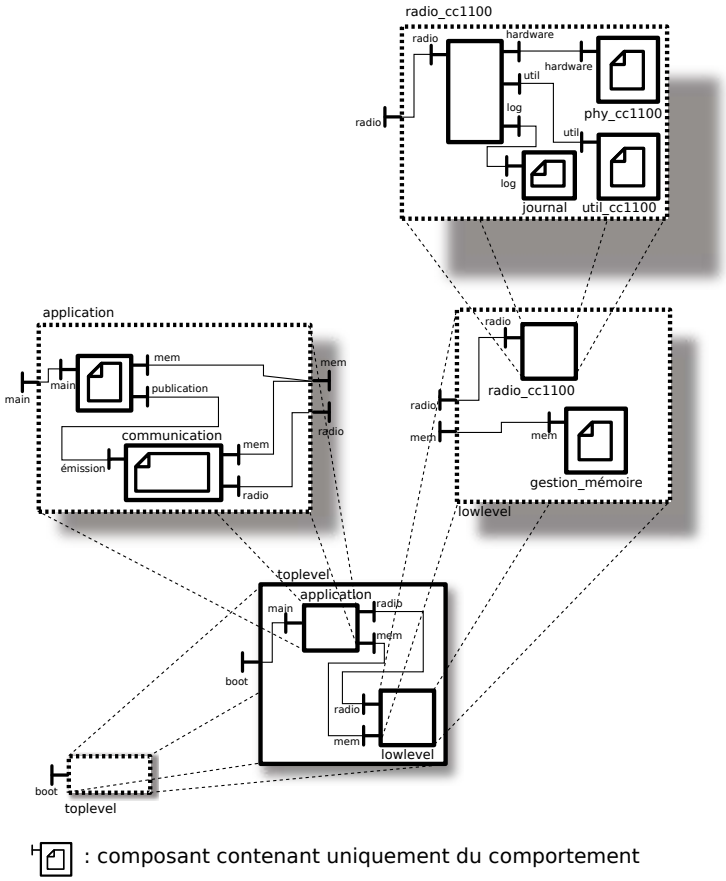


FIGURE 1.4 – Architecture d'un système en FRACTAL (la membrane n'est pas représentée pour alléger le dessin)

La hiérarchie rend difficile le partage de composants représentant des ressources communes à tout le système pour lesquels une unique instance est autorisée. Par exemple, dans la figure 1.4, le composant `journal` qui permet de tenir un journal des actions effectuées dans le composant `radio_cc1100`, pourrait être utilisé dans le composant `application`. Si le journal doit être unique, il est nécessaire d'utiliser la même instance du composant `journal`. Il serait alors obligatoire de rajouter des interfaces et des liaisons aux composants `lowlevel`, et `radio_cc1100` pour permettre la liaison indirecte entre `journal` et `application`. Ces interfaces et liaisons alourdissent inutilement l'architecture. FRACTAL règle ce problème avec la notion de *composant partagé* qui peut être inclus dans plusieurs composants différents mais désignant toujours la même instance. Dans notre exemple, `journal` pourrait donc être partagé entre `application` et `radio_cc1100`, sans nécessiter l'ajout d'interface ou de liaisons aux composants `radio_cc1100`, `lowlevel` et `application`.

FRACTAL ne définit pas de langage de description d'architecture, et cela marque une diffé-

rence avec la plupart des approches à composants existantes. Il définit seulement la façon dont les composants peuvent être manipulés. Une architecture peut être obtenue dynamiquement par création à l'exécution des composants, statiquement, semi-statiquement en créant les composants puis en les déployant, etc. Ceci fait partie des nombreux points laissés ouvert par FRACTAL et que ses instances doivent spécifier.

THINK a fait le choix de décrire l'architecture du système à déployer dans un langage textuel spécifique, que nous appelons simplement ADL dans la suite.

FRACTAL ne possède aucune notion d'activité (*thread*, acteurs, etc). THINK ne spécifie pas plus et se base encore une fois sur le langage C : un composant privilégié, appelé `boot`, est le point d'entrée de l'exécution. Plusieurs activités concurrentes peuvent être utilisées en écrivant le code C des composants de manière à créer ces activités. En d'autres termes, ce sont les composants qui créent dynamiquement des threads.

Nous verrons en section 2.3 comment THINK permet d'étendre tous les comportements calqués sur le langage C.

Analyses. FRACTAL et THINK ne fournissent aucun outil ou technique d'analyse, que ce soit pour la simulation ou la vérification.

Implantation. Ce point n'est absolument pas couvert par le modèle FRACTAL. Ce paragraphe concerne uniquement les choix pris dans THINK.

Le compilateur de THINK génère un ensemble de fichiers en langage C destinés à être compilés par un compilateur C externe pour la plate-forme cible avant chargement et exécution. Ces fichiers C sont obtenus à la fois par réécriture du code fourni avec les composants ainsi que par synthèse de code. Ces manipulations utilisent une représentation du code sous forme d'arbre, ce qui ouvre la voie à des optimisations puissantes, comme décrit dans [LP08, LNMB09]. En particulier, la souplesse permise par FRACTAL, comme l'introspection ou l'intercession, peut être finement configurée. Les coûts qui lui sont associés, comme les tailles de code et de données en mémoire ou encore en temps d'exécution, peuvent être maîtrisés. Potentiellement, toute plate-forme pour laquelle il existe un compilateur C est utilisable avec le compilateur de THINK. Actuellement, le compilateur C de GCC et IAR (fourni par IAR Systems pour plate-formes ARM) sont utilisés.

1.1.7 Autres approches

Parmi les approches existantes mais qui ne sont pas présentées en détails ici, nous pouvons citer Tinap, Metropolis et Vest. MyCCM-HI (« Make Your Component-Container Model-High Integrity »), qui partage beaucoup avec AADL, semble aussi très prometteur. Il affiche des ambitions de vérifications et d'implantation (au dessus d'un système d'exploitation). Nous encourageons donc le lecteur à consulter la page du projet [13] pour de plus amples informations.

TINAP TINAP (« TINAP Is Not only A Profile »), issu des travaux de thèse [Loi08] de Frédéric Loiret, s'intègre aux méthodes de conception dirigées par les modèles. TINAP fournit un espace de conception (ou profil) basé sur un modèle de composant de haut niveau. Ce modèle permet la conception d'applications multi-tâches à « contraintes temps réel embarquées ». [Loi08] détaille l'utilisation de ce modèle à différents niveaux d'abstraction pour la conception d'un

13. MyCCM-HI : http://sourceforge.net/projects/myccm-hi/

système dans sa globalité. L'implantation des systèmes dans TINAP repose sur l'utilisation du patron « conception / programmation orientée membrane ».

Metropolis [GSV02] et son successeur Metro II [DDM+07] utilisent une définition classique des composants : un composant atomique contient du comportement (du code) et un composant composite contient des sous-composants. Leurs interfaces sont composées d'un ensemble de ports pouvant intervenir dans des communications asynchrones (envois d'événements), dans des rendez-vous (synchronisation des exécutions de deux méthodes) et dans la publication d'informations (affichage d'une valeur mesurée par exemple). Chacun des composants possède une activité et leur ordonnancement se fait en trois phases. Les deux premières récoltent les événements émis par des composants et leur attachent un ensemble de valeurs quantitatives (estampille temporelle, température, etc). La dernière phase élit un sous-ensemble de ces événements en fonction des valeurs qui leur sont attachées et ordonne l'exécution des méthodes associées. Pour l'utilisation de composants dont les modes de fonctionnement diffèrent (flot de données et temps continu par exemple), Metro II utilise des adaptateurs. Ils agissent comme des filtres sur les événements (par exemple en effectuant un échantillonnage). Cette méthode s'inspire de l'électronique où les convertisseurs analogiques/numériques sont courants. Metro II fournit une partie frontale (« frontend ») produisant une représentation interne à partir du *méta-modèle* (description d'architecture) metropolis. À partir de cette représentation, plusieurs parties finales (« backend ») peuvent être utilisées : génération de code C++ pour la simulation et l'utilisation de moniteurs, génération d'un modèle utilisable avec le « model-checker » SPIN, etc.

Vest (« Virginia Embedded Systems Toolkit ») est un ensemble d'outils et de bibliothèques. L'outil principal est un éditeur graphique pour la création et l'assemblage de composants. Il permet aussi d'interagir avec des outils de simulation. Les composants dans Vest se repartissent dans deux groupes : composant *logiciel* et composant *matériel*. Le modèle est hiérarchique : des composants *composites* peuvent contenir des sous-composants. Les feuilles de cette hiérarchie sont les composants *sources* qui contiennent du comportement (du code source) ou des composants *matériel* décrivant des éléments précis (processeur, mémoire, ...). À chaque composant est associé un ensemble d'informations dites de « réflexivité » (localisation de fichiers de code sources, paramètres de compilation, temps d'exécution en pire cas, besoins mémoire, ...). Les références [Sta01, SZP+03, SNY+04, SNYH04] ne donnent que très peu d'informations sur la nature exacte des composants et de l'architecture. La gestion des activités dans le système se fait par l'association de *threads* à des composants pendant la construction du système. La particularité de Vest est son orientation « programmation par aspect » [SZP+03, SNY+04], au niveau de l'architecture. Le langage VPAL (« Vest Perspective Aspect Language ») est utilisé pour l'écriture d'aspects perspectifs. Le tissage de ces aspects peut modifier l'architecture du système. Des aspects peuvent aussi être employés pour la vérifications de propriétés comme l'existence d'un ordonnancement ou la présence de ressources mémoires suffisantes sur la plate-forme d'exécution.

1.2 Synthèse et positionnement

Nous donnons dans les paragraphes suivants une synthèse des sections précédentes. Elle se présente en trois parties qui matérialisent, comme nous le verrons dans le chapitre 2, les trois axes principaux de notre contribution.

1.2.1 Intégration du modèle d'exécution et des composants

L'histoire du développement logiciel montre que trop de liberté peut ouvrir la porte à l'introduction de défauts [Lee06]. C'est le cas avec l'utilisation des *threads* qui permet d'exprimer des situations de concurrences complexes, mais qui en contrepartie nécessite une extrême rigueur lorsqu'il est nécessaire de synchroniser plusieurs *threads* (interblocages, situations de courses pour des données partagées, ...). De manière similaire, le langage C est souvent critiqué pour sa trop grande permissivité, qui permet d'introduire des défauts qui ne peuvent pas exister avec d'autres langages. Par exemple, accéder à une adresse mémoire arbitraire est très simple en C et impossible en Java. L'utilisation du langage C reste tout de même incontournable dans certains domaines, en particulier lorsque le logiciel interagit directement avec la plate-forme matérielle. De manière similaire, nous sommes persuadés que la souplesse maximale des *threads* n'est que très rarement nécessaire. L'utilisation d'un modèle d'exécution plus restreint permet :

- de déléguer une plus grande partie du travail de développement aux outils (synchronisation, communication, détection d'erreurs, ...) ;
- une expression des aspects liés aux activités du système plus concise et moins sujette à des erreurs de programmation.

Nous constatons aussi qu'il existe deux tendances parmi l'ensemble des approches présentées ici. Celles mettant l'accent sur la conception et les analyses, qui se présentent en général avec des concepts de haut niveau d'abstraction, et celles mettant l'accent sur l'implantation. Ces dernières possèdent en général des concepts proches de la plate-forme d'exécution, c'est-à-dire à un faible niveau d'abstraction. Aux extrêmes, nous trouvons la rigidité du modèle d'exécution de TinyOS (pas de priorité, ordonnancement LIFO des commandes/événements et FIFO des tâches) d'une part, et d'autre part, la modularité totale du modèle d'exécution de Ptolemy II. Il n'existe pas de modèle d'exécution universel simple convenant à tous les cas d'utilisation. Une grande majorité des cas d'utilisation peut être réalisée soit en utilisant un modèle d'exécution très souple et générique, soit en permettant la création « à la carte » de modèle d'exécution. Nous pensons que la première solution mène à des problèmes similaires aux problèmes présentés pour le modèle à *thread* : la grande souplesse facilite l'introduction de défauts. Nous sommes convaincus que la possibilité de construction sur mesure permet au contraire de minimiser l'introduction de défauts par la meilleure maîtrise de la complexité des primitives manipulées par les développeurs. Cette construction sur mesure est d'autant plus facile que le modèle d'exécution est intégré à la structure en composants du système. Les approches présentées dans ce chapitre font ressortir des différences importantes dans cette intégration. Par exemple, PECOS et AADL font correspondre les activités avec des éléments de l'architecture : composants *actifs* pour le premier, composants *thread* pour le second. A l'opposé, THINK ne définit pas de modèle d'exécution et par conséquent, l'architecture d'un système développé avec THINK ne porte aucune information sur le comportement dynamique du système.

1.2.2 Double objectif : analyse et implantation

L'ensemble des approches présentées s'articule autour des deux objectifs suivants :

- les possibilités d'analyses du système développé : simulation pour l'évaluation de performances, vérification de l'absence d'interblocage, ...
- l'implantation d'un exécutable du système adapté aux plate-formes embarquées : compacité du code et des données et puissance de calcul en adéquations avec les ressources limitées des plate-formes matérielles.

L'obtention d'un seul des deux objectifs est déjà un problème complexe. On constate d'ailleurs que l'ensemble des approches présentées dans ce chapitre se concentre principalement sur un des deux objectifs et considère éventuellement le second. Ptolemy II par exemple se concentre sur les capacités de simulation, ses possibilités d'implantation sont marginales en comparaison. À l'opposé, THINK vise uniquement l'implantation de systèmes.

Nous pensons que cette approche en deux temps n'est pas la plus adaptée pour aboutir à la réalisation des deux objectifs. En effet, chacun de ces deux objectifs implique une solution potentiellement incompatible ou inadaptée à la réalisation de l'autre objectif. Autrement dit, il faut les considérer simultanément. Par exemple, la vérification d'un système développé avec THINK n'est pas envisageable, celle-ci étant équivalente à la vérification d'un programme écrit en langage C faisant un usage intensif des pointeurs. Ce problème est communément reconnu comme difficile. La raison principale de cette impossibilité de vérification est l'utilisation du langage C : il est très adapté à l'implantation, mais pas du tout pour la vérification. Réciproquement, un système analysable peut être difficilement implantable.

La raison est que les deux objectifs favorisent deux approches différentes. Les analyses sont simplifiées par l'utilisation de constructions abstraites (par exemple, des communications du type diffusion atomique). Les analyses nécessitent généralement des ressources matérielles (mémoires, puissances de calcul) et logicielles (système d'exploitation, bibliothèques de code) à l'échelle des dernières avancées techniques : utilisation de processeurs 64bits, cadencés à plusieurs Ghz, associés à plusieurs GiB de mémoire. A l'opposé, les implantations reposent sur des langages (C, assembleur) dont le niveau d'abstraction est faible car très proche de la plate-forme matérielle. L'implantation de constructions complexes (comme la communication en diffusion atomique) peut s'avérer très difficile. Les ressources matérielles disponibles sont aussi à une échelle différente : les processeurs fonctionnent en 8, 16 ou 32bits, cadencés à quelques Mhz, associés à quelques KiB de mémoire, souvent lente.

Pour conclure, nous sommes persuadés qu'il est primordial de prendre en compte de manière équitable les objectifs d'analyse et d'implantation. Il est aussi important de fournir aux développeurs la possibilité de définir leurs propres modèles d'exécution, en fonction du système développé. Ce dernier point passe par le support du modèle d'exécution par l'architecture du système.

1.2.3 Composants, de la conception à l'implantation

On remarque que parmi les approches présentées dans ce chapitre, les concepts de composants et d'architectures sont au cœur de la phase de conception. À l'inverse, les composants peuvent n'être utilisés qu'à la conception et disparaître durant l'implantation, celle-ci ne conservant pas forcément la structure. TinyOS est une illustration de ce procédé : l'architecture est complètement perdue lors de l'implantation et le résultat est un code exécutable monolithique. Une telle approche permet de ne pas payer un sur-coût à l'exécution dû aux composants. L'encapsulation à l'exécution influe sur l'occupation en mémoire avec un code et des structures de données plus complexes mais aussi sur les temps d'exécution. En particulier, il faut traverser les interfaces, ce qui se traduit en général par des indirections supplémentaires. En contrepartie, le code exécutable produit est potentiellement intriqué et devient un ensemble monolithique indissociable. La modification d'une partie du modèle à la conception (l'architecture) impose de procéder à une réimplantation complète de tout le système sans possibilité de réutiliser une partie de l'implantation précédente. La traçabilité du processus d'implantation est aussi plus délicate, car dès les premières étapes de l'implantation, l'architecture disparaît, se « dilue » dans le code généré. Le suivi d'un élément de l'architecture en particulier pendant l'implantation est

difficile.

A l'inverse, les composants et l'architecture peuvent être conservés tout au long de l'implantation et jusqu'à l'exécutable final. Cette approche permet de suivre précisément un composant durant tout son cycle de vie : depuis sa naissance à la conception jusqu'à son exécution. La traçabilité de chacun des éléments de l'architecture est ainsi assurée. Cette traçabilité est un élément important pour la validation d'un processus d'implantation et l'assurance d'une fidélité entre la description du système à la conception et l'exécutable du système. Une autre propriété qu'il est possible d'exploiter est une propriété de localité. La localisation des changements dans le code produit suite à une modification de l'architecture est précisément définie. Ceci peut-être immédiatement exploité dans les opérations de gestion du code après déploiement, comme la création de « patch » binaire ou toute modification locale du code exécutable.

Cette conservation représente un sur-coût. Néanmoins, des travaux [LP08, LNMB09] démontrent qu'avec une série d'optimisations, il est possible de maîtriser ce sur-coût pour obtenir un code exécutable équivalent en termes de performances et empreinte mémoire aux approches « monolithiques ». Ces méthodes d'optimisations sont utilisées lorsque le sur-coût associé à la conservation de la structure à composant devient rédhibitoire. Ces méthodes permettent, en fonction du contexte, de déplacer le code exécutable produit dans un continuum qui va du maintien complet de l'architecture jusqu'à sa disparition partielle ou totale. Dans le cas où l'architecture n'est pas conservée dans l'exécutable, pour assurer la préservation de la fidélité entre description du système à la conception et système exécutable, il est nécessaire de valider les différentes optimisations. Nous sommes convaincus que cette démarche est plus facile et source de moins d'erreurs que l'approche visant à générer directement un code monolithique et optimisé.

Il est également possible que cette disposition joue un rôle dans un éventuel processus de certification de l'outillage ou du code produit. Cette traçabilité entre la représentation du système à la conception et son exécutable apparaît bien comme une caractéristique intéressante dont l'obtention n'est finalement limitée que par la disposition de ressources suffisantes à l'exécution. Elle devrait donc être d'autant plus recherchée dans l'avenir que la disponibilité de ressources à l'exécution tend à s'accroître avec l'avance des technologies micro-électroniques.

A l'échelle d'un système réparti dans lequel certains des sites souffrent d'un manque de ressources ou d'autonomie, comme les réseaux de capteurs, les méthodes d'optimisation citées précédemment peuvent y être sollicitées. Elle rendent possible la préservation d'un même modèle architectural à base de composants en dépit d'une forte hétérogénéité.

Deuxième partie

Contributions

Chapitre 2

Présentation générale

Sommaire

2.1 Introduction

Ce chapitre introduit la démarche qui est à la base de l'ensemble de la contribution de nos travaux et qui se place comme une réponse possible aux attentes de la conclusion du chapitre précédent.

Nous présentons en section 2.2 la démarche que nous avons suivie dans nos travaux. Cette démarche est illustrée par un démonstrateur dans le chapitre 3. Les bases techniques nécessaires sont présentées dans les sections 2.3 et 2.4.

2.2 Une démarche pour l'analyse et l'implantation de modèles

Objectifs. Nous souhaitons créer un langage de modélisation remplissant les trois objectifs présentés en fin du chapitre précédent :

- l'intégration du modèle d'exécution dans le langage de modélisation ;
- la prise en compte équitable des possibilités d'analyse et d'implantation ;
- un processus de compilation permettant la traçabilité des composants durant leur cycle de vie (conception, analyse, implantation et exécution).

Pour obtenir à la fois des possibilités d'analyse et d'implantation, ce nouveau langage doit être traduisible vers deux langages, chacun spécialisé dans une de ces tâches. Nous distinguons dans une première étape la construction de ce nouveau langage, puis dans une seconde étape, la construction des deux traductions. Ces deux étapes sont en réalité dépendantes l'une de l'autre. Plus le nouveau langage s'inspire des deux langages cibles des traductions, plus ces traductions seront simples. En ce sens, le nouveau langage peut être vu comme un rapprochement des deux langages cibles. Cette dépendance entre le langage de modélisation et les choix techniques pour les analyses et l'implantation des modèles marque une différence par rapport aux approches du type MDA [14] où les aspects modèles sont décorrélés des aspects techniques, en particulier pour l'implantation.

Construction d'un nouveau langage de modélisation pour la conception. Cette construction consiste en l'identification d'un ensemble de primitives appartenant à un langage de modélisation et permettant l'expression des systèmes embarqués. Cette construction est pragmatique car les primitives rassemblées se veulent réellement utiles et utilisées. Nous verrons dans le chapitre 3 comment nous avons construit un tel ensemble à partir de l'étude présentée dans le chapitre 1. Les primitives du nouveau langage peuvent aussi bien être inspirées de langages mettant l'accent sur les analyses que de langages plus axés sur l'implantation (y compris des langages de programmations comme C ou Java). Cette construction se fait toujours en gardant les objectifs d'analyses et d'implantation en tête, ainsi seules les primitives pour lesquelles une traduction vers les deux langages cibles est possible sont considérées.

Analyse et implantation du nouveau langage. Pour permettre à la fois d'analyser et d'implanter les systèmes modélisés avec notre nouveau langage, nous définissons deux traductions du modèle du système vers deux modèles dans deux langages différents :

- le premier pour lequel des techniques d'analyses sont utilisables. Des exemples de langages cibles sont les réseaux de pétri ou le langage BIP.
- le deuxième pour lequel des techniques d'implantation existent. THINK, TinyOS ou directement le langage C sont des candidats.

De manière similaire à ce que nous avons présenté pour le cycle de développement dans l'introduction, le problème de la fidélité se pose. Pour assurer la fidélité entre le modèle à la conception et les deux modèles traduits, nous nous assurons que ces derniers sont corrects par construction. Pour cela, chacune des deux traductions est décomposée en un ensemble de règles de traduction élémentaires. Il est alors envisageable de valider chacune de ces règles.

Extension du langage. Le langage peut être étendu de manière itérative. Il est possible de rajouter des primitives à condition de systématiquement compléter les deux traductions. La figure 2.1 illustre la création et l'intégration de ce nouveau langage (`Nlang` dans le schéma) dans le cycle de développement.

2.3 Think

Nous commençons la présentation de THINK [AHJ+09] (brièvement introduit dans la section 1.1) par une introduction aux langages qui lui sont associés. THINK sera ensuite présenté

14. MDA pour Architecture Dirigée par les Modèle (« Model Driven Architecture »). Voir http://www.omg.org/mda/.

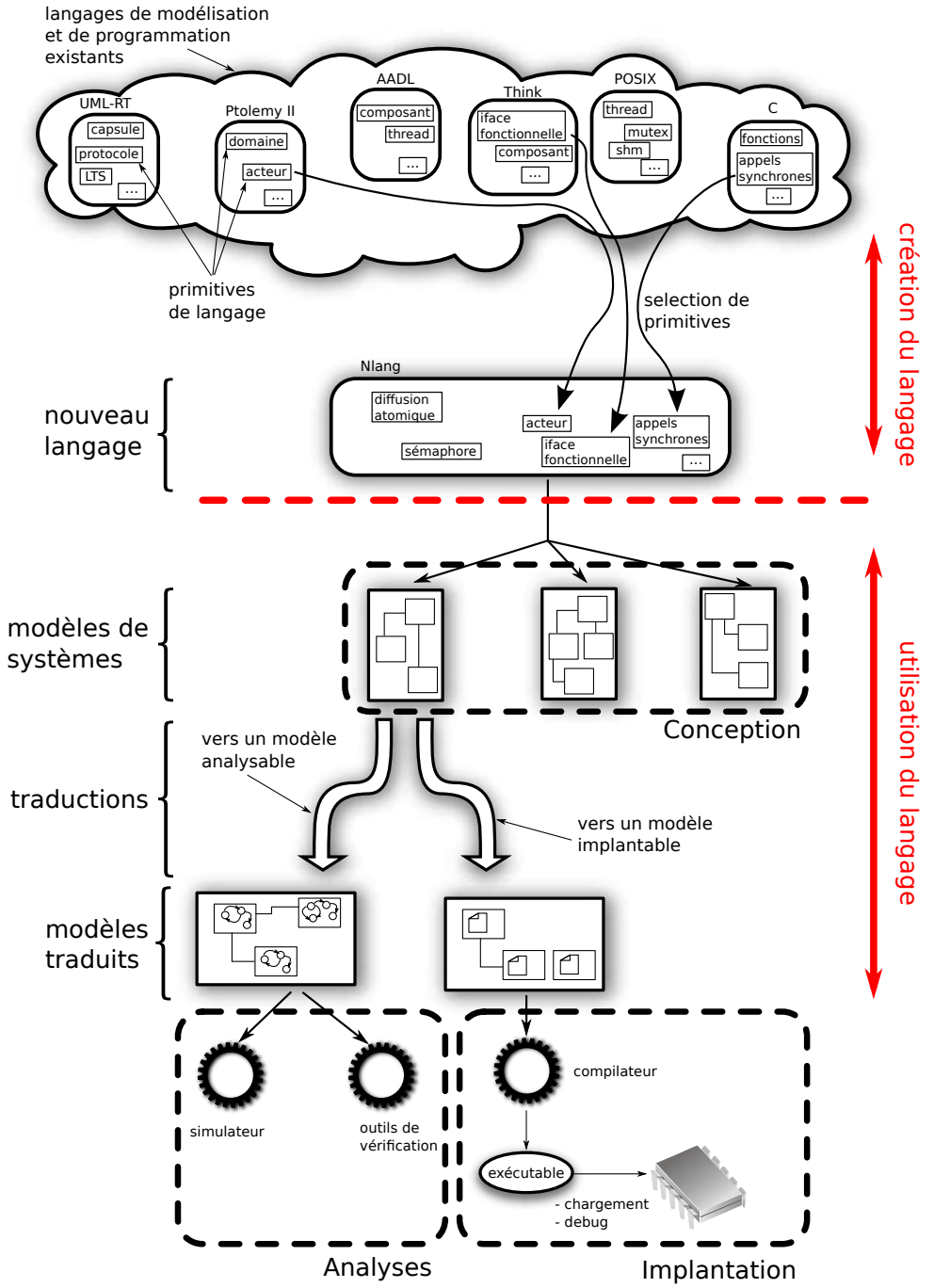


FIGURE 2.1 – Intégration du nouveau langage de modélisation dans le cycle de développement.

selon deux points de vues dans les sections 2.3.2 et 2.3.3. Le premier aborde les aspects d'implantations permis par THINK. Le second présente la façon dont THINK peut être utilisé comme un canevas de construction de compilateur d'architecture. Enfin, une dernière section présente plusieurs cas d'utilisation de THINK mariant ces deux facettes.

2.3.1 Présentation des langages associés

THINK repose sur plusieurs langages :
- le langage ADL pour décrire les composants ;
- le langage IDL pour la définition des interfaces ;
- le langage nuptC pour la programmation du comportement des interfaces serveurs.

Ces trois langages sont présentés brièvement dans les sections suivantes. Pour une meilleure compréhension de ces langages, le lecteur est renvoyé au manuel du programmeur [THI].

ADL est utilisé pour la définition des composants. Ce langage est simple, textuel et lisible (à opposer à un langage basé sur XML par exemple). Une définition de composant contient :
- un nom ;
- une liste d'interfaces serveurs ;
- une liste d'interfaces clientes ;
- un contenu : du code pour les interfaces serveurs et/ou des sous-composants.

Le listings 2.1 illustre les constructions principales du langage. Cet exemple est composé de trois définitions de composants : `foocomp`, `barcomp` et `topcomp`. Les deux premiers sont dit *primitifs*, car ils ne contiennent que du code d'implantation via la directive `content`. Le composant `topcomp` est un composant *composite* car il contient des sous-composants. Ces sous-composants, `foo` et `bar`, sont respectivement des instances des deux définitions `foocomp` et `barcomp`. La dernière liaison du composant `topcomp` relie deux interfaces serveurs. Ce mécanisme, appelé « export-bind », permet la délégation d'une interface serveur d'un parent à un de ses fils. Le même mécanisme existe pour les interfaces clientes. La figure 2.2 illustre de manière graphique une instance de la définition `topcomp` précédente.

```
component foocomp {
 provides Iface_t1 as srv1
 provides Iface_t3 as srv2
 requires Iface_t2 as clt
 content fooimplem
}
component barcomp {
 provides Iface_t2 as clt
 requires Iface_t1 as srv
 content barimplem
}
component topcomp {
  provides Iface_t3 as srv
  contains foo = foocomp
  contains bar = barcomp
  binds foo.clt to bar.srv
  binds bar.clt to foo.srv1
  binds this.srv to foo.srv2
}
```

Listing 2.1 – Exemple simple d'ADL THINK

Le langage **IDL** est utilisé pour définir les types d'interface des composants. Un type d'interface est un ensemble de signatures de méthodes utilisant la syntaxe du langage C. Une liaison entre deux interfaces de type `t` ne peut porter que des invocations de méthodes dont la signature est

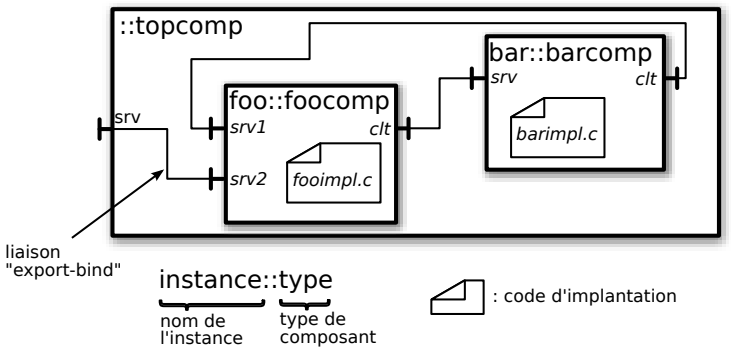


FIGURE 2.2 – Architecture simple avec THINK.

présente dans `t`. Un type d'interface appartient à un paquet (« package ») et possède un nom. En guise d'exemple, le listing 2.2 donne la définition du type d'interface `Framebuffer`, qui peut être utilisé pour exprimer la possibilité de dessiner dans une zone mémoire.

```
package a.package.name;
public interface Framebuffer {
 void ligne(int x1, int y1, int x2, int y2);
 void rect(int x1, int y1, int x2, int y2);
 char *buffer(void);
}
```

Listing 2.2 – Exemple simple d'IDL THINK

Le langage **nuptC** est utilisé pour le code d'implantation des composants. Il s'agit du langage C enrichi par des conventions de nom et des annotations dans les commentaires. La grammaire du langage C n'est donc pas modifiée. La majorité des conventions servent au compilateur pour faire le lien entre les éléments dans le code C et les éléments de l'architecture. Il doit être capable de trouver dans le code d'implantation les fonctions C correspondantes aux méthodes des interfaces d'un composant. Il existe plusieurs annotations et conventions de noms dont nous n'illustrons qu'un sous ensemble dans ce paragraphe. Le langage nuptC est décrit dans le manuel du programmeur [THI]. La figure 2.3 illustre cette correspondance en reprenant le type d'interface du listing 2.2 du paragraphe précédent, dans le cadre d'un composant possédant deux interfaces `srv` et `clt`, respectivement serveur et cliente. Cet exemple fait apparaître les annotations `ServerMethod` et `ClientMethod`, qui permettent la spécification des fonctions C correspondant aux méthodes d'interface.

2.3.2 Think pour la programmation de systèmes embarqués

Comme nous l'avons évoqué dans la section 1.1.6, THINK est particulièrement adapté pour la programmation des systèmes embarqués. Nous rappelons ses points forts à ce sujet :

i. l'interface de programmation est proche des interfaces classiques basées sur le langage C. Cela permet une prise en main rapide par les développeurs systèmes ainsi que la réutilisation de codes existants.

ii. la génération de code C est efficace et permet de maîtriser totalement le compromis entre flexibilité et performance. Cette génération permet de cibler rapidement toute plate-forme

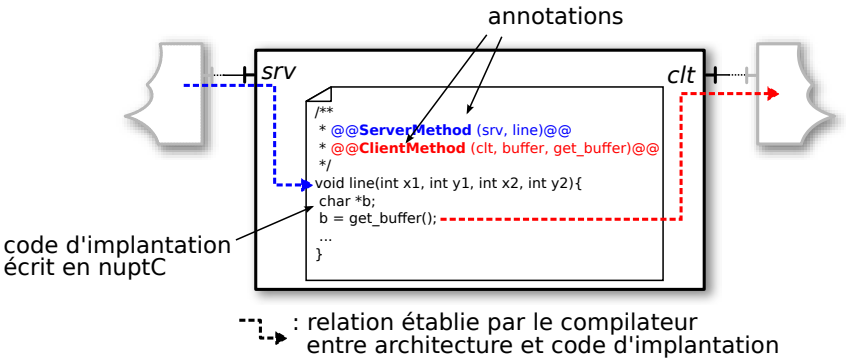


FIGURE 2.3 – Mise en correspondance des éléments d'architecture avec le code d'implantation.

pour laquelle il existe un compilateur C, y compris les plus contraintes en terme de puissance et de mémoire.

iii. le modèle à composants, basé sur FRACTAL, permet l'expression de l'intégralité du logiciel (de l'application jusqu'aux pilotes du matériel) au sein d'une même architecture.

Le niveau d'abstraction offert par THINK nous semble tout à fait adapté pour faire le pont entre notre nouveau langage de modélisation, qui contiendra des primitives plus abstraites que les primitives présentes dans THINK, et le langage C utilisé pour l'implantation par THINK.

Nous avons fait le choix d'utiliser THINK comme pivot pour l'implantation de notre nouveau langage plutôt que d'utiliser directement un langage de programmation comme C ou Ada. Ce choix est justifié principalement par les trois points suivants.

Premièrement, l'utilisation de THINK permet de se reposer sur la chaîne d'outils et la bibliothèque de composants KORTEX développée au sein du projet. Cette utilisation permet d'atteindre directement les plate-formes supportées par THINK sans nécessiter de développements spécifiques. Cela est particulièrement important pour le pilotage du matériel. La bibliothèque de composants KORTEX contient des gestionnaires d'interruption, des pilotes matériels pour les périphériques principaux (puces radio, liens série, capteurs de température, ...) ainsi que des composants pour l'initialisation du matériel (« boot »). Pour atteindre différentes plate-formes matérielles (ARM, AVR, x86, ...), il est nécessaire de supporter différents compilateurs C (GCC, IAR, icc, ...) et/ou différentes extensions du langage C. THINK supporte un ensemble de combinaisons compilateur/plate-forme matérielle et nous permet de nous en abstraire.

Deuxièmement, le modèle FRACTAL n'imposant aucun niveau d'abstraction pour l'utilisation des composants, la traduction de notre nouveau langage vers THINK s'en trouve simplifiée.

Enfin, comme nous le verrons dans la section 2.3.4, de nombreux travaux s'articulent autour de THINK. Même si une combinaison de nos résultats avec ces travaux sort du cadre de ce travail, nous sommes persuadés que c'est un axe de recherche à envisager.

2.3.3 Think comme canevas de construction de compilateur d'architecture

THINK, dans sa version 4 (aussi appelé Nuptse), est plus qu'un simple compilateur. Depuis la version 3, le compilateur de THINK est une application construite avec des composants qui repose sur une implantation en Java du modèle FRACTAL (appelée Julia). L'architecture du compilateur est décrite dans un ADL propre à Julia. Nous verrons comment le compilateur peut être simplement modifié en changeant cette description.

Tout d'abord, nous présentons l'organisation interne du compilateur Nuptse. Nous verrons ensuite ses possibilités d'extension.

De manière classique, un compilateur se décompose en trois parties :
- la partie *frontale* prend en entrée un ou plusieurs fichiers de code source (dans le cas de THINK, ADL, IDL et C) et construit une représentation interne au compilateur. Cette représentation prend souvent la forme d'un arbre, qu'on appelle Arbre de Syntaxe Abstraite (AST pour « Abstract Syntaxt Tree »).
- la partie *médiane* utilise cet AST pour appliquer des analyses et/ou des optimisations (dans le cas de THINK, il s'agit principalement de modification de l'architecture à compiler), en principe indépendantes du type de la sortie du compilateur.
- la partie *finale*, qui, à partir de la représentation interne, génère le code final. C'est le résultat du compilateur, il s'agit d'un ensemble de fichiers C dans le cas du compilateur de THINK.

La compilation d'une architecture avec THINK se déroule en trois phases : le chargement, la compilation et enfin la construction. La terminologie, héritée de l'outil FractalADL qui n'est pas un compilateur, n'est pas la meilleure qu'il soit. Nous parlerons dans la suite de compilation pour désigner le processus complet et d'étape de compilation pour désigner la phase interne au compilateur.

Ces trois étapes, que nous détaillons dans les paragraphes suivant, sont illustrées dans la figure 2.4.

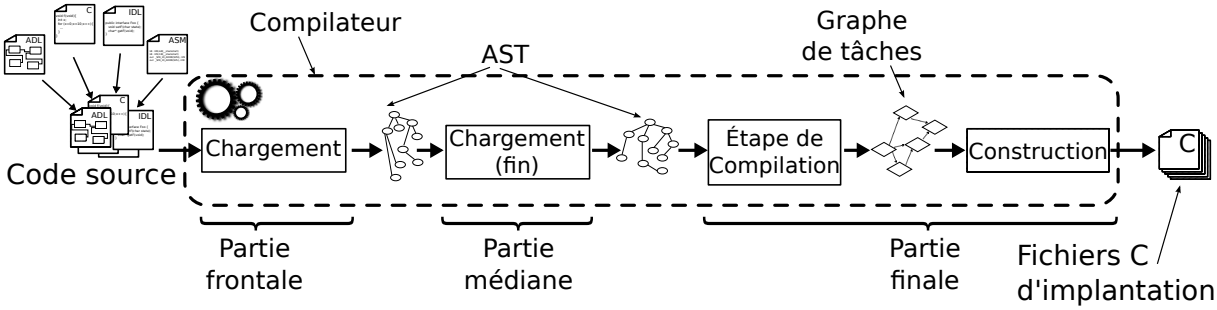


FIGURE 2.4 – Les trois phases du compilateur THINK mises en correspondance avec le découpage classique d'un compilateur.

1re phase : le chargement. Cette phase couvre à la fois la partie frontale et la partie médiane du compilateur. Le chargement consiste en la lecture de l'architecture (fichiers ADL et IDL) à compiler pour construire l'AST. On parle de « chaîne de chargement », car il s'agit d'une suite de composants qui vont être invoqués séquentiellement pour construire de manière incrémentale l'AST. Ces composants sont appelés composants de chargement. C'est aussi au sein de cette chaîne de chargement qu'ont lieu les opérations classiquement rattachées à la partie médiane du compilateur, c'est-à-dire non liées aux langages d'entrée.

2e phase : la compilation. Elle couvre partiellement le partie finale du compilateur. Elle est réalisée par un ensemble de composants (appelés composants de compilation), chacun chargé d'un aspect spécifique de l'étape de compilation (les liaisons, les attributs, le code fonctionnel, etc). Le compilateur parcourt l'AST, et, pour chaque nœud représentant un composant, il invoque séquentiellement les composants de compilation. Chacun de ces composants peut créer des *tâches* de construction, destinées à être exécutées par la phase suivante, une fois le parcourt de l'AST terminé. Ces tâches peuvent être dépendantes entre elles, par exemple, la tâche de création d'un

attribut ne pourra être exécutée qu'après la tâche de création du type des données d'instance d'un composant. Le résultat de cette phase est un graphe de tâches dont les arcs sont les dépendances.

3e phase : la construction. La construction complète la partie finale du compilateur. Les tâches créées durant la phase de compilation sont exécutées une par une dans un ordre compatible avec leurs dépendances. Une exécution de tâche consiste en l'invocation d'un composant de construction avec des paramètres propre à cette tâche. Le résultat de cette phase est le résultat du compilateur THINK, c'est-à-dire un ensemble de fichiers en langage C et les règles de compilation (utilisant un compilateur C classique) pour obtenir un exécutable.

Nous avons vu que la compilation d'une architecture se passe en trois phases, et que chacune des phases est réalisée par un ensemble de composants. La modification du comportement de ces phases se fait simplement par une modification de l'architecture du compilateur. Il est ainsi possible de modifier l'architecture compilée en modifiant la chaîne de chargement, ou de changer le code généré en changeant la partie finale.

2.3.4 Extension du compilateur

Mécanisme d'extensions

Une extension du compilateur de THINK peut prendre deux formes. La première consiste en la modification de l'architecture de compilateur : ajout, suppression et modification de composants. Ce type de modification permet potentiellement de modifier toutes les parties du compilateur mais demande en contrepartie une expertise technique pour leur mise en place.

La seconde consiste en la programmation d'un greffon logiciel (« software plug-in »). La majorité des travaux nécessitant des extensions du compilateur est localisée dans la partie frontale et médiane du compilateur, c'est-à-dire dans la phase de chargement. Pour simplifier ces extensions et réduire l'expertise technique nécessaire à leur programmation, THINK offre des mécanismes pour automatiser l'insertion de composants dans la chaîne de chargement ainsi qu'une méthode générique pour enrichir le langage ADL utilisé pour la description des systèmes développés. Nous présentons ces deux derniers points dans les paragraphes suivants.

Propriétés dans l'ADL. Il est possible d'associer pour toutes les constructions du langage ADL une liste de propriétés. Chaque propriété possède un nom et une valeur. La chaîne de chargement du compilateur remonte automatiquement ces propriétés dans l'AST pour que le reste du compilateur puisse les prendre en compte. Nous verrons que ces propriétés sont utilisées principalement pour l'expression de propriétés non fonctionnelles du système (aspect de sécurité, distribution, qualité de service, etc). Le listing 2.3 donne un exemple d'ADL avec des propriétés qui sont utilisées par la partie finale du compilateur pour guider la génération de code. Par exemple, la propriété `garbage` permet d'activer/désactiver la suppression du code associé à une interface non utilisée, et la propriété `const` indique que l'attribut ne sera jamais modifié à l'exécution.

```
component foocomp {
 provides api.Console as console [garbage=false]
 requires api.Framebuffer as fb
 content fooimplem
 attribute int x = 0 [const=true]
}
```

Listing 2.3 – Exemple d'ADL THINK utilisant des propriétés

Greffon logiciel. Nous avons vu que les modifications du compilateur se font en majeure partie dans la phase de chargement. C'est durant cette phase qu'il est possible de manipuler l'architecture du système compilé, par exemple pour ajouter ou supprimer des composants. Pour simplifier la programmation d'une extension, le compilateur permet l'écriture d'un simple composant respectant une interface donnée. Ce composant est chargé et inséré automatiquement dans la chaîne de compilation. Il reçoit l'AST sur une interface serveur et doit transmettre la version modifiée de cet AST via une interface cliente pour traitement par le maillon suivant de la chaîne. Ce nouveau composant peut reposer sur l'utilisation de propriétés telles que présentées dans le paragraphe précédent. La figure 2.5 résume l'association des mécanismes de propriété et de greffon.

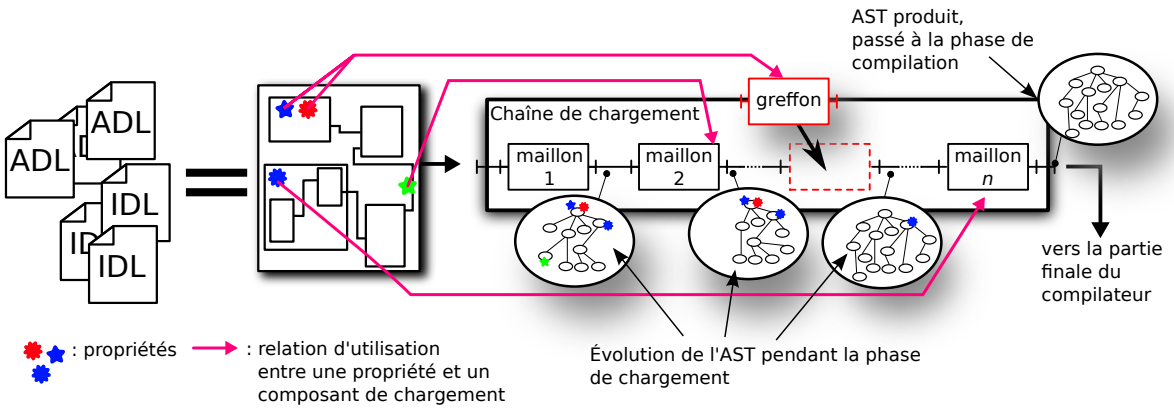


FIGURE 2.5 – Insertion d'un greffon dans la chaîne de chargement du compilateur.

Extensions existantes du compilateur

Nous présentons ici un ensemble d'extensions pour le compilateur qui reposent sur les mécanismes que nous venons de présenter. Cette liste n'est pas exhaustive, mais permet de mettre en valeur la polyvalence que ces extensions procurent au compilateur. Nous abordons les extensions en rapport avec la reconfiguration dynamique, l'isolation et enfin la protection. Le lecteur pourra trouver plus de détails sur ces extensions dans [AHJ+09].

Reconfiguration. La reconfiguration permet la modification d'un système pendant son exécution et peut être utilisée pour appliquer des rustines et des mises à jour, pour implanter des systèmes adaptatifs, pour de l'instrumentation dynamique ou pour supporter des modules de tierce partie. C'est un point important dans les systèmes embarqués où il n'est pas toujours possible d'arrêter un système en fonctionnement. Avec THINK, la reconfiguration dynamique peut être réalisée en plusieurs étapes :

- l'identification des parties du système à reconfigurer ;
- la suspension de leurs exécutions ;
- la modification du système (ajout, suppression de composants) ;
- le transfert de l'état vers les nouveaux composants ;

– la reprise de l'exécution.

La difficulté principale est de savoir si, au cours de l'exécution, un composant donné se trouve dans un état stable. En fonction du modèle d'exécution utilisé, deux mécanismes ont été mis en place. Dans un système à plusieurs *threads*, un compteur de référence est utilisé pour détecter la présence dans un état stable des composants. Il compte le nombre de méthodes en cours d'exécution dans le composant. Lorsque ce compteur tombe à 0, le composant est dans un état stable. Le système peut alors bloquer les nouvelles invocations en attendant la fin de la reconfiguration. Des composants intercepteurs, qui viennent se positionner au cœur des liaisons, comptent les invocations passant par ces liaisons. Dans un système piloté par des événements, un état stable est atteint à chaque terminaison d'un traitant d'événement. Les reconfigurations peuvent ainsi avoir lieu entre chaque traitement d'événements. Des propriétés dans l'ADL sont utilisées pour identifier les composants devant être reconfigurés. Un greffon spécifique est utilisé pour ajouter les composants intercepteurs. La démarche est détaillée dans [PS08].

Isolation. L'isolation permet de contrôler les communications entre différents domaines d'exécution. Ce mécanisme est souvent utilisé pour isoler des parties critiques d'un système, par exemple les pilotes de matériels du reste du système, c'est-à-dire des applications. Des exemples classiques de domaines d'exécutions se trouvent dans les systèmes d'exploitation où le noyau possède tous les droits alors que les applications sont restreintes (elles ne peuvent par exemple pas accéder directement au matériel). THINK permet de distribuer les composants d'un système dans plusieurs domaines différents. Un greffon se charge pendant la compilation d'insérer de nouveaux composants dans le système pour intercepter à l'exécution les communications utilisant des liaisons traversant les frontières entre les domaines.

Protection. La protection est un mécanisme de sécurité permettant de restreindre les accès à certains composants dans le système à l'exécution. Les liaisons concernées par ces restrictions sont surveillées par un moniteur de référence, qui décide à l'exécution, en fonction d'une politique de sécurité, d'autoriser ou non les invocations. Ce mécanisme, associé à la reconfiguration dynamique, permet de changer la politique de sécurité implantée par le moniteur de référence dynamiquement, par exemple lorsque l'appareil exécutant le système change de réseau. Le greffon appelé Cracker se charge d'insérer dans l'architecture du système des composants pour intercepter les invocations transitant sur des liaisons à protéger. Cracker se base sur des informations données par des propriétés dans l'ADL.

2.3.5 Conclusion

THINK permet la programmation de systèmes embarqués fondés sur les composants. Il ne force pas l'utilisation d'un niveau d'abstraction mais permet au contraire l'utilisation des abstractions propres à chaque besoin. Ce dernier point permet d'exprimer l'intégralité du système, depuis les couches basses liées au matériel jusqu'aux couches hautes, associées à l'application, dans un seul modèle. La réutilisation du code existant, en particulier dans la bibliothèque de composants KORTEX, permet d'atteindre une gamme conséquente de plate-formes matérielles à moindre effort. Ce sont pour ces raisons que nous avons décidé d'utiliser THINK dans notre prototype comme pivot pour l'obtention d'une implantation.

De plus, nous avons démontré l'extrême souplesse du compilateur. Il est relativement aisé de reprendre le compilateur de base et de l'étendre de manières différentes. THINK peut être vu comme un canevas de construction de compilateur d'architecture, dont le compilateur fourni n'est qu'un exemple pouvant servir de base à d'autres développements. Nous verrons que notre

prototype utilise intensivement ces possibilités d'extension pour à la fois étendre le langage en entrée du compilateur, mais aussi pour changer totalement la génération de code.

2.4 BIP

BIP [Sif05], pour « Behavior, Interaction, Priority », est un canevas logiciel pour la modélisation de composants temps réels hétérogènes. Dans une première section, nous présentons les caractéristiques de la composition de BIP, qui est son principal point fort. Nous présentons ensuite dans la section 2.4.2 le langage BIP sous ses formes textuelle et graphique. Enfin, la section 2.4.3 présente les outils disponibles.

Il est important de noter que nous présentons ici la première version de BIP. La deuxième version, qui est maintenant disponible et remplace la première, était encore en préparation lors de la réalisation de nos travaux.

2.4.1 Présentation générale

BIP est un canevas logiciel basé sur les composants. Il possède une sémantique formelle et s'intéresse en particulier à la manière de composer les composants entre eux. D'un point de vue abstrait, un composant BIP contient du comportement. Plusieurs composants BIP peuvent être composés grâce à une *glue*, et le résultat est un nouveau composant, voir l'illustration de la figure 2.6.

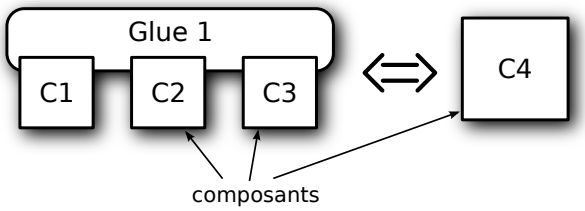


FIGURE 2.6 – Composition de composants BIP.

Cette composition respecte un certain nombre de propriétés, permettant la *constructivité*, qui est la possibilité de construire des systèmes complexes respectant des exigences à partir d'éléments de bases et d'une glue dont les propriétés sont connues. La constructivité est obtenue en respectant l'incrémentalité (*i.e.* un système peut être considéré comme la composition de composants plus petits avec la possibilité d'aplanir ou de décomposer), la compositionalité (*i.e.* il est possible d'inférer les propriétés du système à partir des propriétés des composants qui le composent) et la composabilité (*i.e.* les propriétés essentielles des composants sont conservées pendant la construction du système). Ces trois points sont illustrés dans les figures 2.7, 2.8 et 2.9.

A.Basu présente en détails dans [Bas08] la théorie sous-jacente à ces propriétés.

Reformulé d'une façon différente, nous pouvons dire que BIP consiste en l'empilement de trois couches nommées comportement, interaction et priorité. Le comportement est contenu dans les composants. Les couches interaction et priorité forment la glue dont il est question dans le paragraphe précédent. Ces trois couches sont présentées dans la section suivante.

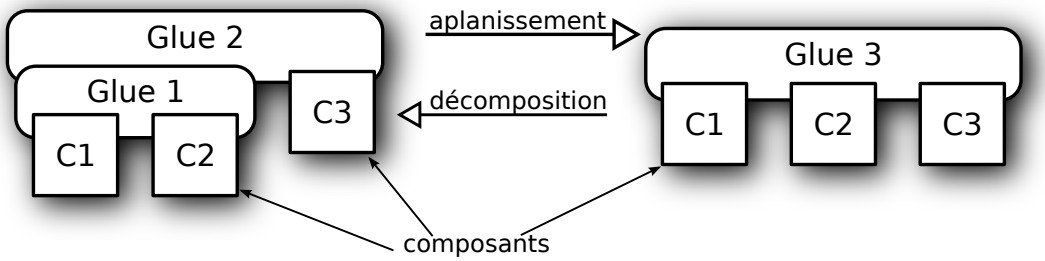


FIGURE 2.7 – Incrémentalité de la composition en BIP.

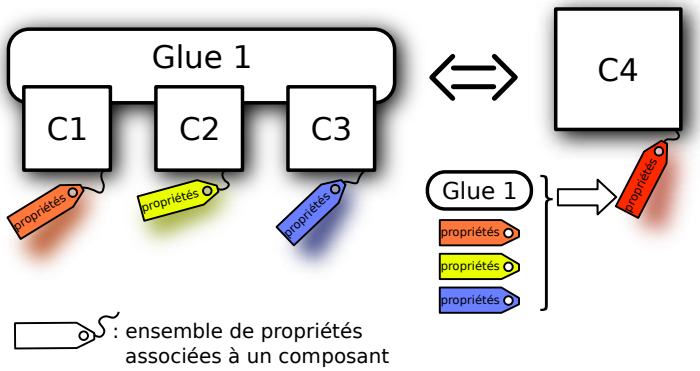


FIGURE 2.8 – Compositionalité de la composition en BIP.

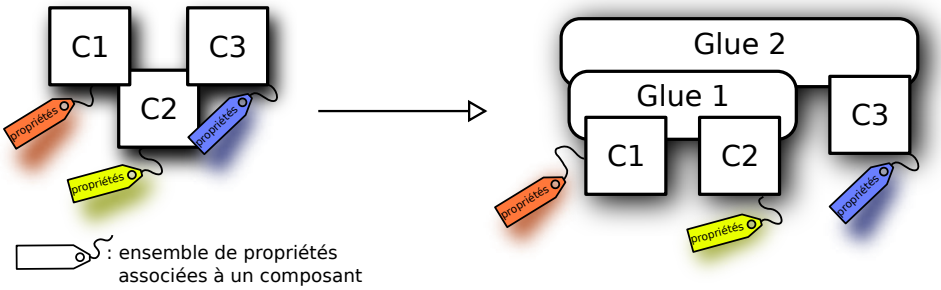


FIGURE 2.9 – Composabilité de la composition en BIP.

2.4.2 Langage BIP

Les trois paragraphes suivants présentent les trois couches qui forment BIP : le comportement, les interactions et les priorités.

Comportement. BIP possède deux catégories de composants. Les composants *atomiques*, ou *atomes*, contiennent du comportement, des *ports* pour interagir avec d'autres composants et des données locales. Les composants *composites* ne peuvent contenir que des sous-composants (atomiques ou composites). Le comportement se présente sous la forme d'un système de transitions d'états (LTS [15] dans la suite) contenu dans les atomes. Chaque transition du LTS est étiquetée par un port, une garde et une fonction. Un composant atomique possède un ou plusieurs ports qui représentent les seuls points d'interaction possibles. La garde est une condition sur

15. LTS pour « Labeled Transistions System »

l'état local du composant. Pour qu'une transition soit exécutable, cette garde doit s'évaluer à « vrai ». La fonction, qui est exécutée lorsque la transition est tirée, permet de modifier l'état local du composant et est exprimée en langage C. Le listing 2.4 est un exemple de définition d'un composant atomique possédant trois ports et une donnée entière. La figure 2.10 donne sa représentation graphique où la garde est préfixée par `g:` et la fonction par `f:`. Si la garde n'est pas spécifiée, alors elle est toujours évaluée à « vrai ». Si aucune fonction n'est à exécuter, elle est omise.

```
component foocomp
 port a,b,c
 data int x
 behavior
  init do x=0; to S1
  state S1
   on a provided x==0 to S2
   on c provided x==1 to S3
  state S2
   on b do x=1; to S1
  state S3
   on b do x=0; to S1
end
```

Listing 2.4 – Représentation textuelle d'un composant atomique BIP.

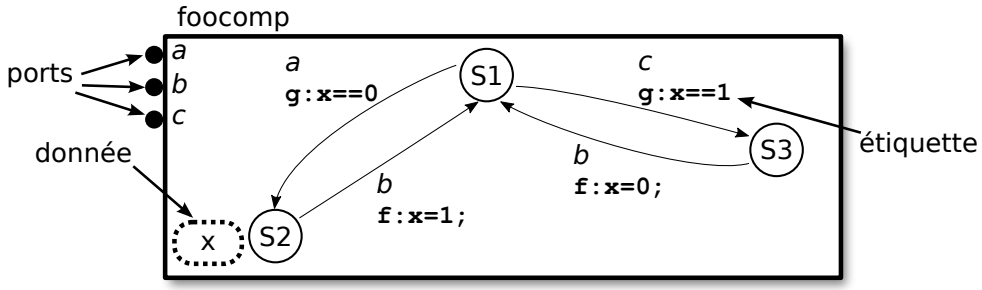


FIGURE 2.10 – Représentation graphique d'un composant atomique BIP.

Interaction. Cette couche permet de contraindre les comportements de plusieurs composants atomiques par la synchronisation de leurs automates (*i.e.* synchronisation des transitions). Elle permet aussi les échanges de données entre les composants. Une *interaction*, qui est un ensemble de ports, est définie par un *connecteur*. Ce dernier relie des ports de différents composants entre eux, et définit un ensemble d'interactions possibles, c'est-à-dire des interactions susceptibles d'être exécutées. Pour être en plus exécutables, les ports d'une interaction doivent être actifs. Un port est dit actif si l'automate sous-jacent se trouve dans un état où l'une des transitions étiquetées par ce port est exécutable, c'est-à-dire que sa garde s'évalue à « vrai ».

Dans la forme la plus simple d'un connecteur, seule l'interaction maximale incluant tous ses ports est possible. Ce type de connecteur est appelé *rendez-vous*. Pour exprimer des interactions plus complexes, BIP fournit deux mécanismes.

Le premier est un système de type pour les ports d'un connecteur. Chaque port peut être soit `synchron`, soit `trigger`. Un synchron est « passif » et ne permet pas d'initier une interaction,

alors qu'un trigger est « actif » et suffisant pour rendre possible une interaction. Ainsi, un connecteur utilisant les types de port définit les interactions suivantes :

- l'interaction maximale définie par l'ensemble de tous les ports du connecteur, quel que soit leurs types ;
- les interactions définies par tous les sous-ensembles de ports du connecteur contenant au moins un port trigger.

Les types synchron et trigger sont aussi appelés respectivement incomplet et complet. La figure 2.11 donne un exemple simple de deux connecteurs : un premier, en bleu, n'utilisant que des ports synchrons (`a`, `b`, `c`), un deuxième, en rouge, possédant aussi un port trigger (`s`, `r2`, `r3`). Le connecteur bleu force les trois composants à passer simultanément de leurs états `S1` à `S2`. Le rouge permet au composant `c1` de passer de `S1` à `S3` et dans le même temps, permet aux composants `c2` et `c3` de passer de leurs états `S2` à `S1`. Les connecteurs similaires (1 port trigger pour plusieurs ports synchrons) sont appelés *diffusion* (« broadcast »).

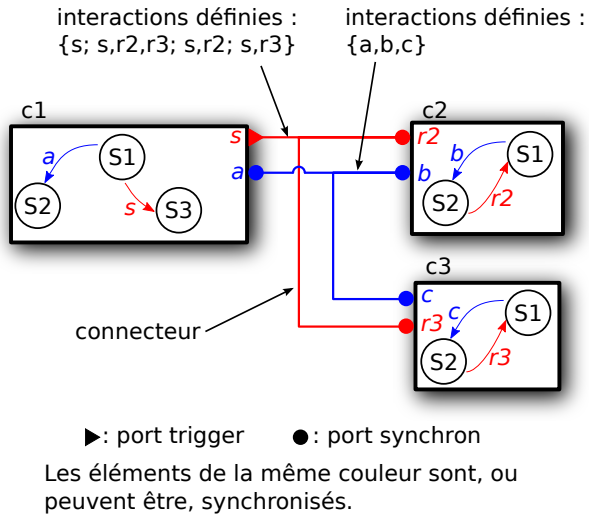


FIGURE 2.11 – Représentation graphique de deux connecteurs BIP.

Le deuxième mécanisme pour définir les ensembles d'interactions permises par un connecteur utilise une expression booléenne sur les ports. Il est ainsi possible de spécifier aussi précisément qu'on le souhaite les ensembles d'interactions autorisés. Il n'existe pas de notation graphique aussi expressive. Il existe néanmoins une notation hiérarchique des connecteurs qui peut convenir dans la plupart des cas, mais celle-ci peut alourdir grandement le dessin. La notation booléenne étant intensivement utilisée dans nos travaux, nous décidons, pour simplifier les représentations graphiques, de ne pas représenter les ensembles d'interactions définies par les connecteurs.

Le listing 2.5 présente un exemple simple de connecteur dont l'ensemble des interactions possibles est donné par une expression booléenne. La figure 2.12 illustre le même connecteur, à la fois sous sa forme hiérarchique, et sous sa forme plus simple et moins expressive telle que nous la rencontrerons dans ce manuscrit. Pour combler ce manque, nous fournirons par la suite soit l'expression booléenne associée, soit une explication dans le texte.

```
connector foo = c1.a, c2.b, c3.c, c4.d
 complete = c1.a||(c1.a && c3.c && c4.d)||
                  (c1.a && c2.b && c3.c && c4.d)))
 behavior
 end
```

Listing 2.5 – Représentation textuelle d'un connecteur BIP

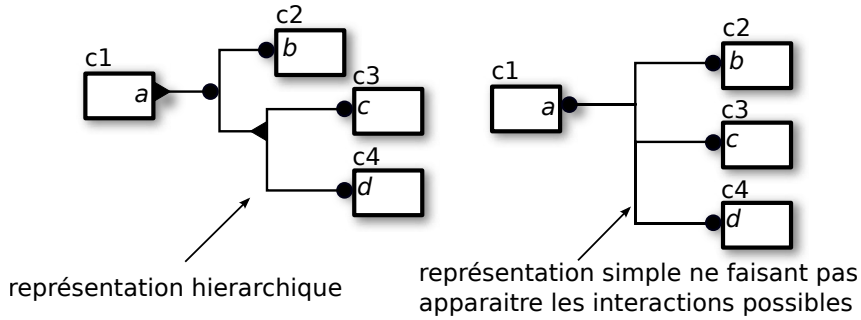


FIGURE 2.12 – Représentations graphiques d'un connecteur BIP sous forme hiérarchique ou plate.

Les connecteurs sont aussi le support des **transferts de données** entre les composants. Il est possible, pour chaque interaction, de spécifier une fonction de transfert qui sera exécutée lorsque l'interaction est exécutée. Enfin, à chaque interaction il est possible d'associer **une garde**. Aussi bien les fonctions de transfert que les gardes ne peuvent agir que sur les données des composants reliés par le connecteur.

La présentation des interactions effectuée dans cette section est suffisante pour nos travaux, mais ne fait qu'effleurer leur puissance. En particulier, [BS08a, BS08b] introduisent une *algèbre de connecteurs*. Pour résumer, les connecteurs permettent d'agrandir l'ensemble des interactions possibles dans le système.

Priorité. Cette couche permet, contrairement aux interactions, de réduire l'ensemble des interactions exécutables. Une règle de priorité se note

$$g \implies l \prec h \tag{2.1}$$

avec `g` une garde et `l` et `h` deux interactions. Lorsque `g` est « vrai » et que `l` et `h` sont exécutables, alors `l` est inhibée (*i.e.* n'est plus exécutable). Les règles de priorité permettent de réduire le non déterminisme dans le système lorsque l'ensemble des interactions exécutables contient plus d'un élément. Il n'existe pas de notation graphique pour les priorités. En guise d'exemple, le listing 2.6 donne la définition textuelle de deux règles.

```
priority
 foo_prio
  if (c1.S1 && c2.x == 10) c2.b, c3.d < c1.a, c4.s
 bar_prio
  c3.go < c2.go
```

Listing 2.6 – Représentation textuelle de règles de priorité BIP

Les priorités peuvent par exemple être utilisées pour implanter des règles d'ordonnancement, pour imposer l'exclusion mutuelle lors de l'accès à une ressource.

2.4.3 Outillage

Présentation générale. Une vue d'ensemble des outils disponibles est donnée dans la figure 2.13. BIP est accompagné principalement d'un compilateur, `bipc` sur la figure, proposant plusieurs sorties. Ce compilateur permet de :

- créer des modèles destinés à être utilisés avec l'environnement de modélisation d'Eclipse EMF [16].
- générer du code C++ pour être utilisé par le moteur d'exécution BIP (« BIP Engine »). Ce dernier permet la simulation ainsi que l'exploration des modèles BIP.
- générer un ensemble de composants THINK, qui peuvent ensuite être compilés vers du code exécutable pour des plate-formes embarquées. Ces travaux sont détaillés dans [PPRS06] et en annexe A.

L'outil `D-Finder` [BBNS09] peut aussi être utilisé pour analyser un modèle BIP et identifier de possibles interblocages. Nous détaillons dans la suite de cette section les possibilités offertes par le moteur d'exécution, qui est l'outil que nous avons principalement utilisé dans nos travaux.

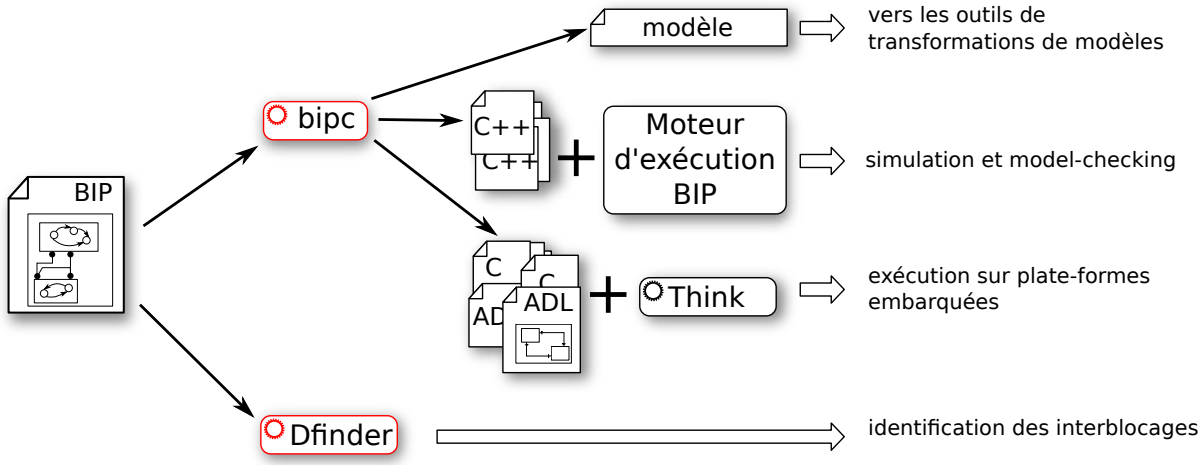


FIGURE 2.13 – Outillage en rapport avec BIP.

Moteur d'exécution BIP. Comme mentionné sur la figure 2.13, ce moteur d'exécution offre des possibilités de simulation et d'exploration. Ce moteur repose sur une boucle d'exécution, présentée sur la figure 2.14.

Au démarrage, les transitions initiales de tous les atomes BIP sont exécutées. Lorsqu'un état stable est atteint (*i.e.* les exécutions sont terminées), l'ensemble de toutes les interactions possibles est construit. Cet ensemble comprend toutes les interactions dont les ports sont actifs et dont les gardes impliquées sont évaluées à « vrai ». Si cet ensemble est vide, alors le système est bloqué. Sinon, l'étape suivante applique les règles de priorité pour filtrer cet ensemble. À l'issue de ces deux étapes, une interaction parmi celles restantes est sélectionnée, puis exécutée.

Le moteur peut itérer cette boucle pour générer une trace d'exécution : c'est de la simulation. Lorsque plusieurs interactions sont exécutables, une seule sera exécutée. C'est pour cette raison que la simulation n'est pas une technique d'analyse exhaustive.

16. EMF, pour « Eclipse Modeling Framework » est disponible sur la page http://www.eclipse.org/modeling/emf/

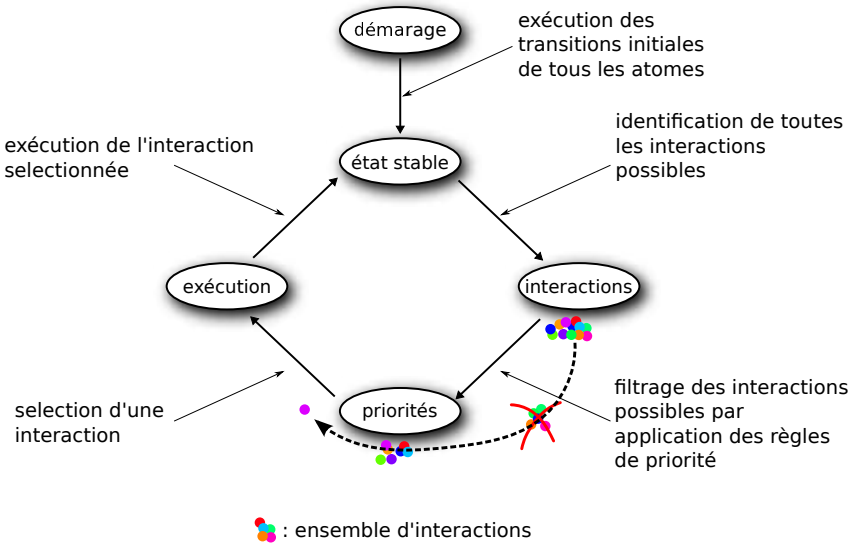


FIGURE 2.14 – Boucle d'exécution du moteur BIP.

Le moteur est aussi capable d'explorer toutes les traces d'exécution. À chaque prise de décision (après l'application des règles de priorités), plutôt que de sélectionner une interaction et d'ignorer les autres, il explore les traces d'exécution correspondant à tous les choix possibles. Cette exploration peut demander des ressources de stockage conséquentes, voire être inapplicable si l'espace à explorer est trop grand. Ce point a déjà été abordé dans l'introduction dans le paragraphe dédié à l'analyse des modèles. Plusieurs outils sont disponibles pour exploiter les résultats de telles explorations. Dans nos travaux, nous avons principalement utilisé des outils pour l'identification des exécutions menant à des blocages, présentés en annexe B.2.

2.4.4 Conclusion

Dans le cadre de nos travaux, BIP offre plusieurs avantages considérables par rapport à d'autres langages de modélisation.

Tout d'abord, malgré un langage relativement simple (syntaxe simple et ensemble de primitives réduit), BIP possède une grande puissance d'expression. En particulier, grâce à ses couches *interaction* et *priorité*, il est possible d'exprimer des systèmes très variés, comme des systèmes orientés flots de données, événementiels, pilotés par le temps (« time triggered »), ...

Les propriétés de composition représentent un atout évident pour une utilisation au sein d'un processus de traduction ou plus généralement de compilation. Ces derniers s'effectuent généralement en plusieurs étapes, chacune enrichissant le résultat de la précédente. BIP permet de garantir que chaque étape n'invalide pas le résultat de l'étape précédente.

Enfin, BIP possède une sémantique formelle qui assure que pour chaque modèle, aucune ambiguïté ou erreur d'interprétation n'est possible. Le modèle se suffit à lui même et ne nécessite aucune information supplémentaire pour son interprétation.

C'est pour ces raisons que nous avons choisi d'utiliser BIP comme cible pour la traduction de notre nouveau langage.

Chapitre 3

Buzz

Talk is cheap. Show me the code.
Linus Torvalds.

Sommaire

3.1 Introduction

Nous avons présenté dans la section 2.2 une méthode pour la construction, l'analyse et l'implantation d'un langage de modélisation. Dans cette méthode le langage de modélisation est traduit vers deux autres langages, l'un disposant de techniques d'analyses, l'autre disposant d'outils d'implantation.

Nous présentons dans ce chapitre un prototype complet démontrant la faisabilité de cette méthode pour un langage de modélisation particulier. Dans les sections 3.2 et 3.3, nous présentons le nouveau langage de modélisation Buzz ainsi que sa syntaxe concrète. Les sections 3.4 et 3.5 détaillent respectivement la traduction de Buzz vers une architecture logicielle à composants Think pour l'implantation et vers un modèle BIP pour les analyses. La section 3.6 complète ces deux présentations en démontrant la relation existante entre les résultats des deux traductions.

3.2 Le langage Buzz

À partir de l'étude menée dans le chapitre 1 ainsi que des besoins récurrents que nous avons constatés dans le domaine des systèmes embarqués monoprocesseur, nous avons isolé un ensemble de primitives et de concepts pour construire le nouveau langage de modélisation Buzz. Cet ensemble est volontairement restreint pour nous permettre la réalisation complète du compilateur associé. Nous souhaitons réaliser une expérience en profondeur, c'est-à-dire couvrant l'intégralité de la démarche, depuis la création du langage jusqu'au déploiement de code exécutable sur des plate-formes embarquées et par la mise en œuvre de techniques d'analyses non triviales. Le langage Buzz n'est pas un langage universel avec un ensemble exhaustif de primitives. Il est un exemple démonstratif et représentatif de la démarche que nous avons présentée dans le chapitre 2.

De plus, une trop grande richesse offerte aux développeurs n'est pas nécessairement bénéfique pour ces derniers. Cette richesse rajoute de la complexité qu'il est nécessaire de maîtriser pour une utilisation correcte des concepts. Les restrictions que nous imposons dans Buzz permettent à la fois de simplifier la mise en œuvre de notre démarche, mais aussi assurent une prise en main plus rapide par les développeurs et évitent les mauvaises utilisations dues à des incompréhensions. Le langage Buzz est caractérisé par les points suivants :

- les composants de Buzz ne structurent pas que les données et le code d'un système (comme dans Think). Ils structurent aussi les activités. Buzz possède des notions de composants actifs et passifs. La notion de composant actif est directement inspirée des modèles à acteurs.
- Buzz permet de choisir pour chaque liaison entre des composants si les invocations de méthodes se font de manière synchrone (*i.e.* appel bloquant) ou asynchrone (*i.e.* par envoi de message) .
- l'ordonnancement des activités n'est pas fixé par le langage mais doit être configuré en fonction des besoins de façon séparée au modèle Buzz.

Buzz distingue les **types (ou définitions) de composants** de leurs instances, de la même manière que les classes sont distinguées des objets dans la programmation orientée objet.

Un type de composant en Buzz possède :

- un nom,
- un ensemble d'interfaces serveurs,
- un ensemble d'interfaces clientes,
- un contenu, qui est le comportement des interfaces serveurs et qui utilise les interfaces clientes,
- un ensemble d'attributs.

Les interfaces d'un composant sont typées. Un type d'interface possède un nom et un ensemble de signatures de méthodes. Un exemple de type de composant est donné dans la figure 3.1. Pour éviter la confusion entre type (ou définition) de composant et instance de composant, nous représentons les types avec des contours en pointillés.

Nous ne détaillerons pas ici la nature du contenu des composants en Buzz et laissons ces précisions pour la section 3.3.

Un système Buzz est **un assemblage d'instances de composant**. Buzz ne permet pas d'assembler les composants de manière hiérarchique. C'est-à-dire que le contenu d'un composant ne peut pas être lui même un assemblage de composants. Nous avons introduit cette limitation afin de simplifier les règles de traduction et la réalisation du compilateur.

Les instances de composants peuvent être liées entre elles par leurs interfaces. Une liaison est point à point et relie une interface cliente à une interface serveur de même type. Les connexions plus complexes, par exemple un client vers plusieurs serveurs, sont volontairement ignorées dans

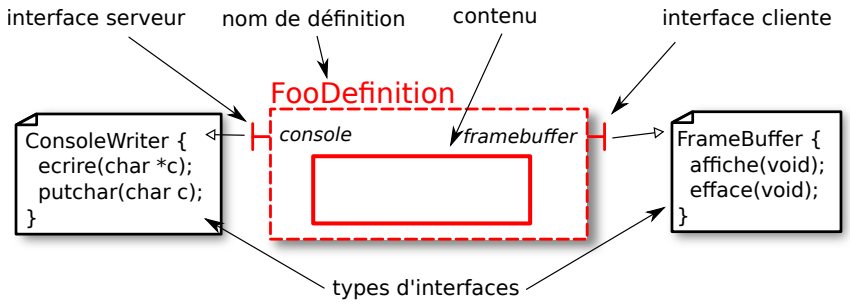


Figure 3.1 – Exemple simple de définition de composant Buzz.

ce langage. Elles auraient grandement complexifié le langage et de son compilateur. Il ne s'agit pas d'une limitation dans l'approche, mais d'une limitation des efforts que nous pouvons investir dans le développement. Rappelons que notre objectif est la réalisation d'un prototype complet de bout en bout.

En plus de ces aspects classiques d'architecture, Buzz associe aux instances et liaisons des propriétés liées à leurs comportements à l'exécution :

- chaque instance de composant peut être *active*, *passive* ou *interrupteur*.
- chaque liaison peut être *asynchrone*, *synchrone* ou *retardée*.
- l'ordonnancement des activités n'est pas imposé par le langage mais doit être spécifié par le développeur de façon séparée à l'architecture.

Nous présentons dans les sections suivantes la signification de ces propriétés.

3.2.1 Activités et composants

Suite à notre étude du chapitre 1 sur les approches à composants existantes, nous avons isolé trois catégories d'instances de composants. Une instance de composant *active* possède une activité propre, de manière similaire à un acteur. Au contraire, une instance *passive* ne possède pas d'activité propre et ne s'exécute que lors d'invocations issues d'autres activités. Enfin, une instance *interrupteur* s'exécute dans un contexte d'interruption matérielle. Cette dernière catégorie est similaire aux composants événementiels de PECOS décrits en section 1.1.4. Ces trois catégories sont détaillées dans les paragraphes qui suivent. Il est important de noter que la catégorie associée à une instance est indépendante de la définition de composant correspondante. Une instance `i` d'une définition `d` peut aussi bien être active, passive ou interrupteur. Évidemment, toutes les combinaisons n'ont pas forcément d'intérêt suivant les composants, nous reviendrons sur ce point plus tard.

Une **instance de composant active** possède sa propre activité. Comme c'est le cas dans le modèle à acteur, ou dans le modèle événementiel, les invocations destinées à ces instances ne s'exécutent pas dans le contexte d'exécution de l'appelant. Les exécutions ont lieu dans le contexte associé à l'instance active désignée par l'invocation. L'utilisation de plusieurs instances actives signifie que ces instances s'exécutent en parallèle. Ce parallélisme implique qu'une instance active peut être invoquée alors même qu'elle s'exécute déjà : cette nouvelle invocation est mise en attente. Une instance active ne traite qu'une seule invocation à la fois (« run-to-completion »). Toujours dans un but de simplification, nous avons imposé que les invocations soient traitées dans leur ordre d'arrivée.

Nous adoptons la convention graphique de la figure 3.2 pour représenter les instances de composants actives. Pour éviter de surcharger les schémas, nous indiquons uniquement le nom

de l'instance et non son type lorsque cela est possible. Dans tous les cas, les noms de types commencent toujours par une majuscule, et les noms d'instances par une minuscule.

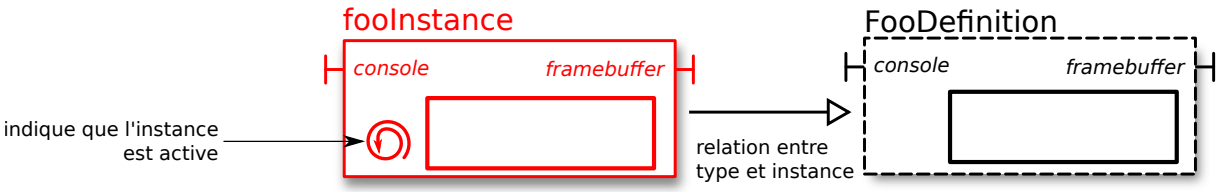


FIGURE 3.2 – Représentation graphique d'une instance de composant active.

Une **instance de composant passive**, ou composant passif, ne possède pas son propre contexte d'exécution. Une instance passive ne peut s'exécuter que dans le contexte d'exécution à l'origine d'une invocation la désignant. Le code appartenant à un composant passif peut donc être exécuté par différents contextes d'exécution. Dans BUZZ, nous avons fait le choix simplificateur de ne pas mettre en place de mécanisme pour interdire la ré-entrance (plusieurs exécutions simultanées du même composant) de ces composants. Les composants passifs peuvent donc être le siège de courses. La figure 3.3 donne la notation graphique d'un composant passif.

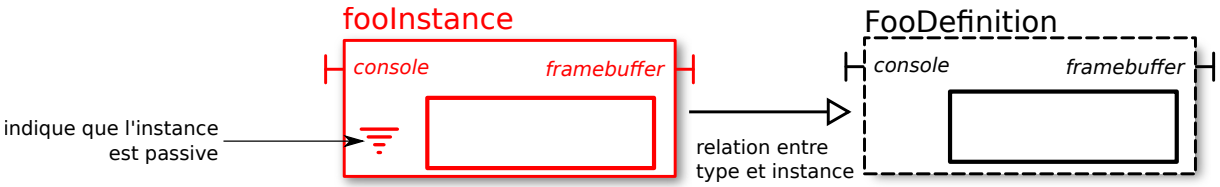


FIGURE 3.3 – Représentation graphique d'une instance de composant passive.

La notion d'**instance de composant interrupteur**, ou composant interrupteur, est fortement liée au contexte d'utilisation embarquée de BUZZ. Un composant interrupteur est le point d'entrée dans le système des événements matériels concrétisés par le mécanisme d'interruptions matérielles. Un composant interrupteur ne possède pas de contexte, comme un composant passif, mais ne nécessite pas non plus d'invocation d'un autre composant pour s'exécuter. C'est le mécanisme d'interruption matérielle qui déclenche l'exécution de ces instances. Ces composants agissent comme des *ponts* entre l'environnement et le système et ne doivent pas être le siège de traitements autres que la simple prise en compte des événements. En effet, l'exécution d'un composant interrupteur a lieu dans un contexte d'interruption, c'est-à-dire que durant cette exécution, aucune autre interruption ne peut être prise en compte. Très souvent, mais cela dépend de la plate-forme matérielle utilisée, cela implique la perte des événements arrivés pendant cette exécution. C'est pour cette raison que les composants interrupteurs ne doivent être utilisés que pour déléguer les traitements à d'autres composants actifs.

La figure 3.4 donne la notation graphique d'un composant interrupteur.

3.2.2 Assemblage de composants

Un système BUZZ est un assemblage d'instances de composants. Ces instances sont reliées entre elles par des liaisons point à point. Ces liaisons représentent l'unique moyen de communication entre les composants. Elles sont utilisées pour la synchronisation, les invocations de méthodes et les transferts de données. Si plusieurs instances actives sont utilisées, il est nécessaire de préciser comment ces activités concurrentes sont ordonnancées. BUZZ fait l'hypothèse

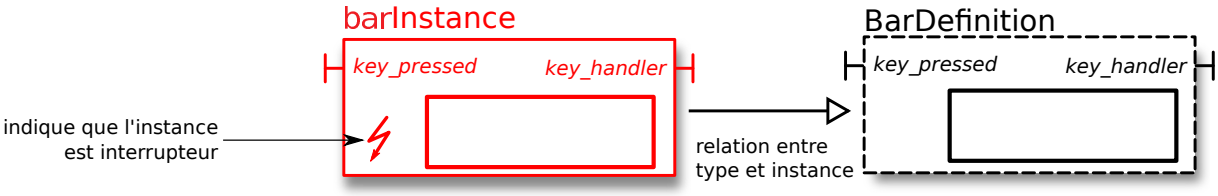


Figure 3.4 – Représentation graphique d'une instance de composant interrupteur.

qu'un seul processeur est disponible, l'ordonnancement est donc responsable de l'entrelacement des différentes activités sur ce processeur.

Nous présentons dans la section 3.2.2 les trois catégories de liaisons qui existent dans Buzz ainsi que la façon dont l'ordonnancement est traité.

Liaisons

Pour faciliter les descriptions qui suivent, nous utilisons le vocabulaire suivant :
- une *invocation* est initiée par un composant *appelant* et destinée à un composant *appelé*.
- une invocation provoque l'exécution du code correspondant à la méthode invoquée, ce que nous appelons la *réaction*.
- le *retour* est le signalement (éventuellement accompagné de données) par l'appelé à l'appelant de la terminaison de la réaction. Toutes les invocations n'ont pas un retour.

Une **liaison synchrone** possède une sémantique similaire à celle des appels de fonctions dans les langages de programmation classiques. L'appelant est bloqué tant que l'appelé n'a pas terminé la réaction correspondante. Une invocation portée par une liaison synchrone possède un retour qui peut contenir des données. La réaction se déroule entre l'invocation et le retour.

Les **liaisons asynchrones** sont majoritairement rencontrées dans les systèmes orientés événements. Ces systèmes possèdent plusieurs composants actifs s'exécutant en parallèle et communiquant par l'échange de messages. Une invocation portée par une liaison asynchrone est non bloquante pour l'appelant et ne possède pas de retour.

Une **liaison retardée** peut être vue comme une liaison asynchrone modifiée. Les invocations sont en effet non bloquantes. Par contre, contrairement au cas asynchrone, les invocations ne sont transmises à l'appelé qu'à la fin de la réaction qui est en cours dans le composant appelant. Une liaison retardée ne porte pas de retour. La réaction se déroule après la fin de la réaction de l'appelant. Ce type de communication peut aussi être obtenu avec une politique d'ordonnancement adéquat. C'est d'ailleurs le comportement souvent observé pour les systèmes dont l'ordonnancement est coopératif et où le processeur est relâché uniquement à la fin de l'exécution d'une méthode.

La figure 3.5 donne la notation graphique des catégories de liaisons. Le diagramme de séquence de la figure 3.6 illustre les enchaînements entre invocation, réaction et retour entre deux composants actifs.

Dans la section 3.2.1, nous avons présenté les différentes catégories d'instances de composants. Nous donnons maintenant les détails sur les différentes combinaisons possibles entre ces catégories d'instance et les catégories de liaison que nous venons d'introduire. En effet, toutes les combinaisons ne sont pas autorisées, parfois pour des raisons techniques, parfois car ces combinaisons n'ont pas de sens ou d'utilité. Le tableau 3.1 résume les combinaisons valides.

La contrainte la plus importante concerne les liaisons asynchrones, qui ne peuvent avoir que des instances actives à leur extrémité serveur. La raison est purement technique. L'appelant

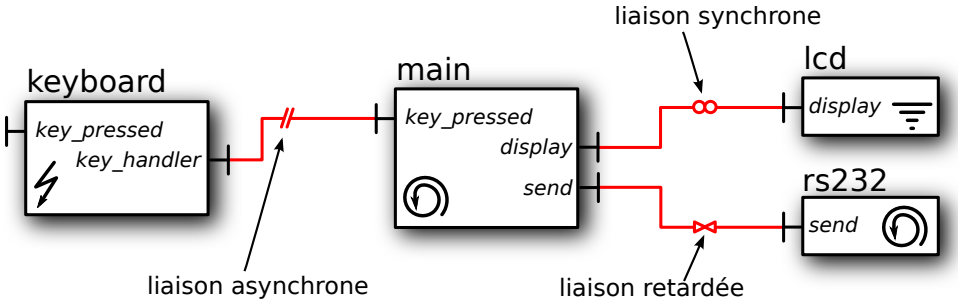


FIGURE 3.5 – Représentation graphique des catégories de liaisons sur un exemple.

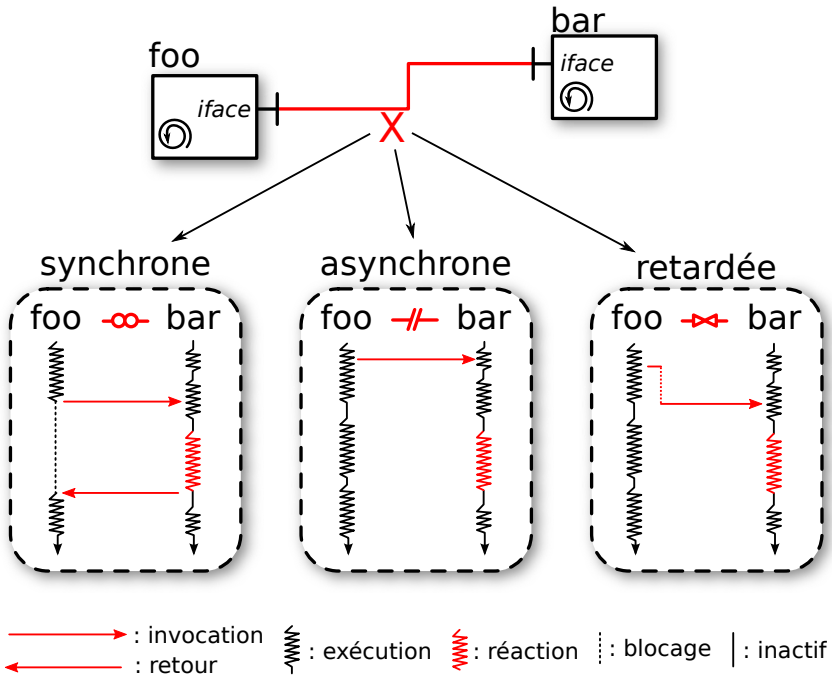


FIGURE 3.6 – Diagramme de séquence pour les différentes catégories de liaison entre deux composants actifs.

n'étant pas bloqué, la réaction ne peut se faire dans son contexte d'exécution. Il existe au moins deux solutions à ce problème :

- imposer que l'appelé dispose d'un contexte d'exécution qui sera utilisé pour la réaction ;
- utiliser un contexte d'exécution anonyme, qui n'est attaché à aucun composant. Par exemple en le créant dynamiquement ou en disposant d'un ensemble de contextes réutilisables (un « pool »).

Dans BUZZ, nous avons écarté la deuxième solution car elle fait intervenir des mécanismes non triviaux. La gestion d'un « pool » ou la création dynamique rendront plus difficile la traduction vers un modèle analysable. D'un point de vue implantation, ce mécanisme existe, par exemple dans polyphonic C# [BCF04]. Dans BUZZ, nous préférons la première solution qui vise à expliciter les activités dans le système. S'il est nécessaire de faire communiquer deux composants de manière asynchrone, alors l'appelé doit être actif.

Une autre restriction concerne les composants interrupteurs, qui ne peuvent pas être utilisés comme client qu'avec des liaisons asynchrones. La raison a été évoquée lors de leurs descriptions.

appelant \ appelé	actif	passif	interrupteur
actif	async.,sync.,retard.	sync.	sync.*
passif	async.,sync.	sync.	sync.*
interrupteur	async.	-	-

async. : asynchrone, sync. : synchrone, retard. : retardée. * : à hauteur d'une seule connexion maximum

Table 3.1 – Combinaisons valides entre composants et liaisons possibles.

Le traitement effectué par un composant interrupteur doit être limité au strict minimum, il n'est donc pas souhaitable d'utiliser une liaison synchrone qui bloquerait l'exécution inutilement. Nous n'avons donc pas traité ce cas. Enfin, pour des raisons de simplification du compilateur (et non de méthode), nous n'autorisons qu'un seul composant à être client d'un composant interrupteur. Cette limitation dans les spécifications du langage n'a pas été remise en cause par les exemples traités dans nos travaux.

Ordonnancement

La politique d'ordonnancement des activités n'est pas spécifiée dans le langage. Celle-ci doit être spécifiée au moment de l'assemblage des composants. Cette idée est semblable à Ptolemy II (description en section 1.1.1) et ses domaines. Buzz fournit un ensemble de politiques d'ordonnancement basiques, mais cet ensemble peut être étendu très simplement, en fonction des besoins. Par exemple, un système ne possédant qu'un seul composant actif sans aucune interaction avec son environnement n'a pas les mêmes besoins qu'un système avec plusieurs composants actifs et plusieurs composants interrupteurs. Buzz fournit un ordonnancement coopératif en tourniquet (« round-robin ») ainsi qu'une variante préemptive avec partage de temps.

Pour permettre la réalisation de politiques d'ordonnancement, Buzz définit plusieurs points durant l'exécution d'un système que peuvent utiliser ces politiques. Ces points, classiquement appelés « hooks », correspondent à des points facilement identifiables, dont les principaux utilisés dans Buzz sont :

- invocation de méthode ;
- fin de traitement d'une interruption ;
- fin d'exécution d'une méthode ;
- interruption périodique (« tick »).

Cette liste n'est pas exhaustive mais représente les points utilisés dans notre prototype. Lorsque l'exécution atteint un de ces points, la politique d'ordonnancement peut effectuer un traitement. Ce traitement peut consister en la mise à jour de données (par exemple, un compteur du temps d'exécution consommé par le contexte actuel) ou au remplacement du contexte qui s'exécute (« context switch »).

Enfin, Buzz permet de contrôler simplement le démarrage de l'exécution d'un système en offrant au développeur la possibilité de spécifier des méthodes qui seront invoquées sur des composants actifs (au plus une par composant).

La figure 3.7 introduit la convention graphique du choix de l'ordonnancement ainsi que des invocations effectuées au démarrage.

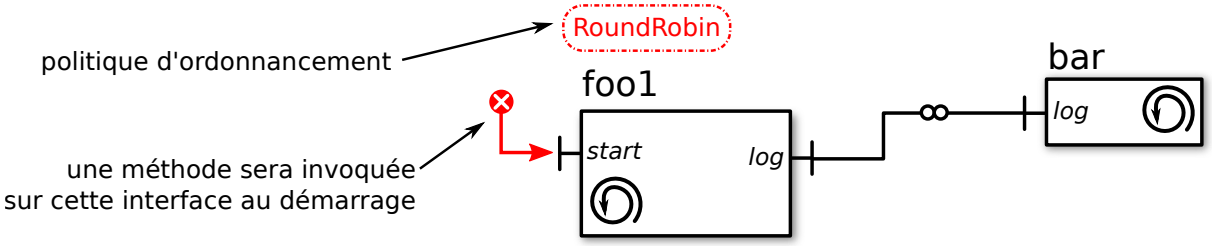


FIGURE 3.7 – Choix de l'ordonnancement et démarrage.

3.2.3 Méta-modèle du langage Buzz

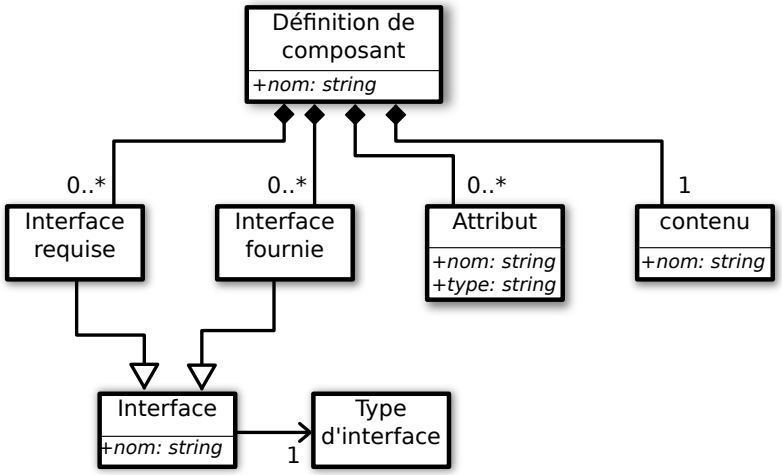


FIGURE 3.8 – Méta-modèle d'une définition de composant BUZZ.

Le langage BUZZ que nous avons présenté dans les sections précédentes peut être en partie résumé par les figures 3.8 et 3.9 qui illustrent respectivement les méta-modèles associés à une définition de composant et à un assemblage de composants. Ces deux schémas font bien ressortir la simplicité du langage.

3.3 Réalisation du compilateur Buzz

Nous avons présenté dans la section précédente le langage BUZZ d'un point de vue abstrait. Nous présentons dans cette section comment nous avons réalisé concrètement un compilateur pour ce langage. L'expression d'un modèle en BUZZ se compose :

i. d'une description de son architecture. C'est-à-dire une description des types de composants ainsi que leur assemblage.

ii. d'une description des types d'interface.

iii. d'une description du contenu des composants.

Nous avons vu en section 2.3.3 que le compilateur de THINK était facilement extensible. En particulier, il est aisé d'enrichir les langages qu'il prend en entrée.

Nous présentons dans la suite comment les langages ADL et IDL de THINK sont utilisés pour exprimer les points i et ii.

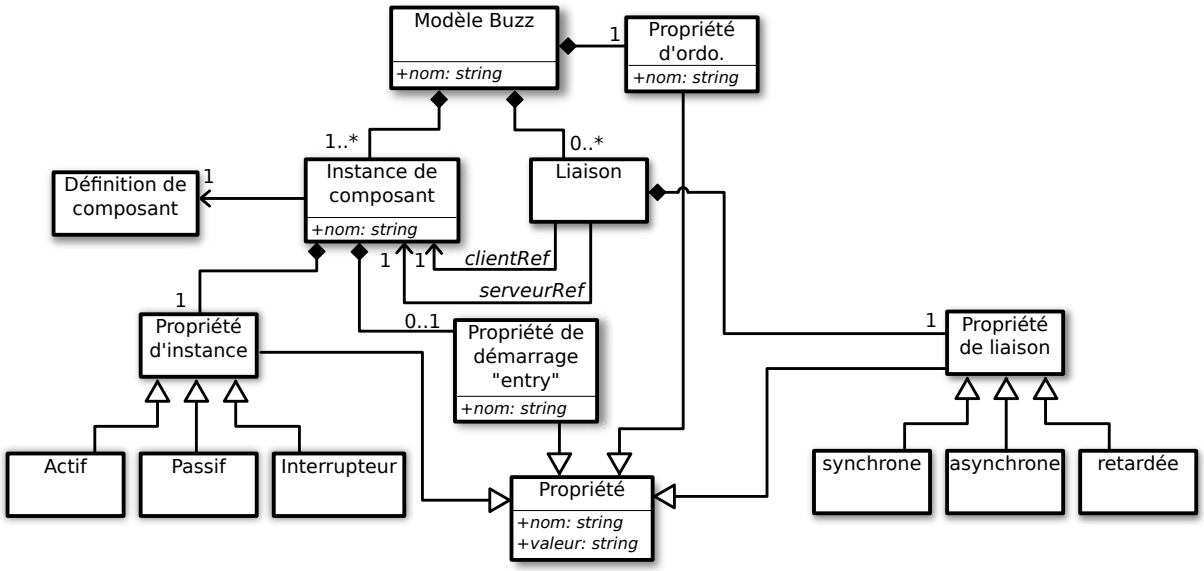


Figure 3.9 – Méta-modèle de l'assemblage de composants Buzz.

Définition de composant. Les notions de types de composant dans Think et Buzz sont quasi identiques, à la seule différence que Buzz ne permet pas de définir des composants composites. Les langages ADL et IDL de Think (voir leur présentation dans la section 2.3.1) sont directement réutilisés dans Buzz. Les listings 3.1 et 3.2 donnent respectivement les codes ADL et IDL correspondant à l'exemple de définition de composant donné dans la figure 3.1 en début de chapitre.

```
component FooDefinition {
 provides a.package.name.ConsoleWriter as console
 requires another.package.name.FrameBuffer as framebuffer
 content contenu_de_foo
}
```

Listing 3.1 – Exemple d'ADL pour une définition de composant.

```
package a.package.name;
public interface ConsoleWriter {
 void ecrire(char *c);
 void putchar(char c);
}
```

```
package another.package.name;
public interface FrameBuffer {
 void affiche(void);
 void efface(void);
}
```

Listing 3.2 – Exemple d'IDL pour deux définitions de types d'interfaces.

Le contenu, spécifié par le mot clé `content` dans le listing 3.1 associe le nom `contenu_de_foo` au contenu du composant. Ce contenu représente le comportement du composant. Dans Think, ce comportement est directement décrit en langage C. Concrètement, les deux traductions, et cela sera détaillé dans les sections 3.4 et 3.5, reposent sur la disponibilité du comportement en langage C ainsi que sous la forme d'un modèle BIP, sans faire aucune hypothèse sur la manière d'obtenir ces deux représentations. Plusieurs solutions ont été envisagées :

i. spécification du comportement en BIP et traduction automatique en C ;

ii. spécification du comportement en C et traduction automatique en BIP ;

iii. spécification du comportement dans un troisième langage et traduction automatique en C et en BIP ;

iv. pas de traduction automatique, spécification du comportement à la fois en C et en BIP ;

Les trois premières solutions ont été écartées, car il n'existe à ce jour aucun outil réalisant ces tâches d'une manière compatible avec nos besoins. L'extraction d'un modèle BIP à partir d'un code C arbitraire est un travail conséquent en lui même, qui implique souvent des restrictions sur le code C supporté, ce qui n'est pas compatible avec la réutilisation de code existant (qui ne respecterait pas ces restrictions). Il n'existe pas à l'heure actuelle de générateur de code C à partir d'un modèle BIP. Enfin, nous ne connaissons aucun langage qui pourrait être traduit à la fois en C et en BIP tout en couvrant tous nos besoins. La réalisation d'une de ces traductions est une tâche non triviale, qui sort du cadre de nos travaux. Nous faisons donc le choix de fournir le comportement à la fois sous la forme d'un modèle BIP et sous la forme d'un code écrit en C. Le contenu d'un composant Buzz est donc un couple composé d'un code en langage C et d'un modèle BIP. Nous faisons l'hypothèse dans nos travaux que ce modèle BIP est fidèle au code C correspondant, malgré le risque d'introduction de différences. Nous ne fournissons pas de méthode ou d'outil pour vérifier ou aider cette tâche. Nous nous intéressons à la composition des composants et non à leur création. Obtenir un modèle BIP fidèle à un code C est un travail complémentaire qui sort de notre étude. De plus, comme nous le verrons en section 3.5.2, le modèle BIP correspondant à un code C n'est pas unique et va dépendre des propriétés étudiées.

L'**assemblage d'instances de composants** Buzz repose aussi sur le langage ADL de Think. Cet assemblage se présente comme une définition de composant composite, c'est-à-dire un composant contenant un ensemble de sous-composants et leurs liaisons. Pour les spécificités de Buzz qui n'existent pas dans Think, nous utilisons le mécanisme de propriété de l'ADL, présenté en section 2.3.4.

Le listing 3.3 donne un exemple complet d'ADL, correspondant à la figure 3.10. Cet exemple décrit l'assemblage de deux composants actifs `foo1`, `foo2` et d'un composant passif `bar`.

```
1 component assemblage [scheduler=RoundRobin] {
2  contains foo1 = FooDefinition [active, entry=main.start]
3  contains foo2 = FooDefinition [active]
4  contains bar  = BarDefinition [passive]
5
6  binds foo1.send_work to foo2.rcv_work [asynchronous]
7  binds foo1.log to bar.log [synchronous]
8  binds foo2.log to bar.log [synchronous]
9 }
```

Listing 3.3 – ADL d'un assemblage de composants en Buzz

Nous détaillons dans les paragraphes suivants les propriétés que nous utilisons dans l'ADL pour exprimer les aspects spécifiques à Buzz.

Buzz définit quatre propriétés applicables à la **déclaration d'instance de composants** dans l'ADL (mot clé `contains`, voir les lignes 2 à 4 du listing 3.3). Les trois premières sont `active`, `passive` et `interrupter`, pour déclarer respectivement que l'instance est active, passive ou interrupteur. Chaque instance possède une de ces trois propriétés. Néanmoins, si rien n'est précisé dans l'ADL, l'instance se voit attribuer la propriété `active` par défaut. La quatrième propriété, `entry`, permet de déclarer une méthode qui sera invoquée au démarrage du système.

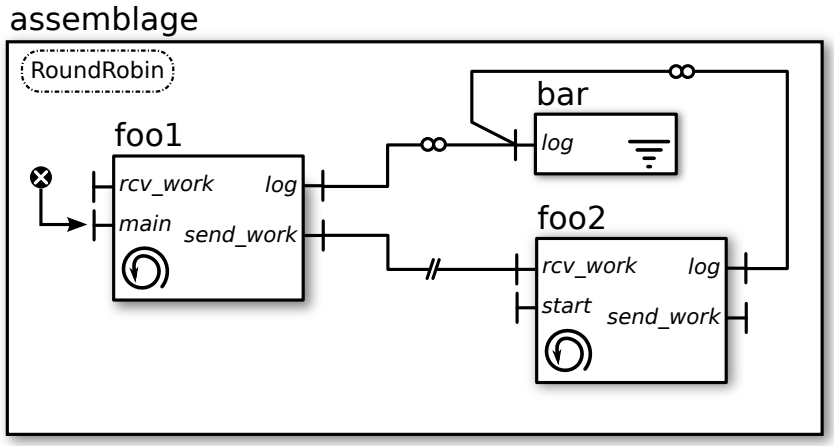


Figure 3.10 – Représentation graphique d'un assemblage de composants Buzz.

La syntaxe utilisée pour désigner une méthode est `interface.methode` (ligne 2 du listing 3.3). Par simplicité, seules les méthodes sans paramètre ni valeur de retour peuvent être utilisées au démarrage.

Buzz définit trois propriétés pour les **déclarations de liaisons** : `asynchronous`, `synchronous` et `delayed`, pour déclarer respectivement qu'une liaison est asynchrone, synchrone ou retardée. Les lignes 6 à 8 du listing 3.3 sont concernées.

Enfin, la dernière propriété, `scheduler`, concerne le choix de la politique d'ordonnancement et s'applique sur la **déclaration de composant composite** qui sert à l'assemblage des composants Buzz. Elle est illustrée à la première ligne du listing 3.3.

3.4 Traduction pour l'implantation d'un modèle Buzz

Nous présentons dans cette section la réalisation de la traduction menant vers une implantation d'un modèle Buzz. La section 3.4.1 présente d'un point de vue technique les principes de réalisation de cette traduction alors que la section 3.4.2 détaille les différentes règles qui la composent. Enfin, la section 3.4.3 présente les optimisations possibles grâce à Think et qui permettent d'atteindre efficacement nos objectifs d'implantation sur plate-forme embarquée.

3.4.1 Principes techniques de réalisation

La réalisation de cette traduction s'appuie doublement sur l'outillage de Think. Tout d'abord, pour ses aspects liés à la programmation de systèmes embarqués, qui en font une cible adaptée à nos besoins, comme détaillée en section 2.3.2. Ensuite, parce que les outils sont extensibles et nous permettent de nous concentrer sur la réalisation de notre traduction, sans avoir à gérer tous les problèmes de la création d'un nouveau compilateur. Ce point a été présenté en section 2.3.3.

La section précédente a présenté comment nous avons enrichi l'ADL de Think pour y intégrer les primitives spécifiques à Buzz. Cette section détaille les règles de traduction de cet ADL enrichi vers une nouvelle architecture de composants pure Think, c'est-à-dire n'utilisant que des primitives présentes dans Think. Nous avons pour cela utilisé un greffon pour le compilateur Think qui se charge de traduire les propriétés spécifiques à Buzz en terme de composants et de liaisons connues par Think. La figure 3.11 illustre le procédé de manière générale.

Nous présentons dans la suite les règles de traduction associées à chaque élément de Buzz.

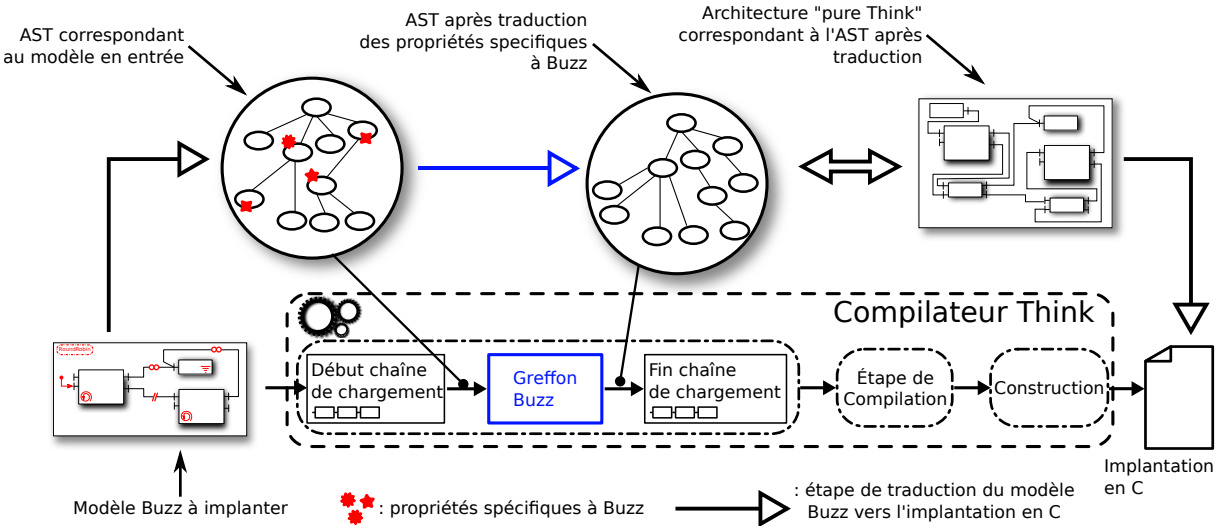


FIGURE 3.11 – Principe de traduction de BUZZ vers THINK.

3.4.2 Règles de traduction

Définitions et contenu de composants

Lors de la description du langage BUZZ en section 3.3, nous n'avons pas défini la nature du contenu des composants mais avons mentionné que chaque définition de composant possède deux parties dans son contenu, une pour chaque voie de traduction. Pour la traduction que nous présentons ici, le contenu est donné en langage C. Ce contenu est strictement identique au contenu utilisé dans les composants THINK et présenté en section 2.3.1. Nous ne revenons donc pas sur ce point.

Instances de composant

Les notions d'**instances de composant passif** BUZZ et instances de composants THINK sont identiques : pas de contexte d'exécution, ni de protection de la ré-entrance. Le greffon n'effectue aucune transformation, ces instances ne nécessitent aucun traitement pour être gérée par THINK.

Les **instances de composant interrupteurs** ne demandent elles non plus aucun traitement par le greffon. La spécificité de ces composants est qu'ils sont exécutés dans le contexte d'interruption. Lors d'une interruption, le matériel bascule du contexte en cours d'utilisation vers le contexte d'interruption et exécute un code prédéfini par le développeur. En THINK, la relation entre interruption et code à exécuter se fait directement dans le code C des composants, où le code correspondant à une méthode est enregistré comme le traitant d'une interruption précise. Dans BUZZ, le contenu des composants est identique à celui des composants THINK, cela comprend aussi la relation avec les interruptions.

Les **instances de composants actives** demandent par contre un traitement conséquent. Ces instances ne correspondent pas aux instances de composants en THINK, il faut donc faire un travail d'adaptation. Elles différent des instances de composants THINK par :

i. un contexte d'exécution qui leur est propre ;

ii. le traitement des invocations reçues les unes après les autres.

Nous créons pour chaque instance active un nouveau composant, chargé de son pilotage. Ce nouveau composant Think, que nous appelons *intercepteur actif*, noté AI, encapsule les données suivantes :

i. un espace mémoire pour héberger le contexte d'exécution (*i.e.* une pile, ou « stack ») associé à l'instance active ;

ii. des files d'attente pour la sérialisation des invocations sur les interfaces serveurs ;

iii. des files d'attente pour la sérialisation des invocations sur les interfaces clientes dans le cas de liaisons retardées.

La création de ce composant implique des changements architecturaux ainsi que de la génération de code pour son contenu. La figure 3.12 illustre les modifications d'architecture effectuées par le greffon. Toutes les liaisons connectées aux interfaces serveurs d'une instance active sont reroutées vers les interfaces serveurs du composant intercepteur correspondant. La même transformation est effectuée pour les liaisons connectées aux interfaces clientes. Pour chaque interface de l'instance active, deux interfaces, une serveur et une cliente, sont ajoutées à l'AI. Le contenu

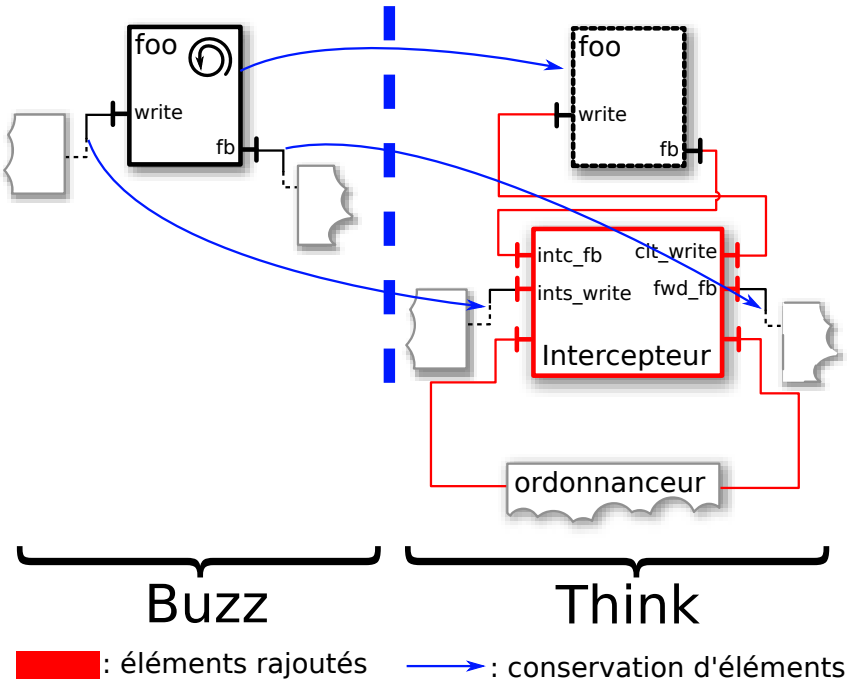


Figure 3.12 – Compilation d'un composant actif.

des AIs, c'est-à-dire du code C, est synthétisé. Ce code peut être divisé en trois parties : envoi/réception des invocations, ordonnancement du traitement des invocations reçues, interaction avec l'ordonnanceur du système. La partie nécessaire à la sérialisation et la mise en attente des invocations de méthode qui entrent et sortent de l'instance active représente la majorité du code. L'ordonnancement dans des invocations suit leur ordre d'arrivée, ce dont se charge une partie du code synthétisé. Pour modifier cet ordre de traitement, par exemple pour tenir compte de priorités, il suffit de modifier la partie du greffon chargée de la synthèse de ce code. Nous verrons dans les paragraphes traitant des liaisons que ce code est aussi dépendant de la catégorie de liaison impliquée. Enfin, une dernière partie de ce code synthétisé sert à interagir avec un nouveau composant chargé de l'ordonnancement des activités dans le système. Cette partie est détaillée dans le paragraphe dédié à l'ordonnancement.

Liaisons asynchrones

La figure 3.13 illustre le fonctionnement du code généré dans l'AI du composant `c2` dans le cadre d'un appel asynchrone du composant `c1` vers `c2`. Comme nous l'avons vu précédemment, les liaisons d'une instance active sont reroutées vers l'AI, c'est ce dernier qui se charge de l'implantation de la sémantique asynchrone.

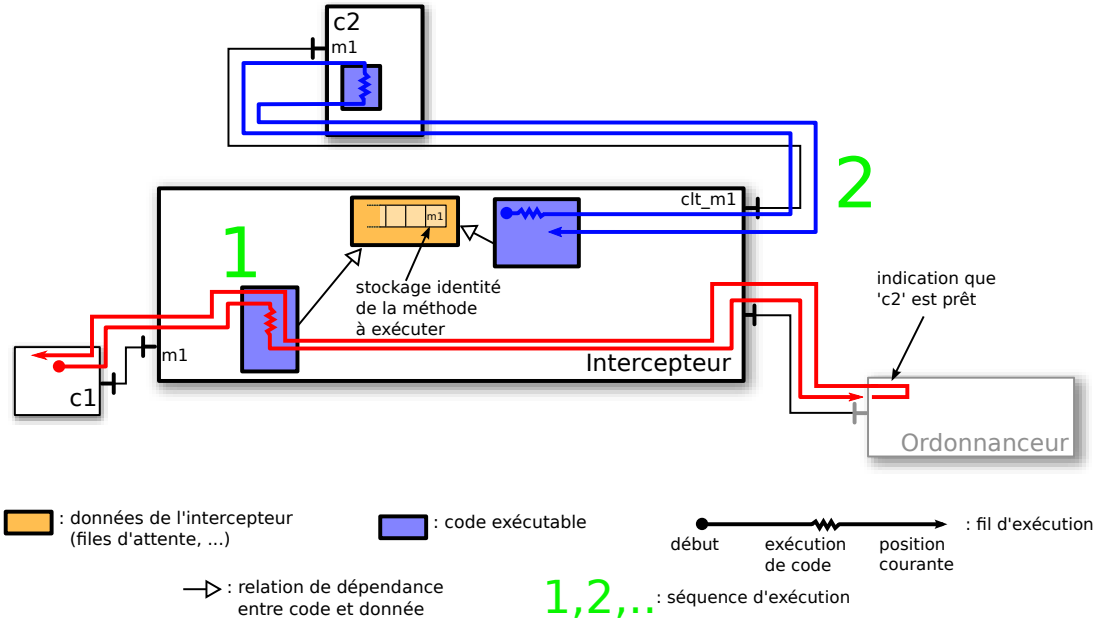


FIGURE 3.13 – Exemple de liaison asynchrone entre deux composants actifs.

Dans un premier temps (noté 1 sur la figure), l'invocation est reçue par l'AI, qui s'exécute alors dans le contexte d'exécution de l'appelant. Cette invocation est sérialisée et stockée dans une file d'attente. Ensuite, l'ordonnanceur du système est notifié que le composant `c2` possède au moins une invocation en attente. Le contrôle est ensuite rendu au composant `c1` qui peut continuer son exécution. Dans un deuxième temps (noté 2 sur la figure), lorsque l'ordonnanceur provoquera l'exécution de l'AI en basculant vers son contexte, l'invocation sera retirée de la file d'attente, désérialisée et transmise à `c2`.

Cette description fait apparaître un risque de course à l'intérieur de l'AI, qui peut être exécuté par plusieurs contextes d'exécution simultanément. Ce risque est illustré sur la figure 3.14. Pour se protéger, l'entrée dans un AI provoque automatiquement la désactivation des interruptions. Cela assure que la file d'attente des invocations ne sera pas manipulée simultanément depuis plusieurs contextes d'exécution.

Liaisons retardées

Contrairement aux liaisons asynchrones qui sont uniquement traitées par l'AI de l'appelé, les liaisons retardées sont aussi traitées par l'AI de l'appelant. Les invocations portées par ces liaisons ne doivent être transmises effectivement à leur destinataire qu'à la fin de l'exécution de la méthode en cours d'exécution chez l'appelant. La figure 3.15 illustre le déroulement du traitement d'une invocation du composant `c1` sur le composant `c2`.

Dans cet exemple, une méthode `m1` de l'interface `i1` de `c1` est en cours d'exécution et invoque une méthode sur l'interface `i2` (noté 1 sur la figure 3.15). Cette invocation est interceptée par

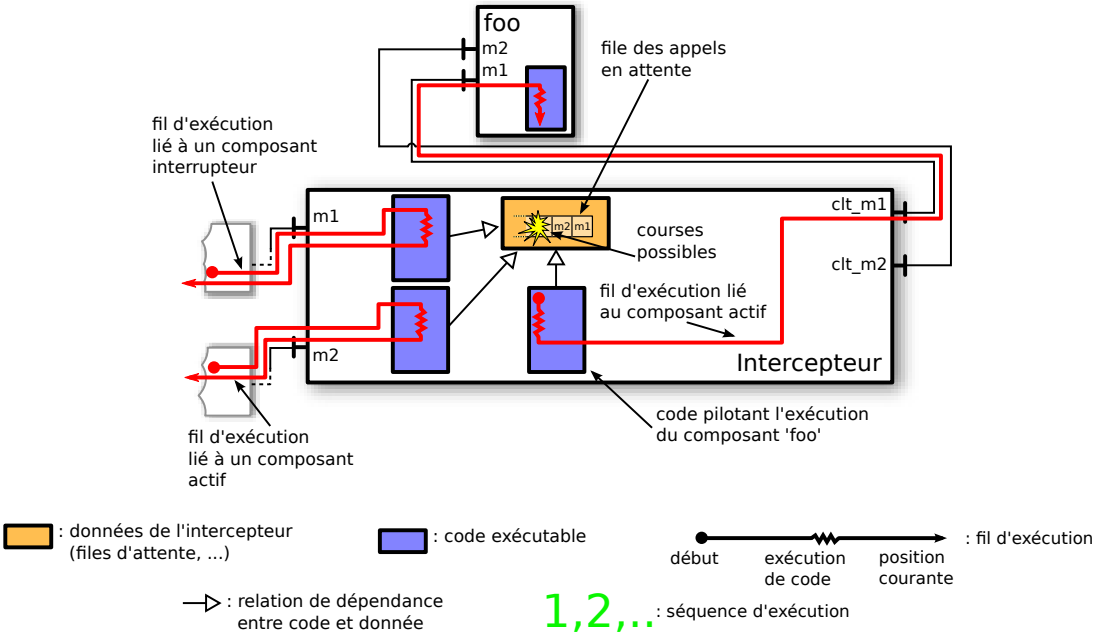


FIGURE 3.14 – Exemple de possibilité de courses lors d'invocations d'un composant actif.

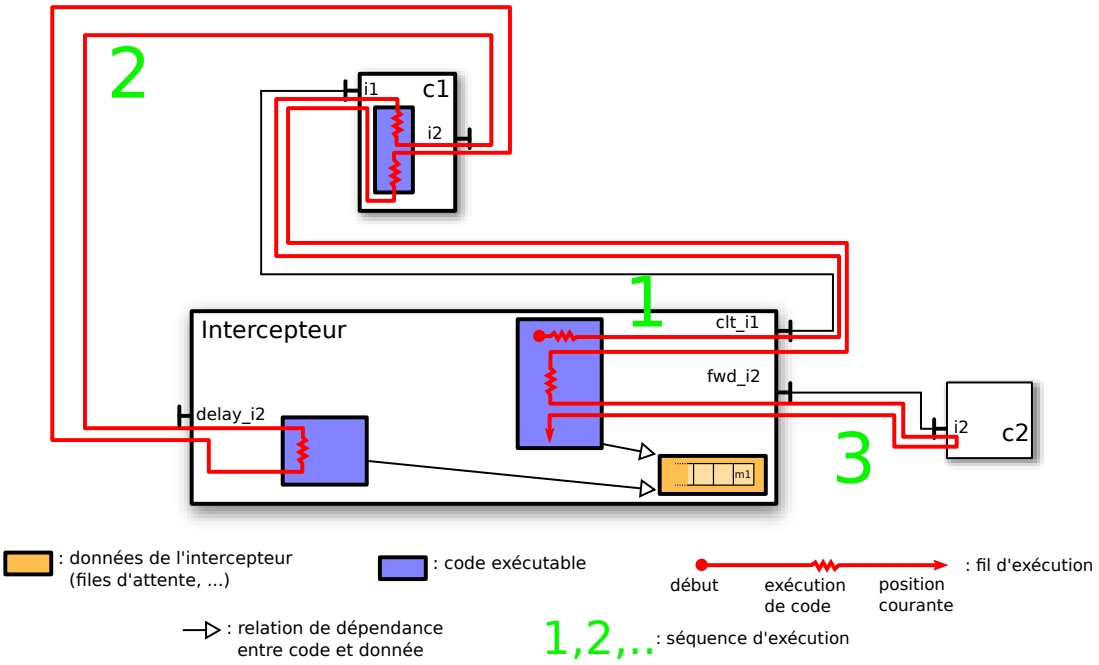


FIGURE 3.15 – Exemple de liaison retardée entre deux composants actifs.

l'AI qui la sérialise et la stocke. Immédiatement, l'AI rend le contrôle au composant `c2` pour qu'il continue son exécution (noté 2 sur la figure 3.15). Lorsqu'il termine l'exécution de la méthode `m1`, le contrôle revient à l'AI, qui purge la file d'attente en transmettant effectivement les invocations aux destinataires (dans notre exemple, une invocation pour le composant `c2`, notée 3 sur la figure).

Liaisons synchrones

La gestion des liaisons synchrones est la plus complexe. Nous découpons sa présentation en fonction de la catégorie du composant appelé.

Une invocation destinée à un **composant passif** ou **interrupteur** est le cas le plus simple, car aucun traitement spécifique n'est nécessaire. La méthode invoquée sur le composant passif est exécutée dans le contexte de l'appelant, de manière similaire à un appel de fonction bloquant en langage C. Le compilateur s'assure, comme nous l'avons indiqué en section 3.2.2, qu'un seul composant est susceptible d'invoquer des méthodes sur chaque composant interrupteur.

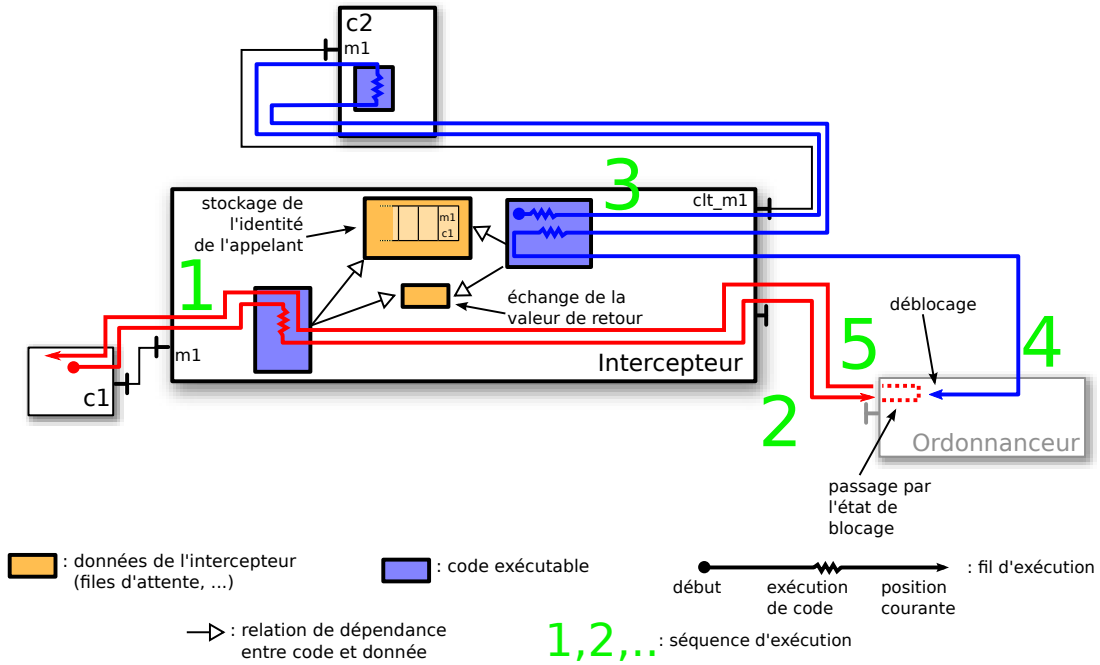


FIGURE 3.16 – Exemple de liaison synchrone entre deux composants actifs.

La figure 3.16 présente le déroulement d'une invocation de méthode synchrone destinée à un **composant actif** `c2` depuis `c1`. De manière similaire au cas asynchrone, l'invocation est interceptée par l'AI, sérialisée et stockée dans une file d'attente (noté 1 sur la figure). En plus, l'identité du composant appelant est stockée. Ensuite, l'AI signale à l'ordonnanceur que `c2` possède au moins une invocation en attente, mais, contrairement au cas asynchrone, demande le blocage de l'exécution du contexte de l'appelant (noté 2 sur la figure). Plus tard, lorsque l'ordonnanceur provoque l'exécution de l'AI, celui-ci retire l'invocation en attente de la file d'attente et la transmet au composant `c2` (noté 3 sur la figure). Lorsque `c2` termine cette exécution et rend le contrôle à l'AI, celui-ci signale à l'ordonnanceur que `c1` (dont l'identité avait été sauvée lors de l'invocation) peut être débloqué, et stocke la valeur renvoyée par `c2` (noté 4 sur la figure). Enfin, lorsque l'ordonnanceur relance l'exécution du contexte précédemment bloqué, l'AI rend le contrôle à l'appelant en renvoyant la valeur de retour stockée (noté 5 sur la figure).

Ordonnancement

L'ordonnancement des activités dans le système est assuré par un nouveau composant, que nous appelons ordonnanceur, qui est créé par le greffon. Sa tâche principale est de pouvoir à

tout moment décider quel contexte d'exécution a le droit de s'exécuter. Cette décision se base sur des critères propres à chaque politique d'ordonnancement, comme nous l'avons présentée en section 3.2.2. Son activité peut être résumée en ces quelques étapes :

i. initialisation des structures de données liées à l'ordonnancement

ii. faire à l'infini :

   (a) attendre un *événement d'ordonnancement* ;

   (b) mettre à jour les structures de données liées à l'ordonnancement ;

   (c) élire un nouveau contexte d'exécution ;

   (d) basculer l'exécution vers le contexte élu.

Nous appelons *événement d'ordonnancement* tous les événements qui provoquent l'exécution de l'ordonnanceur. Cela comprend l'arrivée d'interruptions ou d'invocations directes depuis les AIs. À chaque réception d'un tel événement, l'ordonnanceur met à jour des structures de données. Ces structures contiennent, pour chacun des contextes d'exécution, l'ensemble des données utilisées par l'ordonnanceur lors de sa prise de décision. Comme précisé en section 3.2.2, Buzz ne définit aucune politique d'ordonnancement, et la création d'un système requiert soit d'utiliser une politique prédéfinie dans une bibliothèque, soit d'en définir une sur mesure. Nous fournissons à titre d'exemple des politiques simples, dont les implantations sont détaillées ici. Le composant ordonnanceur est créé par le greffon et est en partie prédéfini et en partie synthétisé.

Nous commençons la présentation par la partie prédéfinie, c'est-à-dire fournie dans une bibliothèque de politique d'ordonnancement.

Pour réaliser les politiques de tourniquets simples fournies dans nos travaux, nous utilisons la structure de données présentée dans le listing 3.4.

```
struct ac_status {
 int activeInterceptorNumber;
 char status;
}
```

Listing 3.4 – Structure de données

Une telle structure est créée pour chaque AI. Le premier champ, `activeInterceptorNumber`, est un identifiant unique d'AI. Le second, `status`, reflète l'état d'exécution du contexte associé à l'AI. Ce champ peut prendre les valeurs suivantes :

- `IDLE` : le contexte est inactif. L'AI correspondant n'a aucune invocation en attente de traitement.
- `READY` : le contexte attend d'être élu pour s'exécuter.
- `BLOCKED` : le contexte est bloqué et attend d'être débloqué avant de pouvoir s'exécuter de nouveau.
- `EXECUTING` : le contexte est actuellement l'élu qui s'exécute.

En plus de cette structure de données, l'algorithme d'élection doit être fourni sous la forme d'un code C, dans une fonction appelée `schedule`. Le listing 3.5 donne la fonction `schedule` dans le cadre de notre tourniquet.

```
list_t *acs;
kt_stack_pointer_t* schedule(void) {
 int idx_toresume, ac_toresume, i;
 kt_stack_pointer_t *ctx_toresume;
```

```
  int j=0;
6
  idx_toresume = find_first_ready();
8
  if (idx_toresume != -1){
10  reorder_queue();
    ac_toresume = get_ac_identifier(idx_toresume);
12  ctx_toresume = get_context(ac_toresume);
    set_executing_status(ac_toresume);
14  } else {
      ctx_toresume = &PRIVATE.contextSched;
16  }
  return ctx_toresume;
18 }
```

Listing 3.5 – Code de la fonction `schedule` pour un tourniquet.

La première ligne déclare une liste, qui sert à représenter notre tourniquet. La ligne 7 récupère l'identifiant du premier AI prêt à s'exécuter (dans l'état `READY`). S'il en existe un, alors la liste `acs` est mise à jour et le contexte élu est exécuté. Sinon, un contexte *qui ne fait rien* est exécuté. Ce contexte dépend de la plate-forme matérielle utilisée. Il peut s'agir d'une simple boucle infinie qui laisse le système sensible aux interruptions, ou il peut s'agir d'un code plus compliqué qui bascule le matériel dans un état de faible consommation en ne laissant que le mécanisme d'interruption activé.

L'ordonnanceur doit fournir un ensemble de fonctions en plus de la fonction `schedule`. Tout d'abord, un ensemble de fonctions est utilisé pour les interactions directes entre les AIs et l'ordonnanceur :

- `yield()` signale que le composant appelant désire rendre la main. Par exemple, dans le cas d'un tourniquet, le contexte appelant est remis en début de liste.
- `idle(id)` signale que l'AI dont l'identifiant est `id` n'a plus de traitement en cours ou en attente.
- `ready(id)` signale que l'AI dont l'identifiant est `id` est prêt à être exécuté, par exemple lors de l'arrivée d'un nouveau traitement ou à la fin d'une période de blocage.
- `block()` signale que le contexte appelant doit être bloqué.

L'ordonnancement peut être préemptif, c'est-à-dire que l'ordonnanceur peut préempter la ressource d'exécution au contexte en cours d'exécution pour élire un nouveau contexte. Cette préemption est réalisée par le mécanisme d'interruption, par exemple avec une interruption périodique, qui provoque l'exécution de l'ordonnanceur. Dans ce cas, c'est la fonction `execute` qui est appelée (à fournir en plus de la fonction `schedule`).

À partir de la partie prédéfinie, le greffon va synthétiser un nouveau composant ordonnanceur. Cette synthèse comprend la définition d'un composant disposant des interfaces nécessaires aux interactions avec les AIs. Si l'ordonnancement est préemptif, le greffon se charge d'ajouter le composant qui suspendra l'exécution pour exécuter l'ordonnanceur.

Le code correspondant à l'initialisation est intégralement synthétisé par le greffon. Ce code est exécuté directement après le démarrage du matériel, assuré par le composant `boot` ajouté automatiquement par le greffon, et se charge d'initialiser les structures de données liées à l'ordonnancement.

La figure 3.17 illustre le résultat de la traduction obtenue à partir de deux composants Buzz actif `foo` et `bar`.

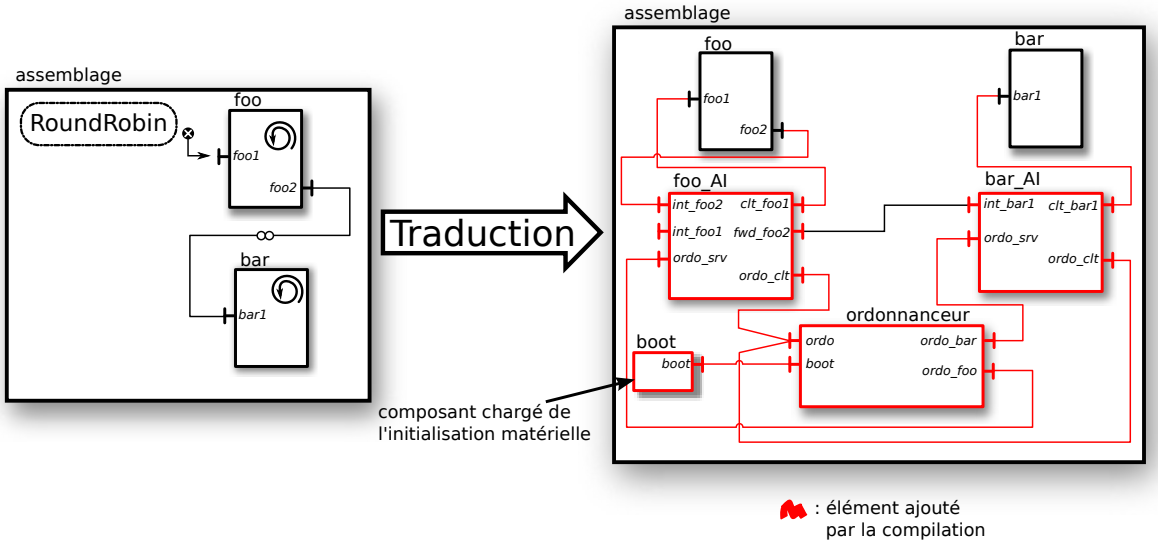


Figure 3.17 – Traduction d'un assemblage de deux composants Buzz.

3.4.3 Optimisations permises par Think

L'utilisation de Think pour l'implantation nous permet de bénéficier directement de certaines de ses optimisations visant les systèmes embarqués.

Architectures statiques. Buzz ne supporte pas la modification de l'architecture pendant l'exécution. Il n'est pas possible d'ajouter ou supprimer des composants, de modifier des liaisons, etc. Comme nous l'avons évoqué en section 1.1.6, Think permet une maîtrise totale du compromis entre flexibilité et sur-côut à l'exécution. Dans Buzz, ces mécanismes de flexibilité ne sont pas utilisés, donc nous ne souhaitons en intégrer aucun à l'exécution. Les liaisons entre tous les composants Think du résultat de la traduction sont implantées concrètement par des appels de fonction ou même directement en intégrant le code appelé dans le code de l'appelant (technique d' « inlining »). Le surcoût généralement associé aux composants disparaît.

Multiples instances de composant. Think permet d'instancier plusieurs fois la même définition de composant. Dans ce cas, des mécanismes de partage de code sont mis en place par le compilateur, mécanismes possédant un coût à l'exécution. Think permet dans les cas où une définition ne possède qu'une seule instance d'économiser le prix de ces mécanismes alors inutiles. Tous les composants introduits lors de la traduction par le greffon sont sujets à cette optimisation.

L'utilisation de Think nous a permis de construire notre traduction en restant au même niveau d'abstraction que le modèle à traduire tout en ne payant aucun prix sur le résultat final de l'implantation par rapport à une solution qui aurait généré du code C classique.

3.5 Traduction pour les analyses d'un modèle Buzz

Cette section est organisée de manière similaire à la section 3.4. Dans un premier temps, la section 3.5.1 présente les principes techniques de la réalisation de la traduction d'un modèle

Buzz vers un modèle en BIP. La section 3.5.2 présente les règles qui composent cette traduction. Enfin, la section 3.5.3 présente une façon d'intégrer le temps dans la traduction.

3.5.1 Principes techniques de réalisation

Comme présenté en section 2.3.4, il est possible d'étendre le compilateur Think par modification de son architecture. Pour la réalisation de la traduction d'un modèle Buzz vers un modèle BIP, nous remplaçons intégralement la partie finale du compilateur. C'est-à-dire, tous les composants du compilateur utilisés pour la génération de code C. Nous remplaçons ces composants par des composants chargés de générer du BIP. La figure 3.18 illustre les changements apportés au compilateur.

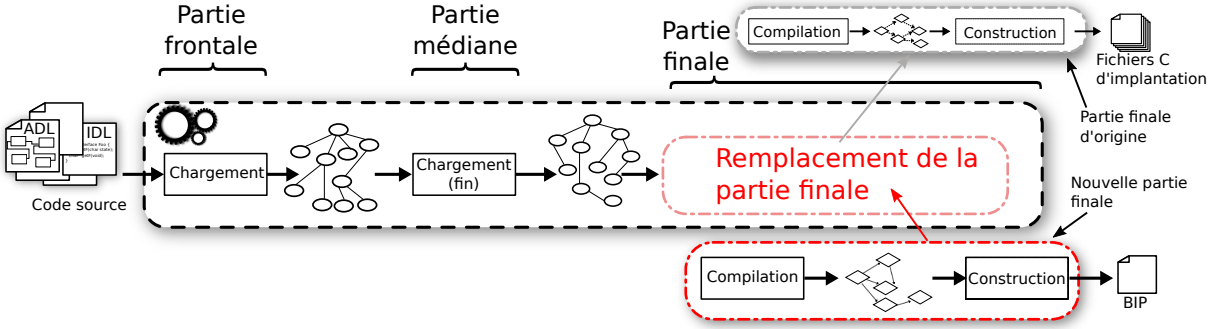


Figure 3.18 – Remplacement de la partie finale du compilateur Think.

Contrairement à la méthode présentée dans la section 3.4.1, il n'est pas nécessaire de modifier l'AST produit par la chaîne de chargement originale pour en retirer les propriétés spécifiques à Buzz. Cet AST enrichi est traité par la partie finale du compilateur spécifique à Buzz qui prend directement en compte ces propriétés. Le principe de traduction est illustré sur la figure 3.19.

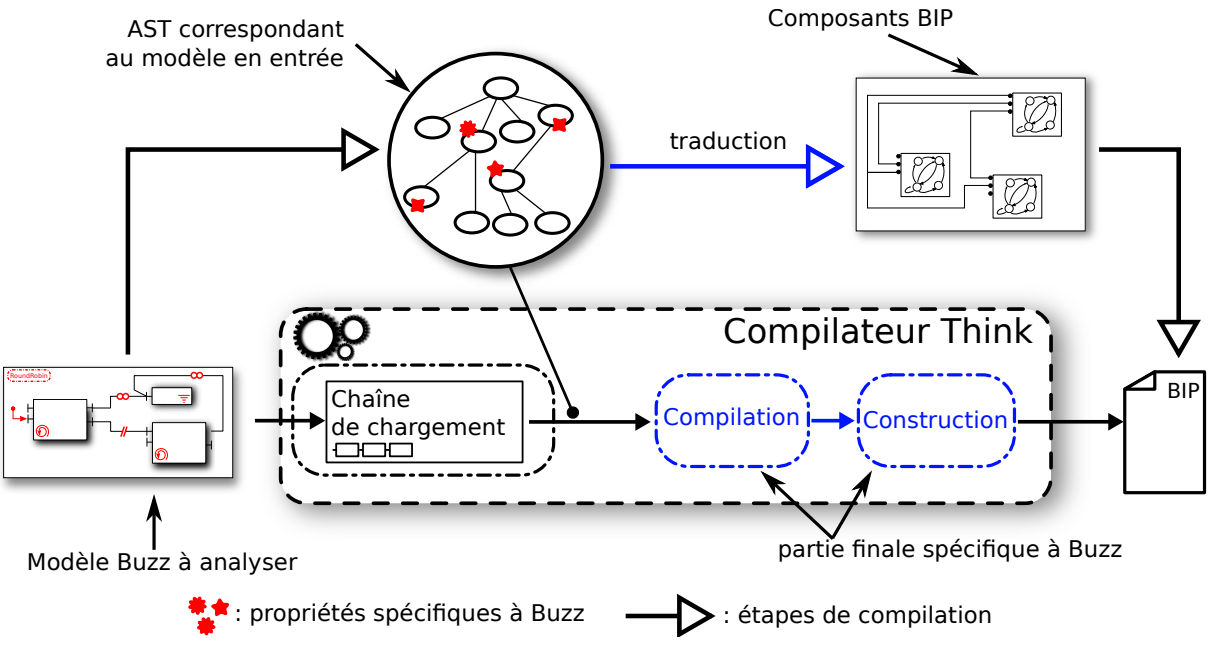


Figure 3.19 – Traduction d'un modèle Buzz vers un modèle BIP au sein du compilateur.

Cette partie finale est composée de 8 composants compilateurs et 7 composants constructeurs (voir la section 2.3.3 pour l'explication des rôles de compilation et de construction des composants). Nous verrons dans la section suivante, qui présente les règles de traduction, que ces composants coopèrent. Cette coopération se fait de manière similaire à la coopération qui se fait entre les composants de la chaîne de chargement : au travers d'une représentation en graphe (appelé ASG pour « Abstract Semantic Graph ») du modèle BIP créé. Nous manipulons dans les parties suivantes trois notions de composants différentes : les composants du compilateur, les composants Buzz et les composants BIP. Au risque d'alourdir le texte, chaque référence à un composant sera systématiquement accompagnée de son type.

3.5.2 Règles de traduction

Nous détaillons ici les règles de traduction qui permettent, à partir d'un modèle Buzz, de construire un modèle BIP. Nous commençons par présenter la nature du contenu des composants Buzz. Ensuite, nous présentons les règles telles qu'elles s'enchaînent dans le compilateur.

Définitions et contenu de composants

Comme nous l'avons rappelé en début de section 3.4.2, le contenu des composants Buzz se présente en deux parties. La première, en langage C, utilisée pour l'implantation. La deuxième est utilisée pour la traduction que nous présentons dans cette section. C'est cette deuxième forme que nous détaillons ici.

De manière similaire à la génération d'un code C d'implantation en Think qui repose sur la présence d'un code C comme contenu pour les composants, la génération d'un modèle BIP à partir d'un modèle Buzz repose sur la présence de modèles BIP pour chacun des composants Buzz, que nous appelons « sous-modèles ». Ces sous-modèles sont manipulés par le compilateur pour construire un modèle BIP pour le système dans sa globalité. Ces sous-modèles possèdent le même rôle que leur pendant en langage C, à savoir spécifier le comportement des méthodes des interfaces serveurs d'un composant. Chacun de ces sous-modèles est un composant BIP atomique dont les ports et l'automate doivent suivre une interface précise.

La liste des ports se décompose en trois groupes :

i. Le premier regroupe les ports utilisés pour la réception des invocations de méthodes sur les interfaces serveurs du composant.

ii. Un deuxième regroupe les ports utilisés pour l'émission des invocations émises par le composant.

iii. Enfin, un troisième groupe rassemble les ports utilisés pour l'ordonnancement, les étapes de calculs interne, etc. Ces ports seront introduits au fur et à mesure dans le texte lorsque cela sera nécessaire.

Aussi bien pour les méthodes clientes que serveurs, les ports doivent être nommés `<nom de l'interface>__<nom de la méthode>` (sans les `< >`).

L'automate prend la forme d'une « marguerite », avec au centre un état appelé `IDLE`. À partir de cet état, un ensemble de transitions mène aux « pétales ». Ces transitions sont étiquetées par les ports utilisés pour les méthodes serveurs (i.). Chaque pétale représente le comportement d'une méthode serveur. Un pétale peut contenir un nombre arbitraire d'états sans aucune convention de nom. Il doit obligatoirement posséder une transition vers l'état `IDLE` étiquetée par le port `terminate`. Ce port fait partie des ports du groupe (iii.) et correspond à la terminaison de l'exécution d'une méthode. Toujours dans un pétale, une transition étiquetée par un port d'une méthode cliente (ii.) correspond à une invocation de méthode vers un autre composant Buzz.

Le niveau d'abstraction utilisé dans les pétales est libre, mais doit au minimum faire ressortir toutes les méthodes qui peuvent être invoquées. Si un pétale utilise des transitions entre deux de ses états qui ne correspondent pas à une invocation de méthode cliente, le port utilisé doit être préfixé par `nada_` (iii.) pour signaler au compilateur qu'il s'agit d'une étape de calcul interne au composant. Enfin, mais nous le verrons plus en détails dans la partie dédiée, les ports représentants des interruptions matérielles doivent être suffixés par `_int`.

La figure 3.20 illustre un composant atomique représentant le comportement de trois méthodes serveurs.

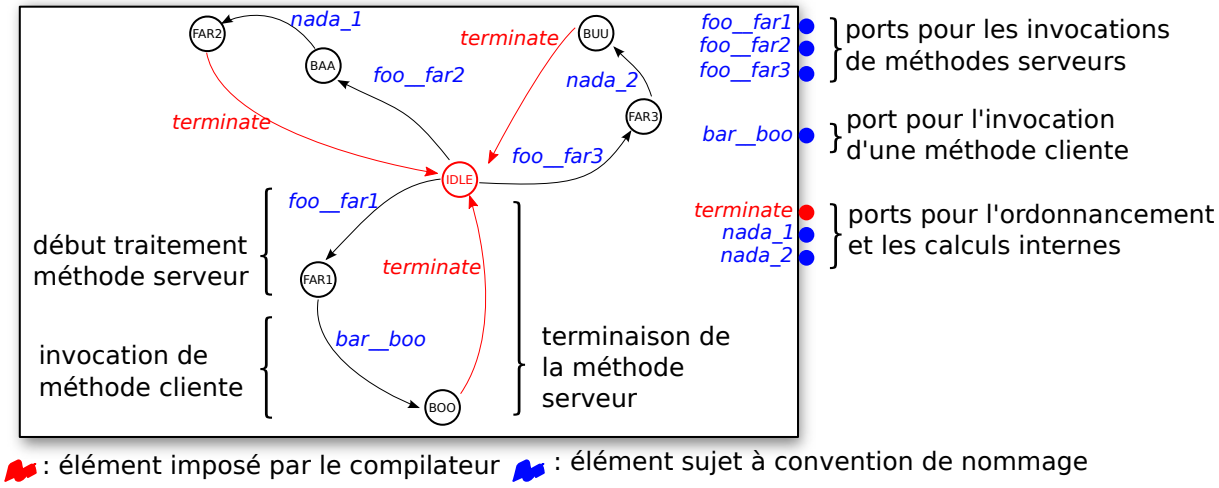


FIGURE 3.20 – Composant atomique BIP associé au contenu d'un composant BUZZ.

Les attributs déclarés dans les définitions de composants sont manipulables dans le modèle BIP en préfixant leurs noms par `att_`. Un attribut nommé `temperature` dans l'ADL devra être manipulé avec le nom `att_temperature` dans le modèle BIP.

Instances de composants actives

Le composant compilateur `ActiveCompiler` se charge de créer une tâche de création d'un composant composite BIP pour chaque instance active dans le système. Chacun de ces composants BIP contient :

- une instance du composant BIP atomique contenu dans le composant BUZZ (tel que défini dans la section précédente). Cette instance s'appelle `core_instance` dans le BIP généré. Nous l'appellerons **cœur** dans la suite de ce manuscrit.
- une instance de composant BIP pour piloter l'exécution du cœur. `active_stub` dans le BIP généré, nous l'appelons composant BIP guide, ou plus simplement *guide*.

Ce composant BIP composite va être complété par chacune des règles présentées dans la suite.

Le rôle du guide est lui aussi détaillé tout au long des différents paragraphes qui suivent. Il possède quatre états, qui correspondent à l'état d'exécution du composant actif :

- `IDLE`, lorsque le composant n'a aucune invocation en attente de traitement ;
- `SUSP`, lorsque le composant est prêt à être exécuté et attend d'être élu pour s'exécuter ;
- `EXEC`, lorsque le composant est en train de s'exécuter ;
- `BLOCKED`, lorsque le composant est bloqué.

Le guide présente un ensemble de ports permettant de le basculer entre ces différents états, dont nous verrons les utilisations dans les paragraphes suivants. Le principe est de ne permettre l'exécution du cœur seulement si le guide est dans l'état `EXEC`. Ainsi, une majorité des transitions du cœur est synchronisée avec une boucle d'exécution présente sur l'état `EXEC` du guide. La figure 3.21 présente le squelette créé pour une instance active Buzz appelée `foo`. Les connexions ne sont pas données précisément sur cette figure car elles sont créées lors d'étapes ultérieures.

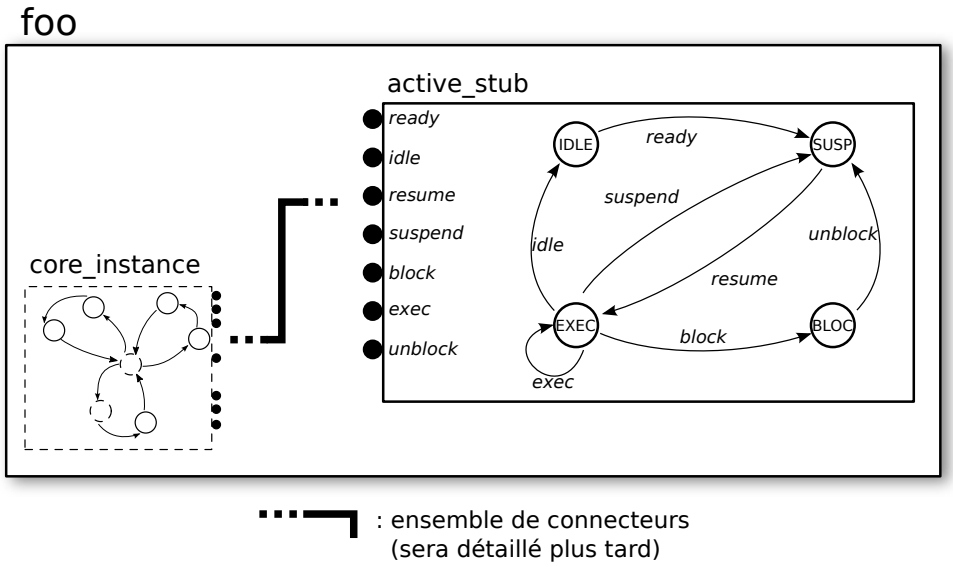


Figure 3.21 – Squelette du composant composite BIP créer pour un composant actif Buzz.

Instances de composants passives

Les instances de composants passives sont aussi gérées par le composant `ActiveCompiler` du compilateur. Le point délicat dans la gestion des composants passifs provient de leur réentrance. En effet, comme nous l'avons vu dans la section 3.2.1, plusieurs contextes d'exécution peuvent exécuter un composant passif, or BIP ne possède aucun support pour exécuter en parallèle plusieurs fois le même composant atomique. Il est donc nécessaire soit de dupliquer ce composant BIP autant de fois que nécessaire, soit de modifier l'automate pour obtenir un comportement équivalent à cette duplication. Nous avons choisi la première solution, plus simple à mettre en place.

Le comportement contenu dans un composant Buzz passif est intégré dans les composants composites BIP des composants actifs qui sont liés au composant passif en tant que client. Autrement dit, tous les composants passifs susceptibles de s'exécuter dans le contexte d'un composant actif voient leurs comportements intégrés aux composants composites correspondants. Cette intégration consiste en l'ajout d'une instance du composant atomique BIP associé au composant passif dans le composant composite. Cette nouvelle instance est pilotée de manière similaire au cœur que nous avons introduit dans le paragraphe précédent. Lorsqu'un composant passif est utilisé par plusieurs composants actifs différents, le composant atomique associé sera dupliqué pour chaque composant composite. Cette duplication permet à chaque composant actif de pouvoir utiliser les composants passifs auxquels il est lié à tout moment de son exécution.

En contrepartie, cette duplication demande un travail supplémentaire pour le partage des données d'un composant passif lorsqu'une instance est accédée depuis plusieurs composants actifs

différents. Dans le cadre de ce travail, nous avons décidé de ne pas gérer ce cas de figure toujours dans un but de simplification du prototype. Il serait tout à fait possible de créer un modèle où le comportement des méthodes serait dupliqué avec un nouveau composant BIP dédié au partage des données. Cela permettrait à la fois les exécutions concurrentes d'un composant passif tout en assurant le partage des données. Dans notre prototype, le partage des données d'un composant entre plusieurs contextes nécessite l'utilisation d'un composant actif.

La figure 3.22 complète la figure 3.21 dans le cas où le composant `foo` est client d'un composant passif `bar`. De nouveau, les connexions ne sont pas détaillées car gérées lors d'une étape future.

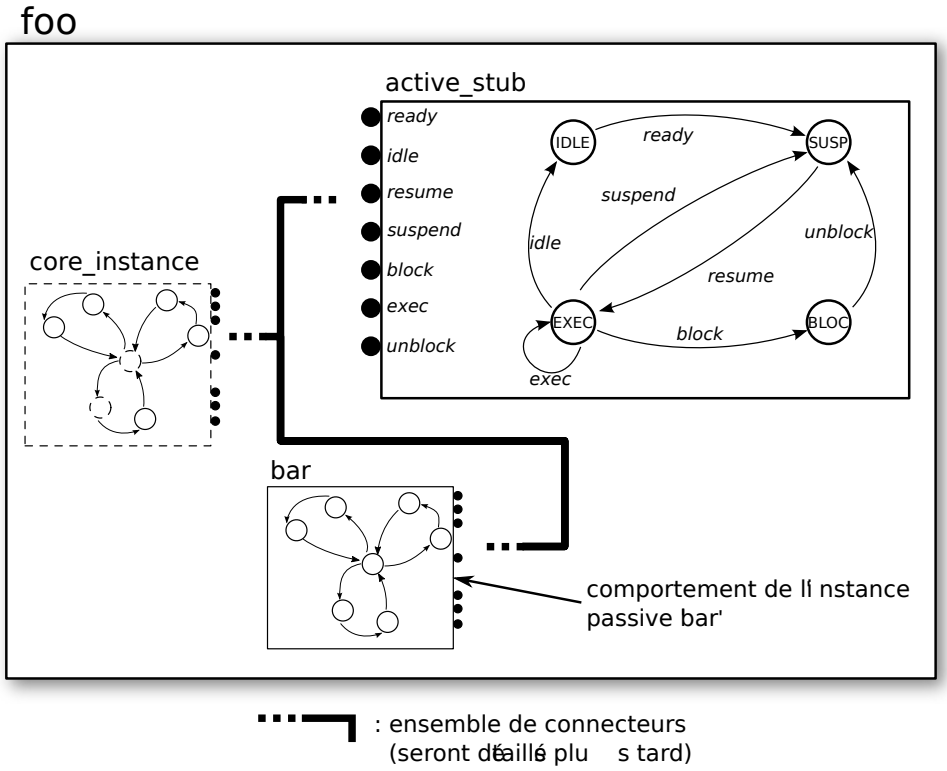


FIGURE 3.22 – Intégration du comportement d'un composant passif dans un composant actif.

Instances de composants interrupteurs

Les instances de composants interrupteurs sont gérées par le composant `InterrupterCompiler` du compilateur. Le composant BIP atomique correspondant à un composant interrupteur doit avoir une forme légèrement différente de celle attendue pour les composants actifs et passifs. En effet, un composant interrupteur est exécuté suite à une interruption matérielle. Nous modélisons les interruptions par l'activation d'un port du composant atomique. Les noms de ces ports spéciaux doivent être suffixés par `_int`. Les pétales traitant les réceptions d'interruption contiennent au plus un état et deux transitions. La première est étiquetée par un port suffixé par `_int` (voir la section 3.5.2) et la seconde par un port pour réaliser une invocation de méthode. Cela correspond aux deux étapes que nous avons décrites en section 3.2.1, à savoir qu'un composant interrupteur reçoit une interruption et la transmet immédiatement à un autre composant pour traitement. Un pétale traitant une interruption peut aussi ne comporter qu'une boucle, qui préemptera simplement le composant en cours d'exécution, sans déclencher de traitant. Enfin,

les pétales correspondant à des méthodes serveurs classiques ont la même structure que celle décrite dans le paragraphe traitant du modèle BIP contenu dans les composants Buzz.

Lorsqu'un composant interrupteur est invoqué depuis un autre composant du système et non par le mécanisme d'interruption, le problème de la réentrance se pose de manière similaire aux composants passifs. La duplication utilisée pour ces derniers n'est pas applicable aussi simplement pour les composants interrupteurs car ils doivent tout même partager le contexte d'interruption. C'est pour cette raison que nous avons introduit la simplification évoquée dans la section 3.2.1, à savoir qu'un composant interrupteur ne peut être exécuté que par un seul contexte d'exécution en dehors du contexte d'interruption.

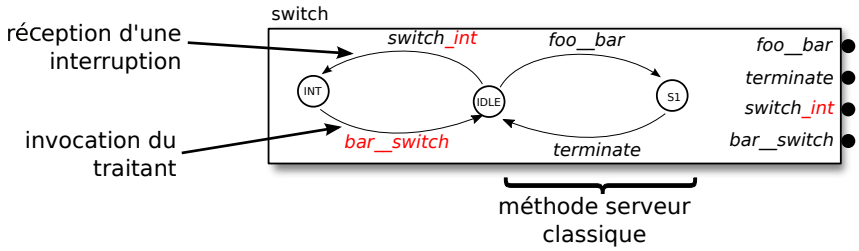


Figure 3.23 – Composant atomique BIP pour un composant interrupteur Buzz.

La figure 3.23 illustre le composant atomique contenu dans un composant interrupteur Buzz, appelé `switch`, utilisé pour la gestion d'un bouton poussoir. À chaque pression, une interruption matérielle est levée. Au niveau du modèle BIP final, ce composant présentera à tout moment cette interruption, comme si elle était systématiquement présente. Ce comportement peut poser problème avec certaines techniques d'analyse (comme l'exploration exhaustive). Plus généralement, les composants interrupteurs devraient contenir un modèle BIP de l'environnement dans lequel s'exécute le système. Ce travail de modélisation ne sera que brièvement abordé dans la section 3.5.3 où nous présentons la création d'un modèle temporisé.

Liaisons synchrones vers des instances passives ou interrupteurs

Les liaisons dont l'extrémité serveur est liée à un composant passif ou interrupteur sont traduites de manière identique. Ce travail est assuré par le composant `BindingCompiler` du compilateur. Cette traduction implique une modification dans le composant BIP atomique correspondant au composant Buzz client. S'il est actif, c'est le cœur qui est modifié, sinon, c'est le composant atomique BIP dupliqué correspondant au composant passif qui est modifié. Nous appellerons *composant client* dans la suite de ce paragraphe le composant BIP modifié et *composant serveur* le composant BIP correspondant au composant Buzz appelé (qui est soit passif, soit interrupteur).

Une invocation synchrone se passe en trois étapes : l'invocation et le blocage de l'exécution du composant appelant, le traitement de la méthode invoquée par le composant appelé et enfin le déblocage de l'appelant. Dans le modèle BIP généré par le compilateur, le blocage de l'appelant est obtenu en ajoutant un nouvel état d'attente dans l'automate du composant client. Cet état est atteint par une transition étiquetée par le port de la méthode invoquée. Cette transition est synchronisée avec la transition de début de traitement du composant serveur. La transition étiquetée par le port `terminate` indiquant la fin du traitement de la méthode est synchronisée avec une transition étiquetée par un nouveau port `resume` ajouté au composant client et sortant de l'état d'attente. Aussi, ces deux transitions sont synchronisées avec la boucle d'exécution du guide. Le caractère « run-to-completion » des exécutions permet de garder les mécanismes

d'invocation, de blocage et de retour relativement simples. Ceux-ci n'ont pas besoin de gérer plusieurs contextes.

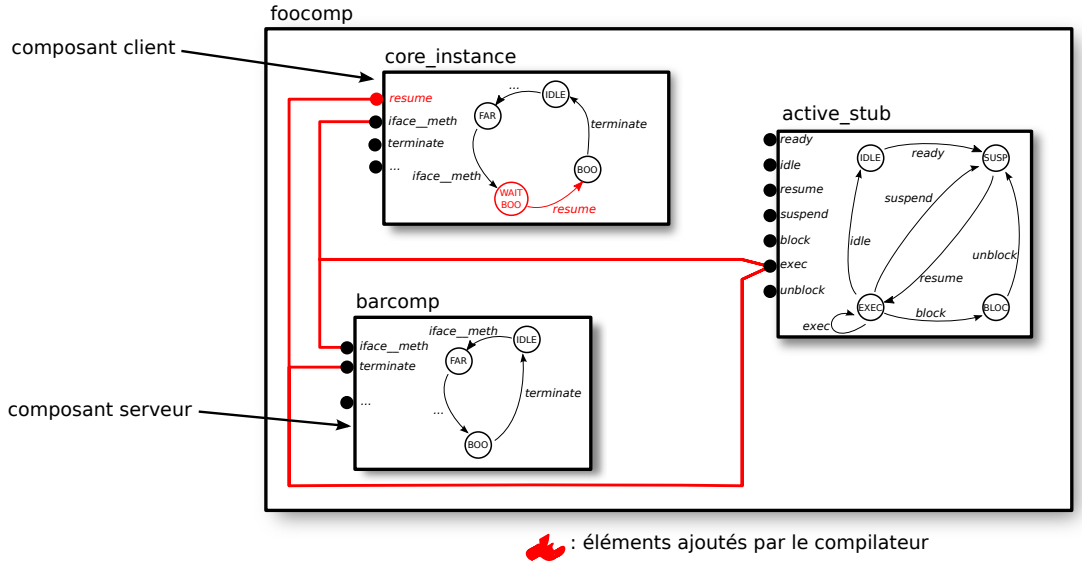


FIGURE 3.24 – Modèle BIP pour une liaison synchrone entre un composant actif et un composant passif.

La figure 3.24 illustre cette transformation dans le cas d'une liaison synchrone entre un composant actif `foocomp` et un composant passif `barcomp` pour l'invocation de la méthode `meth` des interfaces `iface` de chacun des composants. Dans le cas de l'invocation d'un composant interrupteur, le fait que le composant BIP appelé se trouve à l'extérieur du composant composite `foocomp` est la seule différence.

Liaisons vers des instances actives

Les liaisons dont l'extrémité serveur est liée à un composant actif sont les plus complexes à traduire. Pour cette raison, nous présentons dans un premier paragraphe une partie commune qui s'applique à toutes ces liaisons, quel que soit leur mode de fonctionnement, et qui concerne la sérialisation des invocations reçues et le déclenchement de l'exécution du traitement. Ensuite, trois paragraphes présentent les traductions des liaisons asynchrones, retardées et synchrones.

La **sérialisation d'une invocation reçue** consiste à stocker les informations nécessaires au traitement de cette invocation par le contexte du composant appelé. Comme nous l'avons introduit en section 3.2.2, ces invocations sont traitées dans leur ordre d'arrivée. Le composant `BindingBuilder` introduit un ensemble de nouveaux composants atomiques BIP pour imposer cet ordre de traitement. Un premier composant, appelé `input_fifo`, commun à toutes les méthodes des interfaces serveurs, impose l'ordre de traitement (FIFO) en stockant des identifiants des méthodes à exécuter. En plus de ce composant et pour chaque méthode serveur, le compilateur rajoute un composant atomique BIP qui stocke les paramètres propres aux invocations. Ces différents composants permettent de gérer simplement les différents types de paramètres, chaque file d'attente stockant des éléments du même type. Tous ces composants sont similaires, seul les types des éléments contenus dans les files d'attente sont différents. La figure 3.25 illustre ces composants, que nous appelons « FIFO » dans la suite. Pour simplifier le modèle BIP créé, seules les méthodes serveurs effectivement invoquées donnent lieu à cette création de composants.

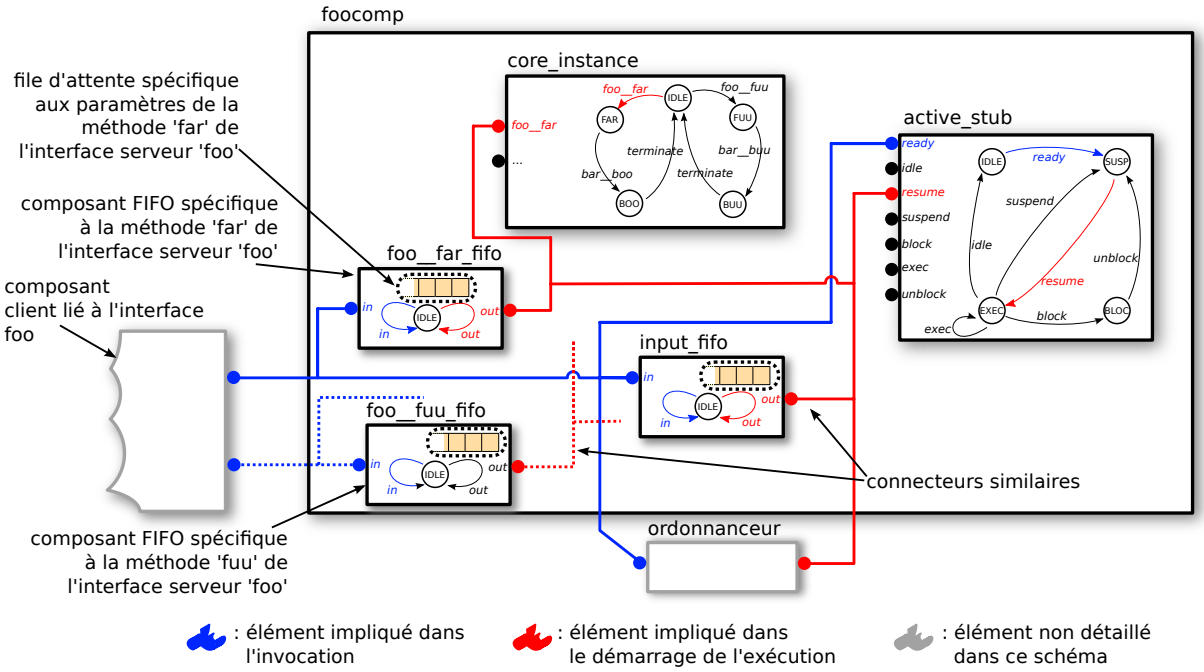


FIGURE 3.25 – Sérialisation des invocations destinées à un composant actif.

La **réception d'une invocation** est prise en charge par un connecteur (en bleu sur la figure 3.25) qui synchronise plusieurs composants :

- le composant FIFO spécifique à la méthode invoquée et le composant `input_fifo`, pour stocker les paramètres de l'appel et l'ordre d'arrivée de l'invocation ;
- le composant ordonnanceur, détaillé plus tard dans le paragraphe traitant de l'ordonnancement, et le composant guide pour faire passer ce dernier de l'état inactif à suspendu dans le cas où le composant actif n'avait aucun traitement en attente avant cette invocation.

Ce connecteur n'impose pas la synchronisation des composants ordonnanceur et guide si ce dernier ne se trouve pas dans l'état inactif.

Le **déclenchement d'une réaction** passe par la synchronisation de plusieurs composants via un connecteur BIP (en rouge sur la figure 3.25) :

- le composant ordonnanceur est synchronisé avec le composant guide pour le faire passer de l'état suspendu à l'état d'exécution. C'est la politique d'ordonnancement qui conditionne cette synchronisation ;
- le composant FIFO correspondant à la méthode à exécuter (c'est-à-dire dont l'identifiant est en tête de la FIFO du composant `input_fifo`) et le cœur (qui passe de l'état `IDLE` au premier état de la méthode à exécuter).

Pour chaque méthode serveur, un tel connecteur est créé. Leur éligibilité pour l'exécution est gardée par la décision de l'ordonnanceur et la méthode se trouvant en tête de la file d'attente du composant `input_fifo`.

Une **liaison asynchrone** ne demande aucun traitement particulier en plus de la sérialisation des invocations du côté du composant appelé, présentée dans le paragraphe précédent (en bleu sur la figure 3.25). Du côté client, un connecteur pour chaque méthode invoquée est créé. La figure 3.26 illustre ce connecteur dans le cas d'une invocation d'un composant actif appelé `foocomp` et complète la figure 3.25. Ce connecteur synchronise la transition étiquetée par le port de la méthode cliente (`bar__buu` dans la figure) et le guide. Ce connecteur est combiné avec les

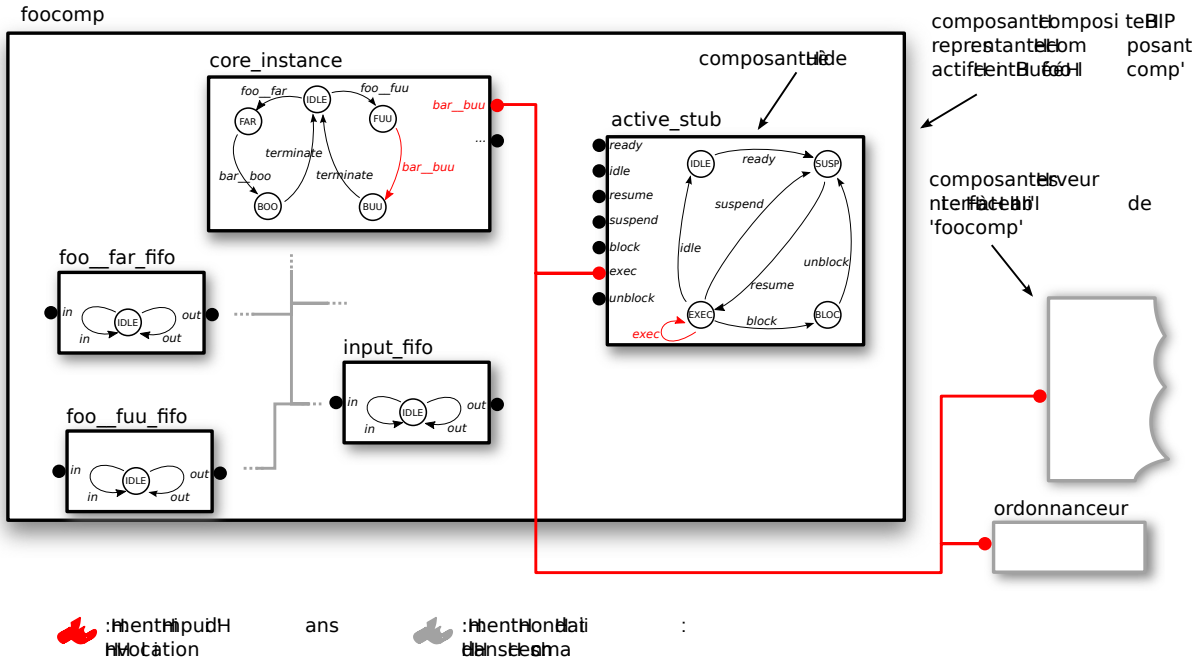


FIGURE 3.26 – Liaison asynchrone vers un composant actif

connexions créées pour la partie serveur (décrites dans le paragraphe précédent). Ainsi, nous obtenons bien le comportement attendu, à savoir que l'appelant n'est pas bloqué.

Les **liaisons retardées** sont une légère variation des liaisons asynchrones. La différence se situe au niveau de la transmission effective de l'invocation au composant appelé. Dans le cas asynchrone, l'invocation est directement transmise au composant serveur. Dans le cas d'une liaison retardée, cette invocation doit être mise en attente et seulement transmise au composant serveur lors de la fin du traitement de la méthode exécutée. La figure 3.27 illustre les modifications apportées au composant composite client par le compilateur. Le connecteur utilisé pour l'invocation, en bleu sur la figure, est lié à un nouveau composant FIFO et non directement au composant serveur. Ce nouveau composant est similaire aux composants FIFO présentés dans le paragraphe traitant de la sérialisation. Un nouvel état `FLUSH` est rajouté juste avant la transition étiquetée par `terminate`. Dans cet état, une boucle étiquetée par un nouveau port `flush` force le vidage du composant FIFO par la transmission des invocations stockées vers les composants serveurs. Le connecteur responsable de cette purge est dessiné en rouge sur la figure.

Les **liaisons synchrones** sont les plus complexes à gérer. Les mécanismes d'ordonnancement n'étant pas présentés avant la section suivante, nous faisons l'hypothèse qu'il existe pour chaque composant actif un connecteur entre son guide et le composant d'ordonnancement lui permettant de se bloquer. Une invocation synchrone et son traitement se découpent en trois parties : l'invocation et le blocage de l'appelant, la réaction dans le composant appelé, le déblocage de l'appelant. Ce paragraphe s'appuie sur la figure 3.28.

La première partie est réalisée par une variation du connecteur utilisé pour l'invocation asynchrone. Les différences sont les suivantes :

- le connecteur utilise le port `block` du guide et non le port `exec`. Le guide passe ainsi de l'état d'exécution à l'état bloqué. Nous verrons dans le paragraphe dédié à l'ordonnancement comment cette demande de blocage est traitée.
- le composant `input_fifo` du composant appelé, en plus de stocker les identifiants de

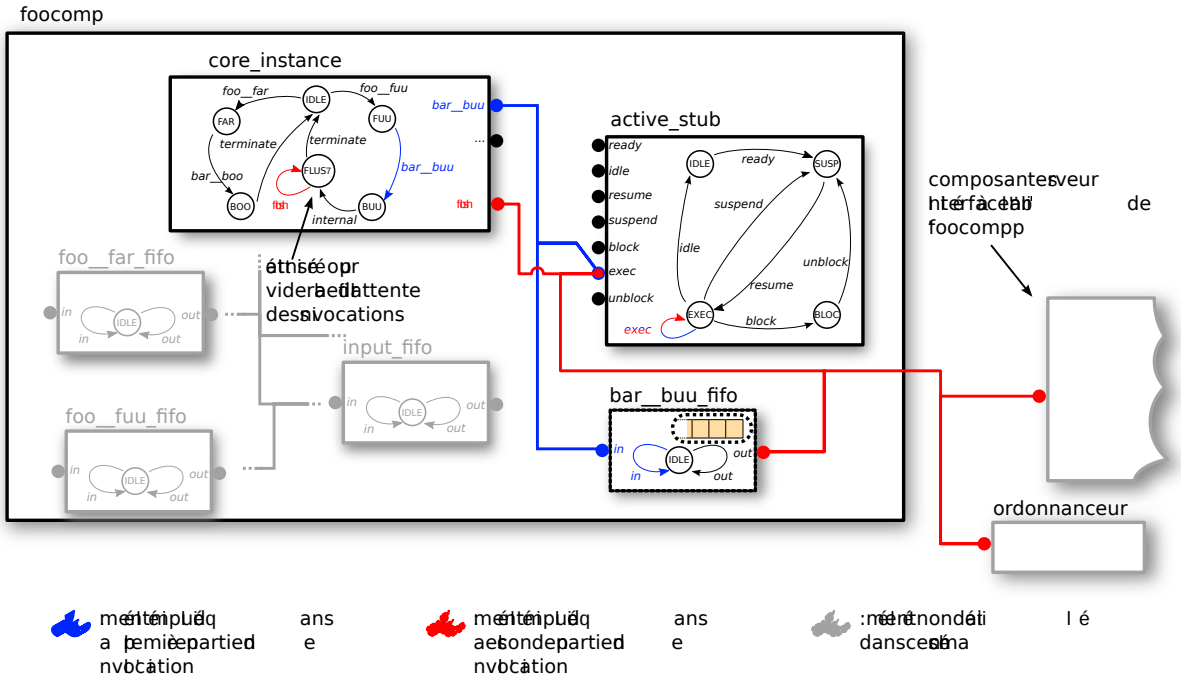


Figure 3.27 – Liaison retardée vers un composant actif

méthodes, stocke les identifiants des composants appelants qui sont détaillés dans le paragraphe traitant de l'ordonnancement.

Autrement dit, le composant appelant transmet l'invocation avec son identifiant et se bloque.

Dans une seconde partie, le composant appelé traite l'invocation. Le mécanisme de déclenchement (en vert sur la figure) de cette réaction est identique à ce qui a été présenté précédemment (voir la figure 3.25).

Enfin, lorsque la réaction se termine, le connecteur utilisé pour la synchronisation de la transition étiquetée par le port `terminate` est aussi lié au composant précédemment bloqué. Ce connecteur permet le passage du guide du composant client de l'état bloqué à l'état suspendu.

Sur la figure, la flèche beige indique le cheminement utilisé par l'identifiant du composant appelant pour permettre son déblocage en fin de réaction.

Ordonnancement

Au niveau du modèle BIP construit, l'ordonnancement contient la politique d'ordonnancement mais aussi l'application de contraintes correspondants à des caractéristiques matérielles des plate-formes visées par les implantations de Buzz.

La **politique d'ordonnancement** des composants actifs dans le système est assurée principalement par un composant que nous avons évoqué à plusieurs reprises précédemment et qui se nomme `ordonnanceur`. Buzz se plaçant dans un contexte mono-processeur (voir le chapitre 3), l'ordonnancement consiste en l'élection du composant actif qui possède le droit de s'exécuter. D'un point de vue matériel, cela correspond à l'élection du contexte d'exécution chargé sur le processeur de la plate-forme. Ce composant ordonnanceur possède une interface très simple : deux données et trois ports.

Le port `fin` et la donnée `find` sont utilisés pour signaler à l'ordonnanceur qu'un composant, dont l'identifiant est copié dans `find`, est maintenant éligible. `fout` et `curd` sont utilisés lors

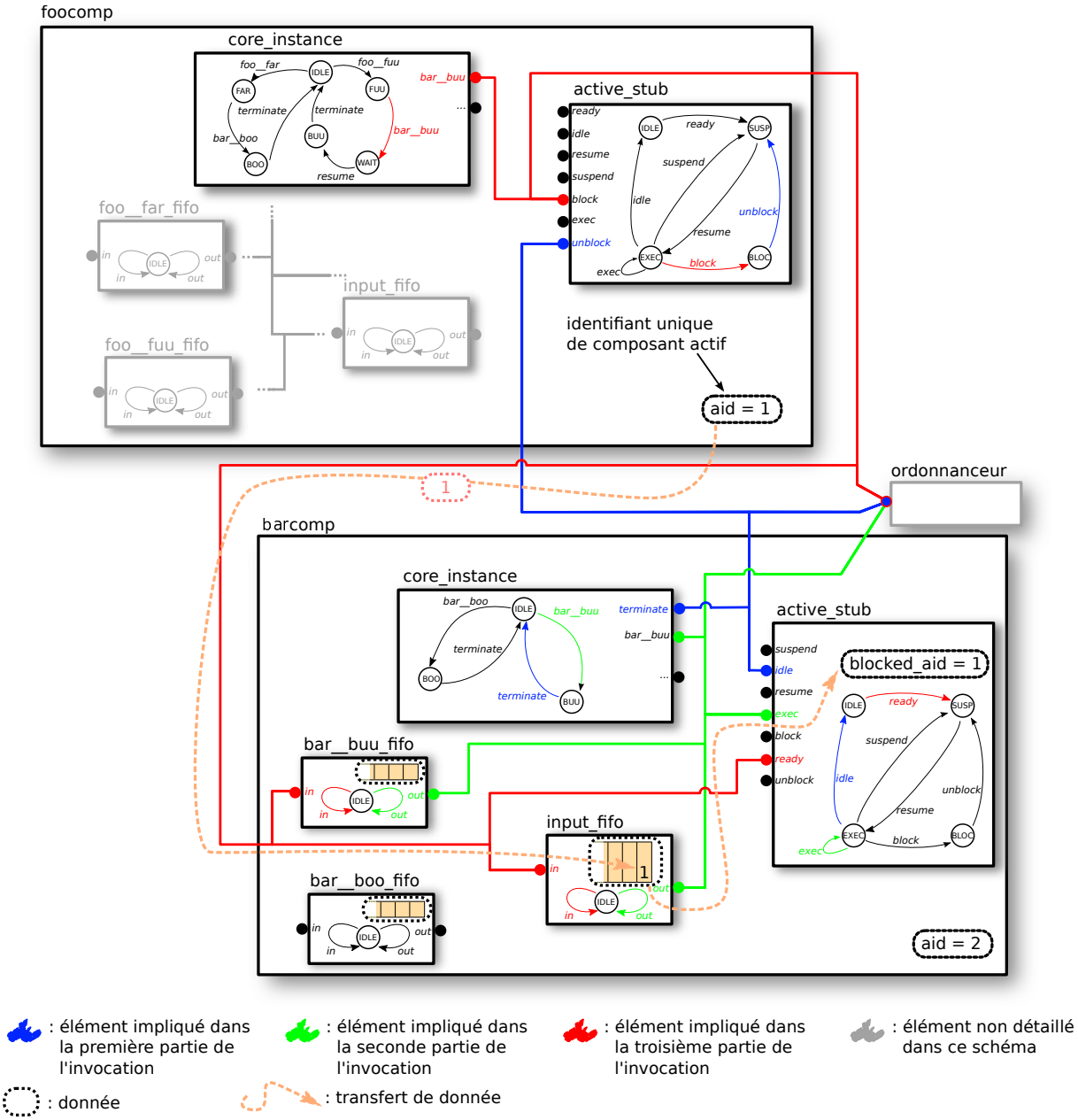


FIGURE 3.28 – Liaison synchrone entre deux composants actifs.

d'invocation désignant un composant actif ou lors du déblocage d'un composant actif suite à une invocation portée par une liaison synchrone.

Le port `fout` et la donnée `curd` sont utilisés par l'ordonnanceur pour appliquer le résultat de l'élection. La donnée `curd` contient l'identifiant du composant actif élu. Ils sont utilisés en particulier pour le déclenchement du traitement d'une invocation en attente par un composant actif. Les connecteurs utilisés pour ce déclenchement, représentés sur la figure 3.25, possèdent une garde qui les désactive si `curd` ne contient pas l'identifiant du composant actif auquel ils sont liés.

Le dernier port, `finrequeue` est utilisé lorsque le composant actuellement élu décide de suspendre son exécution tout en étant candidat pour les prochaines élections.

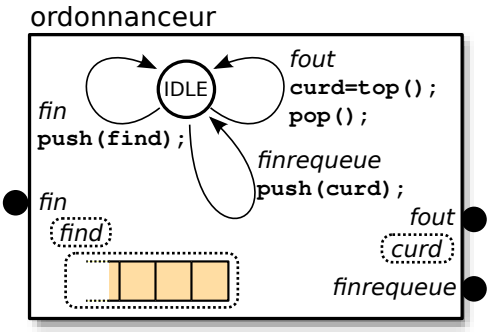


Figure 3.29 – Composant ordonnanceur pour une politique de tourniquet simple.

La figure 3.29 donne le composant tel qu'il est utilisé pour appliquer une politique de tourniquet. Ce composant contient un automate et une simple file d'attente.

L'ordonnancement est aussi chargé de l'**exclusion mutuelle**. Les implantations produites par Buzz visent des plate-formes matérielles mono-processeur (voir la description de Buzz dans le chapitre 3). Nous introduisons pour ce faire un ensemble de règles de priorités dont le rôle est d'inhiber toute nouvelle exécution si un composant guide se trouve déjà dans l'état d'exécution. Ces règles ont la forme suivante :

```
if (c1.active_stub.EXEC)
  {interaction1, interaction2, ...}  < *
```

`c1` désigne le composant BIP composite correspondant au composant actif `c1`. `interaction1` et `interaction2` sont des interactions provoquant le passage à l'état d'exécution d'un composant guide d'un autre composant actif. Cette règle peut se traduire par : « si le composant `c1` s'exécute, alors toutes les interactions menant un autre composant à l'état d'exécution sont interdites ». La figure 3.30 illustre l'application de deux règles dans un système possédant trois composants actifs.

Enfin, c'est le **mécanisme d'interruption** qui est intégré dans l'ordonnancement. Cette tâche, assurée par la plate-forme matérielle en ce qui concerne l'implantation, est réalisée grâce à un ensemble de règle de priorités. Les points importants qu'il est nécessaire de respecter sont :

i. la suspension du composant en cours d'exécution à la réception d'une interruption ;

ii. le masquage des interruptions pendant le traitement d'une interruption.

Le premier point (i.) est assuré par un ensemble de connecteurs qui lient les ports suffixés par `_int` à tous les ports `suspend` des composants guides. Ce connecteur est de type diffusion, c'est-à-dire avec un seul port complet, le port suffixé par `_int` (voir la description des ports BIP complet en section 2.4.2). Un exemple est représenté sur la figure 3.31. Dans cet exemple, le composant `c2` s'exécute lors de la prise en compte de l'interruption représentée par le port `switch_int` et se retrouve suspendu suite à l'exécution de l'interaction représentée en vert. Un ensemble de règles de priorité est employé pour interdire une reprise de l'exécution des composants actifs tant qu'un composant interrupteur traite une interruption.

Le second point (ii.) est assuré par des règles de priorité de la forme :

```
if (! interrupter1.IDLE) interrupter2.signal_int  < *
```

Le port `signal_int` du composant `interrupter2` représente une interruption, qui est interdite par cette règle lorsque le système est en cours de traitement d'une interruption reçue par le composant `interrupter1`.

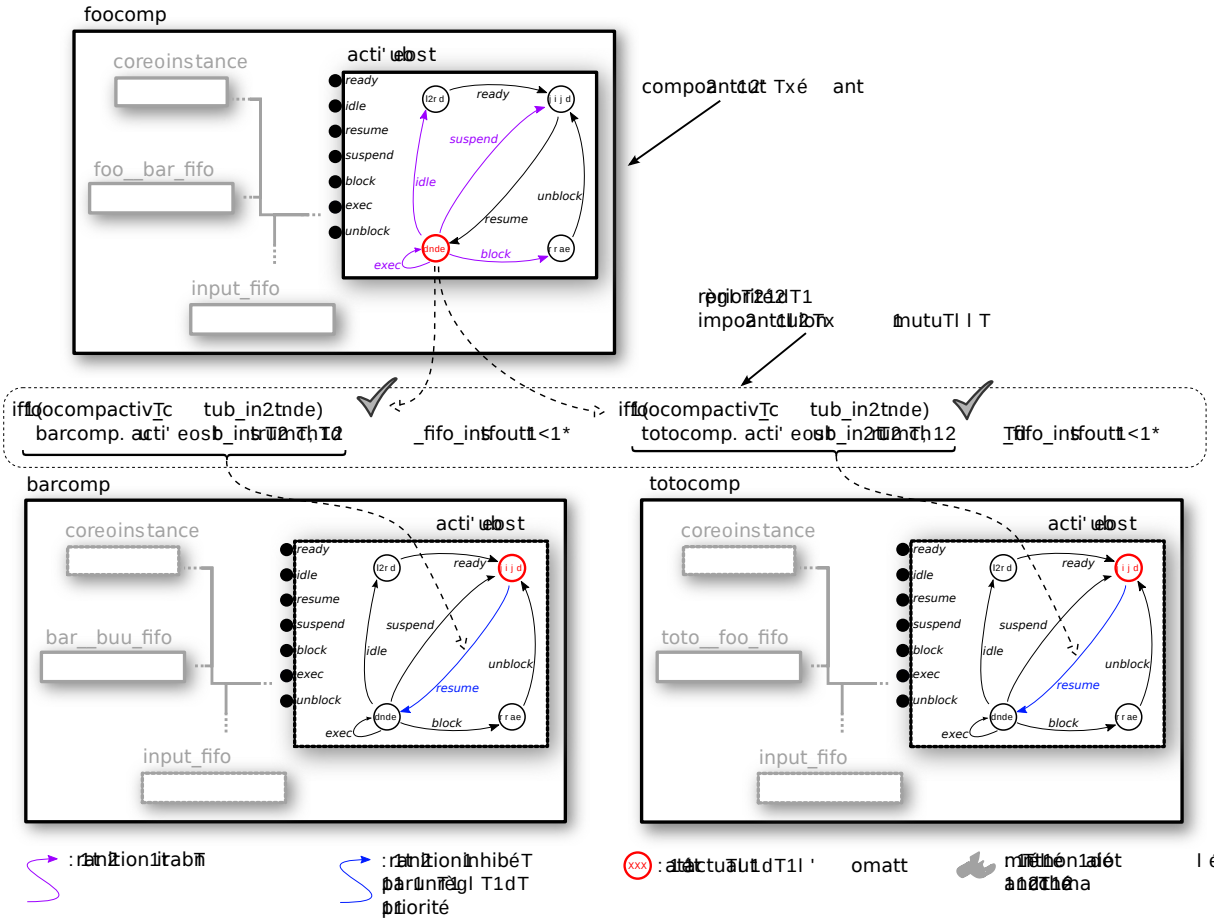


FIGURE 3.30 – Exclusion mutuelle entre les composants actifs.

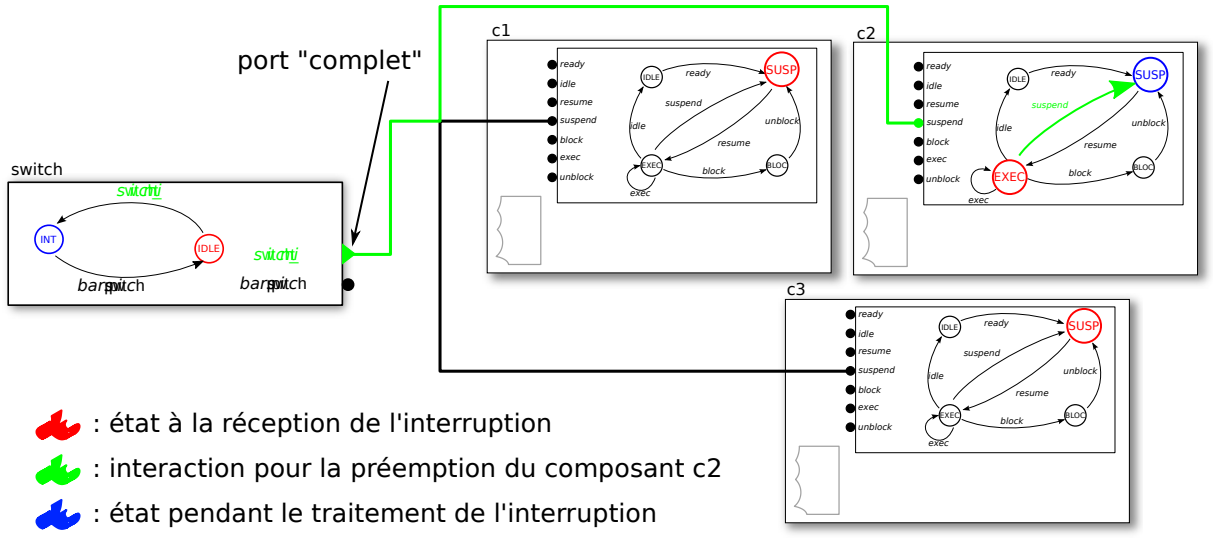


FIGURE 3.31 – Exemple de préemption due à une interruption.

3.5.3 Intégration du temps dans la traduction

Nous présentons dans cette partie une prise en compte sommaire du temps dans la traduction vers un modèle BIP. Cette temporisation illustre une façon d'étendre et d'exploiter le modèle BIP généré par le compilateur. Les résultats qu'il est possible d'obtenir sont à la hauteur de la simplicité de cette extension. Dans un premier temps, nous présentons la manière dont les spécifications temporelles sont introduites ainsi que la spécification des propriétés à vérifier. Ensuite, nous revenons sur les composants interrupteurs et montrons comment affiner leurs modèles en fonction du temps comme évoqué en section 3.5.2.

Temporisation des composants Buzz

Nous décidons, à titre de démonstration simple, de spécifier pour chaque composant Buzz les temps d'exécution minimum et maximum des méthodes serveurs. Nous souhaitons ensuite vérifier que l'exécution du système respecte des contraintes sur les temps de réponses lors de certaines invocations. Le temps considéré est discret et nous appelons l'unité de temps un cycle.

La **spécification des temps d'exécution minimum et maximum** se fait directement dans les composants BIP atomiques contenus dans les composants Buzz. Les temps sont donnés sous la forme d'annotations dans le modèle BIP. Un couple (min/max) est associé aux états des pétales de l'automate. L'exemple du listing 3.6 spécifie que l'exécution de la méthode `m1` de l'interface `foo` dure entre 5 et 6 cycles et celle de la méthode `m2` entre 2 et 5 cycles.

```
behavior
 state IDLE
  on foo__m1 to M1
  on foo__m2 to M2
@minmax=5/6
 state M1
  on terminate to IDLE
@minmax=2/5
 state M2
  on terminate to IDLE
```

Listing 3.6 – Exemple d'annotation du code BIP.

Le composant `BehaviorCompiler` du compilateur est chargé de la prise en compte de ces annotations et procède à des modifications sur tous les composants BIP contenus dans les composants Buzz. Premièrement, une donnée entière appelée `count` est ajoutée dans tous ces composants BIP. Cette donnée est une horloge qui compte le nombre de cycles passés dans chaque état des pétales. Avec un jeu de gardes sur les transitions, le modèle modifié respecte les spécifications des temps minimum et maximum. Pour permettre l'avancée du temps, un nouveau port `clockp` et des boucles étiquetées par ce dernier sont automatiquement ajoutés. La figure 3.32 illustre cette transformation en reprenant l'exemple du listing 3.6. Un connecteur liant tous les ports `clockp` est utilisé pour synchroniser l'ensemble des horloges des composants BIP.

Dans un deuxième temps, nous pouvons spécifier des **contraintes à vérifier**. Cette spécification se fait aussi à l'aide d'annotations mais cette fois sur les transitions étiquetées par un port représentant une invocation de méthode. Ces annotations contiennent une valeur représentant le temps maximum de blocage. Cette annotation n'a de sens que dans le cas où la liaison utilisée est synchrone, le temps de blocage étant systématiquement nul dans les autres cas (les invocations

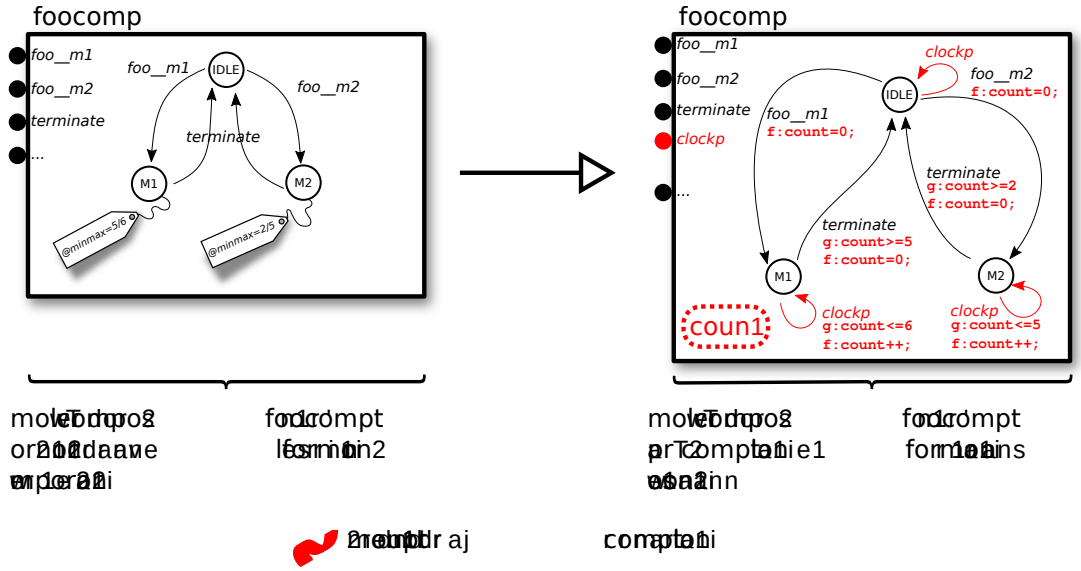


FIGURE 3.32 – Modification du modèle pour la prise en compte du temps.

asynchrones et retardés étants non bloquantes). Le listing 3.7 donne l'exemple d'une contrainte indiquant qu'au plus, l'appel à la méthode `foo` de l'interface `bar` peut bloquer 10 cycles.

```
behavior
 state IDLE
  on bar__f1 to F1
 @foo__bar=10
 state F1
  on foo_bar to ...
```

Listing 3.7 – Exemple de spécification de contrainte de temps d'attente maximum.

À partir de cette spécification, le compilateur rajoute dans le composant une nouvelle horloge qui est remise à zéro lorsque l'automate atteint un état de blocage (voir l'introduction de ces états dans la section 3.5.2). La boucle étiquetée par `clockp` incrémente alors cette nouvelle horloge. Si la valeur de cette horloge dépasse la valeur donnée par l'annotation, l'automate passe dans un état d'erreur pour indiquer le dépassement.

La figure 3.33 illustre l'automate obtenu à partir du composant BIP du listing 3.7. Les gardes ajoutées (en rouge sur la figure) permettent de s'assurer qu'en cas d'erreur, l'automate passe dans l'état `ERROR`. En effet, seule la synchronisation sur le nouveau port `internal` est permise en cas de dépassement.

Temporisation et composants interrupteurs

Les composants interrupteurs sont traités différemment. En effet, les traitements associés à ces composants sont sensés être les plus courts possible. En principe uniquement une invocation de méthode. Nous faisons l'hypothèse simplificatrice que le temps passé dans la prise en compte de ces invocations est insignifiant et donc qu'il n'est pas nécessaire de fournir des temps d'exécution. Cette hypothèse, qui pourrait tout à fait être supprimée, nous permet de simplifier la mise en œuvre de la prise en compte du temps présentée dans le paragraphe précédent. La temporisation de ces composants peut par contre servir à restreindre l'arrivée d'événements modélisant des

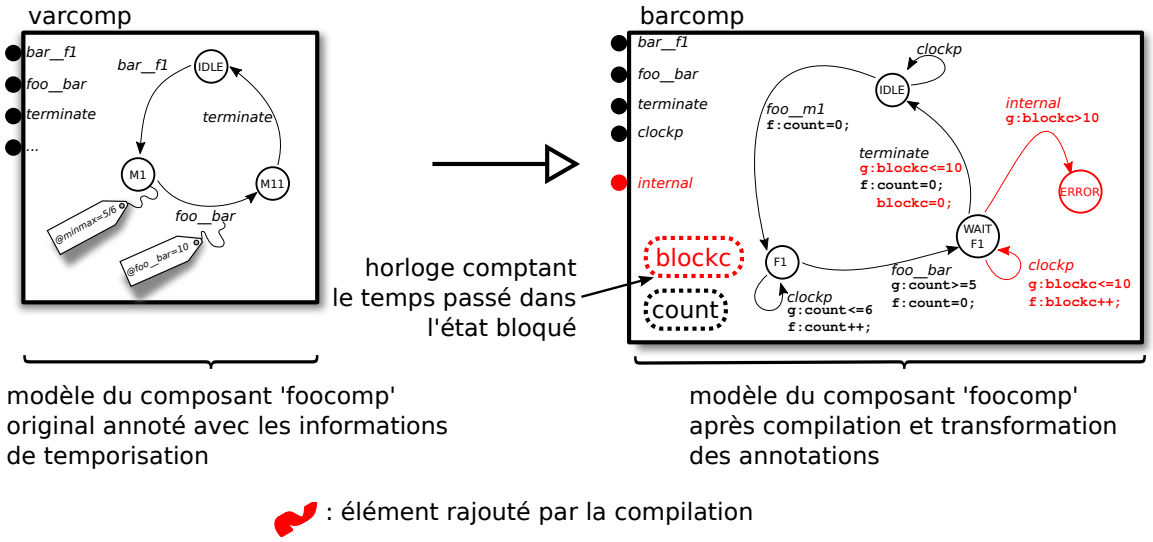


FIGURE 3.33 – Modification du modèle pour la prise en compte d'une contrainte temporelle.

interruptions. Nous avons vu plus tôt, dans la section 3.5.2, que ces composants possèdent des ports spéciaux suffixés par `_int`, qui constituent chacun une interaction possible ne possédant pas de garde et donc toujours éligible. Pour affiner le modèle et limiter les séquences d'exécution où une interruption est tirée à l'infini et « bloque » l'avancement du système, il est possible de conditionner l'arrivée d'interruption en fonction du temps. Un port `clockp` est ajouté ainsi qu'une boucle sur l'état `IDLE` pour faire avancer le temps. Une garde sur la transition modélisant l'arrivée d'une interruption permet de contrôler sa fréquence d'arrivée. Ce modèle peut être vu comme un modèle primitif de l'environnement. En effet, les interruptions d'un système dépendent de l'environnement dans lequel le système s'exécute. L'implantation du système est mise directement en contexte dans l'environnement. Le modèle BIP du système nécessite une modélisation de l'environnement. Nous proposons ici un exemple de construction de modèle simpliste de l'environnement. Une modélisation plus fine demanderait un travail conséquent, qui sort des objectifs de nos travaux.

La figure 3.34 illustre l'exemple d'un composant pour le pilotage d'une liaison série RS232 par laquelle un caractère ne peut arriver que tous les 10 cycles au maximum.

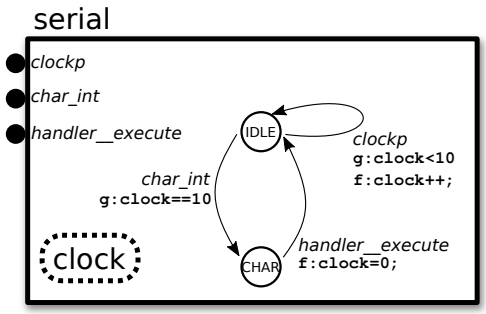


FIGURE 3.34 – Modèle temporisé d'un composant pilotant une liaison série.

3.6 Relation entre les deux traductions

Dans cette section, nous mettons en relation les deux résultats des traductions d'un modèle Buzz simple constitué de quatre composants. Cette mise en relation nous permet d'illustrer la traçabilité des éléments du modèle Buzz au travers des deux traductions. Cette traçabilité joue un rôle important dans la garantie de la fidélité entre les trois représentations d'un système (Buzz, Think, BIP). Plus précisément, nous sommes intéressés par la fidélité des deux résultats des traductions avec le modèle Buzz d'origine. Nous illustrons le devenir d'un modèle Buzz composé de quatre composants, suivant trois points de vue.

Le premier concerne **la conservation de l'architecture** au cours des traductions. La figure 3.35 met en évidence le devenir des éléments d'architecture (composants et liaisons) du modèle Buzz après les deux traductions. On remarque que l'architecture, même si elle n'est pas identique entre le modèle Buzz et les deux résultats, est globalement conservée et est facilement identifiable, aussi bien en ce qui concerne la traduction vers Think que la traduction vers BIP.

Le second point de vue concerne **le pilotage des composants actifs**. De manière similaire à la figure 3.35, la figure 3.36 met en relation les éléments ajoutés pendant les traductions pour assurer le pilotage des composants actifs de manière à respecter la sémantique définie par Buzz (définie dans la section 3.2.1). Les solutions adoptées dans les deux traductions sont proches, à savoir conservation du composant Buzz et de son contenu et ajout de un ou plusieurs composants pour intercepter les invocations entrantes et sortantes. La sérialisation des invocations se fait dans les deux cas en utilisant des files d'attente dédiées (en bleu sur le schéma). Dans le modèle BIP, le pilotage est assuré par plusieurs composants alors qu'en Think, ce pilotage est assuré par un unique composant intercepteur. Pour pousser plus loin la similitude, nous aurions pu regrouper les files d'attente et le guide dans un composant composite BIP.

Le dernier point concerne **la politique d'ordonnancement**. Dans les deux traductions, la majeure partie de l'application de cette politique est assurée par un nouveau composant introduit par le compilateur. La figure 3.37 met en avant ces deux composants (en rouge).

Seule une preuve des outils et des deux traductions peut procurer une confiance totale dans les résultats obtenus. Néanmoins, nous sommes persuadés que la conservation de liens entre le modèle Buzz et ses traductions permet d'atteindre « à moindre coût » un niveau de confiance suffisant. Les deux traductions sont injectives et projettent l'ensemble des modèles exprimables avec Buzz vers deux sous-ensembles des modèles exprimables avec Think et BIP. Nous n'avons pas explicité de traductions inverses, qui traduiraient les modèles de ces deux sous-ensembles vers l'ensemble des modèles exprimables avec Buzz, mais cela serait tout à fait possible. Nous n'avons pas exploré cette possibilité, celle-ci s'inscrivant plus dans une démarche visant à prouver la correction des traductions, ce qui sort du cadre de nos travaux.

3.7 Synthèse

Nous avons présenté dans ce chapitre le langage Buzz qui permet la construction de systèmes embarqués. Un des objectifs du langage est de permettre l'expression d'aspects liés au comportement dynamique du système au niveau de l'architecture, en plus des aspects classiques liés au code et aux données. Ainsi, un composant dans Buzz est une brique structurante à laquelle est attachée en plus de son contenu (code et données) une catégorie correspondant à son mode d'activité dans le système (actif, passif ou interrupteur). Les interactions entre les composants sont elles aussi catégorisées suivant leur mode de fonctionnement (synchrone, asynchrone ou retardée).

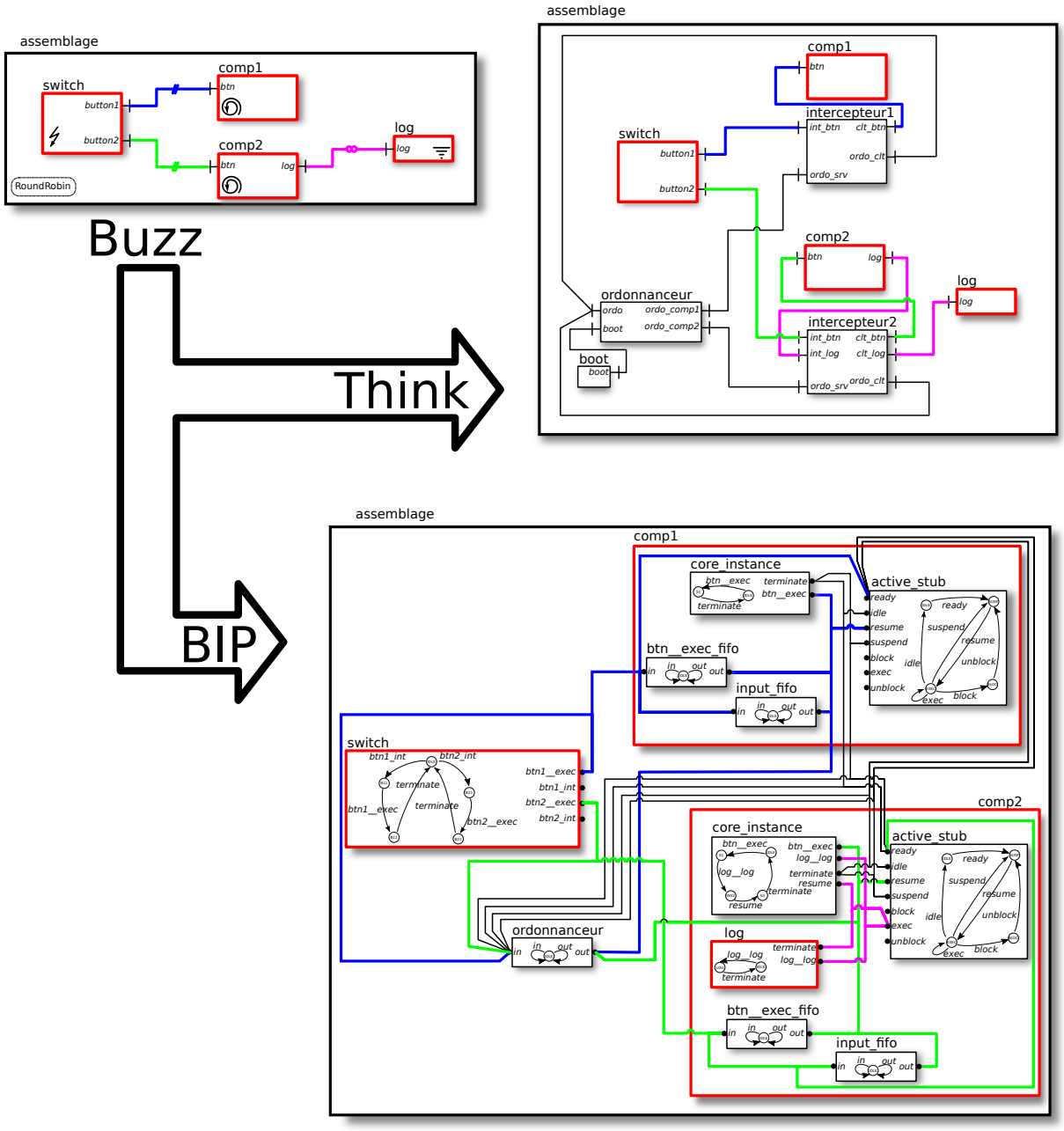
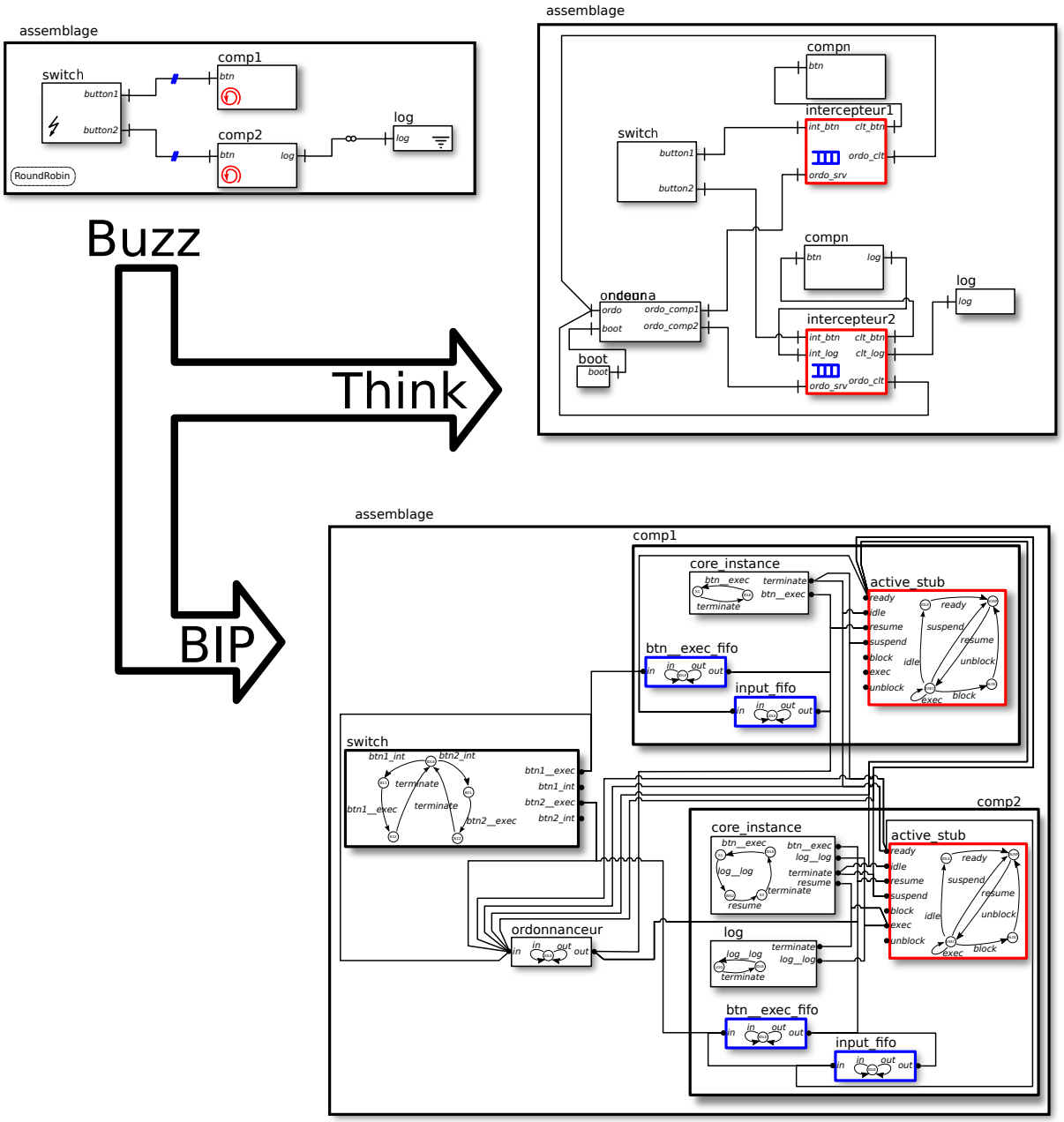


FIGURE 3.35 – Conservation de la structure du modèle BUZZ lors des deux traductions.

Nous avons présenté deux traductions permettant d'obtenir à la fois une implantation et un modèle analysable à partir d'un modèle BUZZ. Cette traduction en une architecture de composants THINK permet de produire simplement et efficacement des implantations adaptées aux cibles matérielles que nous visons. L'effort nécessaire pour cette traduction est nettement inférieur à l'effort qui aurait été nécessaire pour une traduction visant un langage plus classique comme C ou Java. De plus, la traduction vers THINK se fait de manière indépendante des cibles matérielles. Ainsi, les implantations obtenues bénéficient de l'ensemble des possibilités de THINK, en particulier de l'ensemble des plate-formes cibles et des optimisations. Plus généralement, tous les aspects liés au support du matériel sont gérés par THINK. Toutes les améliorations apportées



À chaque couleur correspond un élément du le modèle Buzz et un ensemble d'éléments des deux traductions.

Figure 3.36 – Comparaison du pilotage des composants actifs.

à Think sont directement utilisables avec Buzz.

La traduction vers un modèle BIP produit un résultat qu'il est ensuite possible d'analyser avec les techniques et outils adaptés à cette tâche. Là encore, Buzz ne prend pas en charge cette partie et se place en utilisateur de BIP, comme il se place en utilisateur de Think pour la création d'implantation. Ce modèle capture le comportement de tous les composants du système ainsi que leurs interactions et la politique d'ordonnancement. La prise en charge simple du temps introduite en fin de chapitre illustre la manière par laquelle il est possible d'enrichir la traduction

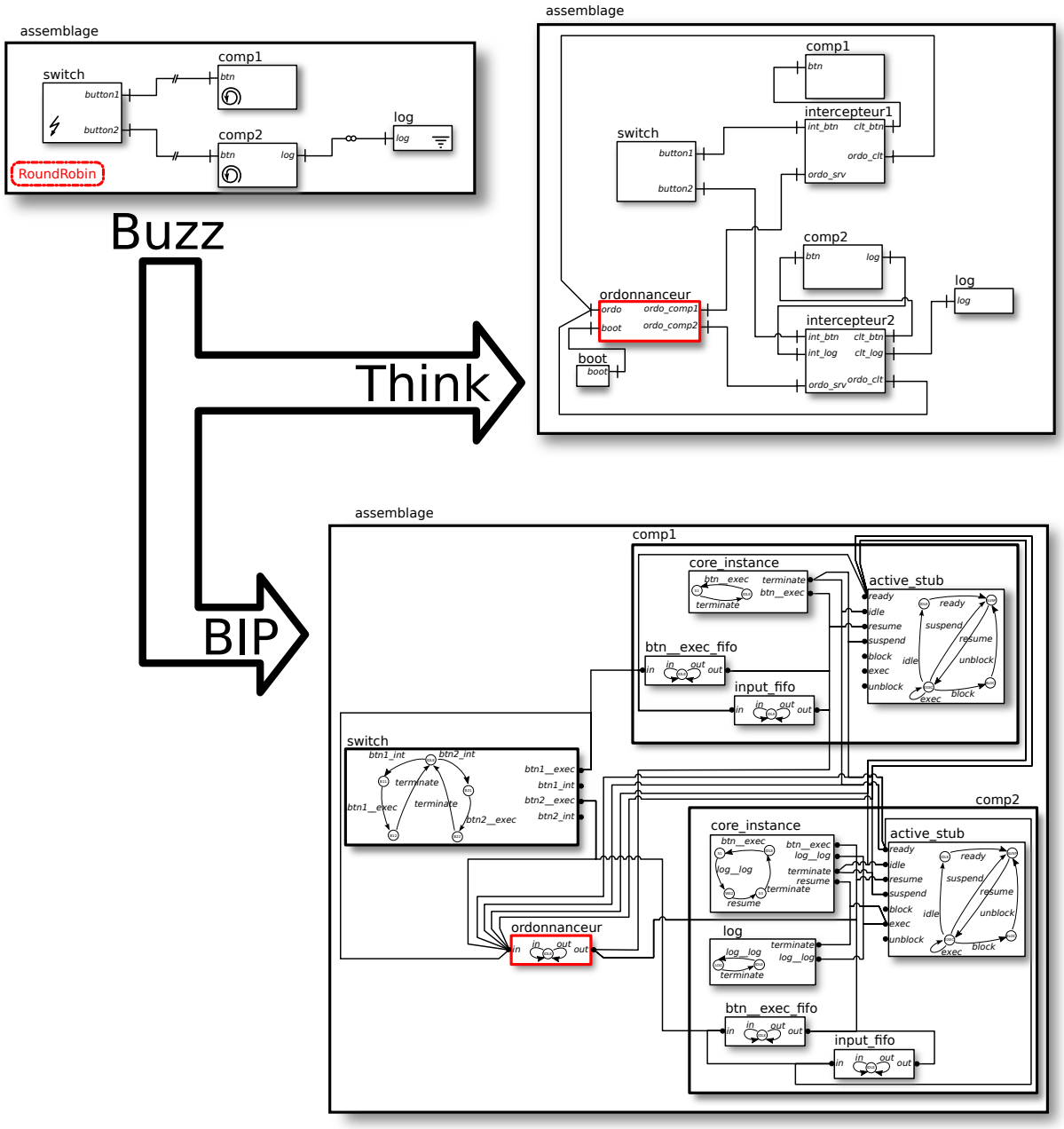


FIGURE 3.37 – Comparaison de l'application de la politique d'ordonnancement.

vers BIP pour y intégrer de nouveaux aspects.

Les deux traductions utilisent autant que possible des constructions similaires pour réduire au minimum l'écart entre l'implantation et le modèle BIP issus d'un même modèle BUZZ. La fidélité entre le modèle BUZZ et ses deux traductions est atteinte par construction, chaque étape de traduction pouvant être validée individuellement.

Comme nous l'avons déjà évoqué dans l'introduction de ce chapitre, BUZZ n'a pas vocation à être un langage exhaustif et complet. Il manque d'ailleurs pour cela des primitives que nous avons volontairement écartées pour ne pas alourdir le langage et son compilateur. Nous avions dans une première version du langage introduit le support de liaisons complexes plus générales

que les seules liaisons point à point actuelles. Le travail de spécification nécessaire a motivé le retrait de ces liaisons qui n'apportaient aucun élément supplémentaire pour notre démonstration. De la même manière, nous avons retiré du langage les travaux visant à supporter des niveaux de priorité pour l'ordonnancement des invocations au sein d'un composant actif.

Troisième partie

Évaluations

Chapitre 4

Évaluations

Sommaire

Nous présentons ici les résultats obtenus à partir du langage Buzz et ses outils, présentés dans le chapitre précédent. La présentation de ces résultats est structurée en deux parties correspondant chacune à un exemple. La première partie s'articule autour du problème du dîner des philosophes. Cet exemple académique permet d'illustrer l'intégralité de la démarche tout en restant suffisamment simple pour être complètement préhensible. La deuxième partie présente la réingénieurie avec Buzz d'un logiciel embarqué existant. Celui-ci a été repris d'une expérience menée par France Telecom R&D dans le contexte des réseaux de capteurs. Ces deux expériences visent à confronter à la réalité le langage Buzz ainsi que son compilateur. En particulier, ces expériences permettent de vérifier que les primitives du langage sont bien adaptées aux systèmes visés. Cette adaptation comprend le niveau d'abstraction ainsi que l'expressivité du langage. De même, pour chacune des expériences, nous menons une étude quantitative sur les résultats obtenus par le compilateur. Pour l'implantation, cela comprend la mesure des tailles des exécutables ainsi que des mesures de performance à l'exécution. Pour le modèle BIP, nous donnons la taille du modèle généré et mettons en œuvre quelques techniques d'analyse.

4.1 Application à un problème classique de concurrence

Cette section présente l'utilisation de Buzz avec un problème classique de concurrence : le dîner des philosophes, présenté par Dijkstra dans [Dij71]. Cet exemple détaille l'étape de conception et présente les résultats de la génération de code exécutable et la génération d'un modèle BIP. Nous montrons avec cet exemple la facilité avec laquelle ce problème classique est modélisé avec les concepts disponibles dans Buzz. Nous montrons également que cette facilité de conception s'accompagne par un gain en terme de développement, une partie du code classiquement écrite par le développeur est alors prise en charge par Buzz pendant la phase d'implantation. Implantation qui sera confrontée à des plate-formes matérielles réelles. Enfin, cet exemple nous permet d'illustrer quels types de propriétés il est possible de vérifier directement à partir du modèle BIP généré par le compilateur.

Comme nous ne trouvons pas raisonnable de manger avec deux fourchettes comme c'est le cas dans l'énoncé classique du problème, nous préférons utiliser des baguettes (« chopsticks »). Le problème est le suivant : n personnes, que nous appellerons « geeks » dans la suite, sont disposées en cercle autour d'une table, avec un bol de riz chacune. Une baguette est placée entre chaque geek. Comme il est nécessaire d'avoir deux baguettes pour manger, tous les geeks à table ne peuvent manger en même temps et ils doivent partager les baguettes pour que personne ne meure de faim. K.M.Chandy et J.Misra ont proposé une solution [CM84] à ce problème. Celle-ci peut être résumée comme suit :

- chaque geek à table peut se trouver dans trois états : rêvasse, veut manger et mange.
- chaque baguette peut être propre ou sale. Une baguette propre devient sale après avoir servi. Une baguette sale est nettoyée quand elle est transmise d'un geek à un autre.
- lorsqu'un geek a faim, s'il possède les baguettes, alors il mange. Sinon, il demande à ces voisins les baguettes manquantes. Lorsqu'il les reçoit, il mange.
- lorsqu'un geek reçoit une demande de baguette, si celle-ci est sale et qu'il n'a pas faim, alors il nettoie la baguette et la donne. Sinon, il nettoiera et cédera la baguette après avoir fini de manger.

Cette solution se décrit simplement avec Buzz en utilisant pour chaque geek à table une instance de composant active. Chaque geek passe de l'état de rêvasserie à l'état de faim suite à la réception d'une interruption, c'est-à-dire suite une invocation d'un composant interrupteur. Chaque geek à table peut :

- envoyer et recevoir des demandes de baguettes ;
- envoyer et recevoir des messages de transmission d'une baguette.

Ces communications se font entre voisins et de manière asynchrones. Elles sont réalisées par des liaisons entre les quatre interfaces (deux clientes et deux serveurs) que possèdent chaque composant geek. Nous utilisons deux attributs (`left_cs` et `right_cs`) pour décrire l'état des baguettes pour chaque geek, qui peuvent prendre les valeurs `manquante`, `propre` et `sale`.

Les quatre interfaces sont du type `Chopstick`, qui définit deux méthodes :

- `void takeit(void)` : pour donner une baguette à son voisin.
- `void giveme(void)` : pour réclamer une baguette à son voisin.

Ces interfaces sont nommées `toleft` et `toright` pour les clientes, et `fromleft` et `fromright` pour les serveurs.

La cinquième interface, serveur et nommée `handler`, est de type `irq.Handler` et définit une unique méthode `void execute(void)` qui signale la réception d'une interruption. Cette méthode est invoquée par les composants interrupteurs sur les composants geeks pour les faire passer de l'état de rêvasserie à l'état de faim.

La figure 4.1 donne la représentation graphique des composants (le composant `sigusr` utilisé

comme composant interrupteur est détaillé plus loin).

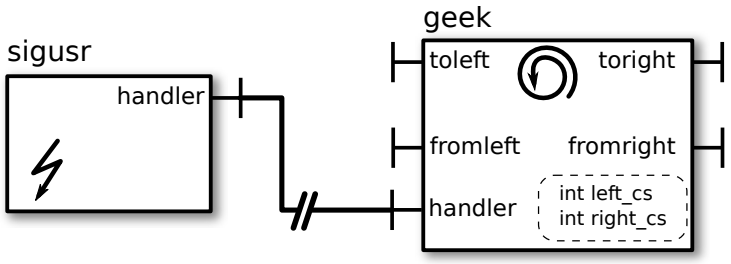


FIGURE 4.1 – Un composant actif geek et son composant interrupteur.

Dans un premier temps, nous prenons une instance du problème avec trois geeks (pour un total de six composants). Ce choix permet de maintenir une taille raisonnable pour la présentation de l'étude sans en réduire l'intérêt. L'architecture qui est utilisée tout au long de cette partie est illustrée dans la figure 4.2. Pour l'ordonnancement des geeks, nous utilisons une politique d'ordonnancement préemptive, sans partage de temps : les décisions se font lors des terminaisons de méthodes ou lors des interruptions.

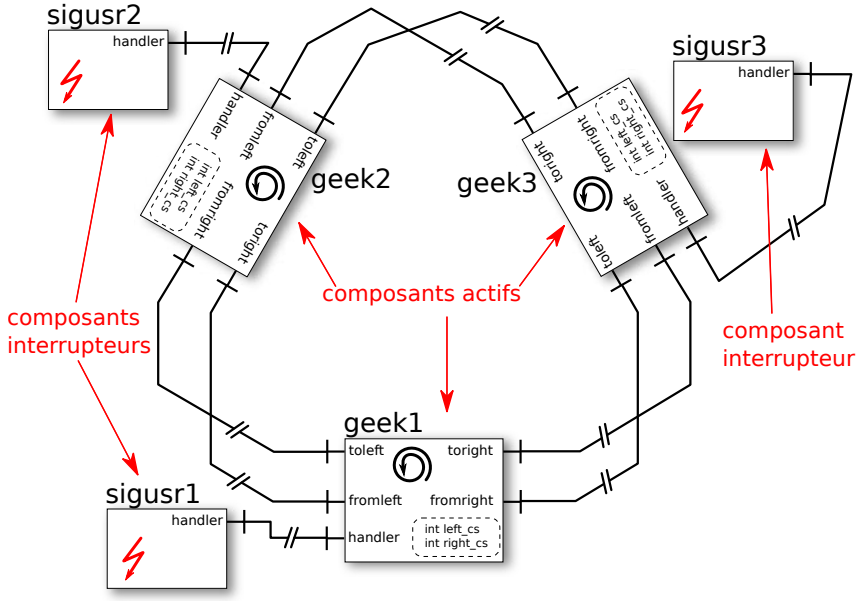


FIGURE 4.2 – Trois geeks mangeurs.

Plate-formes matérielles utilisées. Nous considérons ici les cas des trois plate-formes matérielles supportées par le compilateur de BUZZ. La première, appelée le cognichip, est basée sur un micro-contrôleur AVR 8bits (Atmel ATmega256-1). Cette plate-forme possède 256KiB de mémoire Flash, 8KiB de RAM, deux têtes radio et un lien série RS232. C'est une architecture Harvard [17]. La deuxième, appelée WSN430, est une plate-forme basée sur le micro-contrôleur MSP430 16bits (Texas Instrument) qui nous a été fourni par le laboratoire du CITI de l'INSA Lyon [18]. Cette plate-forme possède 48 KiB de Flash, 10KiB de RAM, une puce ChipCon 1100

17. les mémoires contenant le code et les données sont séparées
18. Projet Worldsens du CITI Lyon : http://worldsens.citi.insa-lyon.fr/

pour les communications radio et un lien série RS232. Ces deux plate-formes sont des exemples « types » du matériel employé dans les réseaux de capteurs en termes de puissance de calcul et d'espace de stockage. Enfin, la dernière plate-forme n'a rien d'embarquée mais permet de déboguer de manière plus confortable : unix (plus précisément, GNU/Linux sur plate-forme matérielle x86). Plus de détails sur cette plate-forme sont donnés en annexe B. Comme nous l'avons vu dans le paragraphe précédent, à chaque geek est associé un composant interrupteur. La nature exacte de ces interrupteurs dépend de la plate-forme utilisée. Ainsi, pour chacune des plate-formes employées dans cette expérience, nous utilisons un ensemble de composants interrupteurs différents :

- pour l'AVR, un timer, un lien série RS232 et un lien radio sont utilisés.
- pour le MSP430, deux timers et le lien série RS232 sont utilisés.
- pour GNU/Linux, trois signaux [19] sont utilisés pour simuler des interruptions. Ces signaux sont générés depuis une application externe de manière aléatoire, périodique ou manuellement.

4.1.1 Évaluation du code exécutable

L'architecture présentée dans la figure 4.2 est modifiée pendant la compilation vers un exécutable (processus détaillé dans la section 3.4). Cette compilation ajoute automatiquement des composants THINK nécessaires à l'exécution du système, comme le composant `boot` (pour le démarrage) ou le composant `irqSafe` (pour la gestion du masque d'interruption). Ces composants ne doivent pas être comptabilisés dans le coût propre de BUZZ, leur présence étant obligatoire.

Les ajouts entièrement imputables à BUZZ sont les intercepteurs des composants actifs et le composant ordonnanceur. Nous vérifions dans une première partie que ce coût reste à l'échelle des plate-formes visées, en particulier en terme d'empreinte mémoire, certaines cibles bénéficiant d'à peine quelques KiB. Dans une deuxième section, nous abordons brièvement les performances à l'exécution en mesurant l'impact de l'ajout des composants intercepteurs et ordonnanceur.

Nous donnons dans les paragraphes suivants des valeurs pour les trois plate-formes considérées. Ces valeurs sont fortement dépendantes de la plate-forme, du compilateur C, des options de compilations, ... Il ne faut donc pas comparer les tailles entre les différentes cibles. Les mesures ont été effectuées en utilisant le compilateur C de GCC, dans des versions différentes, en fonction des disponibilités sur chacune des plate-formes :

- 3.2.3 pour le MSP430.
- 4.3.2 pour l'AVR.
- 4.3.3 pour GNU/Linux.

Même si nous verrons qu'il n'existe pas d'écart sensible dans les coûts liés à BUZZ sur MSP430, l'utilisation d'une version ancienne (3.x) de GCC doit être pénalisante par rapport à l'utilisation de versions récentes (4.x) qui mettent l'accent sur de meilleures optimisations (par exemple en compacité de code produit).

Mesures des tailles du code et des données

Taille de code. BUZZ rajoute des composants qui sont créés pendant la compilation et dont le code est en partie fourni dans une bibliothèque (algorithme d'ordonnancement des composants actifs) et en partie synthétisé (intercepteurs, partie spécifique à l'architecture de l'ordonnanceur). Le tableau 4.1 donne les tailles de code imputables à BUZZ. Les trois instances de composants

19. les signaux sont un moyen de communication asynchrone entre les processus utilisés sur les systèmes d'exploitation de type Unix.

	Ordonnanceur	Intercepteur	Taille totale
MSP430	956	894	10182
AVR	1276	1342	17750
GNU/Linux x86	1241	1393	9207
Coût moyen	9%	10%	

Les tailles sont données en octets. La colonne intercepteur ne détaille qu'une des trois instances présentes dans le système.

TABLE 4.1 – Coût en taille du code.

	Ordonnanceur	Intercepteur	taille totale
MSP430	44	604	2218
AVR	41	348	2299
GNU/Linux x86	460	8720	27476
coût moyen	1%	24%	

Les tailles sont données en octets et regroupent les données non initialisées (section `BSS`) et les données initialisées (section `DATA`). La colonne intercepteur ne détaille qu'une des trois instances.

TABLE 4.2 – Coût en taille des données.

applicatifs (les geeks) étant relativement simples, le coût associé à BUZZ peut sembler disproportionné. En réalité, c'est la simplicité de l'application qui biaise ces proportions, le coût de BUZZ augmentant avec la complexité de l'architecture (*i.e.* nombre de composants et de liaisons) et non avec la taille du code applicatif (qui dans cet exemple est quasi nul). Le coût de BUZZ reste tout à fait à l'échelle des plate-formes utilisées avec à peine quelques KiB. Les tailles totales sont aussi fortement dépendantes des plate-formes utilisées. Les raisons principales sont :

- présence de logiciel qui sera lié dynamiquement pour GNU/Linux (la bibliothèque C standard par exemple), qui n'est donc pas compté, alors que ces mêmes services sont liés statiquement pour les deux autres plate-formes (et donc comptabilisés).
- l'absence d'unité de calcul en nombres flottants (FPU) sur l'AVR pénalise cette plate-forme qui doit héberger de lourdes routines logicielles (plusieurs KiB) pour compenser cette absence.

Taille des données. La tableau 4.2 présente le coût en données de l'ensemble des composants ajoutés par BUZZ. Il faut noter que ce coût, même s'il représente une part importante de la taille totale, est facilement maîtrisable. La plus grande partie de la mémoire reservée aux données est occupée par les piles d'exécution (256 octets sur AVR et MSP430 et 8KiB pour GNU/Linux) de chacun des composants actifs ainsi que des files d'attente nécessaires à la sérialisation des appels destinés aux composants actifs. Dans cet exemple, nous avons fixé la taille des files d'attente à quatre éléments. Comme les communications entre composants actifs sont asynchrones, aucune file d'attente de « déblocage », utilisée pour les communications synchrones, n'est créée.

Évolution du coût avec la complexité de l'architecture. Le coût global de l'infrastructure rajoutée par BUZZ est une fonction linéaire de la complexité de l'architecture compilée et peut être calculé à priori. Par exemple, le coût en taille des données est obtenu par :

$$C_D = \sum_{i \in \mathrm{CA}} C_{\mathrm{IA}_i} + C_{\mathrm{ord}}$$
$$C_{\mathrm{IA}_i} = T_{\mathrm{pile}} + \sum_{j \in \mathrm{IFS}_i} T_{\mathrm{file}_i}$$

avec (les coûts considérés ici ne concernent que les données) :

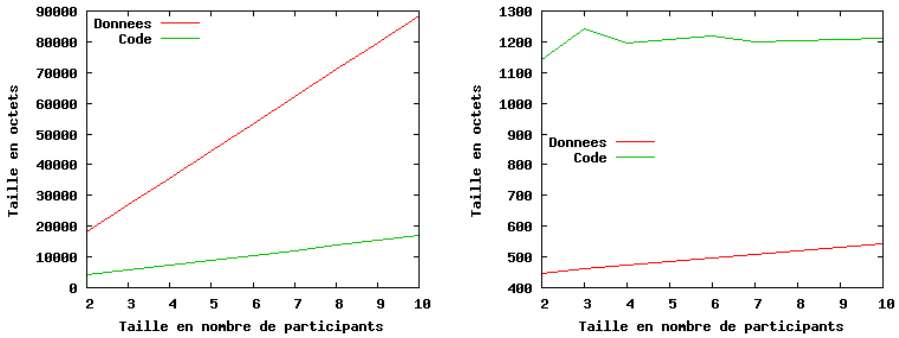
- C_D : coût global de Buzz ;
- C_{IA_i} : coût de l'intercepteur du composant actif i ;
- C_{ord} : coût de l'ordonnanceur ;
- $\mathrm{IFS_i}$: ensemble des interfaces serveurs et des interfaces clientes utilisées avec une liaison retardée pour le composant actif i ;
- CA : ensemble des composants actifs ;
- T_{pile} : taille d'une pile d'exécution ;
- T_{file_i} : taille d'une file d'attente pour l'interface. i.

Tous ces éléments dépendent de la plate-forme cible, les tailles des types des données n'étant pas constantes d'une architecture matérielle à l'autre (taille des pointeurs et des entiers, qui varient dans notre cas entre 16 et 32 bits). Ensuite, certains éléments dépendent de la configuration du système, comme la taille des piles ou des files d'attente. Certaines tailles sont directement fonctions de la complexité architecturale. Par exemple, la taille des éléments contenus dans une file d'attente dépend du type de l'interface associée à cette file ainsi qu'aux arguments des méthodes de l'interface. Une formule similaire peut être obtenue pour calculer le coût en taille de code.

Pour se convaincre de cette augmentation linéaire du coût associé à Buzz, nous avons simplement fait varier le nombre de participants au dîner et mesuré les tailles de code et des données des exécutables obtenus pour GNU/Linux x86. Les graphes de la figure 4.3 présentent ces résultats. La partie gauche 4.3a donne les évolutions des tailles de code et des données de tout ce qui est ajouté par le compilateur de Buzz alors que la partie de droite 4.3b ne donne que les évolutions liées à l'ordonnanceur. Les tailles de chacun des intercepteurs sont strictement identiques et ne sont donc pas présentées séparément (ces tailles sont prises en compte dans le cumul du graphe 4.3a). On remarquera que la taille du code de l'ordonnanceur est relativement constante, mais que de légères variations existent, sans doute provoquées par les optimisations du compilateur C.

Mesure des performances

Nous souhaitons mesurer l'impact sur les performances de l'utilisation de Buzz. Nous prenons un exemple simple constitué de deux composants geeks actifs que nous connectons avec une liaison synchrone plutôt qu'asynchrone comme précédemment. Nous nous plaçons ainsi dans le cas le plus défavorable en terme de sur-coût à l'exécution, ce type de liaison étant le plus complexe des types proposé dans Buzz. Une invocation synchrone est en effet découpée en six invocations dans l'architecture Think (voir la figure 4.4) et implique deux changement de contextes. Pour simplifier les mesures, nous avons utilisé un exécutable sur GNU/Linux, fonctionnant sur un processeur Intel Centrino cadencé à 1.7GHz. Nous souhaitons mesurer le rapport entre la durée totale d'une invocation sur le temps passé dans les composants ajouté par Buzz (intercepteurs et ordonnanceur). Ces temps étant relativement faible à l'échelle de la plate-forme utilisée, nous avons calculé la moyenne suite à 1'000'000 d'invocations. Malgré cela, nous avons jugé les mesures obtenues non significatives et par conséquent, nous ne les reportons pas ici. Celles-ci sont



(a) Coût global de Buzz (b) Coût de l'ordonnanceur

FIGURE 4.3 – Évolution du coût de Buzz avec la complexité de l'architecture (mesures faites sur l'exécutable pour GNU/Linux).

de l'ordre de la nanoseconde alors que la résolution (contrainte par le matériel et GNU/Linux) des outils de mesures est de l'ordre de la milliseconde. De plus, les optimisations de performance n'ont pas été étudiée dans nos travaux, ces performances n'ayant jamais été limitantes. Nous les avons jugées suffisantes lors de nos études. Nous avons cependant récolté des mesures nous permettant de comparer les temps passés dans chacun des composants ajoutés par le compilateur (les étiquettes font référence à la figure 4.4) :

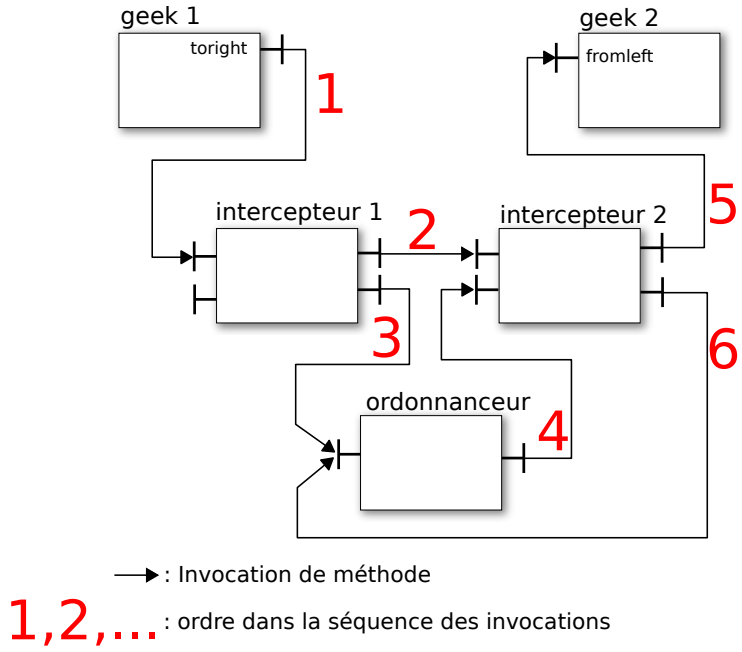
- intercepteur 1 : 12% (étiquette 1)
- intercepteur 2 : 23% (étiquettes 2 et 4)
- ordonnanceur : 65% (étiquettes 3 et 6)

Notre analyse montre qu'une grande partie du temps est passée à la protection des données partagées (file d'attente pour le stockage des invocations dans l'intercepteur 2 et structure interne à l'ordonnanceur) en particulier dans l'ordonnanceur. En effet, le code généré « sur-protège » ces données, parfois inutilement. Il serait possible de supprimer certaines de ces protections en repensant une partie du code généré par le compilateur. Ce sur-coût étant relativement faible, nous n'avons pas pousser plus en avant cette étude.

Conclusion

Que ce soit en terme de taille de code ou des données, l'exécutable généré par notre prototype respecte les contraintes matérielles de l'embarqué. Le coût de l'utilisation de Buzz reste raisonnable en laissant assez de ressources mémoire disponibles pour le code applicatif. De plus, ce coût est totalement et simplement prévisible et peut être remonté comme indicateur pour guider la phase de conception. Il est ainsi possible de prévoir avant de passer à l'implantation si la taille des données du système est compatible avec la plate-forme matérielle visée. Cette prévisibilité du coût lié au code généré s'inscrit parfaitement dans l'approche de Think visant à une maîtrise complète de la génération du code. Néanmoins, seule une application réelle peut confirmer cette conclusion. C'est pour cette raison que nous avons mené l'expérience décrite dans la section 4.2.

Les performances obtenues dès le début de nos travaux n'ont jamais été rédhibitoires pour toutes les études que nous avons menées. Nous n'avons donc pas apporté de soin particulier à l'optimisation de ces performances. Néanmoins, nous avons identifié plusieurs points qui pourraient être améliorés, comme les protections lors des accès concurrents aux structures de données internes à Buzz (files d'attente dans les intercepteurs et dans l'ordonnanceur) ou encore les nombreuses copies de données. Les protections utilisées verrouillent souvent des parties trop large



- 1 : invocation de la méthode `giveme()` par le geek 1
- 2 : transmission de l'invocation à l'intercepteur lié au geek 2
- 3 : blocage du geek 1 dans l'attente du retour de l'invocation
- 4 : exécution du geek 2 via l'exécution de son intercepteur
- 5 : exécution de la méthode `giveme()`
- 6 : passage du geek 2 dans l'état inactif et basculement de contexte pour reprise de l'exécution du geek 1 (exécution consistant dans les retours des invocations 3 et 1)

FIGURE 4.4 – Flot d'exécution lors d'une invocation de méthode synchrone entre deux composants actifs.

du code. Nous n'avons pas affiné ces verrous par souci de simplification des développement. Cet affinage augmenterait la complexité du code à générer et donc les risques d'introduction de défauts.

4.1.2 Évaluation et utilisation du modèle BIP

Nous présentons dans cette partie les résultats obtenus par analyse du modèle BIP généré par BUZZ. Dans un premier temps, le modèle BIP généré est non temporisé. Dans un deuxième temps, nous présentons une solution différente au problème du dîner, pour laquelle nous générons une version temporisée du modèle BIP.

La construction du modèle BIP se fait par la composition des modèles BIP contenus dans les composants BUZZ, à savoir : les composants actifs geeks et les composants interrupteurs. Comme mentionné dit en section 3.5.2, BUZZ ne fournit ni méthode ni outil pour faire correspondre le modèle BIP et le code C des composants. Nous faisons l'hypothèse de l'existence du modèle et du code C. En guise d'exemple, une partie du modèle BIP du composant geek est donnée dans la figure 4.5. Seul les débuts des méthodes `takeit()` sont illustrés.

Les modèles BIP contenus dans les composants interrupteurs modélisent les interactions entre l'environnement et le système. Il n'y a donc pas forcément un modèle unique. Le modèle le plus simple pour une interruption est de toujours la rendre possible, ce qui inclut tous les compor-

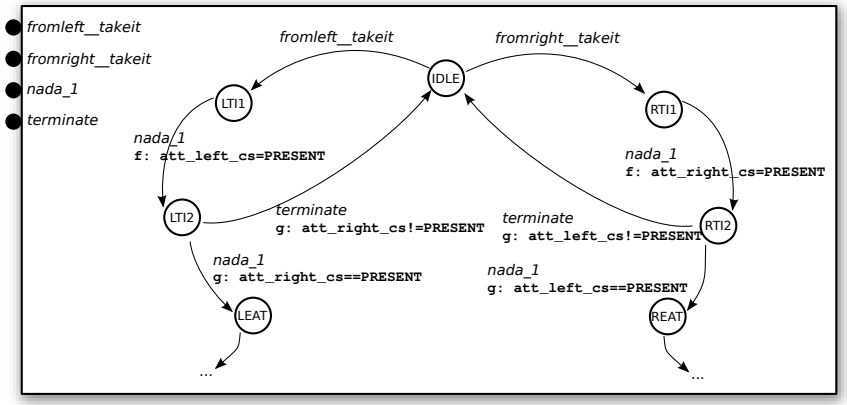


FIGURE 4.5 – Modèle BIP des méthodes de réception d'une baguette

tements possibles (abstraction maximale). Le problème de ce modèle simple est qu'il comprend aussi des comportements dégénérés qui posent problème lors de certaines analyses. Lorsqu'une interruption est systématiquement présente, le système va « boucler » entre la prise en compte de l'interruption et l'envoi d'un message au composant actif chargé de traiter cette interruption. Le reste du système n'ayant aucune chance de s'exécuter, les files d'attente débordent systématiquement. Il est nécessaire de choisir un modèle plus restreint ne faisant pas apparaître ces comportements tout en rendant compte des comportements du système. Une restriction efficace et simple est de fixer une fréquence d'apparition de ces interruptions, mais cela requiert une notion de temps, que nous ne prendrons pas en compte avant la seconde partie.

Modèle BIP non temporisé

Nous faisons le choix de limiter le nombre d'occurrences de chacune des interruptions à une valeur fixée. Le comportement global contiendra des suites dégénérées (où le système ne peut s'exécuter), mais elles seront en nombre fini, de taille finie et facilement identifiables. Les modèles BIP utilisés sont donnés dans la figure 4.6. Ces modèles sont uniquement donnés en exemple, la création de modèles BIP réalistes de l'environnement sort du cadre de ce travail.

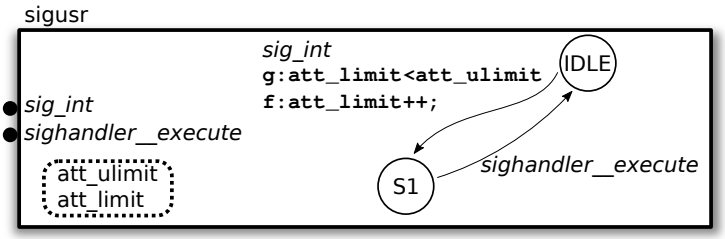


FIGURE 4.6 – Modèles BIP des interrupteurs.

Le modèle BIP créé par le compilateur (1930 lignes sans compter les commentaires) est composé de :

- 32 définitions de composants ;
- 81 règles de priorité ;
- 45 définitions de connecteurs ;
- 82 définitions d'états.

Le modèle ainsi créé ne possède pas d'évolution infinie, toutes les exécutions aboutissent à un état d'arrêt. La figure 4.7 présente une illustration d'une partie de l'espace des états parcourus. Des outils, présentés en annexe B, permettent par exemple de vérifier la propriété « Pour tous les états d'arrêts, tous les geeks ont mangé à leur faim ».

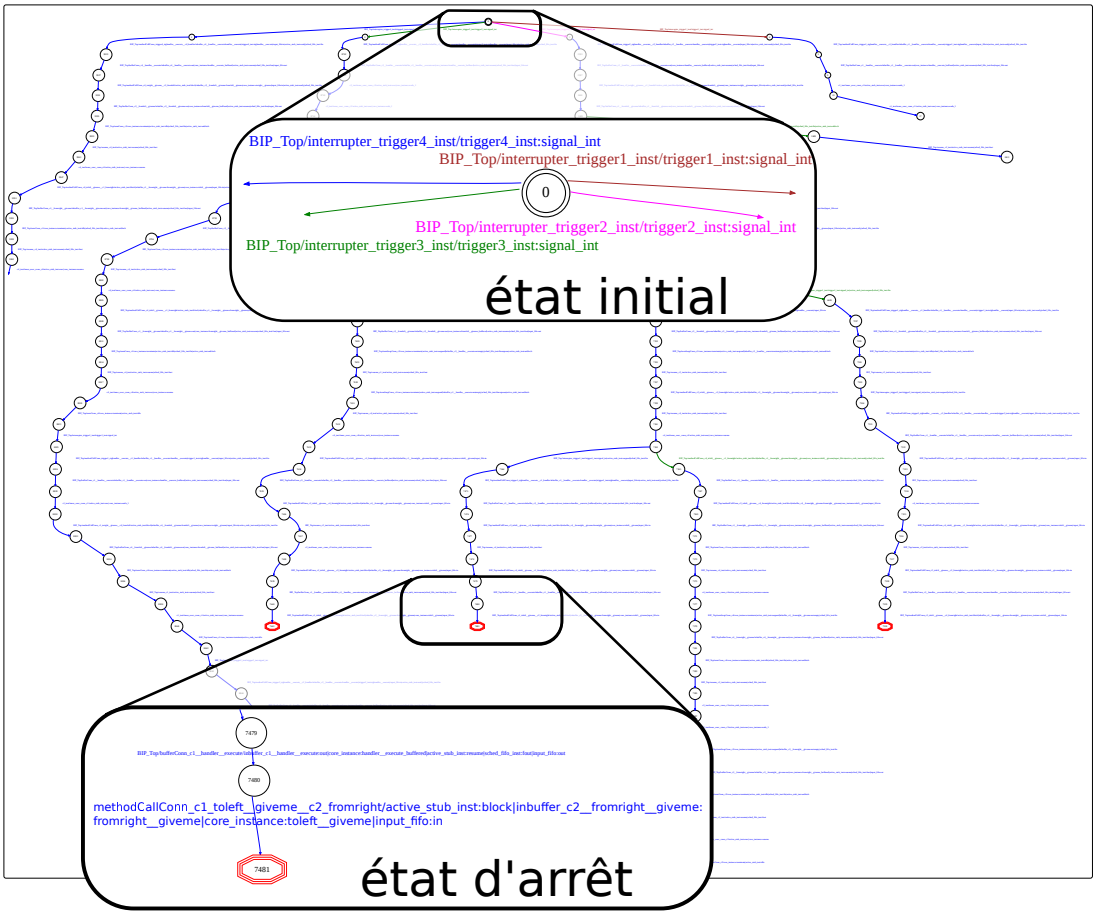


FIGURE 4.7 – Représentation graphique de chemins aboutissants à des blocages.

Le tableau 4.3 donne les tailles et les temps d'exploration des espaces d'états en fonction du nombre d'interruptions émises par chaque interrupteur, pour un dîner à trois participants. Les explorations ont été effectuées sur une machine équipée de processeurs Intel Xeon 3.20Ghz et de 4GiB de mémoire vive.

Très rapidement, cette exploration brutale n'est plus applicable, l'espace d'état demandant des ressources en mémoire trop importantes. Les techniques évoquées dans la section 2.4 doivent être envisagées.

Le modèle BIP généré permet d'autres analyses. Dans le cas précédent les files d'attente sont volontairement surdimensionnées pour éviter les débordements. Le tableau 4.4 illustre la manifestation de débordements lorsque nous réduisons cette taille dans le cas où une seule interruption est émise par chaque interrupteur. Nous voyons clairement que plus ces files sont petites, plus les débordements sont nombreux et plus les terminaisons normales sont rares. L'étude d'une trace menant à un débordement permet de se convaincre de l'existance de ce débordement. Par exemple lorsque les files d'attente sont de taille 3, il est possible d'arriver à un état où le geek 3 possède en attente à la fois les messages de transmission des baguettes ainsi

interruptions pour le premier geek	1	2	3	4
états	4556	29003	127994	*
temps d'exploration	<2s	18s	1m57s	*
états d'arrêt	22	44	70	*

* : exploration interrompue par manque de mémoire.

TABLE 4.3 – Tailles et temps d'exploration de l'espace d'état.

taille des files d'attente	1	2	3	4 et plus
états	1027	3701	4518	4556
débordements	93	41	3	0
états d'arrêt	2	18	22	22

TABLE 4.4 – Modification des tailles des files d'attente en entrée de composants actifs.

que les messages suivants de demande de récupération de ces baguettes (soit un total de quatre messages pour une file de taille 3).

Modèle BIP temporisé

Comme nous l'avons présenté dans la section 3.5.3, BUZZ ne prend en charge actuellement qu'une seule vérification automatique liée au temps : le non dépassement d'une limite sur le temps de blocage lors d'un appel synchrone. Or la solution du dîner que nous avons étudiée jusqu'ici n'utilise que des liaisons asynchrones. La vérification fournie par BUZZ n'est donc pas applicable. Nous présentons dans ce paragraphe une autre solution au dîner, cette fois utilisant des liaisons synchrones. De plus, par rapport à la solution précédente, celle-ci présente des situations où des interblocages sont possibles et permettra l'illustration de leur détection. Cette adaptation montre avec quelle facilité et avec très peu de modifications de l'architecture il est possible de porter un système asynchrone vers un système synchrone.

Approche synchrone du dîner. Nous prenons la solution suivante au problème du dîner :

- tous les geeks à table sont gauchers, sauf un qui est droitier.
- un geek gaucher qui a faim commence par saisir la baguette gauche, puis ensuite la droite. S'il lui manque une baguette, il en fait la demande à son voisin et se bloque en attendant d'obtenir la baguette avant de passer à la suite. Un droitier procède de manière identique mais dans l'ordre inverse (baguette droite en première, puis la gauche).
- au début du repas, tous les gauchers possèdent leurs baguettes gauches. Le droitier ne possède par contre aucune baguette (un des gauchers possède donc deux baguettes).

Le caractère « run-to-completion » des composants actifs impose qu'un geek qui décide de manger ne répondra aux requêtes qui lui sont envoyées qu'après avoir mangé. Le modèle BIP complet d'un composant geek gaucher est donné dans la figure 4.8.

L'architecture du système est identique à celle de la partie précédente présentée par la figure 4.2, à l'exception des liaisons entre geeks qui sont synchrones.

L'exploration de l'espace d'état met en évidence la présence de situations d'interblocages, où tous les geeks sont bloqués dans l'attente d'une réponse d'un de leur voisin. Il est facile de se convaincre de l'existence de tels blocages. Ils se produisent lorsqu'un geek possède une baguette

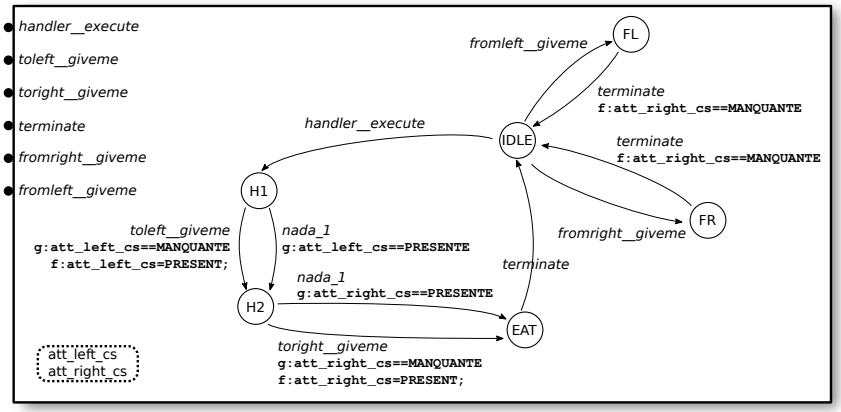


FIGURE 4.8 – Modèle BIP d'un geek gaucher.

nombre d'interruptions par interrupteur	1	2	3	4
taille de l'espace d'états	1091	63946	976393	6920891*
états d'arrêt	8	12	12	*
interblocages	0	65	804	*

* : exploration interrompue par manque de mémoire.

TABLE 4.5 – Détection d'interblocages dans le dîner à 3 geeks synchrones.

(la droite par exemple) et se bloque pour obtenir l'autre (la gauche). Dans ce cas, son voisin de droite n'aura pas accès à la baguette qu'ils se partagent. Si cette baguette restait disponible, alors il n'y aurait pas d'interblocage. Nous avons testé en mélangeant droitiers, gauchers, en changeant la distribution initiale des baguettes et le nombre d'interruptions pour chacun des interrupteurs. Nous avons constaté que certaines configurations sont sujettes à des interblocages et d'autres non. Le tableau 4.5 donne les résultats de l'évolution du nombre d'interruptions pour un dîner à trois geeks.

Le modèle temporisé est obtenu en annotant le modèle original et en modifiant le modèle des composants interrupteurs. Comme indiqué dans la section 3.5.3, cette prise en compte est sommaire. Les annotations sur le modèle BIP des composants geeks sont utilisées pour spécifier quels sont les temps d'exécution en meilleurs et pires cas, pour chacun des états des modèles des composants. Ces temps doivent être mesurés (par exemple par analyse du code compilé) ou estimés, ce qui sort du cadre de ce travail. Nous utilisons ici des temps totalement fictifs. Le listing 4.1 donne un extrait du modèle annoté. Nous nous plaçons dans un cas simple où l'automate passe exactement un cycle (unité de temps) dans chaque état.

```
...
behavior
 state IDLE
  on handler__execute to HANDLER_EXECUTE
  on fromleft__giveme to FROMLEFT_GIVEME

 @minmax=1/1
 state HANDLER_EXECUTE
  on toleft__giveme
```

```
    provided (att_left_cs == MANQUANTE)
    do att_left_cs = PRESENTE;
    to HANDLER_EXECUTE1

  on nada_1
    provided (att_left_cs == PRESENTE)
    to HANDLER_EXECUTE1

 @minmax=1/1
 state HANDLER_EXECUTE1
  on toright__giveme
    provided (att_right_cs == MANQUANTE)
    do att_right_cs = PRESENTE;
    to EAT

  on nada_1
    provided (att_right_cs == PRESENTE)
    to EAT
...
```

Listing 4.1 – Modèle BIP d'un geek annoté avec les valeurs minimales et maximales d'exécution

Enfin, les modèles des composants interrupteurs sont aussi modifiés pour prendre en compte le temps. La figure 4.9 présente le modèle d'un composant interrupteur utilisé sur GNU/Linux (de manière textuelle et graphique). L'interruption est levée périodiquement tous les `att_period` cycles (`period` étant un attribut du composant `sigusr` permettant de configurer la période). Le port `clockp` utilisé dans les modèles des interrupteurs est le port utilisé pour la synchronisation de tous les composants temporisés dans le système. Ne pas offrir de synchronisation sur ce port revient à bloquer le temps et donc rendre impossible l'exécution du reste du système. Seule la levée d'interruption est possible lorsque le temps est bloqué. C'est ce qui est utilisé dans le modèle présenté ici : quand la période est atteinte, le temps est gelé et ne pourra être dégelé qu'après avoir levé l'interruption.

```
behavior
 state IDLE
  on clockp
    provided (clock_value<att_period)
    do clock_value++;
    to IDLE
  on sig_int
    provided (clock_value==att_period)
    do clock_value=0;
    to SIGNAL
 state SIGNAL
  on sighandler__execute to IDLE
end
```

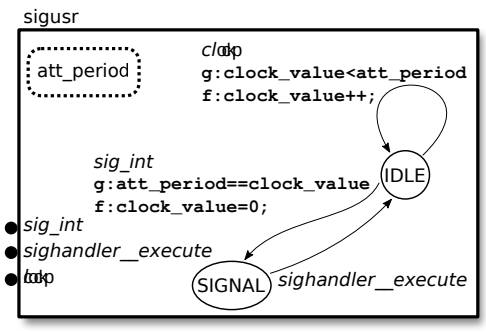


FIGURE 4.9 – Modèle BIP d'un composant interrupteur périodique.

périodes d'interruptions	5,5,5	50,50,50	40,50,60	75,50,50	75,50,125
taille de l'espace d'états	1059322	6900	14640	10562	9913
interblocages/débordements	119149	0	6	0	0

Les périodes d'arrivées d'interruptions sont données pour chaque interrupteur.

TABLE 4.6 – Évolution du nombre de blocages en fonction des fréquences d'interruptions, pour le dîner à trois geeks.

Temps de blocage accepté	50	18	17	15	10
dépassements	0	0	4	28	148

TABLE 4.7 – Vérification du temps de blocage maximum dans l'attente d'une baguette.

À partir de ce modèle temporisé, il est possible de vérifier si des interblocages sont possibles ou non. Le tableau 4.6 présente les résultats pour différentes valeurs de périodes. Il regroupe les états d'interblocages et les erreurs dues à des débordements de files d'attente. On constate que la configuration des périodes d'arrivées des interruptions est déterminante de la présence ou non d'interblocages.

À partir de ce modèle, il est possible de vérifier des propriétés liées au temps. Le compilateur de BUZZ ne supporte de manière automatique que la vérification du non dépassement d'un temps maximum d'attente de retour d'appel synchrone. La spécification d'une telle contrainte se fait par annotation comme présenté dans la section 3.5.3. Le listing 4.2 donne un extrait du modèle précédent enrichi de l'expression de la contrainte : « l'appel synchrone de la méthode `giveme()` via l'interface `toleft` doit retourner au plus après 30 cycles ».

```
@minmax=1/1
@toleft__giveme=30
state HANDLER_EXECUTE1
  on toleft__giveme
   provided (att_left_cs == MANQUANTE)
    ...
```

Listing 4.2 – Modèle BIP enrichi avec la spécification d'une contrainte sur le temps de blocage.

L'analyse du modèle BIP généré par le compilateur permet de vérifier que cette contrainte est bien respectée dans le système. Le tableau 4.7 donne quelques valeurs en exemple. Nous utilisons pour ces mesures des périodes d'arrivées d'interruptions de 50 cycles. Nous avons vérifié que les différentes invocations aux méthodes `giveme()` mettent au plus 18 cycles à répondre. En faisant descendre cette limite, on constate un nombre grandissant d'erreurs.

À partir du modèle généré, il est possible, comme dans le cas du modèle non temporisé, d'écrire des observateurs chargés de vérifier d'autres propriétés, comme les temps de réponse (pas uniquement pour les appels synchrones).

Conclusion

Nous avons vu dans cette section que le modèle BIP généré par BUZZ permet d'obtenir des résultats d'analyse pertinents. L'ensemble des propriétés auxquelles nous nous sommes intéressés (débordement de file d'attente, interblocage et temps de blocage lors d'un appel synchrone) n'est pas exhaustif et nous a seulement permis de vérifier la faisabilité de l'outil.

Les limitations que nous avons mis en évidence dans cette partie n'indiquent pas de problème dans la méthode que nous avons utilisée mais font seulement ressortir des choix simplificateurs pour l'implantation des outils. Comme nous l'avions déjà évoqué dans la section 3.5.3, la création d'un modèle BIP temporisé ainsi que son exploitation sont volontairement restreints. Dans cet exemple, nous avons seulement employé un outil d'exploration de l'intégralité de l'espace des états, outil qui possède des limitations connues (rapidement contraint par l'espace mémoire utilisé). Lors de la réalisation de ces tests, d'autres outils et techniques d'analyse étaient en développement mais pas encore applicables à nos travaux. Nous nous sommes concentrés sur la création correcte par construction d'un modèle BIP et non de son exploitation, qui est un travail à part entière.

Ces limitations sont donc uniquement liées au caractère de prototype de nos outils, et non à la démarche.

4.2 Ré-ingénierie d'une application de routage pour réseau de capteurs

4.2.1 Introduction

Nous présentons dans cette section la ré-ingénierie d'une application logicielle développée de manière classique avec THINK. Le but de cette ré-ingénierie est d'utiliser les outils et concepts de BUZZ pour permettre ensuite la comparaison entre le prototype original et le prototype issu de BUZZ. En particulier, nous pourrons constater un gain net lors de la phase de conception en terme de facilité d'expression. Cela grâce aux concepts de BUZZ qui sont d'un niveau plus élevé que ceux utilisés dans l'application d'origine qui utilise THINK.

Cette application logicielle appartient au domaine des réseaux de capteurs sans-fils (WSN, « Wireless Sensors Network »). Un WSN est constitué d'un ensemble de nœuds (aussi appelés *capteurs*) communiquant entre eux par des connexions radios. Le plus souvent, ces nœuds sont chargés de mesurer des valeurs (température, pression, etc) et de les transmettre à une autre entité pour traitement (stockage, calcul de statistique, facteur de prise de décision,...). Chacun de ces nœuds est constitué d'une plate-forme matérielle sur laquelle s'exécute un logiciel qui assure les fonctionnalités de mesures et de communications. Le déploiement d'un WSN consiste en la dispersion des nœuds sur une zone géographique, par exemple dans une forêt en les larguant depuis un avion, dans un entrepôt en les fixant aux murs ou encore dans le béton d'un ouvrage.

Dans tous les cas, les nœuds sont dépourvus d'accès à des ressources énergétiques permanentes. Ils doivent tout de même avoir une autonomie de plusieurs semaines au minimum, jusqu'à plusieurs années. Leurs accessibilités n'est souvent pas garantie et il n'est pas question de redéployer trop régulièrement de nouveaux nœuds. La consommation énergétique est donc un point crucial à optimiser dans le développement des nœuds. Cela comprend bien sûr le matériel qui doit être étudié pour consommer aussi peu que possible (*e.g.* micro-contrôleur et puce radio à faible consommation), ce qui se traduit par des ressources en calcul et en mémoire très faibles. Le logiciel doit être optimisé pour ces faibles ressources matérielles. Les protocoles utilisés dans les communications doivent à leur tour être taillés en fonction de ces contraintes (*e.g.* en minimisant la quantité des informations transmises).

La fiabilité du logiciel est aussi un point important. Dans le cas général, il n'est simplement pas possible de mettre à jour le logiciel qui s'exécute sur les nœuds. Un défaut peut rendre le WSN totalement inutilisable et demander un redéploiement complet après avoir corrigé le logiciel. Lorsque les mises à jour sont possibles, elles peuvent s'avérer très coûteuses, celles-ci

devant être diffusées à tous les nœuds du WSN :

- coût en mémoire, chaque nœud doit être capable de télécharger les correctifs. Ces correctifs peuvent se présenter sous différentes formes (actions de reconfiguration, *patchs* binaires, logiciel complet de remplacement) et utiliser une part importante de la mémoire disponible sur la plate-forme.
- coût en énergie pour la transmission radio et l'application de ces correctifs. Les communications étant de loin la fonction la plus gourmande en énergie sur ce type de plate-forme.

Pendant leurs vies, les WSN peuvent avoir une topologie changeante : les nœuds disparaissent (panne matérielle, épuisement des ressources énergiques, défaut logiciel, ...), apparaissent (déploiement de nouveaux nœuds) ou se déplacent (nœuds attachés à des véhicules, des animaux).

Les algorithmes de routages classiques ne permettent pas de répondre efficacement aux contraintes associées aux nœuds d'un WSN (consommation énergétique, faible capacité de calcul et de stockage) et s'adressent en général à des topologies fixes (réseaux en étoile, en anneau, ...).

C'est dans ce contexte que s'inscrivent les travaux de Thomas Watteyne qui ont abouti à la mise au point des protocoles de communication suivants. Un protocole pour le contrôle d'accès au support, « 1-hopMAC » [WBD+06], et un algorithme de routage utilisant des coordonnées virtuelles [WDABB08]. Nous ne présentons pas ces protocoles ici et plus de détails sont disponibles dans [Wat08, WBDA08]. Ces protocoles définissent deux classes pour l'ensemble des nœuds :

- les nœuds basiques, ou « nœuds capteurs ». Ils échantillonnent et transmettent régulièrement des mesures (température, luminosité, pression). Ces nœuds servent aussi de relais pour le routage des paquets au sein du réseau.
- les « nœuds puits » récoltent les données remontées par les nœuds capteurs. Ces puits sont reliés à une machine permettant le traitement des données (stockage par exemple).

Un vue globale du réseau est présentée dans la figure 4.10.

Un prototype logiciel mettant en œuvre cette pile protocolaire a été développé par Thomas Watteyne. Il utilise pour chacun des nœuds la plate-forme matérielle WSN430, présentée précédemment dans la section 4.1. Nous présentons en section 4.2.2 ce prototype logiciel tel qu'il a été utilisé dans les travaux d'origine. Ensuite, la ré-ingénierie vers Buzz et ses résultats sont détaillés en section 4.2.3.

4.2.2 Présentation du prototype logiciel original

L'environnement Think a été utilisé pour le développement du prototype original. Nous n'aborderons que les aspects qui rentrent en compte dans nos travaux, c'est-à-dire les aspects de génie logiciel.

L'architecture en composants du logiciel est donnée dans la figure 4.11. Elle se décompose en quatre couches[20], chacune réalisée par un composant :

- le composant `appli` pour la couche *application*. Il est chargé de l'échantillonnage des mesures de température et de luminosité. Régulièrement, il provoque l'envoi de données vers un puits.
- les composants `net` et `link` pour les couches *réseau* et *liaison*. Ces couches se placent entre les couches *application* et *physique*. Elles se chargent de l'implantation de l'algorithme de routage.
- le composant `phy` pour la couche physique. Il est chargé du pilotage du matériel radio.

20. Cette notion de couche et d'empilement de protocoles de communication est classique dans les réseaux informatiques et est définie par l'OSI (« Open Systems Interconnection »).

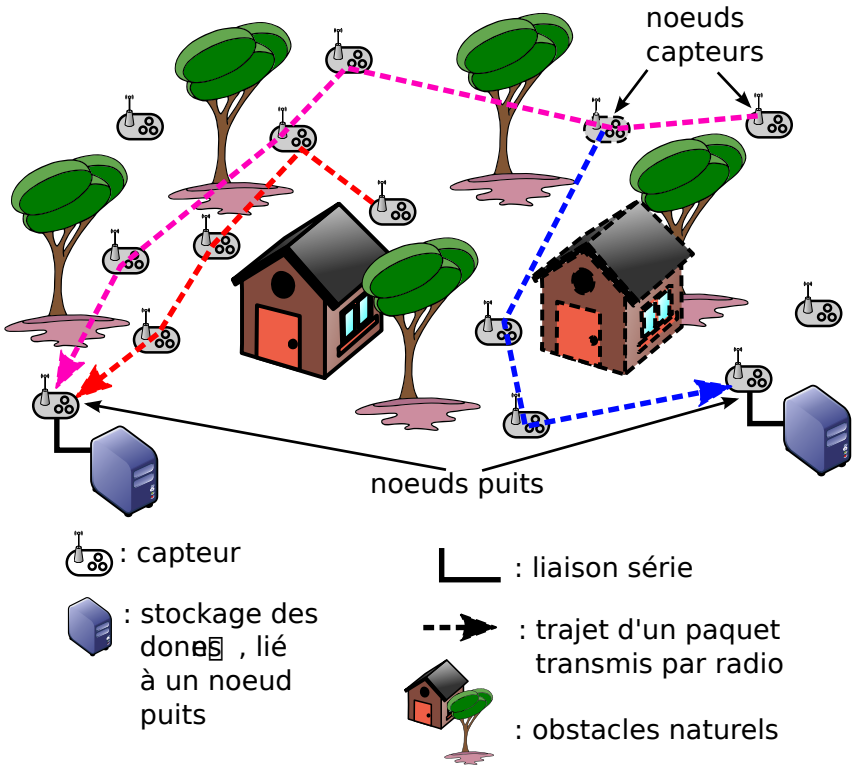


FIGURE 4.10 – Un exemple de réseau de capteurs.

Le composant nommé `thread` sur la figure 4.11 est en réalité un ordonnanceur coopératif à tourniquet qui à tour de rôle, invoque systématiquement les composants `appli`, `link` et `net`, que ceux-ci aient du travail en attente ou non (par exemple, l'application peut être invoquée par le composant `thread` alors qu'elle est en attente d'une mesure : elle ne peut rien faire). Les quatre composants principaux (`appli`, `link`, `net` et `phy`) sont liés entre eux et communiquent de manière synchrone (mécanisme d'appel de fonction classique). Ces quatre composants sont aussi pilotés par le mécanisme d'interruption :

- des « timers » provoquent l'échantillonnage des mesures pour l'application, du canal radio pour la couche liaison, ainsi que l'envoi de données par l'application.
- le pilotage de la radio et des capteurs.

Ce mécanisme d'interruption est une source de parallélisme à l'exécution, qui demande de protéger et de synchroniser certains accès. Pour cela, des files d'attente sont créées, et pendant leur accès, les interruptions sont masquées pour éviter les accès concurrents.

Ainsi, les quatre composants principaux peuvent être exécutés depuis trois sources différentes : par l'ordonnanceur, par un autre composant lors d'une invocation de méthode et par le mécanisme d'interruption.

Cet entrelacement des exécutions, ajouté à l'utilisation des mécanisme d'intercessions de FRACTAL (certaines liaisons sont modifiées en fonction de l'état dans lequel se trouve le système) rend la compréhension et la maintenance du système difficile. Son bon fonctionnement dépend de l'enchaînement correct des diverses exécutions, ce qui en l'état, est difficilement vérifiable et rend l'identification et la correction de défaut difficile. Typiquement, le système peut se bloquer dans l'attente d'un événement qui n'arrive jamais. Pour se prémunir des blocages, un mécanisme matériel de « watchdog » est utilisé : un compte à rebours spécial redémarre le système lorsqu'il

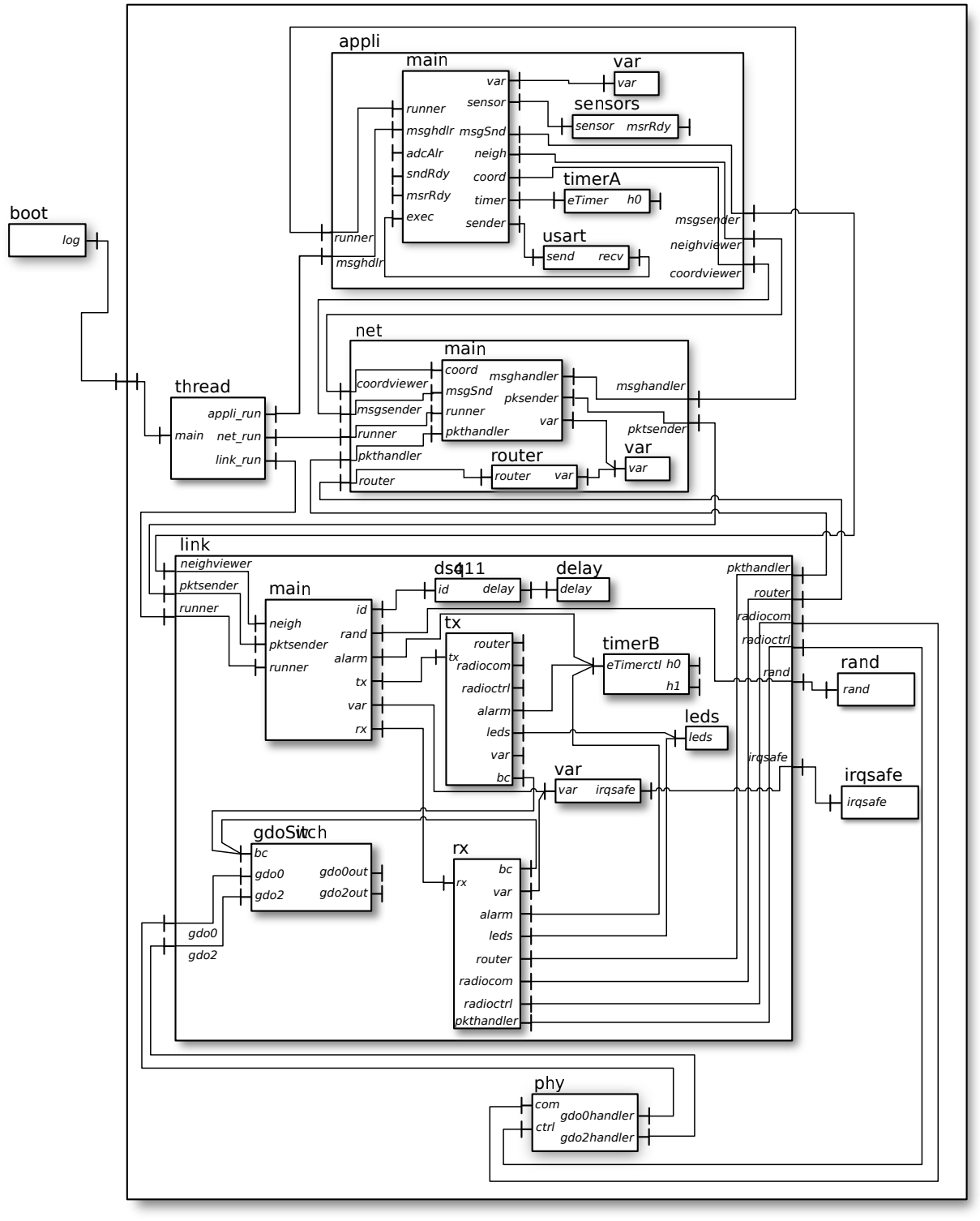


FIGURE 4.11 – Architecture du prototype original utilisant THINK.

expire. Le système doit donc réinitialiser suffisamment souvent ce compte à rebours, preuve de son activité. Ce mécanisme permet de retrouver un fonctionnement normal après une faute, mais il n'est pas une solution au problème initial.

Résultats. Ce prototype a rempli correctement sa fonction, à savoir implanter la pile protocolaire et démontrer son fonctionnement. Ces résultats sont disponibles en ligne [21]. Néanmoins, d'un point de vue logiciel, ce système est difficile à maintenir. L'ADL ne documente qu'une partie du système : l'architecture à l'initialisation, qui sera modifiée pendant l'exécution. L'intégralité des éléments caractérisant le comportement à l'exécution est enfouie dans le code fonctionnel des composants. Chacune des interruptions (dont les sources sont le matériel radio, les « timers » et les capteurs) peut provoquer une combinaison des éléments suivants :

- exécution de un ou plusieurs composants de l'architecture (potentiellement en concurrence avec le fil d'exécution *normal* du système) ;
- configuration des sources d'interruptions :
  - réarmement ou annulation de « timers » ;
  - demande de mesure à un capteur ;
  - interactions avec le matériel radio.
- modification de l'architecture.

Ces réactions dépendent implicitement (par des conditions dans le code fonctionnel) de l'état du système. Par exemple, une interruption ne sera pas systématiquement traitée par le même composant.

Pour résumer, les problèmes suivants ont été identifiés dans ce prototype logiciel :

- difficulté de localiser le code exécuté par les différentes activités du système ;
- synchronisation et sérialisation manuelles des appels entre les activités ;
- l'ordonnancement des composants est simpliste et parfois provoque des exécutions qui pourraient être évitées (gaspillage d'énergie et de temps d'exécution).

4.2.3 Présentation de la ré-ingénieurie du prototype en utilisant Buzz

Cette section présente la ré-ingénieurie du prototype initial en utilisant Buzz. L'objectif est de corriger les défauts relevés dans la section précédente. Cette ré-ingénieurie comprend quatre étapes, qui sont détaillées dans la section suivante. La première consiste en la suppression de la hiérarchie dans l'architecture, Buzz ne supportant que les architectures plates. Cette étape est rendue nécessaire par les limitations imposées par le prototype de notre outil et non par la démarche, comme nous l'avons indiqué précédemment en section 3.2. La deuxième étape consiste en l'identification des composants actifs, passifs et interrupteurs. Nous verrons qu'après cette deuxième étape, une troisième étape de suppression, fusion et/ou découpage de certains composants est nécessaire. Enfin, toutes les liaisons doivent être caractérisées avec leur mode de communication (synchrone, asynchrone, retardé). Nous présentons ensuite les résultats liés aux phases de conception, d'implantation et d'analyse. Enfin, nous concluons cet exemple.

Aplanissement. Le prototype original ne possédant pas de composants instanciés plusieurs fois, l'aplanissement peut se faire simplement. Il suffit de retirer les membranes des composants composites et de préfixer les noms des sous-composants par le nom du composant composite qui disparaît. Cette technique est illustrée par la figure 4.12 pour une partie du composant `appli`.

Composants actifs, passifs et interrupteurs. Cette étape consiste à identifier quels sont les composants actifs, passifs et interrupteurs. Le découpage en composants composites du prototype initial conduit naturellement à rendre actifs les trois composants `appli`, `net` et `link`. Ceux-ci n'existant plus car ils ont été « éclatés » par l'étape d'aplanissement, ce sont leurs anciens

21. Page internet du projet « Sense and Sensitivity » : http://senseandsensitivity.rd.francetelecom.com/

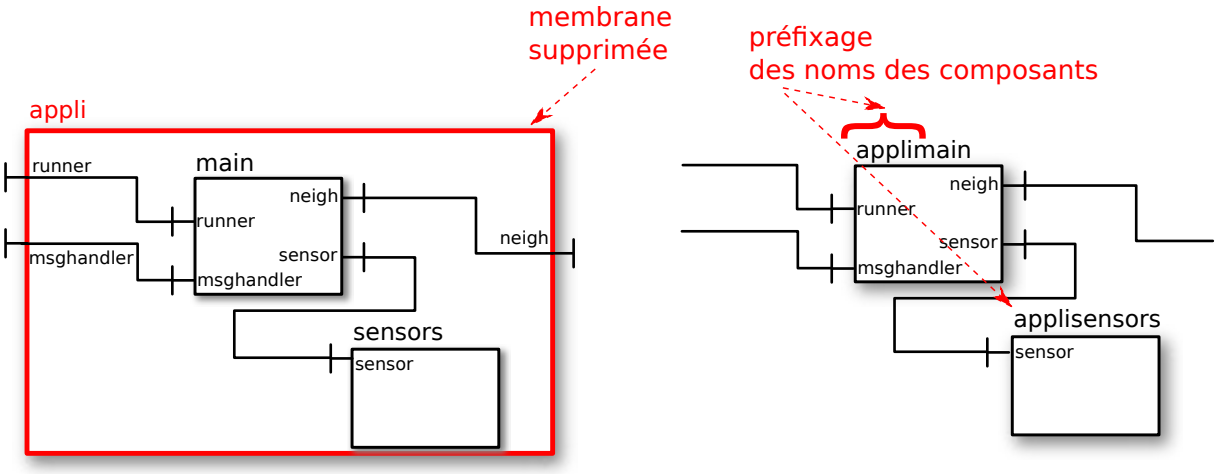


FIGURE 4.12 – Aplanissement de la hiérarchie.

sous-composants appelés `main` qui sont actifs, c'est-à-dire, après les changements de noms de l'étape précédente : `applimain`, `linkmain` et `netmain`. Les composants du prototype initial qui sont des traitants d'interruption (il est nécessaire de lire le code fonctionnel pour extraire cette information du prototype initial) deviennent des composants interrupteurs :

- `applitimerA` : « timer » utilisé par l'application pour déclencher les échantillonages des capteurs ou l'envoi de données.
- `applisensors` : capteurs levant une interruption quand une mesure précédemment réclamée est prête.
- `linktimerB` : « timer » utilisé par la couche liaison.
- `physpi` et `linkgdoSwitch` : sources d'interruptions liées au matériel radio.

Les composants restants sont des composants passifs.

Suppression, fusion et découpage de composants. L'utilisation de BUZZ permet de supprimer le composant `thread` qui gérait l'entrelacement des activités manuellement. Cette gestion est assurée par l'ordonnanceur synthétisé par BUZZ.

Dans le prototype initial, les interruptions peuvent provoquer l'exécution de n'importe quel composant. BUZZ interdit ce comportement et oblige les composants interrupteurs à n'invoquer que des composants actifs.

Pour respecter cette règle, des interfaces sont ajoutées à plusieurs composants actifs, parfois en fusionnant plusieurs composants ensemble. La figure 4.13 illustre le cas du composant `linkmain`, qui fusionne avec les composants `linkrx` et `linktx`. Ces deux composants étaient en grande parties invoqués par le système d'interruption (via le composant `linkgdoSwitch`). Ces invocations se faisaient via des liaisons créées dynamiquement ; elles sont maintenant portées par des liaisons asynchrones explicites.

Liaisons. Toutes les communications entre les composants (actifs et passifs) sont synchrones. Ce choix a été pris pour ne pas avoir à modifier le code fonctionnel des composants, écrit dans un cadre synchrone. Seules les liaisons entre composants interrupteurs et composants actifs sont asynchrones (cela est imposé par le prototype de BUZZ).

Comme la modification de l'architecture à l'exécution n'est pas permise par BUZZ, toutes les liaisons doivent être présentes dans l'ADL (*i.e.* au déploiement). Cela permet une meilleure

structuration en faisant apparaître explicitement toutes les possibilités qui étaient présentes implicitement dans le code du prototype initial (du fait de sa dynamique). La figure 4.14 illustre cette modification.

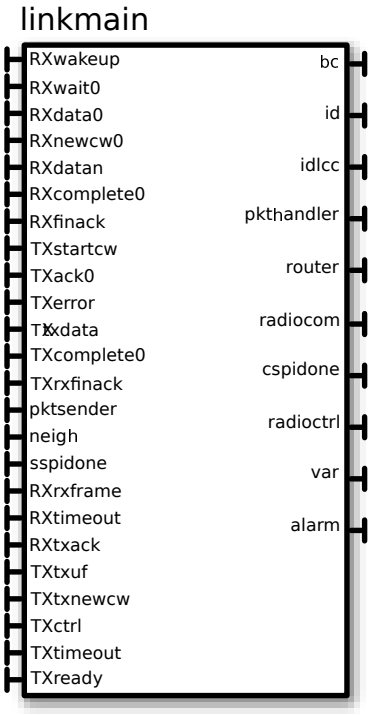


FIGURE 4.13 – Composant `linkmain`.

4.2.4 Résultats de conception

L'architecture du système, illustrée dans la figure 4.14, peut être considérée comme un document de conception. Contrairement à la description de l'architecture du prototype original, l'architecture BUZZ rend compte de la dynamique du système, ce qui en fait un document essentiel. L'utilisation de BUZZ permet de répondre aux problèmes soulignés dans la présentation du prototype initial :

- Les activités présentes dans le système ressortent directement : il possède exactement trois activités concurrentes et plusieurs sources d'interruptions qui invoquent ces trois activités via des liaisons explicites ;
- La gestion des communications (synchronisation et sérialisation des invocations de méthodes) entre les composants actifs est gérée automatiquement par BUZZ ;
- L'ordonnancement est plus riche. En particulier, l'état d'inactivité du système où aucun composant actif n'est prêt à s'exécuter est facilement détectable. Cela permet d'exploiter les modes de faible consommation offerts par le matériel.

4.2.5 Résultats de génération de code

Le tableau 4.8 donne une comparaison entre les exécutables obtenus à partir du prototype initial et à partir du prototype issu de BUZZ. Ce tableau fait apparaître très clairement une augmentation de la taille du code de chacune des couches, sauf pour la couche physique. Cette

	Prototype initial	Prototype Buzz	rapport	
application	3438	4904	+42%	code
réseau	3196	5626	+76%	
lien	8428	12723	+50%	
physique	4838	2226	-53%	
ordonnanceur	274	852	+210%	
Total	25940	33222	+28%	
application	1064	982	-7%	données
réseau	2184	490	-77%	
lien	3166	1404	-55%	
physique	554	42	-92%	
ordonnanceur	12	58	+383%	
Total	7404	4092	-44%	

Table 4.8 – Comparaison des codes du prototype initial et du prototype issu de Buzz pour chacune des couches (les tailles sont données en octets).

augmentation est due aux intercepteurs qui embarquent une complexité qui n'était pas présente dans le prototype initial, chacun pesant en moyenne 4000 octets. Ce poids, relativement important en comparaison à la taille du code initial, est compensé par des optimisations plus agressives possibles grâce à l'architecture intégralement statique générée par Buzz (la taille finale n'est pas simplement l'addition de la taille des intercepteurs au code initial). L'ordonnanceur est relativement plus lourd avec Buzz, ce qui s'explique par la plus grande richesse offerte (l'ordonnanceur initial est une simple boucle d'une dizaine de lignes). Dans l'absolu, cette taille reste faible.

La taille des données est plus faible dans la version issue de Buzz. Les raisons sont multiples :

- Buzz ne permet pas l'utilisation de hiérarchie de composants (économie de certaines structures de données).
- certains composants ont été fusionnés (en particulier les composants liés à la couche liaison).
- les tailles des files d'attente peuvent être plus facilement modifiées et ont été réduites.
- des optimisations agressives sont employées par le compilateur (évoquées plus haut).

4.2.6 Résultats de génération de modèle BIP

Pour la génération d'un modèle BIP, nous devons dans un premier temps créer la partie BIP du contenu des composants Buzz. Contrairement à la phase d'implantation qui réutilise le code C du prototype initial, le contenu BIP des composants doit être entièrement écrit pour cet exemple. Nous présentons cette création en deux parties. La première concerne les composants actifs et passifs, la seconde les composants interrupteurs. Pour simplifier ces tâches, nous utilisons des modèles BIP non temporisés.

Les contenus BIP des **composants actifs et passifs** sont obtenus à partir du code C du prototype initial. Nous faisons le choix de maintenir les modèles BIP associés aussi simples que possible, c'est-à-dire à un haut niveau d'abstraction. En effet notre démarche n'est pas dépendante du niveau de détails des modèles BIP. Le compilateur Buzz peut donc être utilisé pour générer à la fois des modèles à grains fins et à gros grains. Les automates créés rendent compte des

interactions avec les autres composants au travers des invocations de méthodes. Les séquences d'invocations sont naturellement modélisées par plusieurs transitions successives. Lorsque le code C utilise des structures de contrôle du flot d'exécution (boucles ou conditions), l'automate que nous construisons est non déterministe. L'automate ainsi créé possède un comportement plus large que celui qui sera observé lors de l'exécution de l'implantation du système. Ce point a été abordé dans l'introduction qui traite des limitations des analyses liées au niveau d'abstraction utilisé dans les modèles. La figure 4.15 présente un extrait du code C du composant Buzz `applimain` ainsi que l'intégralité de l'automate de son modèle BIP. Nous construisons des modèles BIP similaires pour l'ensemble des composants Buzz actifs et passifs.

Les contenus des **composants interrupteurs** sont obtenus différemment. Comme nous l'avons vu en section 3.5.3, les modèles BIP de ces composants n'ont pas de correspondance avec du code C mais modélisent l'environnement dans lequel le système s'exécute. Dans cet exemple, l'environnement est complexe et sa modélisation est une tâche difficile. Un nœud d'un réseau de capteur peut recevoir des signaux radio de plusieurs autres nœuds, avec des qualités de réception différentes. Ces différents signaux peuvent entrer en collision ou même ne pas suivre le protocole prévu (en cas de défaut logiciel ou de partage du canal radio avec un autre WSN par exemple). Nous avons décidé dans le cadre de cet exemple de nous limiter à un modèle primitif de l'environnement, similaire à ce que nous avons décrit pour l'exemple précédent en section 4.1.2. C'est-à-dire que notre modèle se place à un niveau d'abstraction élevé et possède un comportement plus général que le comportement observé de l'environnement. Ceci est particulièrement vrai pour les modèles BIP des composants interrupteurs `linkgdoswitch` et `physpi` liés au matériel radio. Les modèles pour les composants `applitimerA` et `linktimerB` sont plus simples à obtenir : une interruption ne peut subvenir qu'après la mise en place d'un compte à rebours. Il en va de même pour le composant `applisensors` qui ne lève une interruption qu'après la requête d'une mesure. Ce composant est illustré sur la figure 4.16. Une interruption, modélisée par le port `ADC12_int` ne peut avoir lieu que si la donnée `sense` a été préalablement positionnée à 1 par l'invocation de la méthode `sense`, modélisée par le port `sense__sense`.

La génération du modèle BIP produit 6500 lignes (sans les commentaires) et possède 61 définitions de composants BIP. 42 définitions de composants sont instanciées par les 3 composants composites des composants actifs :

- 36 composants BIP FIFO spécifiques aux méthodes serveurs et 3 (un par composite BIP) FIFO généraux. Ces composants sont présentés en section 3.5.2.
- 3 composants guides.

5 définitions de composants sont utilisées par les composants passifs, 6 définitions sont utilisées par les composants interrupteurs et une définition de composant est utilisée par l'ordonnanceur. Enfin, une définition est utilisée pour le composant composite de plus haut niveau.

Avec le modèle BIP créé par Buzz pour ce système, nous sommes capables d'utiliser les outils de simulation fournis avec BIP. Néanmoins, pour les raisons évoquées précédemment au sujet de la prise en compte de l'environnement, la simulation doit être assistée par l'utilisateur. À chaque pas de la simulation, l'utilisateur doit décider si le système suit une exécution normale ou si une interruption survient.

Pour pouvoir utiliser des techniques d'analyse de modèles BIP plus poussées comme le « model checking » ou des techniques spécifiques à la détection d'interblocages, il est nécessaire d'enrichir les modèles BIP des composants interrupteurs de façon à obtenir une modélisation de l'environnement plus précise. Il est aussi possible d'abstraire certaines parties du modèle. Par exemple, les modèles BIP liés à la couche physique du système peuvent être modifiés pour abstraire les signaux radio bas niveau et ne gérer que la notion de paquet. Les modèles nécessaires sont alors beaucoup moins complexes à construire. Il n'est évidement plus possible d'analyser

les échanges radios (*e.g.* les collisions).

4.2.7 Conclusion

Cet exemple de ré-ingénieurie nous a permis de montrer la modélisation d'un système embarqué réel avec Buzz. La première partie permet de mettre en vis-à-vis le modèle de conception obtenu en utilisant des concepts classiques avec Think et le modèle obtenu en utilisant Buzz. Cette comparaison illustre certains points abordés dans le premier chapitre de ce manuscrit en section 1.2. Les concepts de composants actifs, passifs et interrupteurs simplifient le modèle que doit fournir le développeur. Nous constatons aussi que la simple lecture du modèle Buzz fournit plus d'informations que la lecture du modèle Think original. Les activités sont directement identifiables. L'utilisation de liaisons dont le mode de fonctionnement est explicite dans le modèle permet d'alléger ce dernier : certains composants dans le prototype original n'ont comme seul rôle que celui de réaliser des communication asynchrones (*e.g.* sous-composant `var` du composant `appli`).

L'étude de l'implantation obtenue grâce à Buzz nous a permis de mettre en évidence que le code généré est adapté aux contraintes liées aux plate-formes embarquées que nous visons. Le coût de Buzz que nous avons mesuré sur cet exemple est tout à fait acceptable, que ce soit en terme d'empreinte mémoire ou de performance à l'exécution.

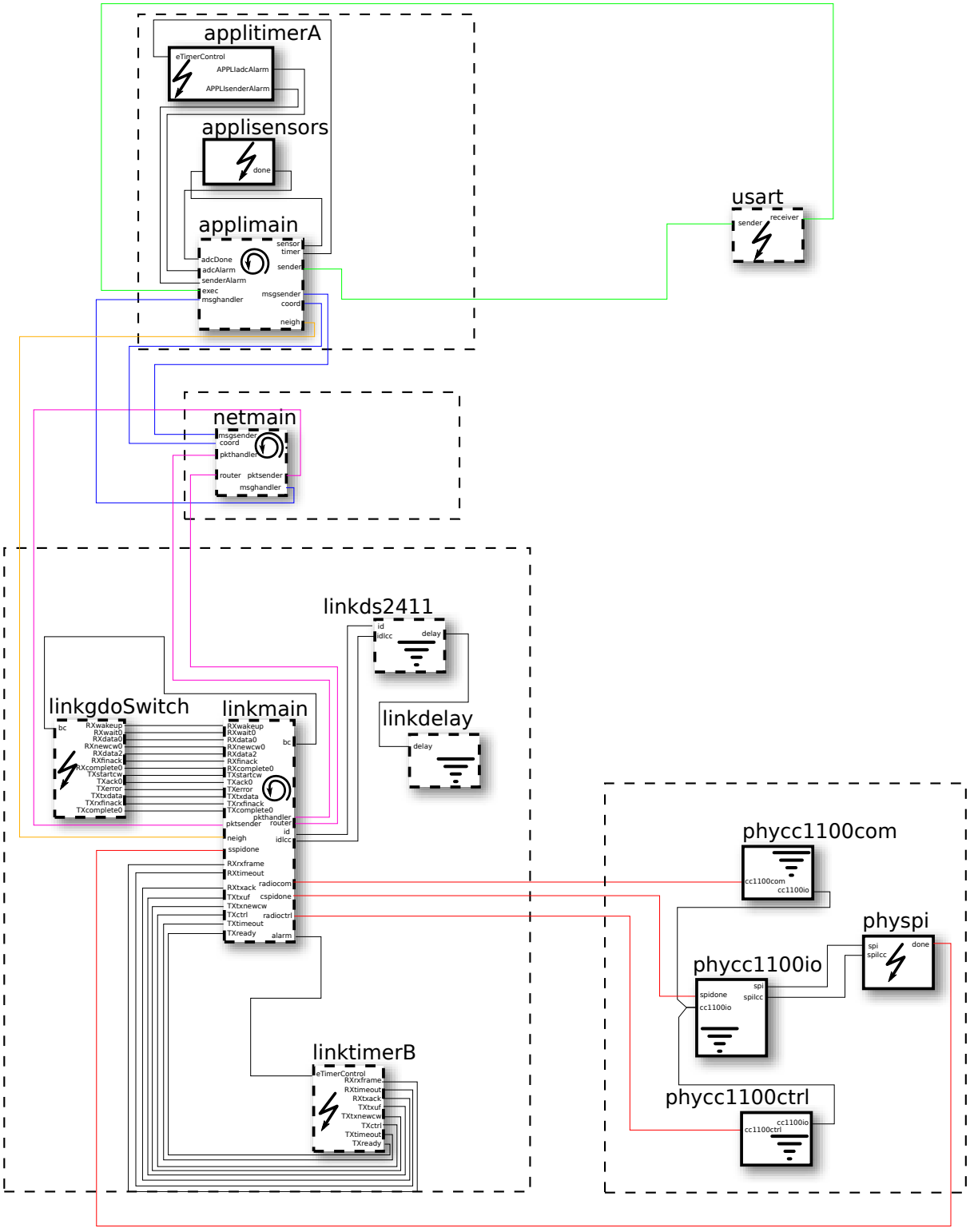
Enfin, nous avons pu vérifier que la création d'un modèle BIP pour un système complet est concluante. Dans cet exemple, nous avons généré un modèle BIP relativement simple, que ce soit au niveau de la modélisation du comportement des composants ou au niveau de la modélisation de l'environnement. Malgré cette relative simplicité, nous pouvons utiliser les outils de simulation de BIP. Si nous avions voulu pousser l'expérience plus loin avec des analyses plus complexes, nous aurions dû procéder différemment. En effet, nous nous sommes basés sur un prototype logiciel existant, pour lequel la documentation et le support était minimum. Nous avons décidé de modifier au minimum le code C de ce prototype durant la ré-ingénieurie, ce qui nous a forcé à conserver certains choix dans la structure des composants parfois inadaptés. Le prototype initial possède aussi certaines lacunes souvent rencontrées dans ce genre de logiciel (prototype de démonstration) et n'a pas été développé en suivant des méthodes de génie logiciel (interactions implicites entre différentes parties du logiciel au travers des interruptions, variables globales, programmation par effet de bord, ...).

4.3 Conclusion sur l'évaluation

Cette évaluation a montré que le **pouvoir expressif** de Buzz permet l'expression de systèmes correspondants à nos objectifs. C'est-à-dire des systèmes s'exécutant sur des plate-formes matérielles restreintes (faible puissance de calcul, faible capacité mémoire). L'utilisation des différents types de composant (actif, passif et interrupteur) et de liaison (asynchrone, synchrone et retardée) s'est avérée tout à fait adaptée. Comme mentionné à plusieurs reprises, Buzz ne se veut pas universel et les primitives qui le composent ont été choisies de manière arbitraire à titre d'illustration. Néanmoins, ces évaluations nous ont permis de vérifier que ces primitives correspondent aux concepts souvent utilisés de manière implicite. En particulier, cela comprend les utilisations de threads, les communications asynchrones ou le traitement des interruptions tels que décrits dans l'introduction de ce manuscrit, qui sont souvent sources d'introduction de défauts. L'intégration dans le langage des ces concepts permet de mieux guider la conception. L'architecture résultante contient plus d'informations et est moins chargée que l'architecture d'un système au comportement équivalent exprimée dans un autre langage (*e.g.* Think).

Aussi bien sur l'exemple académique du dîner que sur l'application embarquée sur des nœuds de réseaux de capteurs, les résultats d'**implantation** nous ont confortés sur la faisabilité de la démarche. Avec relativement peu d'efforts sur l'optimisation du code généré par Buzz, les résultats sont tout à fait acceptables pour ce type de systèmes. Ces résultats confirment que l'utilisation de Think comme base d'implantation est un choix valide. Il nous a permis d'arriver rapidement à des résultats d'implantations acceptables. La réutilisation des composants de la bibliothèque Kortex ainsi que de toute la chaîne de génération de code du compilateur Think réduit les efforts de développement nécessaires tout en assurant une maîtrise totale du système, ce qui n'est pas forcément le cas lors de l'utilisation d'un système d'exploitation tiers. De plus, notre compilateur reste à l'état de prototype, ce qui laisse espérer plusieurs voies possibles d'amélioration du code généré (*e.g.* plus compact, plus performant).

La génération d'un modèle BIP à des fins d'**analyses** est elle aussi concluante. Le premier exemple, de part sa taille et sa relative simplicité, nous a permis de mettre en œuvre une série d'analyses démontrant les possibilités offertes par la disposition d'un modèle formel du système. Le modèle BIP produit pour la deuxième application est lui de taille trop importante pour pouvoir appliquer les mêmes techniques d'analyse que celles employées dans le premier exemple. Nous n'avons hélas pas pu mener d'analyses poussées sur ce modèle par manque de temps. Une simulation permet de se convaincre de la correction du modèle BIP créé. Nous sommes convaincus qu'avec l'arrivée de nouvelles techniques et outils d'analyse de modèles BIP, la quantité d'effort nécessaire à l'obtention de résultats d'analyses plus complexes sera grandement réduite.

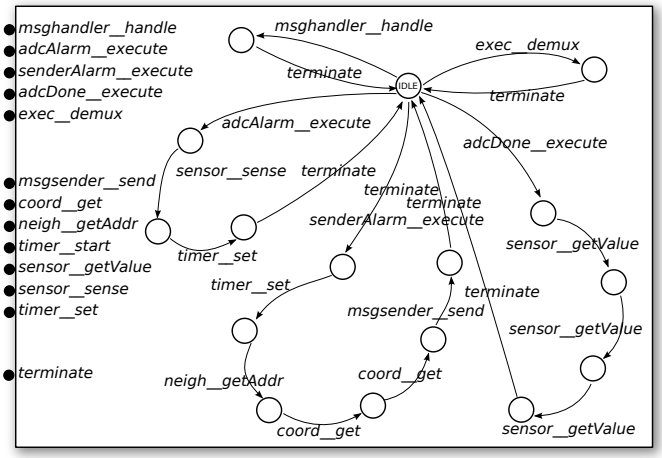


: groupe de composants correspondant aux composants composite du prototype original

Toutes les liaisons sont synchrones, à l'exception des liaisons des composants interrupteurs vers les composants actifs qui sont asynchrones.

FIGURE 4.14 – Architecture du système utilisant BUZZ.

```
void adcAlarm__execute(void){
 kt_u32 period;
 sensor__sense(my.nextadc);
 // convert it to ticks
 period = SAMPLE_PERIOD;
 period = period _SEC;
 timer__set(period, FROM_NOW,
            0, ADCALARM);
}
```



Seule la méthode `execute` de l'interface serveur `adcAlarm` est donnée en code C. Les noms des états de l'automate ont été retirés pour alléger le schéma.

FIGURE 4.15 – Modèle BIP du composant BUZZ `applimain`.

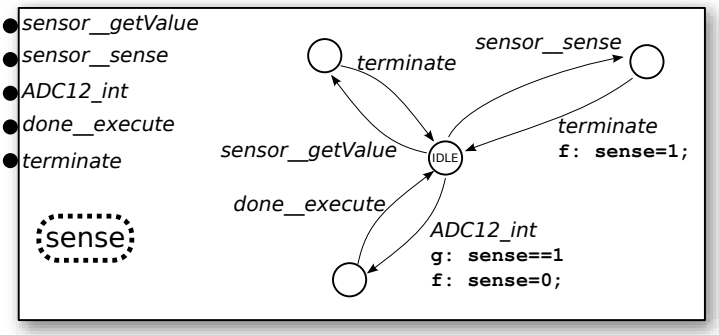


FIGURE 4.16 – Modèle BIP du composant `applisensors`.

Conclusions et Perspectives

1 Contributions de la thèse

Dans, cette thèse, nous avons étudié l'implantation des logiciels embarqués à base de composants complètement caractérisés quand à leur comportement dynamique. Nous avons identifié et résolu plusieurs difficultés :

- le pouvoir expressif du langage de conception doit être suffisant pour satisfaire au double objectif (implantation et analyse) fixé, tout en restant proche des langages utilisés actuellement par les développeurs ;
- la préservation de la fidélité entre l'implantation et le modèle sur lequel portent les analyses doit être garantie pour assurer la pertinence des résultats d'analyse.

Notre solution est suffisamment réaliste pour avoir permis le développement d'un prototype complet, nommé Buzz, que nous avons appliqué à des exemples significatifs.

1.1 Construction d'un langage de conception

Nous proposons une démarche pour la construction d'un langage de conception permettant à la fois l'implantation et l'analyse des systèmes développés. Cette construction de langage procède de manière **itérative**, ce qui permet l'ajout de nouvelles primitives dans le langage à moindre effort. À chacune d'elle correspondent des règles de **traduction vers deux langages** spécialisés dans les tâches d'implantation et d'analyse. Pour assurer la correction de ces règles, les distances entre le nouveau langage et ses deux langages cibles sont maintenues aussi réduites que possible. Pour cela, le langage construit est **inspiré des deux langages cibles**, et c'est l'universalité de ces deux langages (Fractal et BIP dans notre prototype) qui autorise la » couverture« de langage adaptés aux domaines et aux développeurs.

La **fidélité** entre le système développé, son implantation et son modèle analysable est garantie par construction.

Une construction itérative. Les objectifs d'implantations et d'analyses sont obtenus par traduction du nouveau langage vers deux langages spécialisés dans ces deux tâches. L'emploi de deux traductions permet la réutilisation des outils et techniques relatifs aux deux langages cibles. Cette réutilisation épargne de lourds développements et représente aussi un énorme gain en qualité des résultats car l'ensemble des possibilités offertes par les deux langages sont directement acquises. Contrairement aux approches classiques où les traductions sont mises en place de manière indépendante de la construction du langage source, nous les intégrons au plus tôt dans celle-ci. C'est-à-dire qu'à chaque introduction de primitive dans le langage, les règles de traduction correspondantes doivent être ajoutées aux deux traductions. Cette approche itérative assure systématiquement qu'en fin de chaque itération, le langage est à la fois implantable

et analysable. Cette systématisation permet aussi d'assurer que chaque primitive possède une sémantique clairement définie, celle-ci étant nécessaire à la mise en place des traductions.

Un asservissement inversé. Nous procédons de manière inverse aux approches classiques en ce qui concerne la relation d'asservissement entre langage source et langages cibles. Nous procédons au mariage des deux langages cibles des traductions pour construire le nouveau langage au lieu de fixer le nouveau langage et de construire des traductions à posteriori. Cette remontée d'informations des langages cibles vers le nouveau langage permet de choisir judicieusement les primitives sélectionnées de manière à obtenir des règles de traduction aussi simples que possible grâce au maintient de distances relativement faibles entre langage source et langages cibles.

Des traductions fidèles. La fidélité entre les modèles exprimés dans le langage source et leurs traductions est obtenue en exploitant, rapprochant et identifiant les concepts d'architectures et de composants (et la possibilité de construire des architectures par composition de composants) présents aussi bien dans le nouveau langage que dans les deux langages cibles. Nous nous efforçons de maintenir l'architecture du modèle original au travers de deux traductions isomorphes. C'est cet isomorphisme qui garantit la fidélité recherchée.

Un langage à la carte. Nous avons introduit une démarche pour la construction de langages de conception implantables et analysables « à la carte », c'est-à-dire de langages constitués uniquement de primitives sélectionnées pour un contexte précis (*i.e.* un langage dédié, ou DSL [22]). Cette spécialisation permet de ne pas encombrer le langage avec des concepts qui ne seront pas (ou mal) utilisés. Cela assure des modèles plus concis et donc plus lisibles, pour lesquels la détection et la localisation de défauts est plus aisé. Cette construction « à la carte » permet d'apporter une solution plus élégante au problème classique posé par la différence entre un langage et le domaine dans lequel il est employé. Lorsqu'il est nécessaire d'exprimer une construction qui ne fait pas partie du langage, il est nécessaire soit de contourner ce manque (par exemple par l'utilisation de *patterns*), et donc de surcharger inutilement le modèle, soit de modifier le langage. Notre démarche permet cette modification, ce qui améliore l'adéquation entre le langage construit et ses domaines d'applications.

Nous rappelons dans le paragraphe suivant les caractéristiques principales du prototype Buzz qui est une illustration de notre démarche. Un langage orienté flots de données construit avec notre démarche est envisagé par le projet MIND [23]

1.2 Buzz : prototype de langage et compilateur associé

Nous avons développé un prototype de langage, appelé Buzz, construit suivant notre démarche. Buzz se base sur deux traductions visant Think et BIP pour respectivement l'implantation et l'analyse des systèmes. Ce dernier met l'accent sur l'expression du comportement dynamique du système grâce à des primitives spécialisées (communications et activités). La réalisation de ces traductions s'est faite par extensions du compilateur actuel de Think, appelé Nuptse. Ces extensions exploitent à la fois le mécanisme de « plugin » offert par Nuptse ainsi que le remplacement de certains de ses composants. Nous avons montré avec deux expériences différentes les bénéfices de notre approche. La première illustre sur un exemple académique la réutilisation directe des outils et techniques acquis grâce à BIP et Think. La seconde, qui détaille

22. DSL pour « Domain Specific Language ».
23. Document de présentation du projet MIND : http://www.minalogic.com/posters/Mind.pdf

le portage d'un système fonctionnant sur un réseau de capteurs, met particulièrement en valeur la lisibilité accrue du modèle Buzz face au modèle original et les capacités de récupération de code tiers (code « legacy »).

Limitations du prototype

Nous détaillons dans les paragraphes suivants un ensemble de limitations de notre prototype qui peuvent être levées par des développements.

Spécification de l'ordonnancement. Le prototype actuel permet la prise en compte de politiques d'ordonnancement arbitraires par la fourniture à la fois d'un code C et d'un modèle BIP, tous deux reposant sur un ensemble de poignées (« hooks ») mises à disposition par Buzz (voir les section 3.4.2 et 3.5.2). L'utilisation d'un langage dédié à l'expression de politiques d'ordonnancement à partir duquel les codes en C et BIP seraient générés permettrait une meilleure séparation des préoccupations.

Modèle de l'environnement. Nous avons montré dans le chapitre 4 que l'intégration d'un modèle de l'environnement dans le modèle BIP généré par Buzz n'est actuellement pas guidé et doit se faire de façon entièrement manuelle. C'est un frein à l'utilisation d'analyses mettant en jeu l'environnement du système. Cet handicap pourrait être corrigé, de manière similaire au point présenté précédemment pour l'ordonnancement, par une séparation de cette modélisation, par exemple à l'aide d'un formalisme adapté.

Hiérarchie comme support de l'abstraction. Dans l'introduction, nous avons présenté brièvement les enjeux liés au choix de l'abstraction utilisée pour l'analyse d'un système. Buzz n'impose pas de niveau de granularité de la modélisation au sein d'un composant mais il ne permet pas d'abstraire une partie composée de plusieurs composants. Il parait intéressant d'intégrer la notion de hiérarchie (Buzz est limité à une architecture plate par simplicité), d'autant plus que les langages Think et BIP supporte cette notion. Il deviendrait possible de changer la correspondance *un à un* qu'il existe actuellement entre modèle de comportement et composant Buzz. Il serait alors possible d'associer à un composant composite, contenant plusieurs sous-composants, un seul modèle de comportement.

Utilisation de BIP version 2. Contrairement aux points précédents, ce paragraphe décrit une facilité offerte par une évolution technique et non pas par une limitation.

À l'époque où ces travaux ont débuté, seule la première version du langage BIP était disponible. La seconde version est maintenant disponible et procure un nombre important d'améliorations. En particulier, le support de connecteurs structurés et des concepts d'encapsulation plus proches du génie logiciel que la première version. Ces deux améliorations permettraient la simplification des règles de traduction et de leur mise en œuvre dans le compilateur. Une conséquence directe serait une vérification de ces traductions plus aisée et un modèle généré plus modulaire. En effet, dans la première version du langage BIP, une définition de connecteur ne peut faire intervenir que des sous-composants contenus dans le composant où elle se trouve. S'il est nécessaire de connecter un nouveau composant à ce connecteur, il peut être nécessaire de déplacer la définition du connecteur à un niveau supérieur dans la hiérarchie. BIP 2 propose une solution à ce problème avec la possibilité de composer hiérarchiquement les connecteurs.

2 Perspectives

Cette thèse a présenté l'intégration d'une vue comportementale à l'architecture à composant d'un système au travers de Buzz. À partir de cette architecture enrichie, nous procédons à l'application de *patterns* architecturaux pour aboutir à une implantation d'une part, et à la construction de modèles formels du système pour la vérification d'autre part.

Ces travaux pourraient être étendus pour la prise en compte d'autres aspects que le comportement, comme les optimisations, la reconfiguration, l'isolation, la protection, la qualité de service, ... De telles extensions permettraient la création d'un environnement de développement plus complet où toutes les vues du système en développement seraient modifiables indépendamment les unes des autres. Cet outil serait capable à la fois de combiner les différentes vues pour générer une implantation correcte vis-à-vis de chacune d'elles, mais aussi de construire un modèle formel du système sur lequel baser des vérifications globales. Nous avons montré dans [AHJ+09] que ces différentes vues peuvent être intégrées à l'architecture des systèmes et donner lieu à des implantations par des mécanismes identiques à ceux mis en œuvre dans Buzz. Néanmoins, un outil intégré devra en plus fournir une solution à la combinaison des différentes vues. Cette combinaison ne consiste pas en la simple application successive des supports de chacune des vues ; sa correction est une tâche complexe. L'outil doit être capable à la fois de présenter au développeur une abstraction du système pour chacune des vues tout en assurant la correction du système en développement. Les travaux présentés dans cette thèse fournissent un premier élément de réponse. Notre démarche, qui permet de garantir la fidélité entre le modèle formel et l'implantation du système, peut servir de base pour la vérification de la correction du mariage des différentes vues du système. Il serait alors possible de modifier les aspects liés à la sécurité d'un système tout en étant capable d'assurer que les contraintes énoncées dans une vue liée à la qualité de service sont toujours respectées.

Quatrième partie

Annexes

Annexe A

BIP2THINK & Nesc2BIP

Sommaire

Cette annexe présente deux expériences articulées autour du langage BIP. La première repose sur un prototype de compilateur permettant de traduire un modèle BIP en une architecture de composants THINK. Ce compilateur est mis en pratique sur un exemple de système orienté flots de données : un encodeur vidéo MPEG. La seconde expérience repose aussi sur un prototype de compilateur mais cette fois du langage nesc, orienté événements (voir la description de TinyOS en section 1.1.5) vers BIP. Nous illustrons ce second prototype sur un exemple d'application fonctionnant sur un réseau de capteurs.

Ces deux expériences mettent en valeur le pouvoir expressif du langage BIP, qui permet ici la description de manière concise de systèmes aux fonctionnements différents (flots de données vs. événementiel).

A.1 Traduction de BIP vers Think

Cette expérience explore deux voies : l'utilisation du langage BIP comme langage de programmation pour un système orienté flots de données, et l'utilisation de THINK comme pivot pour l'implantation sur plate-formes embarquées. Nous présentons dans les paragraphes suivants une traduction de BIP vers THINK. Cette traduction est évaluée sur un exemple d'encodeur vidéo que nous implantons sur un iPod.

A.1.1 Principes

Pour traduire un modèle BIP vers une architecture de composants THINK, nous procédons de la manière suivante. Une partie des composants est prédéfinie et réutilisée pour chaque traduction. Ces composants sont rassemblés dans une bibliothèque spécifique et concernent en grande

partie l'implantation du moteur d'exécution BIP, décrit en section 2.4.3. Le compilateur `bipc`, qui implante la traduction de BIP vers THINK, utilise à la fois des composants de cette bibliothèque, des composants de la bibliothèque KORTEX (décrite en section 2.3.2) pour le pilotage du matériel ainsi que des composants synthétisés et spécifiques au modèle BIP compilé. La figure A.1 résume le procédé.

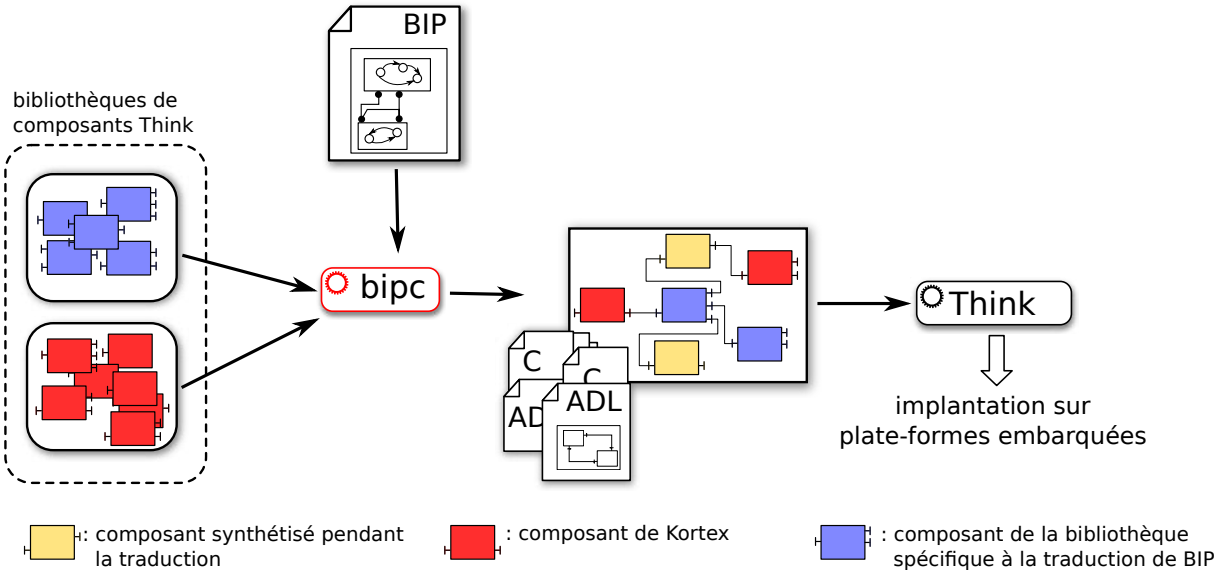


FIGURE A.1 – Traduction de BIP vers THINK.

Pour simplifier la création de notre prototype de compilateur, nous nous sommes intéressés à un sous-ensemble du langage BIP :

- pas de hiérarchie de composant, uniquement un modèle plat (*i.e.* pas de composant composite) ;
- les connecteurs lient des ports synchrons et au plus un trigger. C'est-à-dire des connecteur de type *rendez-vous* ou *diffusion* (voir leurs descriptions en section 2.4.2).

Nous présentons dans un premier temps les règles de traduction qui produisent de nouveaux composants THINK puis ensuite les composants prédéfinis pour l'implantation du moteur d'exécution.

Composant atomique. Tous les composants atomiques BIP sont traduits par des composants THINK spécifiques dont les contenus sont principalement issus des traductions des automates en code C. Les interfaces des composants BIP, c'est-à-dire les données et les ports, sont traduites par des interfaces serveurs sur les composants THINK correspondants. Le type d'interface THINK pour les données, appelé `Data`, contient deux méthodes `get` et `set`, pour respectivement lire et écrire. Le type d'interface THINK pour les ports, appelé `Port`, définit en particulier les méthodes :

- `isSynced`, pour évaluer la garde de la transition associée avec ce port.
- `execute`, pour déclencher la fonction associée à la transition en attente sur le port.

Connecteur. Chaque connecteur du modèle BIP est traduit en un composant THINK. Ce composant dispose d'une interface cliente pour chaque port que le connecteur relie. Ces interfaces seront liées aux interfaces serveurs de type `Port` correspondantes sur les composants THINK représentant les atomes BIP. Si le connecteur dispose de gardes sur les données de certains

composants BIP, alors des interfaces serveurs sont ajoutées au composant THINK. Elles seront elles aussi liées aux composants THINK des atomes BIP correspondants via les interfaces de type `Data`. Ce composant est chargé de calculer les interactions possibles et d'en déclencher l'exécution si nécessaire. Il dispose d'une interface serveur de type `Connector` qui définit les méthodes suivantes :

- `execute`, pour déclencher l'exécution de l'interaction maximale associée au connecteur.
- `isLegal`, pour tester s'il existe au moins une interaction possible sur le connecteur.

De plus, ce composant peut inhiber des interactions pour l'application des règles de priorités. Les méthodes utilisées sont :

- `inhibit(id)` pour désactiver l'interaction dont l'identifiant est fourni en paramètre.
- `isInteractionLegal(id)`, pour tester si une interaction donnée est possible ou non.

Priorité. Les règles de priorité sont rassemblées dans un unique composant THINK. Ce composant est lié à tous les composants connecteurs impliqués dans au moins une règle de priorité via leurs interfaces de type `Connector` ainsi qu'aux composants THINK correspondants aux composants atomiques BIP dont les données sont nécessaires (*i.e.* utilisées dans une garde), via leurs interfaces de type `Data`. Ce composant priorité exporte une unique interface de type `Priority` comportant une seule méthode `apply` pour appliquer les règles de priorité.

Moteur d'exécution BIP. Le moteur d'exécution BIP est implanté par un composant de la bibliothèque de composants prédéfinis. Ce composant implante la boucle d'exécution présentée dans la section 2.4.3. Il est lié au composant chargé d'appliquer les priorités ainsi qu'à tous les composants connecteurs. À chaque itération de la boucle, le moteur invoque les méthodes `isLegal` sur tous les connecteurs pour construire une liste d'interactions possibles. Il invoque ensuite la méthode `apply` du composant priorité, qui va éventuellement invoquer les méthodes `inhibit` de certains connecteurs. Enfin, il invoque la méthode `execute` d'un connecteur après le filtrage. La figure A.2 illustre la traduction d'un modèle BIP constitué de trois composants, de deux connecteurs et d'une règle de priorité.

A.1.2 Évaluation sur un encodeur vidéo

Pour évaluer cette traduction, nous nous sommes basés sur un encodeur vidéo pour lequel nous disposons d'un modèle BIP. Ce modèle BIP est lui même issu d'une expérience de réingénierie sous forme de composants BIP d'une application monolithique écrite en C de manière classique. Le modèle BIP, en plus d'expliciter les flots de données qui transitent entre les composants, permet d'appliquer des règles d'ordonnancement sans modification du code fonctionnel, tel que décrit dans [CFLS05]. L'algorithme consiste en une succession d'étapes qui s'exécutent sous la forme d'un « pipeline » en s'échangeant des données, c'est-à-dire que le système est orienté flots de données. La figure A.3 illustre partiellement le modèle BIP de cet encodeur. La structure fait clairement ressortir les flots de données.

Le modèle BIP contient 20 composants, 34 connecteurs ainsi qu'un ensemble de règles de priorités pour assurer l'ordonnancement des exécutions des composants.

Le résultat de la traduction est une architecture composée de 56 composants THINK, d'environ 6300 lignes de code C et 1000 lignes d'ADL. Nous avons ensuite implanté cette architecture vers un exécutable pour un iPod video. Cette plate-forme matérielle consiste en deux cœurs ARM7tdmi (nous n'en utilisons qu'un seul) cadencés à 80Mhz et plusieurs MiB de RAM. Nous évaluons à la fois la faisabilité de l'approche et le sur-coût (compacité du code final et performances à l'exécution) par rapport à une approche classique basée sur du code C monolithique.

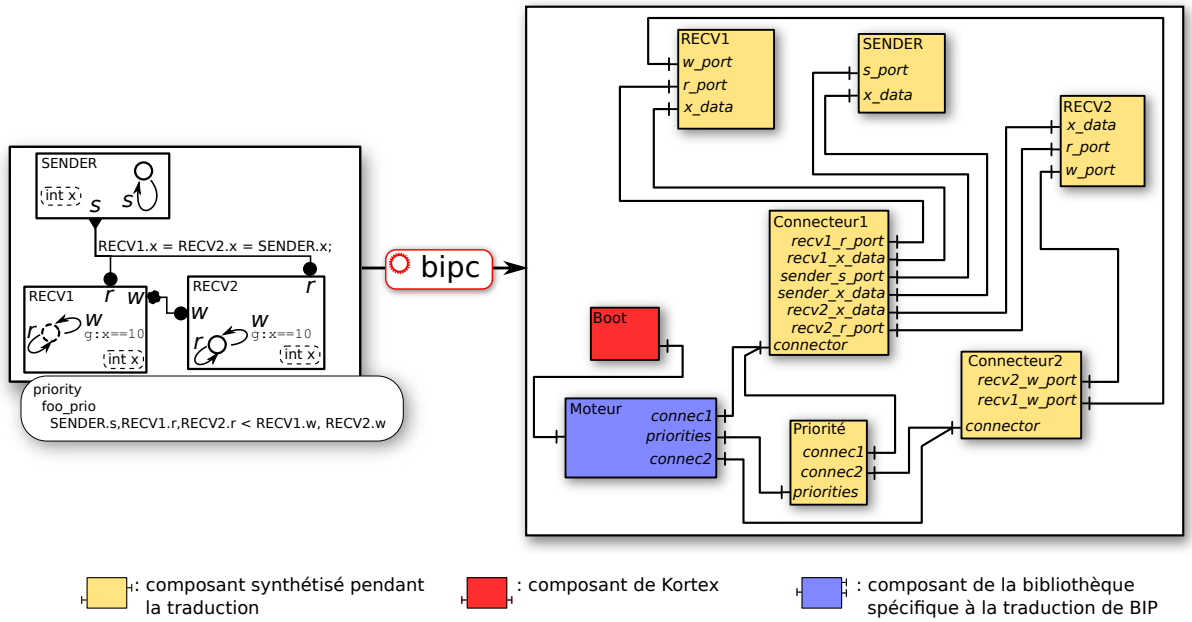


FIGURE A.2 – Exemple simple de traduction d'un modèle BIP en une architecture de composants THINK.

L'exécutable produit par notre chaîne d'outils pèse 300KiB, contre 216KiB pour la version monolithique, soit une augmentation d'environs 38%. Pour les performances à l'exécution, le tableau A.1 présente les résultats mesurés pour deux vidéos différentes. On constate que notre approche réduit environs de moitié la vitesse d'encodage des deux vidéos.

	Vidéo (*pixels, nombre d'images*)	Temps de traitement (*secondes*)	Vitesse (*images par seconde*)
BIP+THINK	320×240, 40	500	0,08
	64×48, 161	77	2
monolithique	320×240, 40	200	0,203
	64×48, 161	37	4,3

TABLE A.1 – Mesure de performances de l'encodeur vidéo sur iPod.

Ces résultats peuvent en partie s'expliquer par le manque de maturité de notre outil de traduction, pour lequel aucun effort d'optimisation n'a été entrepris. Une analyse plus fine montre qu'une grande partie du coût à l'exécution est imputable à 66% aux exécutions des connecteurs BIP et 33% à l'évaluation des règles de priorité. Ces performances pourraient être grandement améliorées par des techniques d'optimisation, que cela soit au niveau du modèle BIP [MB09] ou au niveau de notre traduction.

A.1.3 Conclusion

Plus de détails peuvent être obtenus dans [PPRS06]. Cette expérience démontre que BIP permet l'expression de systèmes orientés flot de données et qu'il est possible d'en dériver une implantation réaliste sur plate-forme embarquée. Cette expérience nous permet aussi de vérifier

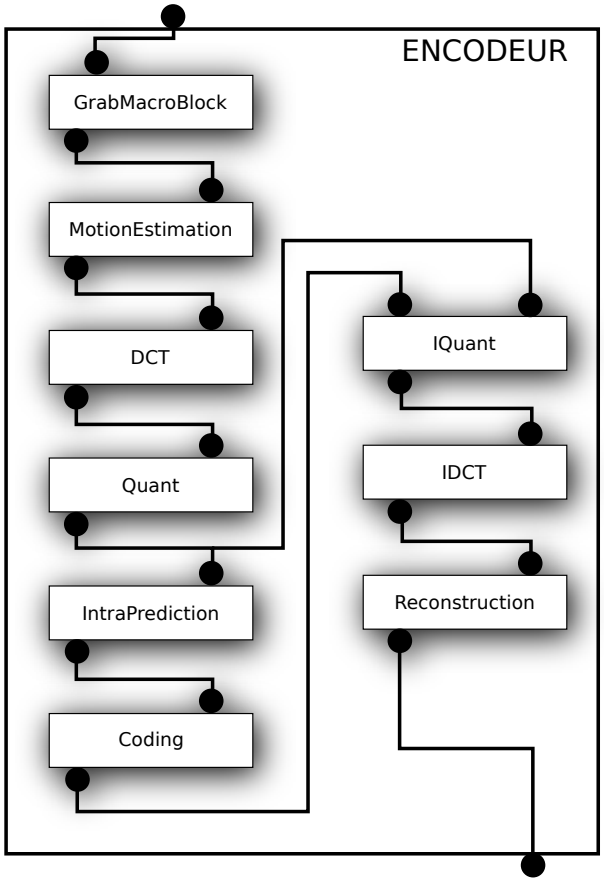


FIGURE A.3 – Modèle BIP de l'encodeur vidéo.

le gain représenté par l'utilisation de THINK comme pivot pour l'implantation plutôt qu'une génération directe de code en langage C. Cette utilisation de THINK nous a permis de nous abstraire de la majorité des problèmes liés à la gestion du matériel car déjà pris en compte par la bibliothèque de composants KORTEX. Il est important de souligner que cette expérience a été faite alors que THINK était en version 2. À cette époque, le nombre des optimisations disponibles était très réduit par rapport à ce qui est disponible aujourd'hui dans la version 4, décrite en section 2.3.

Néanmoins, nous avons noté une limitation dans l'utilisation du langage BIP comme langage de programmation de système embarqué. La première version du langage BIP que nous avons utilisé ne permet pas l'expression de certaines contraintes ou propriétés indispensables à la programmation de ces systèmes, plus particulièrement pour tous les aspects liés au matériel (*i.e.* interruption, placement mémoire, ...). Des travaux sont actuellement en cours sur la deuxième version du langage pour combler cette limitation.

A.2 Traduction de nesC vers BIP

Cette expérience utilise le langage BIP pour la modélisation et la vérification de systèmes en réseaux. La méthode est appliquée aux systèmes utilisant TinyOS et donc programmés avec le langage nesC (voir la description dans la section 1.1.5). Elle se démarque des approches plus classiques en construisant un modèle global pour le réseau. C'est-à-dire qu'un modèle BIP

est construit en composant les modèles des logiciels s'exécutant sur les nœuds du réseau, les modèles de la plate-forme matérielle utilisée par ces nœuds ainsi que les communications radio. Cette approche est très différente des solutions existantes qui se limitent à la simulation de la plate-forme matérielle et des communications radio en utilisant le code exécutable obtenu par la compilation du code nesC. Cette expérience montre avec quelle facilité le langage BIP peut être utilisé pour l'expression de systèmes existants, en capturant à la fois la structure du système original (les composants nesc) ainsi que son modèle d'exécution (événementiel). Plus de détails sont disponibles dans [BMP+07].

A.2.1 Principes de la traduction

Nous présentons la modélisation en langage BIP d'un réseau de capteurs initialement programmé en nesc. Cette modélisation comprend l'intégralité du réseau, c'est-à-dire les nœuds et leur environnement, en particulier les échanges radio. Cette présentation se fait en trois parties. La première présente la modélisation en BIP des composants nesc de l'application, c'est-à-dire les composants développés par les utilisateurs. La deuxième partie présente la modélisation de TinyOS, c'est-à-dire l'ordonnancement de l'exécution des composants applicatifs et leurs communications. Enfin, la dernière partie présente la composition des modèles BIP des nœuds, à l'aide d'un ensemble de connecteurs et de règles de priorité, pour obtenir un modèle BIP global pour tout le réseau.

Composants applicatifs. Nous utilisons un traducteur automatique qui prend en entrée un code nesC non ré-entrant et génère en sortie un ensemble de composants BIP, de règles de priorité et de connecteurs BIP. Des annotations dans le code nesc permettent à notre outil d'extraire simplement la structure du code nesC (définition de traitant d'événement, émission de signaux, etc). La méthode consiste en la création d'un composant atomique BIP pour chaque traitant de commande, événement et tâche définis dans le code nesC. Notre outil crée la structure seulement, le comportement de ces composants atomiques est laissé à la charge du développeur. Les raisons de cette limitation sont les mêmes que celles données en section 3.5.2 pour le contenu des composants Buzz (*e.g.* problème de l'extraction du comportement d'un code arbitraire). La contrainte de non ré-entrance sur le code nesC peut être supprimée en utilisant un modèle BIP plus riche, ce que nous n'avons pas souhaité dans ce travail dans le but de simplifier notre premier prototype de traducteur.

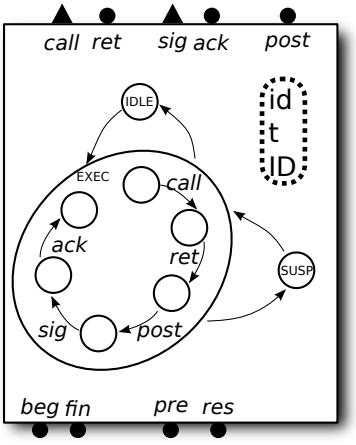


Figure A.4 – Squelette de composant atomique BIP pour les traitants nesC.

La figure A.4 illustre le squelette de composant BIP utilisé pour les trois catégories de traitant (commande, événément et tâche). Le comportement est spécifié par un automates à trois états : `IDLE`, `SUSP` et `EXEC`. L'état `EXEC` est en réalité décomposé en plusieurs états qui correspondent au comportement spécifique du composant (la partie à la charge du développeur dont il est question plus haut).

Les ports sont classés dans deux catégories :

- les ports `beg`,`fin`,`pre` et `res` servent pour les transitions de début, de fin, de préemption et de reprise de l'exécution du traitant. Ils sont impliqués dans les interactions avec TinyOS ou dans les interactions modélisant les mécanismes d'appel/retour des traitants de commandes. Ces ports sont synchrons car ils doivent être déclenchés par d'autres composants.
- les ports `call`, `ret`, `sig`, `ack` et `post`, servent pour les transitions d'appel et retour des commandes, signalement et acquitement des événenements, et envoi de tâches. Les ports `call` et `sig` sont des triggers car à l'origine d'une interaction de type diffusion.

Le squelette contient aussi un identifiant unique, l'identifiant du composant appelé ainsi que l'identifiant de la tâche envoyée.

TinyOS. Notre modèle de TinyOS est la composition de deux ensembles de composants : les ordonnanceurs d'événements et de tâches, et les composants correspondant à la plate-forme matérielle : timers, capteurs, radio.

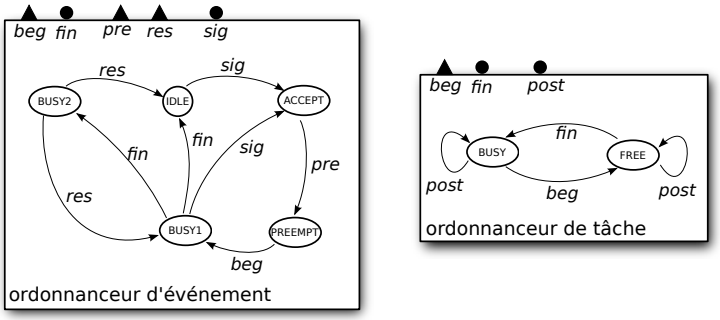


FIGURE A.5 – Ordonnanceurs d'événement et de tâche.

L'ordonnanceur d'événements, partiellement illustré dans la figure A.5, est responsable des événements générés par les composants BIP correspondant au matériel. Si un composant s'exécute lors de la réception par l'ordonnanceur d'un événement e sur son port `sig`, alors l'ordonnanceur le préempte en se synchronisant sur son port `pre` et empile son identifiant. Ensuite, l'ordonnanceur déclenche l'exécution du traitant d'événement correspondant à e en diffusant e au travers de son port `beg`. Lorsqu'il se trouve dans l'état `BUSY1`, l'ordonnanceur d'événement peut à la fois recevoir de nouveaux événements par son port `sig` ou des notifications de fin de traitement par son port `fin`.

L'ordonnanceur de tâche, partiellement illustré dans la figure A.5, reçoit de nouvelles tâches à exécuter par son port `post` et les stocke dans une file d'attente. Il ne peut déclencher l'exécution d'une tâche qu'à la seule condition que l'ordonnanceur d'événement se trouve dans son état `IDLE` (c'est-à-dire qu'aucun événément ou commande n'est actuellement en train de s'exécuter).

Les composants BIP pour les timers, capteurs et contrôleurs radio sont modélisés de manière similaire.

Construction du modèle d'un nœud. Dans ce paragraphe, nous décrivons la composition des modèles BIP décris précédemment à l'aide de connecteurs pour la construction des modèles des nœuds. Ces connecteurs se répartissent en deux groupes.

Le premier modélise les interactions entre les composants applicatifs au travers des commandes (`call`) et des émissions de signaux (`signal`). Pour un appel de commande, un connecteur *Call* et un ensemble de connecteurs $Return_i$, comme illustré dans la figure A.6. Le connecteur

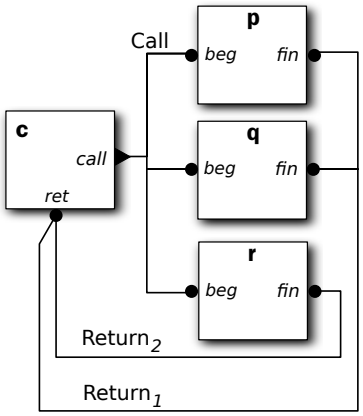


FIGURE A.6 – Connecteurs BIP pour un appel de commande.

Call diffuse au travers du port `call` de l'appelant `c` vers les ports `beg` des appelés `p`, `q` et `r`. Le composant `c` peut soit appeler les composants `p` et `q`, soit le composant `r`. Cette sélection se fait par l'utilisation d'une garde sur les identifiants. Les connecteurs $Return_i$ synchronisent les ports `fin` des composants appelés avec le port `ret` du composant appelant. Le traitement des signaux est identique au traitement des commandes. Cependant, les signaux représentant des événements matériels sont traités séparément par l'ordonnanceur d'événements.

Le deuxième groupe de connecteurs traite des interactions entre les composants BIP correspondants aux composants applicatifs et les composant BIP correspondant à TinyOS. La figure A.7 illustre ces connecteurs. Les connecteurs $TBegin$ et $EBegin$ traitent respectivement des

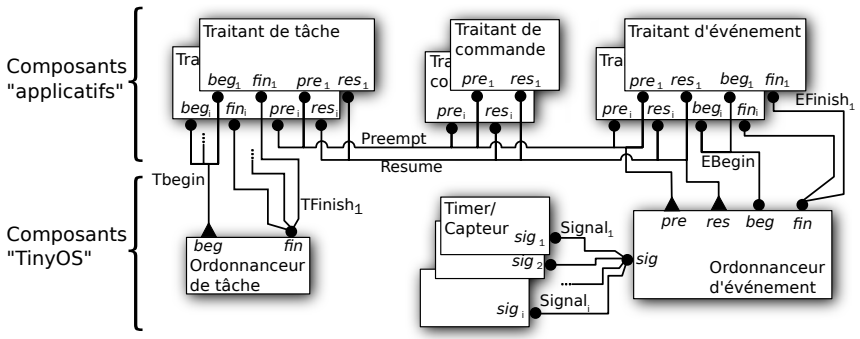


FIGURE A.7 – Architecture globale du modèle BIP.

interactions entre les traitants de tâches/ordonnanceur de tâche et traitants d'événements/ordonnanceur d'événements. Les connecteurs $TFinish_i$ et $EFinish_i$ sont utilisés par les traitants de tâches et d'événements pour notifier de leur terminaison. Le connecteur *Preempt* provoque la préemption des composants applicatifs. Le connecteur *Resume* est utilisé pour la reprise de l'exécution du dernier composant suspendu. Enfin, les connecteurs $Signal_i$ sont utilisés pour

signaler les événements générés par le matériel. La prise en compte de nouvelles tâches se fait au travers de connecteurs liant le port `post` de l'ordonnanceur des tâches et les ports `post` des composants applicatifs (ces connecteurs ne sont pas représentés sur la figure A.7).

Construction du modèle global du réseau. Cette construction consiste essentiellement en la modélisation des communications radio entre les nœuds qui composent le réseau. Ces communications sont modélisées par des connecteurs BIP liant les composants BIP représentant les contrôleurs radio de chacun des nœuds (ces composants font partie du deuxième groupe décrit précédemment). Ces composants possèdent un port `broadcast` et un port `listen` pour respectivement envoyer et recevoir des données. Nous ne considérons dans cet exemple que les réseaux à topologie fixe. Le port `broadcast` d'un nœud `n` est lié à l'ensemble des ports `listen` des nœuds susceptibles de recevoir (*i.e.* les nœuds à portée de radio) des données de `n`. Cette liaison est assurée par un connecteur BIP, sur lequel il est possible d'utiliser des conditions d'activation, pour modéliser par exemple un lien avec perte.

A.2.2 Évaluations

Nous considérons trois exemples, fournis avec TinyOS : `BlinkTask`, `SenseToLeds` et `SenderReceiver`. Le premier exemple illustre l'utilisation de techniques de vérifications. Les deux autres comparent notre méthode à des techniques spécifiques de simulation. Naturellement, avec cette approche plus généraliste, nous pourrions nous attendre à des résultats en terme de performance en dessous des résultats obtenus par les techniques spécialisées pour cette tâche. De plus, l'utilisation de modèles riches (non déterministes) et non de modèles déterministes pourrait aussi avoir un impact négatif sur les résultats. Néanmoins, les résultats expérimentaux présentés ici ne mettent pas en évidence de différence significative, en comparaison par exemple avec [ECZ06].

`BlinkTask` décrit un nœud avec une variable `state` représentant l'état de ses LEDs. Cette variable est partagée entre la tâche `processing`, qui la lit, et le traitant d'événement `Timer.fired()`, qui la modifie. Pour `BlinkTask`, nous générons un modèle BIP temporisé comprenant 4 composants BIP atomiques (les composants applicatifs), 3 composants pour TinyOS (2 ordonnanceurs et 1 « timer ») et 11 connecteurs. Une exploration exhaustive de l'espace des états permet la détection d'états erronés où un événement est signalé par le « timer » alors que la tâche `processing` est en cours d'exécution (situation de course sur la variable `state`). Les chemins menant à ces erreurs peuvent être obtenus en modélisant un composant BIP observateur qui garde trace des séquences d'interactions. Par exemple, l'espace d'états analysé possède 28701 états et 46197 transitions pour les intervalles de temps d'exécution suivant (l'unité est un cycle et les intervalles sont donnés sous la forme $[min, max]$) :

- période de signalement du « timer » : 50 ;
- `Timer.fired()` : $[2, 9]$;
- `Leds.redOn()` : $[2, 7]$;
- `Leds.redOff()` : $[2, 7]$;
- `processing` : $[20, 32]$.

Les valeurs choisies assurent un comportement correct de l'exemple. Cependant, le changement de la période du « timer » en une valeur inférieure à 48 provoque l'apparition d'erreurs, détectées par le composant observateur.

Le deuxième exemple est `SenseToLeds` qui est un réseau composé de nœuds échantillonnant des données d'un capteur photo. Chacun des nœuds affiche ses données au travers de ses LEDs. Le logiciel de chaque nœud se compose de 4 composants. La traduction vers un modèle BIP produit 8 composants BIP applicatifs, 4 composants pour TinyOS (2 ordonnanceurs, 1 « timer » et un

capteur) ainsi que 21 connecteurs. Nous considérons dans cet exemple un réseau composé de 250 nœuds de ce type, sans aucune communication radio. Pour un temps virtuel de 300 secondes, en considérant un timer de fréquence 4Hz sur chacun des nœuds, la simulation requiert 600 secondes sur une station de travail classique. Comme nous l'attendions, le temps de simulation augmente linéairement avec le nombre de nœuds du réseau.

Le troisième exemple `SenderReceiver` est un réseau d'émetteurs et de récepteurs utilisant des canaux de communication sans perte et une topologie fixe. Chaque émetteur est connecté à un nombre fixe de récepteurs y. Chaque récepteur n'est lié qu'à un seul émetteur (pas de collision possible). Les nœuds émetteurs exécutent l'application `CntToLedsAndRfm` fournie dans la base d'exemples de TinyOS. La figure A.8 donne les temps de simulation de 300 secondes virtuelles en fonction du nombre d'émetteurs x et de récepteurs par émetteur y. Chaque « timer » possède une fréquence de 4Hz.

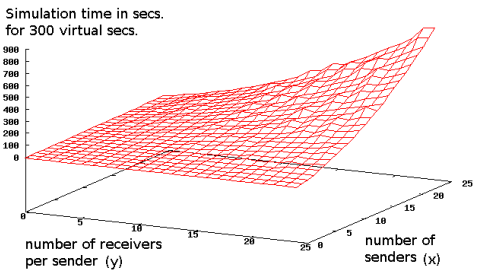


FIGURE A.8 – Temps de simulation de l'exemple `SenderReceiver`.

A.2.3 Conclusion

Cette expérience applique à TinyOS une méthode pour la modélisation et la vérification de systèmes en réseau. Cette méthode est générale et peut être appliquée à la construction de modèles globaux pour des systèmes hétérogènes. Elle consiste en la modélisation de la plate-forme d'exécution par une machine abstraite pilotant l'exécution du logiciel applicatif. Pour cela, une formalisation du langage dans lequel l'application est écrite doit être fournie, en terme de primitives offertes par la plate-forme. Cette tâche est non triviale. La formalisation devrait se placer au niveau d'abstraction le plus adapté. La granularité doit être choisie de manière à capturer tous les éléments impliqués dans les propriétés à vérifier. De plus, pour maintenir la complexité du modèle aussi faible que possible, il doit être possible d'ignorer les séquences de calculs n'impliquant aucun de ces éléments. La génération de modèle BIP à partir de nesc peut être adaptée à n'importe quel autre langage de programmation. L'analyseur de code que nous avons développé peut être modifié de façon à identifier les parties dans le code source générant les événements pertinents ainsi que pour déterminer le niveau de granularité du modèle. Nous avons dépensé 2 hommes-mois pour le développement de notre méthode pour TinyOS. Pour d'autres plate-formes, des efforts supplémentaires seraient nécessaires pour la mise sous forme de composants à un niveau d'abstraction adapté. Un tel investissement semble être la seule façon de surmonter les limitations actuelles des méthodes de conceptions basées sur les modèles et pour la conception de systèmes dont la qualité est garantie.

Annexe B

Buzz

Sommaire

Nous présentons dans cette annexe l'utilisation du compilateur présenté en chapitre 3 ainsi qu'un ensemble d'outils qui ont été développés pour faciliter l'utilisation du compilateur et de ses résultats. Cette annexe est organisée comme suit. La section B.1 détaille l'ensemble des paramètres acceptés par le compilateur ainsi que la méthode de lancement de ce dernier. La section B.2 présente un ensemble d'outils qui peuvent être employés conjointement au compilateur. Enfin, la section B.3 présente brièvement des développements annexes pouvant être utilisés en dehors du cadre de Buzz.

L'ensemble du code dont il est question dans cette annexe est distribuée avec une licence libre (GNU GPL et LGPL [24] suivant les cas) sur le site du projet Think.

B.1 Utilisation du compilateur Buzz.

Cette section présente dans un premier temps l'ensemble des paramètres acceptés par le compilateur et dans un second temps la méthode utilisée pour exécuter une compilation.

B.1.1 Paramètres acceptés

Le compilateur de Buzz accepte, en plus des paramètres du compilateur Nuptse, un ensemble de paramètres spécifiques. Ces paramètres sont tous préfixés par `buzz`. Nous donnons ici la liste complète de ces paramètres avec une description détaillée.

24. GNU « General Public License » et « Lesser General Public License ». Voir le texte complet à l'adresse : http://www.gnu.org/licenses/

Paramètres relatifs à la génération du modèle BIP

buzz-bip-output=`fichier` Ce paramètre est utilisé lors de la génération d'un modèle BIP et permet de spécifier le nom du fichier qui contiendra le résultat.

buzz-bip-clock=`true|false` Lorsque ce paramètre est positionné à `true`, le modèle BIP qui sera créé contiendra un connecteur et le nécessaire pour synchroniser tous les composants temporisés du système. Si ce paramètre est positionné à `false`, la génération du modèle BIP ignorera toutes les informations de temporisation. Plus de précisions sont données en section 3.5.3.

buzz-bip-debug=`true|false` Lorsque ce paramètre est positionné à `true`, le modèle BIP généré contiendra du code additionnel pour aider à la recherche de défaut dans une partie du code C généré. Ce code instrumente en particulier la gestion des files d'attente et la sérialisation/désérialisation (utilisées lors de l'exploration exhaustive) des données contenues dans les composants BIP.

buzz-bip-simu-state-trace=`true|false` Lorsque ce paramètre est positionné à `true`, chaque tirage de transition pendant la simulation provoquera un affichage supplémentaire dans la trace comprenant les états de départ et d'arrivée ainsi que les valeurs d'un ensemble de données. Seules les données listées par l'annotation `DEBUG_DATA` dans le code BIP sont affichées par ce biais. Cette annotation est attachée aux composants atomiques. Le listing B.1 donne un exemple d'utilisation de cette annotation et le listing B.2 donne un exemple de trace affichée à la simulation. Sur ce dernier, l'interaction exécutée par le simulateur est donnée sur les lignes 1 à 5. Ce sont les affichages à partir de la ligne 6 qui sont rajoutés.

```
@DEBUG_DATA=att_req_l,att_req_r,att_left_cs,
            att_right_cs, att_status
component geek
  data int att_req_l, att_req_r, att_left_cs,
           att_right_cs, att_status
  ...
```

Listing B.1 – Exemple d'annotation de `DEBUG_DATA` pour le composant `geek`

```
BIP_Top/methodCallConn_c2_toright__giveme__c1_fromleft/
     active_stub_inst:exec
    |inbuffer_c1__fromleft__giveme:fromleft__giveme
    |core_instance:toright__giveme|input_fifo:in
    |active_stub_inst:ready|sched_fifo_inst:fin
active_stub_c2 ::
    EXEC -> EXEC
inbuffer_c1__fromleft__giveme__def ::
    SINGLE -> (OVERFLOW_PRUNE_9)? SINGLE
activated__c2geek ::
    HANDLER_EXECUTE -> HANDLER_EXECUTE1;
    att_req_l: 1, att_right_cs: 3, att_req_r: 1,
    att_status: 1, att_left_cs: 0,
fifo_input_c1_def ::
    SINGLE -> (OVERFLOW_PRUNE_0)? SINGLE
```

```
16 active_stub_c1 ::
       IDLE -> SUSPEND
18 sched_fifo_def ::
       SINGLE_FIFO_SCHED -> (OVERFLOW_ERROR_3)? SINGLE_FIFO_SCHED
```

Listing B.2 – Exemple de trace de simulation enrichie avec des informations supplémentaires pour chaque transition.

buzz-bip-extra-debug=`true|false` Ce paramètre, lorsqu'il est positionné à `true`, permet de générer des commentaires supplémentaires dans le code BIP. Ceci dans le but de pouvoir identifier plus rapidement les problèmes dans la génération de code BIP. Ces commentaires sont attachés à chaque élément du code BIP (déclaration de données, de ports, transitions, garde, etc). Chaque commentaire contient le nom du fichier source Java du compilateur, le numéro de la ligne et le nom de la méthode responsable de la création de l'élément auquel il est attaché. Le listing B.3 donne un exemple de code généré (les commentaires ajoutés sont préfixés par `[GENINFO]`).

```
1  // [GENINFO] Utils.java:1050::copyAtom()
   component activated__active1barcomp
3  ...
   // [GENINFO] Utils.java:1171::activate()
5   data params_t input_params
    behavior
7    initial to IDLE
     // [GENINFO] Utils.java:1064::copyAtom()
9    state IDLE
      // [GENINFO] Utils.java:1074::copyAtom()
11    on // [GENINFO] Utils.java:1055::copyAtom()
      s__methodfoo_buffered// [GENINFO] Utils.java:1086::copyAtom()
13     to S11
...
```

Listing B.3 – Exemple de code contenant des informations supplémentaires pour identifier les sources en cas d'erreur de génération.

buzz-bip-overflow-error=`true|false` Ce paramètre est similaire au précédent. S'il est positionné à `true`, l'exploration est interrompue lorsqu'une file d'attente dépasse sa capacité (tentative d'ajout d'un nouvel élément alors que la file est pleine), sinon, l'exploration continue (la branche qui contient l'erreur est élaguée).

buzz-bip-caller-time=`true|false` Ce paramètre active le support de la vérification des contraintes temporelles telles que décrites dans la section 3.5.3.

buzz-bip-caller-time-error=`true|false` Ce paramètre est utilisé lorsque le modèle BIP généré est utilisé avec une exploration exhaustive. S'il est positionné à `true`, lorsqu'un dépassement d'une contrainte temporelle (telle que décrite dans la section 3.5.3) est détecté, l'exploration est interrompue. S'il est positionné à `false`, l'exploration continue (mais élague la branche qui était explorée) et signale le dépassement dans la trace produite.

Paramètres relatifs à la génération d'une implantation

buzz-leds-debug=`true|false` Lorsque ce paramètre est positionné à `true`, le code généré fait l'hypothèse qu'un composant capable de piloter des LEDs est présent dans l'architecture. La disponibilité et le comportement du code généré dépend de la plate-forme matérielle sous-jacente. L'AVR que nous avons utilisé dispose de 8 LEDs alors que la plate-forme WSN430 n'en dispose que de 2.

buzz-trap-component=`composant` Ce paramètre est obligatoire et doit contenir le nom du composant qui sera utilisé pour la gestion des contextes. Par exemple, pour la plate-forme WSN430, ce paramètre peut être `msp430.irq.lib.trap`.

buzz-tick-component=`composant` Lorsque l'ordonnancement utilise un signal périodique pour par exemple faire du partage de temps, le compilateur insère un composant `tick` qui dépend de la plate-forme matérielle. Sur la plate-forme basée sur l'AVR que nous avons utilisé, ce paramètre peut prendre la valeur `avr.atm2561.irq.lib.tick`.

buzz-irqsafe-component=`composant` De manière similaire aux paramètres précédents, ce paramètre désigne le composant qui sera utilisé pour manipuler le masque des interruptions. Ce paramètre est obligatoire.

buzz-stack-size=`stack_size` Ce paramètre permet de paramétrer la taille des piles qui sont utilisées par chaque composant actif. Sur Unix, nous utilisons une taille de `8192` (8KiB). Cette pile se trouve dans chaque intercepteur actif.

buzz-delayed-call-enable=`true|false` Ce paramètre active/désactive le support des liaisons retardées. Ce support pourrait être désactivé automatiquement lorsqu'aucune liaison de ce type n'est utilisée, mais par simplicité, nous le faisons manuellement.

buzz-aci-max-delayed-call=`delay_fifo_size` Ce paramètre permet de configurer la taille des files d'attente utilisées pour stocker les invocations portées par des liaisons retardées. Ces files se trouvent dans les intercepteurs actifs.

buzz-aci-max-aci-blocked=`blocked_fifo_size` Ce paramètre permet de configurer la taille des files d'attente utilisées pour stocker les composants bloqués lors d'invocations synchrones désignant un composant actif. Ces files se trouvent dans les intercepteurs actifs.

buzz-aci-params-queue-size=`method_fifo_size` Ce paramètre permet de configurer la taille des files d'attente, spécifiques à chaque méthode et utilisées pour sérialiser les invocations de méthodes désignant un composant actif. Ces files se trouvent dans les intercepteurs actifs.

buzz-aci-max-pending-call=`fifo_size` Ce paramètre permet de configurer la taille de la file d'attente présente dans chaque intercepteur actif et qui est commune à toutes les méthodes. Cette taille représente le nombre maximal d'invocations en attente que peut posséder un composant actif.

buzz-aci-extra-trace=`in|out|inout|none` Ce paramètre permet, si la plate-forme matérielle le supporte, l'affichage d'informations supplémentaires au cours de l'exécution.
- `in` : pour obtenir des informations lors de la réception d'invocations par les intercepteurs.
- `out` : pour les invocations émises par les intercepteurs.
- `inout` : regroupe les deux types d'information.

buzz-join-call-enable=`true|false` Active ou désactive le support des appels *joins*. Ce support est présent à titre expérimental et ne possède pas d'équivalent pour la génération de modèle BIP.

buzz-aci-max-join-tok=`queue_size` Ce paramètre permet de configurer la structure de données stockant les invocations utilisées dans le cadre d'un appel *join*.

B.1.2 Lancement du compilateur et organisation d'un projet

Cette section détaille l'exécution du compilateur ainsi que l'organisation d'un projet que nous avons utilisé. Par projet nous entendons l'ensemble des fichiers sources nécessaires (ADL, C, BIP, IDL) ainsi que les scripts de compilation.

La méthode présentée ici repose sur l'utilisation de l'outil GNU make[25]. Chaque projet est organisé comme suit :
- un répertoire racine, que nous appelons `root` dans la suite ;
- un sous-répertoire de `root` nommé `src/` contenant tous les fichiers sources spécifiques au projet ;
- un fichier `Makefile`, point d'entrée principal pour l'exécution du compilateur ;
- un (ou plusieurs) fichier `Makefile.<arch>`, avec `<arch>` l'architecture de la plate-forme matérielle employée. Ce fichier contient les paramètres spécifiques à la plate-forme matérielle.

Le listing B.4 donne un exemple de l'ensemble des fichiers d'un projet. Cette structure peut être automatique générée avec l'utilisation du programme nuptCinit.

```
geekdiner
|-- Makefile
|-- Makefile.unix
|-- Makefile.atm2561
|-- Makefile.msp430
`-- src
    |-- api
    |   `-- Chopstick.idl
    |-- geek.adl
    |-- geek.bip
    |-- geek.c
    `-- geekdiner.adl
```

Listing B.4 – Exemple de projet Buzz

Le fichier `Makefile` contient un ensemble de variables qu'il est possible de modifier pour paramétrer le lancement du compilateur. Ces variables influent directement sur les paramètres détaillés dans la section précédente ainsi que sur la génération de code assurée par le backend

25. GNU make est disponible sur http://www.gnu.org/software/make/.

C de Nuptse. Le listing B.5 donne un court extrait d'un `Makefile`. Toutes les variables sont documentées directement dans ce fichier.

```
DEBUG_LEVEL=ERROR

##
# How to use this Makefile
# ------------------------
...
# do you want buzz to generate debug info in BIP code ?
BUZZ_BIP_DEBUG=false

# do you want buzz to generate extra debug info in BIP code ?
# Such info includes full code annotation (to know which compiler
# part generated each piece of BIP code)
BUZZ_BIP_EXTRA_DEBUG=false

# Do you want extra debug output in simulation ? This includes color
# output for transitions in BIP simulator. Beware that this should
# not be used when doing exploration, as output will also occur.
BUZZ_BIP_SIMU_TRACE=false
...
# Do you want Nuptse's C backend to generate debug information or not
# (do you want gdb to display your code (currently, you wont be able
# to manipulate methods/attributes as their names are mangled during the
# process)
SOURCE_DEBUG=false
...
```

Listing B.5 – Extrait de `Makefile`

Un fichier `Makefile.unix` est donné en exemple dans le listing B.6. Il spécifie en particulier quelles sont les commandes à utiliser pour lancer le compilateur C (`CC`), l'éditeur de liens (`LD`) ainsi que leurs arguments (`CFLAGS` et `LDFLAGS`). Toutes les règles et variables implicites définies dans le manuel de GNU make s'appliquent et peuvent être surchargées dans ce fichier. Des variables permettent de passer des paramètres directement au compilateur, aussi bien pour l'implantation que la génération de code BIP :

- `NUPTSE_EXTRA_ARGS`, pour passer des paramètres utilisés pour l'implantation et la génération de code BIP ;
- `NUPTSE_C_EXTRA_ARGS`, pour passer des paramètres utilisés uniquement pour l'implantation ;
- `NUPTSE_BUZZ_EXTRA_ARGS`, pour passer des paramètres utilisés uniquement pour la génération de code BIP.

```
CC=gcc
LD=gcc
CFLAGS=-Wall -c -Os
LDFLAGS=
NUPTSE_EXTRA_ARGS=--global-typedefs-file=unix/kortex_types_unix.h
NUPTSE_C_EXTRA_ARGS=--buzz-leds-debug=false \
```

```
                -buzz-trap-component=unix.irq.lib.trap \
                -buzz-tick-component=unix.irq.lib.tick \
                -buzz-stack-size=8192 \
                -buzz-irqsafe-component=unix.irq.lib.irqsafe
NUPTSE_BUZZ_EXTRA_ARGS=
```

Listing B.6 – Extrait de `Makefile.unix`

Le lancement s'effectue simplement à l'aide de la commande `make` suivi d'une des cibles disponibles :

- `bip` : pour générer le modèle BIP du système. Le compilateur procédera à la génération d'un modèle BIP avec et sans informations supplémentaires pour la simulation (paramètre `buzz-bip-simu-state-trace`).
- `flat-elf` : pour générer un exécutable au format ELF [26]

Il existe d'autres cibles pour automatiquement charger l'exécutable créé sur la plate-forme matérielle et lancer le débugueur. Toutes les cibles disponibles sont documentées dans le fichier `Makefile`.

B.2 Outils annexes au compilateur Buzz.

Génération squelette BIP à partir de code nuptC Comme nous l'avons décrit dans la section 3.5.2, chaque composant doit contenir à la fois du code d'implantation en C et un modèle de son comportement en BIP. Pour assister cette tâche, l'outil `genbip` permet, à partir d'un code source en C contenu dans un composant Buzz, de créer un squelette en BIP qu'il suffit ensuite de compléter. Cet outil s'occupe en particulier des déclarations BIP des données, des ports. L'automate résultat est simpliste et consiste en une marguerite possédant un état spécifique à chaque méthode serveur.

Outils d'analyse des tailles binaires L'outil `comp-size` permet d'extraire la taille des différentes sections d'un exécutable au format ELF. Cet outil permet en particulier de donner des statistiques rapides sur les tailles relatives des différents éléments. Le listing B.7 donne un exemple d'affichage. Il permet aussi dans certains cas (cela dépend beaucoup des optimisations qui ont lieu lors de la génération de code C et de sa compilation) d'extraire des statistiques pour tous les composants Think présent dans un objet ELF.

```
    Total : bss ( 87288 ) text ( 36236 ), data ( 2748 ) [data+bss:  90036]
     Buzz : 87236 (99.9%)  16853 (46.5%)   1268 (100.%) [data+bss: 88504 (99.9%)]
Scheduler :   436 (.499%)   1213 (3.34%)    108 (8.51%) [data+bss:   544 (.614%)]
  AI (10) : 86800 (99.4%)  15640 (43.1%)   1160 (42.2%) [data+bss: 87960 (97.6%)]
   AVG AI :  8680 (9.94%)   1564 (4.31%)    116 (4.22%) [data+bss:  8796 (9.76%)]
```

Listing B.7 – Exemple d'affichage de l'outil `comp-size`

B.3 Autres développements

Nous présentons dans cette section deux développements suffisamment indépendant pour être utilisé en dehors du cadre de nos travaux. Le premier concerne un ensemble de composants ajouté

26. ELF, pour *Executable and Linkable Format*, est le format de fichier exécutable principalement utilisé par les outils GNU (compilateur, débugueurs, etc)

à la bibliothèque Kortex pour le développement de systèmes sur Unix. Le second introduit brièvement la bibliothèque de classe Java qui nous a permis de mettre en œuvre relativement facilement l'ensemble des manipulations de modèle BIP présenté dans cette thèse.

Nous avons aussi développé une série d'outils que nous ne détaillons pas dans ce manuscrit :

- un outil capable de créer une représentation graphique d'un diagramme de séquence à partir de l'observation d'une exécution du système sur la plate-forme cible. Ce programme s'appelle `seqdiag`.
- l'outil `bipplayer` se place au dessus du simulateur BIP pour afficher de manière plus exploitable l'évolution du système au cours de la simulation.
- un ensemble d'outils pour manipuler les résultats produits par une exploration exhaustive. Cela comprend l'isolation des interblocages et l'affichage de l'état du système (données et automates), le rejeu d'une séquence d'interactions

B.3.1 Environnement d'exécution Unix

Comme nous l'avons évoqué dans le chapitre 4, nous avons utilisé des implantations s'exécutant au dessus du système d'exploitation GNU/Linux, ceci dans le but de diagnostiquer et corriger plus facilement les problèmes dans les parties de code qui sont identiques entre les différentes cibles. Cela comprend notamment les codes des ordonnanceurs et des intercepteurs actifs. Néanmoins, l'environnement d'exécution offert par GNU/Linux et celui offert par les composants de Kortex pour l'AVR ou le WSN430 sont très différents en terme de fonctionnalités. En particulier, la gestion des contextes d'exécution au dessus de GNU/Linux se fait généralement à l'aide des *thread* POSIX. Ceux-ci étant pilotés par l'ordonnancement du noyau Linux, il est délicat d'ajouter l'ordonnanceur de Buzz par dessus pour obtenir un comportement identique à ce qui est observé avec le seul ordonnanceur de Buzz sur l'AVR ou le WSN430. Pour cette raison, nous avons développé un environnement simpliste permettant de gérer des contextes d'exécution au niveau d'un processus utilisateur au dessus de GNU/Linux. Cette solution au problème du manque de maîtrise de l'ordonnancement en espace utilisateur est relativement courante. Ainsi, l'ordonnanceur de Buzz ne gère pas le partage du processeur (cela reste bien sûr à la charge du noyau Linux), mais il gère le partage des ressources allouées au processus dans lequel il s'exécute. De même, il n'est pas possible simplement depuis un processus en espace utilisateur d'interagir directement avec le mécanisme matériel d'interruption, lui aussi géré par le noyau Linux. Ainsi, nous avons construit un mécanisme similaire utilisant les signaux POSIX pour simuler les interruptions.

Tous les composants ont été intégrés dans une branche de développement, appelée `buzz-unix` de la bibliothèque Kortex. Voici la liste complète de ces composants :

- le composant `unix.irq.lib.irqsafe` est utilisé pour masquer et démasquer les signaux, de manière similaire au masquage/démasquage des interruptions.
- le composant `unix.irq.lib.sigusr` est paramétré par un identifiant de signal. Il possède une interface cliente sur laquelle est invoquée une méthode `execute()` lorsque le signal par lequel il a été paramétré est reçu.
- le composant `unix.irq.lib.tick` active l'envoi périodique d'un signal au système. À chaque réception de ce signal, la méthode `execute()` de son interface cliente `timerhandler` est invoquée.
- enfin, le composant `unix.irq.lib.trap` permet de sauver/restaurer des contextes d'exécution.

B.3.2 Manipulation de modèles BIP

Comme nous l'avons évoqué en section 3.5.1 qui présente la création d'un modèle BIP à partir d'un modèle Buzz, nous disposons de code permettant la représentation d'un modèle BIP sous la forme d'un graphe (ASG). Ce type de représentation est adapté aux tâches de transformation et de génération de code. Nous fournissons avec le compilateur de Buzz un « package » Java nommé `bip` qui contient un ensemble de classes utilisées pour représenter un modèle BIP : `component` (atome et composite), `connector`, `port`, ... En plus de cela, nous mettons à disposition un ensemble de méthodes pour la manipulation de ce graphe : recherche de connecteurs suivant certains critères (« tous les connecteurs qui sont liés à un port spécifié », ...), renomage, manipulation des automates contenu dans les atomes, etc. Enfin, nous accompagnons cette bibliothèque d'un « parser » permettant le chargement de fichiers BIP et d'un visiteur [27] pour la création de fichiers BIP à partir d'un ASG.

Cette bibliothèque a été écrite avant les efforts de développements effectués sur BIP 2 et qui fournissent maintenant un canevas logiciel basé sur EMF (« Eclipse Modeling Framework »), autrement plus puissant que ce que nous avons développé.

27. Le « pattern » visiteur est un pattern classique en compilation.

Bibliographie

[AAD] SAE. Architecture Analysis & Design Language (standard SAE AS5506), September 2004, available at http ://www.sae.org.

[ABS] Annex Behavior Specification SAE AS5506.

[AHJ+09] Matthieu Anne, Ruan He, Tahar Jarboui, Marc Lacoste, Olivier Lobry, Guirec Lorant, Maxime Louvel, Juan Navas, Vincent Olive, Juraj Polajovic, Jacques Pulou, Marc Poulhiès, Stephane Seyvoz, Julien Tous, and Thomas Watteyne. Think : View-based support of non-functional properties in embedded systems. In *Proceedings of the 6th International Conference on Embedded Software and Systems (ICESS-09)*, 2009.

[Bas08] Ananda Shankar Basu. *Modélisation à base de Composants de Systèmes Temps réel Hétérogènes en BIP*. PhD thesis, UJF, dec 2008.

[BBF+08] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frederic Lang, and François Vernadat. Fiacre : an Intermediate Language for Model Verification in the Topcased Environment. In *ERTS 2008*, Toulouse France, 2008.

[BBNS09] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. D-finder : A tool for compositional deadlock detection and verification. In *CAV*, pages 614–619, 2009.

[BCF04] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26(5) :769–804, 2004.

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J-B. Stefani. The Fractal Component Model and Its Support in Java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11–12) :1257–1284, September 2006.

[BCS02] E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and dynamic software composition with sharing, 2002.

[BMP+07] Ananda Basu, Laurent Mounier, Marc Poulhiès, Jacques Pulou, and Joseph Sifakis. Using bip for modeling and verification of networked systems – a case study on tinyos-based networks. *Network Computing and Applications, IEEE International Symposium on*, 0 :257–260, 2007.

[BS08a] Simon Bliudze and Joseph Sifakis. The algebra of connectors—structuring interaction in BIP. *IEEE Transactions on Computers*, 57(10) :1315–1330, 2008.

[BS08b] Simon Bliudze and Joseph Sifakis. Causal semantics for the algebra of connectors. *Formal Methods in System Design*, 2008. (Submitted).

[CFLS05] J. Combaz, J-C. Fernandez, T. Lepley, and J. Sifakis. QoS Control for Optimality and Safety. In *Proceedings of the 5th Conference on Embedded Software*, September 2005.

[CLZ06] Elaine Cheong, Edward A. Lee, and Yang Zhao. Viptos : A graphical development and simulation environment for tinyos-based wireless sensor networks. Technical Report UCB/EECS-2006-15, EECS Department, University of California, Berkeley, Feb 2006.

[CM84] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4) :632–646, 1984.

[CRBS08] M.Yassin Chkouri, Anne Robert, Marius Bozga, and Joseph Sifakis. Translatin aadl into bip - application to the verification of real-time systems. In *Model Based Architecting and Construction of Embedded Systems*, 2008.

[DDM+07] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, and Qi Zhu. A next-generation design framework for platform-based design. In *DVCon 2007*, February 2007.

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1 :115–138, 1971.

[DS02] L. A. J. Dohmen and L. J. Somers. Experiences and lessons learned using uml-rt to develop embedded printer software. pages 475–484. 2002.

[ECZ06] Edward A. Lee Elaine Cheong and Yang Zhao. Joint modeling and design of wireless networks and sensor node software. Technical Report UCB/EECS-2006-150, EECS Department, University of California, Berkeley, November 2006.

[EJL+03] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming Heterogeneity : The Ptolemy Approach. *Proceedings of the IEEE*, 91(1) :127–144, January 2003.

[FH07] Peter Feiler and Jorgen Hansson. Flow Latency Analysis with the Architecture Analysis & Design Language (AADL). Technical Note ADA475162, CMU and SEI, 2007.

[FLV03] P.H. Feiler, B. Lewis, and S. Vestal. The SAE Architecture Analysis and Design Language (AADL) Standard : A basis for model-based architecture-driven embedded systems engineering. In *RTAS Workshop on Model-driven Embedded Systems*, pages 1–10, 2003.

[FSLM02] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. Think : A Software Framework for Component-Based Operating System Kernels. In *USENIX Annual Technical Conference*, 2002.

[GCW+02] Thomas Genssler, Alexander Christoph, Michael Winter, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Bastiaan Schönhage, Peter Müller, and Chris Stich. Components for embedded software : the pecos approach. In *CASES '02 : Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 19–26, New York, NY, USA, 2002. ACM.

[GF96] Daniel Gaudreau and Paul Freedman. Temporal analysis and object-oriented real-time software development : a case study with room/objectime. In *IEEE Real-Time Technologies and Applications Symposium*, pages 10–12, 1996.

[GH08] Olivier Gilles and Jérôme Hugues. Applying wcet analysis at architectural level. In Raimund Kirner, editor, *8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. also published in print by Austrian Computer Society (OCG) under ISBN 978-3-85403-237-3.

[GLvB+03] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The NesC language : A holistic approach to networked embedded systems. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[GSV02] Gregor Goessler and Alberto Sangiovanni-Vincentelli. Compositional modeling in metropolis. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *Proc. EMSOFT'02*, October 2002.

[HBS73] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proc. of the 3rd IJCAI*, pages 235–245, Stanford, MA, 1973.

[HLL+03] C. Hylands, E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, and H. Zheng. Ptolemy ii - heterogeneous concurrent modeling and design in java, 2003.

[JHR+07] Erwan Jahier, Nicolas Halbwachs, Pascal Raymond, Xavier Nicollin, and David Lesens. Virtual Execution of AADL Models via a Translation into Synchronous Programs. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software EMSOFT 2007*, pages 134 – 143, Salzburg Austria, 2007. ASSERT.

[KRP+93] Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael González Harbour. *A practitioner's handbook for real-time analysis*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[Lee06] Edward A. Lee. The problem with threads. *Computer*, 39(5) :33–42, 2006.

[LHJ+01] Edward A. Lee, C. Hylands, J. Janneck, J. Davis II, J. Liu, X. Liu, S. Neuendorffer, S. Sachs M. Stewart, K. Vissers, and P. Whitaker. Overview of the ptolemy project. Technical Report UCB/ERL M01/11, EECS Department, University of California, Berkeley, 2001.

[LNMB09] O Lobry, Juan Navas Mantilla, and Jean-philippe Babau. Optimizing component-based embedded software. In *2nd IEEE International Workshop on Component-Based Design of Resource-Constrained Systems, annual IEEE International Computer Software an Applications Conference, COMPSAC-09*, July 2009.

[Loi08] Frédéric Loiret. *Tinap : Modèle et infrastructure d'exécution orienté composant pour applications multi-tâches à contraintes temps réel souples et embarquées*. Thèse de doctorat, Université des Sciences et Technologies de Lille, Lille, France, may 2008.

[LP08] Olivier Lobry and Juraj Polakovic. Controlling the performance overhead of component-based systems. In Cesare Pautasso and Éric Tanter, editors, *Software Composition*, volume 4954 of *Lecture Notes in Computer Science*, pages 149–156. Springer, 2008.

[MB09] Joseph Sifakis Marius Bozga, Mohamad Jaber. Source-to-source architecture transformation for performance optimization in bip. Technical report, Verimag, Centre Équation, 38610 Gières, May 2009.

[Met] *MetaH Users Manual*.

[MKH03] Jamison Masse, Saehwa Kim, and Seongsoo Hong. Tool set implementation for scenario-based multithreading of uml-rt models and experimental validation. In *RTAS '03 : Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 70, Washington, DC, USA, 2003. IEEE Computer Society.

[MYC] MyCCM-HI : http://sourceforge.net/apps/trac/myccm-hi/.

[NAD+02] Oscar Nierstrasz, Gabriela Arévalo, Stéphane Ducasse, Roel Wuyts, Andrew P. Black, Peter O. Müller, Christian Zeidler, Thomas Genssler, and Reinier van den Born. A component model for field devices. In *CD '02 : Proceedings of the IFIP/ACM Working Conference on Component Deployment*, pages 200–209, London, UK, 2002. Springer-Verlag.

[PPRS06] Marc Poulhiès, Jacques Pulou, Christophe Rippert, and Joseph Sifakis. A methodology and supporting tools for the development of component-based embedded systems. In *Monterey Workshop*, pages 75–96, 2006.

[PS08] J. Polakovic and J.-B. Stefani. Architecting Reconfigurable Component-Based Operating Systems. *Journal of System Architecture*, 54(6) :562–575, 2008.

[Sel98] Bran Selic. Using uml for modeling complex real-time systems. In *LCTES '98 : Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 250–260, London, UK, 1998. Springer-Verlag.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time object-oriented modeling*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[Sif05] J. Sifakis. A framework for component-based construction. In *SEFM05, pages 293-300*, pages 293–300. IEEE Computer Society, 2005.

[SNY+04] John A. Stankovic, Prashant Nagaraddi, Zhendong Yu, Zhimin He, and Brian Ellis. Exploiting prescriptive aspects : a design time capability. In *EMSOFT '04 : Proceedings of the 4th ACM international conference on Embedded software*, pages 165–174, New York, NY, USA, 2004. ACM.

[SNYH04] John A. Stankovic, Prashant Nagaraddi, Zhendong Yu, and Zhimin He. Vest user's manual. Technical report, UVa, 2004.

[Sta01] John A. Stankovic. Vest - a toolset for constructing and analyzing component based embedded systems. In *EMSOFT '01 : Proceedings of the First International Workshop on Embedded Software*, pages 390–402, London, UK, 2001. Springer-Verlag.

[SZP+03] John A. Stankovic, Ruiqing Zhu, Ram Poornalingam, Chenyang Lu, Zhendong Yu, Marty Humphrey, and Brian Ellis. Vest : An aspect-based composition tool for real-time systems, 2003.

[Szy97] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Professional, December 1997.

[Szy03] Clemens Szyperski. Component technology : what, where, and how ? In *ICSE '03 : Proceedings of the 25th International Conference on Software Engineering*, pages 684–693, Washington, DC, USA, 2003. IEEE Computer Society.

[THI] Documentations du projet Think : http://think.ow2.org/documentations.html.

[Ves00] Steve Vestal. Formal Verification of the MetaH Executive Using Linear Hybrid Automata. In *IEEE Real Time Technology and Applications Symposium*, pages 134–144, June 2000.

[Wat08] Thomas Watteyne. *Energy-Efficient Self-Organization for Wireless Sensor Networks*. PhD thesis, INSA de Lyon, November 2008. number 2008-ISAL-0082.

[WBD+06] T. Watteyne, A. Bachir, M. Dohler, D. Barthel, and I. Aug<E9>-Blum. 1-hopMAC : An Energy-Efficient MAC Protocol for Avoiding 1-Hop Neighborhood Knowledge. In *International Workshop on Wireless Ad-hoc and Sensor Networks (IWWAN)*, 2006.

[WBDA08] Thomas Watteyne, Dominique Barthel, Mischa Dohler, and Isabelle Augé-Blum. WiFly : Experimenting with Wireless Sensor Networks and Virtual Coordinates. Research Report RR-6471, INRIA, 2008.

[WCkMT] P. Whittaker, M. Coldsmith, k. Macolini, and T. Teitelbaum. Model checking uml-rt protocols.

[WDABB08] Thomas Watteyne, Mischa Dohler, Isabelle Augé Blum, and Dominique Barthel. *Localization Algorithms and Strategies for Wireless Sensor Networks*, chapter Beyond localization : communicating using virtual coordinates. IGI Global, 2008.