Génération de système d'exploitation pour le ciblage de logiciel multitâche sur des architectures multiprocesseurs hétérogènes dans le cadre des systèmes embarqués spécifiques.

L. Gauthier

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

T H E S E

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Microélectronique

préparé au laboratoire TIMA
dans le cadre de **l'Ecole Doctorale EEATS**

présentée et soutenue publiquement

par

Lovic GAUTHIER

le 5 décembre 2001

Titre :

Génération de système d'exploitation pour le ciblage de logiciel multitâche sur des architectures multiprocesseurs hétérogènes dans le cadre des systèmes embarqués spécifiques

Directeur de thèse :

Ahmed Amine Jerraya

JURY

M. Guy MAZARÉ	, Président
M. Yves SOREL	, Rapporteur
M. Diederik VERKEST	, Rapporteur
M. Ahmed Amine JERRAYA	, Directeur de thèse
M. Giovanni DE MICHELI	, Examinateur
M. Christian BERTHET	, Examinateur

Remerciements

Une thèse est souvent considérée comme un «travail personnel». Cependant, c'est un travail qui ne peut être pleinement accompli sans le soutien pratique et moral d'amis et de collègues dévoués. Et du dévouement il en faut pour pouvoir supporter l'humeur lunatique d'un thésard en train de rédiger ! C'est pourquoi je tiens à remercier :

- Mon directuer de thèse, Ahmed Jerraya pour son accueil, son soutien et pour la liberté qu'il m'a laissée dans mon travail.
- Mes collègues de l'équipe SLS pour leur active collaboration et leur bonne humeur et en particulier mes compagnons d'arme pour l'application VDSL : mon voisin de bureau Damien Lyonnard, Gabriella Nicolescu, Amer Baghdadi, Wander Cesario et Yanick Paviot.
- Dhanistha Panyasak, Gabriella Nicolescu, Sonja Amadou, Guillaume Allègre, Damien Lyonnard, Wander Cesario, Ahmed Jerraya et Yanick Paviot qui m'ont beaucoup aidé dans les corrections du manuscrit.

Et bien sûr je remercie également ma famille, mes amis et tous les membre du laboratoire que je n'ai pas cités mais que je n'oublie pas.

Table des matières

Table des figures

Introduction

Contexte : la conception des systèmes embarqués spécifiques

Cette thèse a pour cadre la conception des systèmes électroniques embarqués spécifiques. C'est-à-dire les systèmes électroniques ayant des fonctionnalités spécifiques aux applications pour lesquelles ils sont employés.

Dans de tels systèmes, le logiciel prend une place de plus en plus importante, et sa complexité augmente d'autant : il n'est pas rare de nos jours de rencontrer des systèmes avec plusieurs processeurs de types différents[1], sur chacun se trouvant de nombreuses tâches logicielles. Pour pouvoir réaliser de tels logiciels, des systèmes d'exploitation sont couramment utilisés pour prendre en charge l'exécution concurrente et la communication de ces tâches.

Ces systèmes sont composés de plusieurs couches, comme l'illustre la figure 1. La couche la plus abstraite est la couche de l'application logicielle, elle communique avec une couche logicielle de plus bas niveau qui représente le système d'exploitation. Ensuite, vient le réseau de communication matériel, puis les composants matériels.

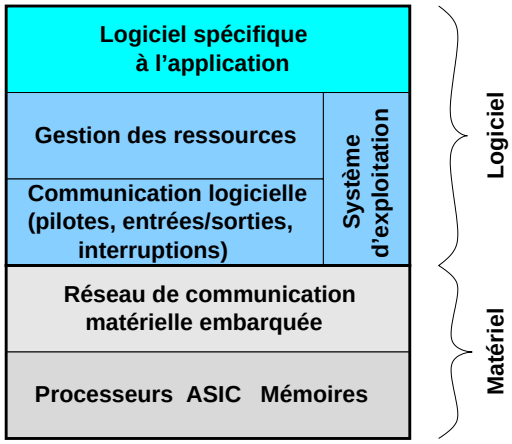

FIG. 1 – *Représentation d'un système embarqué sous forme de couches*

Les difficultés rencontrées pour la conception de tels systèmes

Les systèmes embarqués spécifiques étant très largement utilisés dans les applications récentes, il est important de pouvoir les développer rapidement. Cependant, de nombreuses

1. Il est courant d'utiliser des architectures constituées d'un processeur à usage général pour le contrôle et d'un processeur de traitement du signal pour les calculs.

difficultés se présentent lorsqu'il s'agit de développer de tels systèmes.

La complexité des systèmes embarqués

La première difficulté rencontrée par les concepteurs est la complexité des systèmes actuels : ils doivent fournir des fonctionnalités de plus en plus élaborées, et peuvent donc être constitués de composants logiciels ou matériels variés.

Les composants réutilisables permettent d'éviter aux concepteurs d'avoir à développer à chaque fois toutes les fonctionnalités requises. Cependant, pour qu'il y ait un gain de temps, l'intégration de ces composants au reste du système ne doit pas demander trop d'efforts aux concepteurs.

Une autre voie pour surmonter la complexité consiste à élever le plus possible le niveau d'abstraction des descriptions des systèmes à concevoir. Cette méthode, utilisable en conjonction avec la précédente, peut donner aux concepteurs une vision plus générale et plus simple des systèmes. Des méthodologies et des outils sont alors nécessaires pour guider le passage des descriptions de haut niveau aux réalisations finales.

La conception des interfaces entre les composants

Pour que le système fonctionne, les divers composants logiciels et matériels doivent être interconnectés. Il est rare que les composants puissent être directement reliés entre eux. Il est donc nécessaire de développer des interfaces qui adaptent leur communication.

La conception de ces interfaces est la clef de l'étape d'intégration des divers composants du système. C'est un travail long, fastidieux du fait de la très grande diversité des interfaces à créer.

Pour le logiciel, ces interfaces sont souvent comprises dans les systèmes d'exploitation. Elles sont spécifiques aux processeurs et plus généralement aux architectures : elles doivent donc être développées en partie ou en totalité pour chaque nouvelle application.

Vérification des systèmes embarqués spécifiques

La complexité et la diversité des composants des systèmes embarqués spécifiques rendent la validation du système complet très difficile. Les validations formelles requièrent souvent des modèles trop distincts, et des temps de calculs trop importants pour être utilisés pour un système complet, tandis que les validations par simulation ne peuvent pas être exhaustives.

Très souvent, faute de modèle de simulation de haut niveau, ces validations générales ne sont effectuées qu'au niveau du prototype. En cas de problème, il est souvent nécessaire de reprendre la conception depuis le début, ce qui occasionne des coûts et des délais de développement très importants.

Vers une automatisation plus importante de la conception

Comme nous l'avons vu dans la section précédente, la conception des systèmes embarqués spécifiques est une tâche de plus en plus difficile. Cette difficulté est si importante de nos jours, que ces systèmes sont plus limités par les habitudes de conception que par la technologie.

L'objectif général encadrant cette thèse est de proposer un flot de conception pour les systèmes embarqués spécifiques qui faciliterait le travail des concepteurs. Ce flot traite des architectures dans lesquelles les processeurs sont traités comme des composants destinés à

exécuter des fonctions bien définies. Il se concentre sur les communications et la conception des interfaces. Plus précisément ce flot propose :

- de décrire le système à un niveau d'abstraction élevé
- d'automatiser le plus possible les passages aux niveaux d'abstraction inférieurs, jusqu'à la réalisation finale
- de pouvoir simuler l'ensemble du système à toutes les étapes de sa conception

La figure 2 représente ce flot. Il part d'une description du système sous la forme de boîtes interconnectées au travers de ports par des canaux de communication. Chaque boîte représente un composant, et possède un comportement qui peut être caché. Cette description initiale abstraite est raffinée dans un premier temps pour déterminer les protocoles de communications et les diverses allocations. Puis les interfaces matérielles et logicielles des divers composants sont générées.

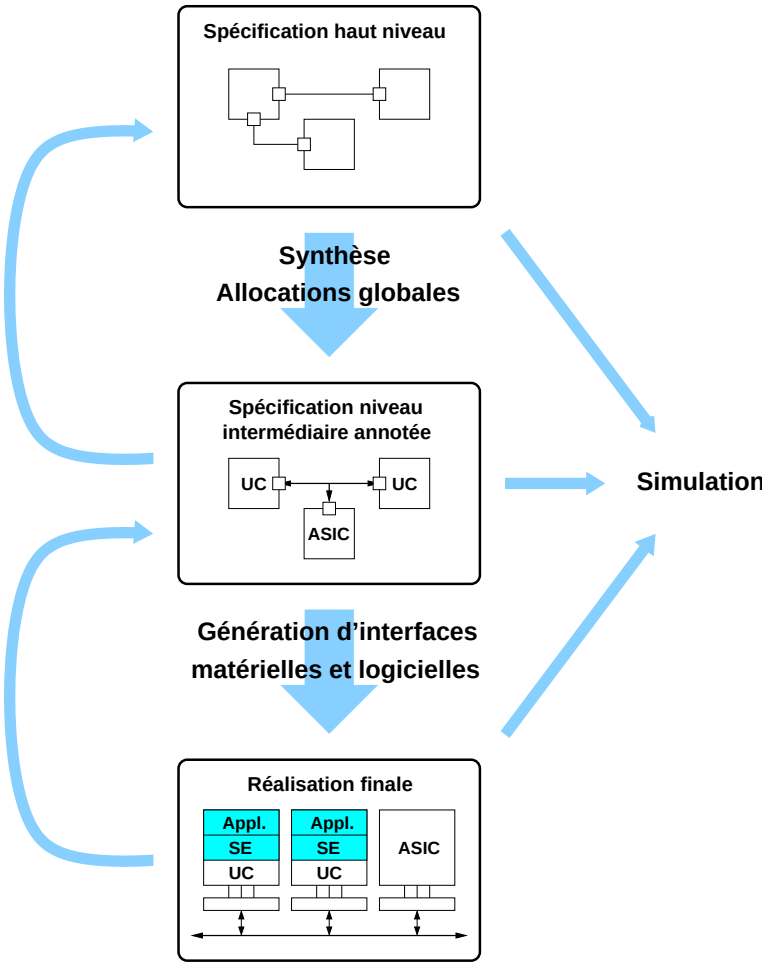

FIG. 2 – *Objectif : un flot de conception pour les systèmes embarqués spécifiques*

Contribution : ciblage automatique du logiciel avec génération de systèmes d'exploitation

Structure de la thèse

Cette thèse est décomposée en cinq chapitres. Cependant comme le montre la figure 3, il est possible de distinguer trois grandes étapes : la première est consacrée à la présentation de l'état de l'art sur les sujets connexes à la thèse, la deuxième propose un flot de ciblage logiciel avec génération de système d'exploitation, enfin la dernière montre son application sur un exemple concret.

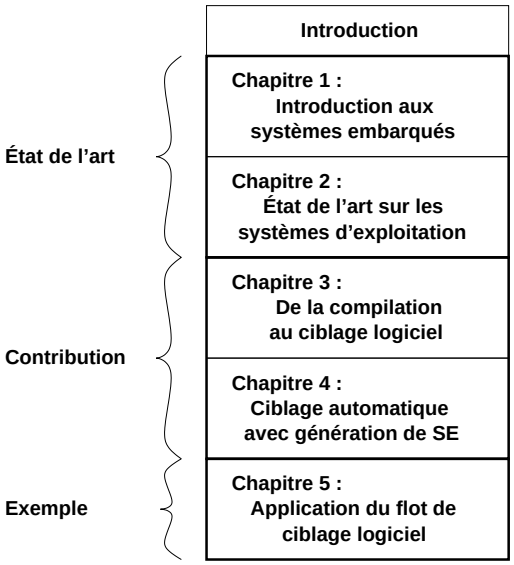

FIG. 3 – *Structure du mémoire de thèse*

Premier chapitre : Introduction aux systèmes embarqués

Ce chapitre a pour but de situer le travail de cette thèse. Pour ce faire, il présente dans une première section l'architecture et les méthodes des diverses générations des systèmes embarqués et notamment les systèmes sur une puce. Dans la deuxième section sont présentés quelques modèles qui peuvent être utilisés pour concevoir ces systèmes, c'est-à-dire les décrire, les simuler, les synthétiser ou les générer. Différentes méthodes de conception sont présentées dans la section suivante. Enfin la dernière section est consacrée au flot de conception défini dans l'équipe SLS du laboratoire TIMA.

Deuxième chapitre : État de l'art sur les systèmes d'exploitation

Le domaine couvert par le terme système d'exploitation est très vaste. Le but de ce chapitre est d'en préciser le sens dans le cadre du travail de cette thèse. La première section présente les systèmes d'exploitation en général en donnant quelques définitions, quelques propriétés et quelques exemples. La deuxième section est plus précisément consacrée aux systèmes d'exploitation pour les systèmes embarqués. Enfin la dernière section s'intéresse à la conception du logiciel pour systèmes embarqués avec l'utilisation de systèmes d'exploitation.

Troisième chapitre : De la compilation au ciblage logiciel

Ce chapitre définit la notion de ciblage logiciel. Une première section rappelle les principes de base de la compilation. La section suivante présente le ciblage logiciel et met en évidence ce qui le distingue de la compilation. La troisième section énumère les types de représentation utilisés pour le ciblage logiciel, et la dernière section donne les spécificités du ciblage logiciel pour les architectures embarquées spécifiques.

Quatrième chapitre : Ciblage automatique avec génération de systèmes d'exploitation

Ce chapitre décrit l'outil de génération de systèmes d'exploitation développé dans le cadre de la thèse. La première section décrit les représentations utilisées pour la spécification d'entrée et pour la bibliothèque de génération. La seconde section présente l'outil de ciblage.

Cinquième chapitre : Application du flot de ciblage logiciel

Ce dernier chapitre illustre l'utilisation du flot de ciblage pour une application. Cette application, une partie d'un modem VDSL, est présentée dans la première section. La deuxième section présente la bibliothèque de système d'exploitation qui a été développée pour mener à bien ce projet. Enfin la dernière section évalue les résultats obtenus.

Chapitre 1

Les systèmes embarqués spécifiques

Sommaire

Par définition un système embarqué est un système inclus dans un autre système. De par cette définition, le domaine des systèmes embarqués est très vaste : il dépasse largement le cadre de cette thèse. Ce chapitre précise la partie de ce domaine qui nous intéresse : il propose une définition de systèmes embarqués restreinte à l'électronique et à l'informatique. Puis il présente en détail les systèmes embarqués spécifiques d'une part et les systèmes sur une puce d'autre part, qui sont le véritable cadre de la thèse. Ces précisions étant effectuées, ce chapitre se consacre à la présentation d'un état de l'art pour la conception de tels systèmes : quelles sont leurs principales évolutions, comment peuvent-ils être modélisés et comment sont-ils conçus. Pour clore cet état de l'art, le flot général de conception développé au sein de l'équipe SLS du laboratoire TIMA est décrit. Le travail de cette thèse est une étape de ce flot.

1.1 Les systèmes embarqués concernés par la thèse

Les systèmes électroniques sont de plus en plus présents dans la vie courante. Les ordinateurs et micro-ordinateurs sont des systèmes électroniques bien connus. Mais l'électronique se

trouve maintenant embarquée dans de très nombreux objets usuels : les téléphones, les agendas électroniques, les voitures. Ce sont ces systèmes électroniques enfouis dans les objets usuels qui sont appelés systèmes embarqués.

1.1.0.1 Caractéristiques des systèmes embarqués

Les systèmes embarqués ont pour but de permettre aux objets usuels de réagir à l'environnement. Ils peuvent aussi apporter une interface avec l'utilisateur. La structure de base de ces systèmes est donnée sur la figure 1.1 : l'environnement est mesuré par divers capteurs[1]. L'information des capteurs est échantillonnée pour être traitée par le coeur du système embarqué. Puis le résultat du traitement est converti en signaux analogiques qui génèrent les actions sur l'environnement (afficheur d'informations pour l'utilisateur, actionneurs, transmission d'information, etc.).

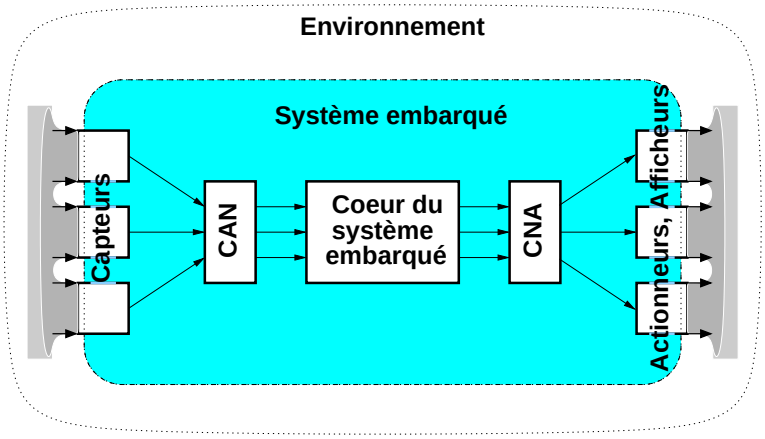

FIG. 1.1 – *Un système embarqué dans son environnement*

1.1.0.2 Les contraintes des systèmes embarqués

Les systèmes embarqués possèdent des contraintes particulières par rapport aux autres systèmes électroniques :

- l'encombrement : ils doivent souvent être transportés et doivent donc être de taille réduite.
- la consommation : étant transportés, ils n'ont souvent accès qu'à des ressources limitées en énergie (piles, batteries) ; il est donc important qu'ils consomment le moins possible.
- le temps : ils sont souvent employés dans des applications ou le temps est important.
- la sécurité : ils sont souvent employés dans des domaines tels que l'aéronautique.

1.1.1 Les systèmes embarqués spécifiques

Lorsqu'un système est utilisé pour une tâche bien précise, il est souvent plus efficace et économe s'il est spécifique à cette fonctionnalité que s'il est général. Les systèmes embarqués sont très souvent utilisés dans ces conditions, et il est donc intéressant qu'ils soient conçus spécifiquement pour les fonctions qu'ils doivent remplir. Notamment, les contraintes citées

1. L'utilisateur est aussi assimilé à l'environnement.

dans la section précédente ne peuvent souvent être respectées que si le système est conçu dès le départ pour pouvoir les respecter. Il est donc de par sa conception même spécifique.

Le problème qui se pose alors est que, pour chaque nouvelle fonctionnalité, il faudra concevoir un système spécifique différent de ceux déjà existants. Le flot de conception, présenté dans la section 1.4, et dont le travail de cette thèse fait partie, s'intéresse à ce type de système embarqué, et tout particulièrement aux systèmes sur une puce, présentés dans la section suivante.

1.1.2 Les systèmes monopuces

Les progrès réalisés par les fondeurs de circuits permettent maintenant d'envisager l'intégration sur une même puce d'un système embarqué complet. Ces systèmes monopuces (*SoC: System on a Chip*) apportent des changements importants dans les flots de conception classiques [61].

Dans les systèmes électroniques classiques, une grande difficulté était les problèmes électriques dûs aux grandes dimensions des cartes électroniques. Ces problèmes limitaient notamment la vitesse de communication, ce qui faisait qu'il pouvait y avoir sur une carte des composants allant à grande vitesse, mais ne pouvant communiquer entre eux qu'à vitesse réduite. C'était particulièrement critique pour la communication entre le processeur et la mémoire : quelle que soit la vitesse du processeur, il devait aller lire ses instructions en mémoire. Pour pallier ces problèmes il était nécessaire d'utiliser des caches. L'inconvénient des caches est qu'ils sont des systèmes très complexes à étudier et qu'ils sont de gros facteurs d'indéterminisme. Avec les systèmes monopuces, la communication reste toujours un goulet d'étranglement, car elle est très consommatrice de surface, mais avec un facteur bien moindre. En effet le fait d'avoir sur la même puce l'ensemble du système raccourcit les chemins de communication et facilite la construction d'architecture s'accordant aux localités de calcul de l'application. Ces facilités permettent ainsi souvent de s'affranchir des caches. Les systèmes monopuces sont aussi moins encombrants et surtout, ils peuvent consommer moins : en effet les données doivent transiter par des chemins beaucoup moins longs, l'énergie nécessaire à cette transmission est donc plus faible.

Les systèmes monopuces apportent aussi des changements dans les habitudes de conception. Notamment, la frontière entre le logiciel et le matériel n'est plus aussi nette : en effet avec les anciens systèmes le matériel était déjà conçu lorsqu'il fallait concevoir le logiciel. Avec les systèmes monopuces, les deux doivent être conçus en même temps. Cela augmente la complexité de la conception, mais cela offre aussi plus de liberté [63] : chaque partie peut être réalisée en logiciel, en matériel, ou de manière mixte.

1.1.3 Architecture des systèmes embarqués

Dans cette section nous présentons les architectures supportées par trois générations d'outils de conception.

1.1.3.1 Les systèmes embarqués de première génération

Partie matérielle des systèmes embarqués de première génération

Les premiers systèmes embarqués supportés par des outils tels que **COSYMA** [42] et **vulcan** [99] étaient très simples : ils étaient constitués par un processeur qui contrôlait un nombre restreint de CIAS (Circuits Intégrés à Applications Spécifiques) ou *ASIC* qui étaient

appelés périphériques. Cette architecture est représentée sur la figure 1.2. Les communications de cette architecture se situent au niveau du bus du processeur et sont de type maître/esclave : le processeur est le maître et les périphériques sont les esclaves.

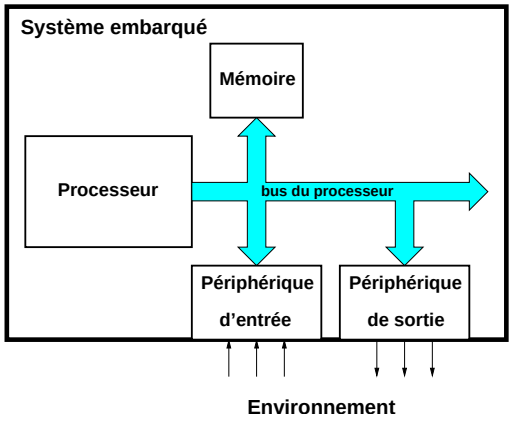

FIG. 1.2 – *Architecture embarquée de première génération*

Les périphériques de ces architectures étaient essentiellement des capteurs et des actionneurs (contrôleurs magnétiques, sorties, etc.). Le processeur est dédié au calcul et au contrôle de l'ensemble du système.

Les microcontrôleurs assemblent sur une même puce le processeur et des périphériques.

Partie logicielle des systèmes embarqués de première génération

Ne devant pas exécuter de nombreuses opérations simultanées (le nombre de périphériques et de fonctions étant restreint), les parties logicielles étaient constituées d'un seul programme. La réaction aux événements était effectuée par le biais de routines de traitement d'interruptions.

Cette partie logicielle était décrite directement en langage d'assemblage ce qui permettait d'obtenir un code efficace et de petite taille.

1.1.3.2 Les systèmes embarqués de deuxième génération

Les premiers systèmes embarqués ne pouvaient fournir que des fonctions simples ne requérant que peu de puissance de calcul. Leur architecture ne peut pas supporter les fonctionnalités requises pour les systèmes embarqués actuels à qui il est demandé non seulement d'effectuer du contrôle, mais aussi des calculs complexes tels que ceux requis pour le traitement numérique du signal. De nouveaux outils tels que **N2C** [114][28] (dans ses nouvelles version) permettent de traiter des architectures plus complexes.

Partie matérielle des systèmes embarqués de deuxième génération

L'architecture des systèmes embarqués de deuxième génération est composée d'un processeur central, de nombreux périphériques, et souvent de quelques processeurs annexes contrôlés par le processeur central. Le processeur central est dédié au contrôle de l'ensemble du système. Les processeurs annexes sont utilisés pour les calculs ; il s'agit souvent de processeurs spécialisés comme les *DSP*. Une telle architecture est représentée sur la figure 1.3. Dans une telle architecture, plusieurs bus de communication peuvent être nécessaires : chaque processeur dispose de son bus de communication.

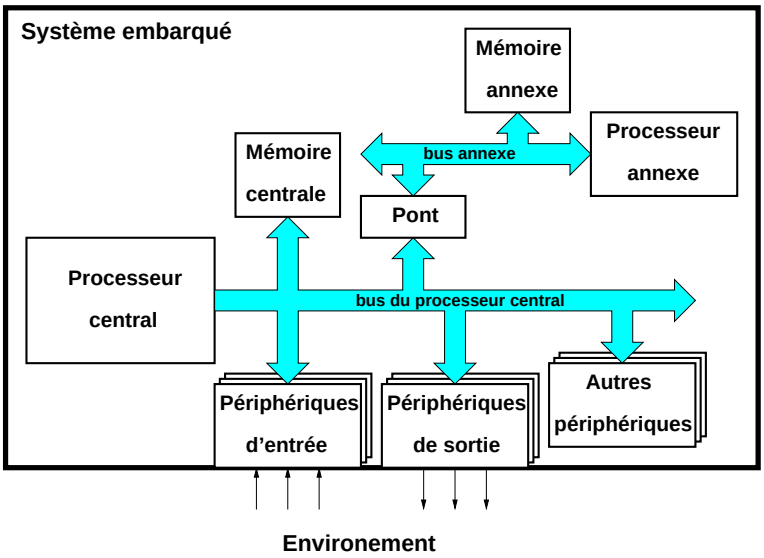

FIG. 1.3 – *Architecture embarquée de deuxième génération*

Partie logicielle des systèmes embarqués de deuxième génération

La partie logicielle des systèmes embarqués de deuxième génération est répartie sur plusieurs processeurs (le processeur principal et les processeurs annexes). Les systèmes actuels sont trop complexes pour pouvoir être gérés par un unique programme sur le processeur principal. Il est donc nécessaire d'avoir une gestion multitâche sur ce processeur, et un système d'exploitation est couramment employé dans ce but.

Le logiciel du processeur central est souvent décrit dans un langage de haut niveau tel que le C. Le logiciel des processeurs annexes est souvent trop spécifique pour être entièrement décrit dans un langage de haut niveau, et l'utilisation des langages d'assemblage est nécessaire.

1.1.3.3 Les systèmes embarqués de troisième génération

Les progrès de l'intégration permettent d'envisager des circuits pouvant contenir plusieurs milliers de portes [23]. Il devient donc techniquement possible de fabriquer des systèmes embarqués pouvant remplir toutes les fonctionnalités souhaitées.

Parties matérielles des systèmes embarqués de troisième génération

Pour pouvoir supporter conjointement les besoins en puissance et en flexibilité, ces architectures comprennent de plus en plus de processeurs, qui peuvent chacun se comporter en maître : l'architecture couramment utilisée, basée sur un processeur central contrôlant le reste du système, n'est donc plus suffisante.

Alors qu'auparavant le goulet d'étranglement était les ressources en calcul, de nos jours il est situé plutôt au niveau des communications. Ce sont elles qui définissent désormais l'architecture, et non plus les ressources de calcul [61]. La figures 1.4 donne des exemples d'architectures centrées sur les communications. Dans cette figure, tous les éléments (processeur, *ASIC* ou mémoires), sont traités de la même manière. Le premier exemple est basé sur des communications par bus : ce modèle de communication consomme peu de surface, mais risque de devenir un goulet d'étranglement. Le deuxième est basé sur des communications en barres croisées très performantes mais aussi très coûteuses en surface. Le troisième exemple donne une solution intermédiaire, par réseau commuté. Enfin le dernier exemple montre qu'il est

possible de mixer plusieurs modèles de communication, et d'apporter de la hiérarchie dans l'architecture.

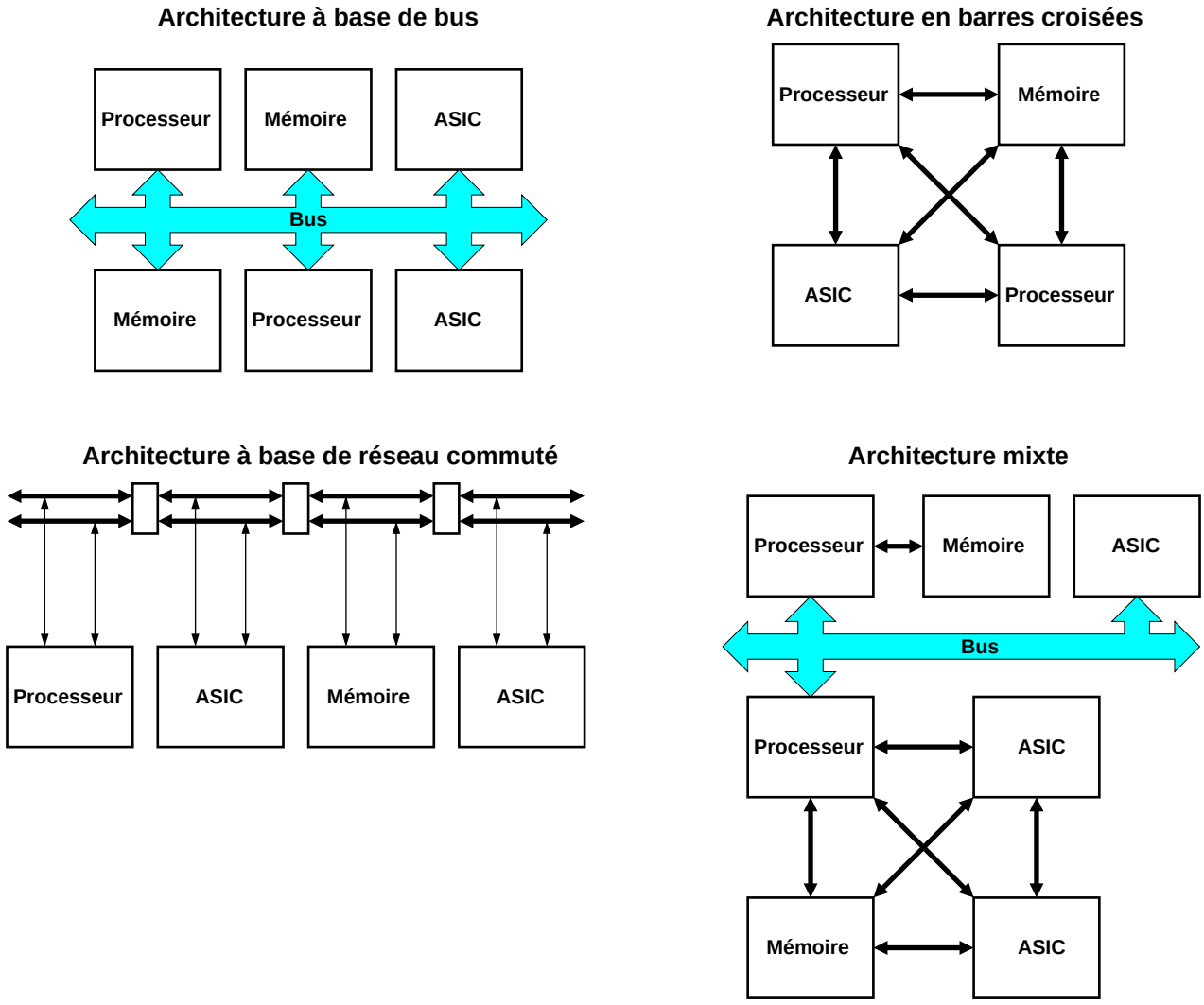

FIG. 1.4 – *Architectures embarquées de troisième génération*

Autour de ce modèle d'architecture centré sur les communications, se greffent les autres modèles d'architectures : architecture des éléments de calcul et des mémoires. L'architecture des éléments de calcul consiste à définir quels sont les éléments principaux et quels sont leurs périphériques de manière à les grouper dans une architecture locale. L'architecture des mémoires sert à définir quelles sont les mémoires locales à un groupe et quelles sont celles qui seront partagées.

Parties logicielles des systèmes embarqués de troisième génération

Les parties logicielles ont beaucoup gagné en importance dans les systèmes embarqués [118]. Plusieurs systèmes d'exploitation sont parfois nécessaires pour les divers processeurs de l'architecture. De plus, la complexité et la diversité des architectures possibles font qu'il devient de plus en plus nécessaire d'abstraire les tâches logicielles des détails du matériel. Toute cette complexité est donc reportée dans les systèmes d'exploitation, qui deviennent de plus en plus complexes.

Cette complexité logicielle et matérielle entraîne de nombreuses alternatives. En particulier,

l'aspect multiprocesseur apporte des alternatives pour les systèmes d'exploitation : il peut y avoir un seul système pour tous les processeurs (solution difficilement applicable lorsque les processeurs sont hétérogènes), ou il peut y avoir un système par processeur (solution qui peut être plus coûteuse).

1.2 Représentation des systèmes embarqués

Dans cette section, nous allons donner un aperçu rapide des différentes représentations utilisées pour décrire le logiciel ou le matériel dans les systèmes embarqués.

Dans un premier temps nous étudierons les représentations structurelles. Puis nous nous intéresserons aux représentations comportementales, avant de présenter quelques autres types de représentations. Ce chapitre se terminera avec la présentation de quelques représentations bien connues.

1.2.1 Représentations structurelles

Les représentations structurelles permettent de décrire comment est constitué un système : quels sont ses composants, quelles sont leurs relations, où sont leurs positions et leurs interconnexions.

Nous allons présenter dans cette section deux types de représentations structurelles : les représentations topologiques, et les représentations relationnelles.

1.2.1.1 Représentations topologiques

Introduction

Dans ces représentations chaque élément correspond à un composant dans le système réel. Ce type de représentation est tout à fait adapté pour décrire l'architecture d'un circuit. De ce fait de très nombreux langages de description de circuit proposent ce type de représentations ; c'est le cas pour **VHDL** [8] ou **EDIF** [38]. Ils peuvent aussi être utilisés pour des descriptions de haut niveau, comme c'est le cas pour le **SDL** [58].

Les concepts pour les représentations topologiques

Il existe de très nombreuses représentations topologiques ; cependant, elles utilisent en général les mêmes concepts [85]. Les éléments de base utilisés sont : les modules, les ports et les canaux (les noms peuvent varier suivant les représentations).

- Les modules représentent des sous-systèmes. Ils peuvent communiquer entre eux par l'intermédiaire des ports et des canaux.
- Les ports représentent les interfaces des modules, c'est-à-dire que c'est par l'intermédiaire de leurs ports que les modules interagissent avec le monde extérieur.
- Les canaux relient des ports entre eux. Ils matérialisent l'interaction entre les modules : ainsi deux modules connectés par un canal peuvent interagir, tandis que deux modules non connectés ne le peuvent pas.

Certaines représentations topologiques enrichissent ces concepts de base en les rendant hiérarchiques. C'est très souvent le cas pour les modules : dans des langages tels que le **VHDL** [8], les modules peuvent contenir des sous-modules. La figure 1.5 donne un exemple de ce type de représentation : deux modules **M1** et **M2** contiennent des sous-modules **m1**, **m2** et **m3**. Ces modules communiquent au travers des canaux **C1** et **C2** pour **M1** et **M2** et des canaux

c1 à **c5** pour **m1**, **m2** et **m3**, par l'intermédiaire des ports **P1** à **P4** pour **M1** et **M2**, et des ports **p1** à **p6** pour **m1**, **m2** et **m3**.

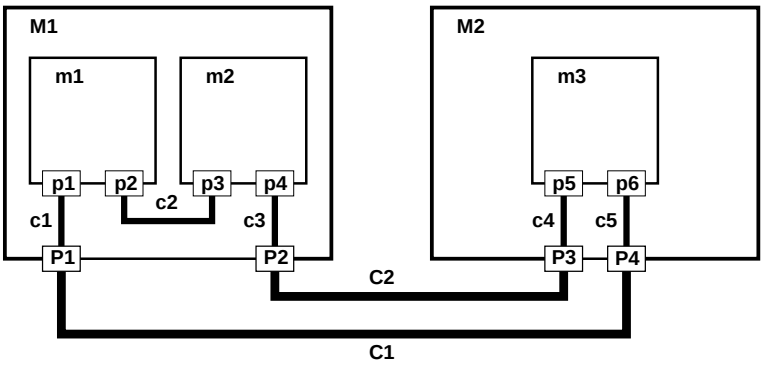

FIG. 1.5 – *Exemple de représentation topologique d'un système*

Ces définitions, très générales, peuvent être utilisées avec des sémantiques très différentes, et notamment des niveaux d'abstraction très différents. Par exemple, en **VHDL** [8], les modules représentent des circuits, les ports sont leurs plots d'entrée ou de sortie, et les canaux sont les fils les reliant. Ainsi les ports et les canaux respectivement, représentent uniquement les points de communication et les médias véhiculant la communication. De plus, les interactions sont limitées à des transmissions de données élémentaires. En **SDL** par contre, les canaux encapsulent à la fois un média et un comportement. En outre, les données transmises sont des messages complexes.

Les instances

Certaines représentations topologiques ajoutent un autre concept permettant de matérialiser explicitement les composants : il s'agit du mécanisme d'instanciation.

Une instance est l'utilisation d'un élément. Cela permet d'utiliser plusieurs fois un élément avec des paramètres particuliers sans avoir à dupliquer les informations de base, et sans ambiguïté entre les diverses instances.

Les instances sont souvent utilisées dans les descriptions de matériel tel que **VHDL** [8] : elles permettent notamment de définir des bibliothèques de composants (modules), avec lesquelles l'utilisateur peut décrire un circuit. Il suffit pour cela d'instancier les composants souhaités : l'ensemble des instances obtenu est la description du circuit.

1.2.1.2 Représentations relationnelles

Introduction

Dans les représentations relationnelles un système est défini par des propriétés et des relations avec d'autres systèmes. Elles ne décrivent pas la décomposition du système comme les représentations topologiques mais plutôt la sémantique du système.

Approche objet

L'approche objet [29][19] est un concept de modélisation plébiscité dans le monde de la programmation. Elle représente les systèmes sous la forme d'**objets**.

Un **objet** est défini par ses **attributs**, et ses **méthodes** (qui symbolisent son comportement). L'**état** d'un **objet** correspond aux valeurs de ses attributs à un instant donné. Chaque **objet** dispose aussi d'une **identité**, indépendante de son **état**, et qui le distingue de tous les autres **objets**.

Les **objets** sont regroupés en **classes** : une **classe** décrit le domaine de définition d'un ensemble d'**objets**. Ainsi tout **objet** appartient à au moins une classe.

Pour construire des **objets**, et décrire les relations qu'ils peuvent avoir, l'approche objet apporte les paradigmes suivants :

- l'**encapsulation** : la **spécification** et la **réalisation** d'une **classe** sont séparées. La **spécification** donne les **attributs** des **objets** de la **classe**. Cette **spécification** précise aussi la visibilité de ces **attributs** vis à vis des autres objets. La **réalisation** décrit le comportement de l'objet, elle est cachée aux autres objets.
- l'**héritage** : une **classe** peut **hériter** d'une autre **classe**, elle dispose alors automatiquement des attributs et méthodes de cette dernière. Il est possible de définir des **attributs** ou **méthodes** supplémentaires pour la **classe** fille.
- le **passage de messages** : les **objets** interagissent entre eux par le biais de **messages**.
- le **polymorphisme** : le **polymorphisme** permet avec un même **message** de provoquer des opérations différentes suivant les situations. Par exemple l'envoi d'un **message** à deux **objets** différents pourra produire des réactions différentes.

1.2.2 Représentations comportementales

Les représentations comportementales décrivent comment le système fonctionne, c'est-à-dire comment il évolue, comment et pourquoi il réagit, etc. Il existe une infinité de comportements différents, et le nombre de descriptions comportementales est lui-même très grand. Cette section va en présenter quelques-unes souvent utilisées pour décrire des systèmes embarqués.

1.2.2.1 Équations

La première approche pour représenter le comportement d'un système est de suivre une approche «physique» en le représentant par une série d'équations dont les paramètres représentent le monde extérieur : cela peut être le temps, une tension, etc.

Ces modélisations sont souvent présentées sous la forme d'un graphe flot de données [72] comme sur la figure 1.6. Dans un tel graphe, les nœuds représentent des opérateurs, et les arcs représentent des dépendances de données.

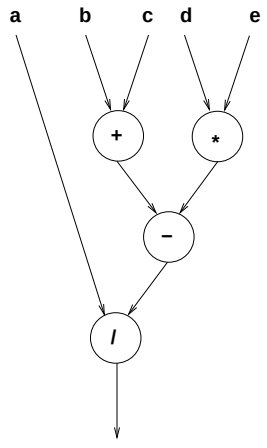

FIG. 1.6 – *Exemple de représentation flot de données*

Équations différentielles

Les modèles à base d'équations différentielles sont similaires à des modèles physiques. L'utilisation de tels modèles revient à résoudre ces équations. Si nous prenons l'exemple d'un filtre il pourra être modélisé par des équations différentielles en fonction du temps et du signal d'entrée. Il est alors possible de résoudre ces équations avec diverses valeurs du temps et du signal d'entrée pour obtenir le résultat en sortie.

Ce type de modélisation est couramment utilisé dans la modélisation des circuits analogiques comme **Spice** [109]. En effet, ils permettent une modélisation fine du comportement «physique» d'un système. Il est par contre inadapté pour les descriptions logiques (électronique numérique) et logicielles. De plus, il demande des temps de calculs très élevés, ce qui limite son utilisation.

Équations aux différences finies

Les équations aux différences finies sont les échantillonnées des équations différentielles.

Elles sont plus simples à calculer que les équations différentielles. Par contre elles requièrent l'utilisation d'une horloge générale pour l'échantillonnage, ce qui rend difficile la modélisation des irrégularités temporelles comme les événements. Les systèmes interactifs, basés sur les événements, ne peuvent donc pas être couverts par ces modèles.

Cette représentation est couramment utilisée en traitement du signal. Elle peut être réalisée en logiciel notamment avec les processeurs *DSP*. **Simulink** [83] de **Matlab** [53] est un simulateur basé sur cette représentation souvent utilisé pour le traitement du signal.

1.2.2.2 Machines à états finis

Les machines à états finis consistent en un graphe orienté, dont les nœuds représentent les états du système, et dont les arcs représentent les transitions.

Les machines de Moore et de Mealy

Les deux types de machines à états finis sont les machines de Moore et les machines de Mealy.

- Pour les machines de Moore les actions ne dépendent que des états. L'exemple de la figure 1.7 présente un algorithme de calcul et sa représentation en machine de Moore.
- Pour les machines de Mealy les actions dépendent des états et des entrées. L'exemple de la figure 1.8 présente le même algorithme de calcul que précédemment et sa représentation en machine de Mealy.

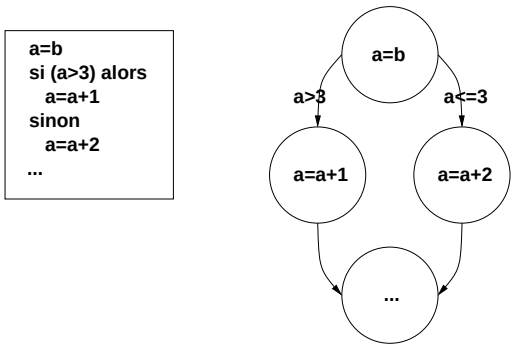

FIG. 1.7 – *Exemple de machine de Moore*

Remarque : il a été démontré [11] que ces deux représentations étaient équivalentes.

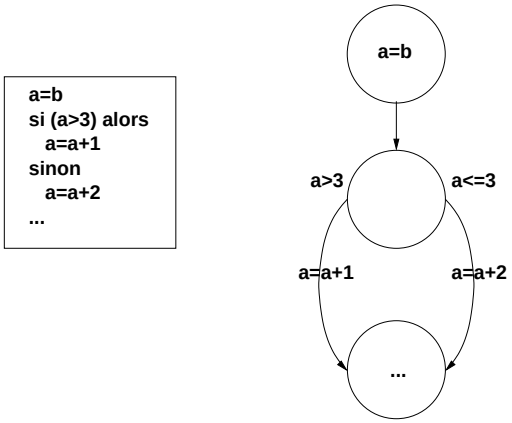

FIG. 1.8 – *Exemple de machine de Mealy*

Domaine d'application des machines à états finis

Dans leur forme de base, ces représentations se prêtent bien aux analyses formelles [16]. Elles sont aussi facilement réalisables en matériel qu'en logiciel [73][25]. C'est aussi un modèle qui peut être exécutable, ce qui permet de faire des simulations, en plus des vérifications formelles. Cependant, elles ne permettent pas d'exprimer des mécanismes tels que la récursivité. De plus, le nombre d'états peut devenir très élevé, même pour décrire des comportements de faible complexité [73].

Les machines à états finis ont souvent étés étendues, avec l'ajout de variables, ou d'une hiérarchie par exemple. Ces extensions augmentent le pouvoir d'expression de ces représentations, cependant, elles en réduisent les possibilités d'analyse formelle.

Dans l'industrie électronique numérique elles sont utilisées pour décrire les parties contrôles de circuits. Elles sont aussi utilisées dans l'industrie logicielle pour décrire les pilotes de périphériques, les interfaces homme-machine, etc.

1.2.2.3 Les réseaux de Pétri

Les réseaux de Pétri [91] représentent les systèmes comme un ensemble de places et de transitions. Chaque place peut contenir des jetons. L'état d'un réseau de Pétri est donné par un marquage, c'est-à-dire un nombre de jetons donné pour chaque place. Les aspects dynamiques sont modélisés par les franchissements des transitions par les jetons.

Fonctionnement d'un réseau de Pétri

Dans un réseau de pétri, le passage d'un état à l'autre est effectué en traitant toutes les transitions du réseau. Lorsqu'une transition est franchie, un jeton est supprimé de chaque place amont, et un jeton est ajouté dans chaque place aval. Pour qu'une transition soit franchie, il faut qu'il y ait au moins un jeton dans chaque place amont. La figure 1.9 donne un exemple de réseau de pétri dans plusieurs états consécutifs. Dans la figure, il y a un indéterminisme au niveau de l'état 3 : une seule transition ne peut être franchie pour la première place.

Il existe de nombreuses extensions à ce modèle simple. Par exemple, une extension possible consiste à ajouter des poids aux transitions. La figure 1.10 illustre cette extension : une transition ne peut être franchie que si chaque place amont possède au moins autant de jetons que le poids amont. Au franchissement de la transition ces jetons sont supprimés, et dans chaque place aval, autant de jetons que le poids aval sont ajoutés. Dans cette figure nous avons repris le même réseau de Pétri que la figure 1.9, mais cette fois ci, les poids font en sorte que l'état

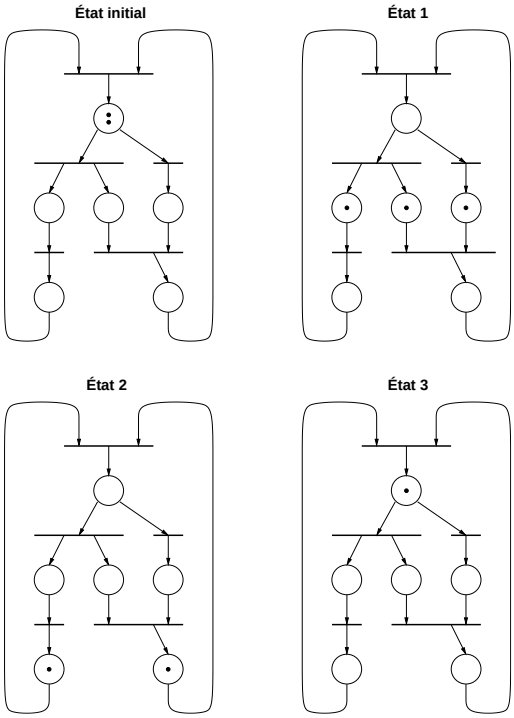

FIG. 1.9 – *Exemple de réseau de Pétri*

3 est l'état initial.

Une autre extension consiste à associer un temps à chaque transition, ce qui fait du réseau de Pétri un modèle de simulation temporelle. Il est également possible d'associer des attributs aux jetons et de conditionner leur franchissement des transitions par ces attributs.

Domaine d'application des réseaux de Pétri

Les réseaux de Pétri permettent de représenter des systèmes à très grand nombre d'états sans que le modèle soit volumineux (contrairement aux machines à états finis). Il est possible d'effectuer des vérifications formelles sur ces modèles, cependant certaines propriétés requièrent beaucoup de calculs pour être vérifiées (par exemple, vérifier si un réseau est vivant ou s'il a un nombre fini d'états [73]).

Les réseaux de Pétri sont employés pour la simulation et l'analyse dans des domaines variés (développement, production, etc.). Ils sont aussi utilisés pour la synthèse de circuits localement synchrones et globalement asynchrones [92].

1.2.3 Modèles basés sur la communication

Ce sont des modèles mixtes structurels et comportementaux : le système est décomposé en plusieurs unités comportementales communiquant entres elles. Ces unités peuvent elles-mêmes être des systèmes. Les communications expriment le comportement global du système, ainsi que sa structure relationnelle. Les unités comportementales expriment la structure topologique du système.

Pour le comportement, ces modèles associent des actions à des communications (alors appelées événements) : les actions ne sont exécutées que lorsque l'événement correspondant arrive. Un ensemble d'actions peut être associé à un ensemble d'événements sous le nom de processus.

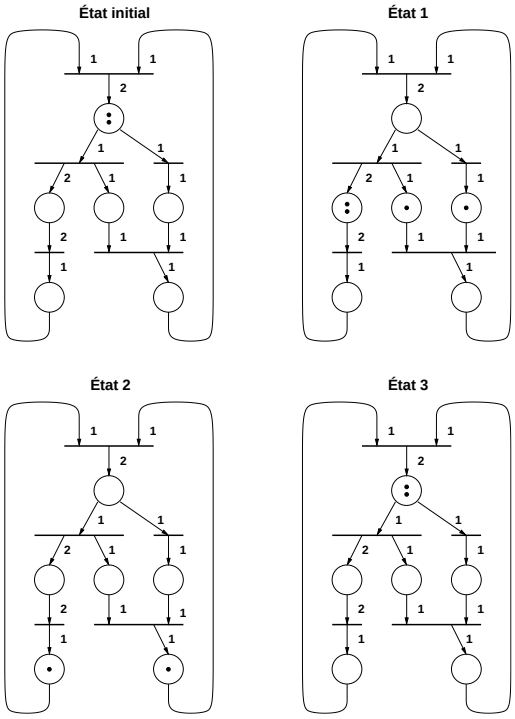

FIG. 1.10 – *Exemple de réseau de Pétri avec poids sur les transitions*

Ces modèles sont souvent utilisés pour décrire et simuler des circuits numériques.

Les modèles à événements discrets

Avec ces modèles, les événements sont générés par les actions. En cas de simulation, une première action est exécutée, ce qui génère d'autres événements qui déclenchent à leur tour des actions. La simulation s'arrête lorsqu'il n'y a plus d'événements générés. La figure 1.11 illustre le fonctionnement de ce type de modèle : trois processus **P1**, **P2** et **P3** sont en interaction par l'intermédiaire des signaux (événements) **a**, **b**, **c**, **d**, **e** et **f**. Les chronogrammes représentent les valeurs des signaux, leurs transitions, et l'activité des processus.

La notion de temps est souvent ajoutée à ce type de modèles sous la forme d'événements activés dans l'ordre chronologique à chaque fois que le système devient stable. Par exemple **VHDL** [8] et **Verilog** [82] sont des modèles à événements discrets avec notion de temps. Récemment un nouveau langage a vu le jour utilisant cette représentation : il s'agit de **SystemC** [102].

Ce type de modèle peut être réalisé en logiciel ou en matériel ; il est cependant très lent dans sa version logicielle [73].

Les modèles à événements synchrones

Dans ces modèles, les événements proviennent d'une horloge globale. Les divers noeuds de calculs (processus) sont exécutés à chaque coup d'horloge et produisent instantanément le résultat. La figure 1.12 illustre le fonctionnement de ce type de modèle.

Ces modèles peuvent être réalisés aussi bien en matériel qu'en logiciel. Dans le cas du logiciel, la vitesse d'exécution est bien supérieure aux modèles à événements discrets [73]. **Esterel** [17] **Signal** [15] ou **Lustre** [51] sont des langages à événements synchrones. Esterel commence à être utilisé dans l'industrie. **SystemC** [102] permet lui aussi d'utiliser ce modèle.

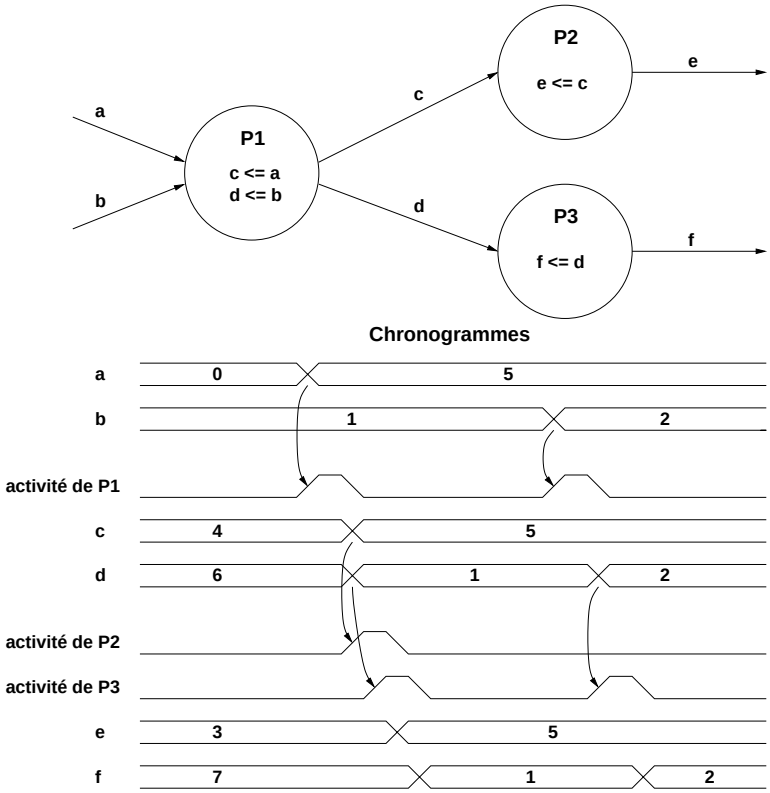

FIG. 1.11 – *Modèle à événements discrets*

Les modèles à passages synchrones de messages

Avec ces modèles, les événements sont en fait des messages entre processus. Les processus ne peuvent s'exécuter qu'à la réception d'un message, et ils ne peuvent recevoir un message qu'à la fin de la transmission de leurs messages. La figure 1.13 illustre le fonctionnement de ce type de modèle : les processus P1 et P2 sont en exécution, puis P1 doit envoyer une donnée à P2. P1 cesse alors son activité, mais la communication ne peut pas débuter car P2 ne cherche pas à consommer la donnée. Dès que P2 à besoin de cette donnée, il cesse son activité et la communication se déclenche. Lorsqu'elle est achevée, les deux processus P1 et P2 peuvent reprendre leur activité.

Ce type de modèle induit des synchronisations locales entre tous les processus, pour un fonctionnement globalement asynchrone.

CSP [54] est un langage basé sur ce modèle. Ce modèle est également utilisé dans des langages de programmation concurrente comme **Lotos** [18] ou **Occam** [78]. Il commence à être utilisé pour la description de circuits globalement asynchrones [92].

1.2.3.1 Les langages de programmation

Les langages de programmation sont des modèles de description de logiciels. Prévus pour s'exécuter sur un processeur, ils se présentent sous la forme d'une suite séquentielle d'instructions. En général, ces langages ne disposent pas de notion de parallélisme, ni de notion temps.

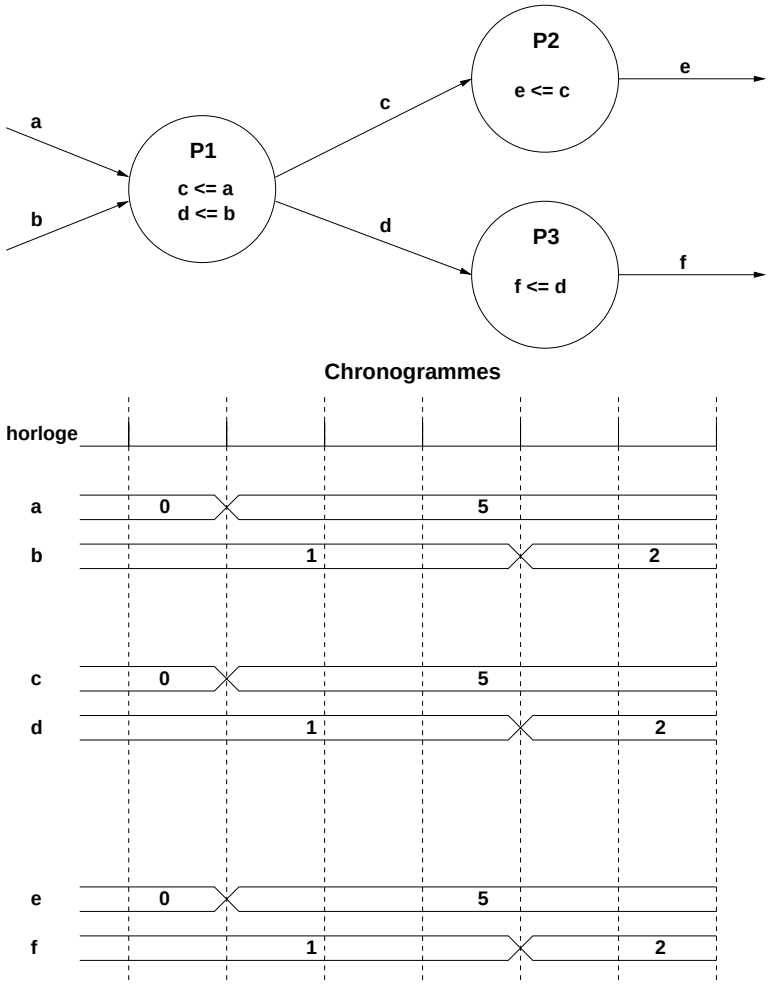

FIG. 1.12 – *Modèle à événements synchrones*

Les langages impératifs

Dans les langages impératifs, les instructions sont des opérations élémentaires (opérations arithmétiques et logiques, contrôles, etc.).

Ces langages sont généralement exécutables après compilation, et conviennent parfaitement pour la description de logiciel efficace. Ils peuvent aussi être interprétés, mais avec une rapidité bien inférieure. Ils ne se prêtent pas facilement aux vérifications formelles, il est donc nécessaire de les tester pour les valider.

C [66] est un exemple bien connu de langage de programmation impératif.

Les langages fonctionnels

Les langages fonctionnels sont constitués uniquement de fonctions. Certaines fonctions sont élémentaires, et permettent de construire d'autres fonctions complexes.

Ces langages sont plus difficiles à compiler que les langages impératifs. Par contre ils possèdent souvent un formalisme mathématique bien défini [69].

lisp [48] est un exemple de langage de programmation fonctionnel.

Les langages d'assemblage

Les langages d'assemblage sont une représentation symbolique du langage machine directement exécutable par les processeurs. Ces langages décrivent une suite séquentielle d'instruc-

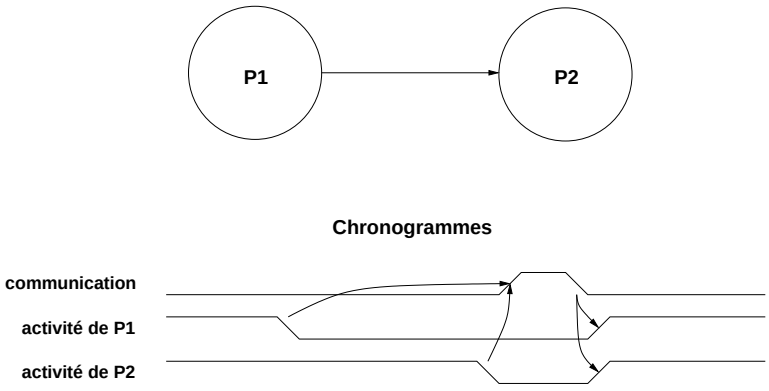

FIG. 1.13 – *Modèle à passage de messages synchrones*

tions que le processeur doit exécuter.

Le langage machine est une succession de 0 et de 1 qui sont interprétés comme des instructions à exécuter par le processeur. C'est une représentation difficile à utiliser pour l'homme. Le langage d'assemblage symbolise chaque instruction du processeur par des mots (appelés mnémoniques) plus facilement utilisables par un programmeur.

Étant directement exécutables par le processeurs, les langages machines dépendent du processeur utilisé : en effet peu de processeurs sont compatibles entre eux, et notamment peu de processeurs disposent du même jeu d'instructions. Il en est de même pour les langages d'assemblage.

1.2.4 Quelques exemples de représentations et leurs utilisations

1.2.4.1 Le langage C

Le langage **C** [66] est un langage de programmation impératif, avec quelques notions fonctionnelles. C'est un langage de haut niveau, qui permet aussi de programmer à bas niveau, ce qui en fait un langage de choix pour la programmation de logiciel performant, et la programmation système.

Ce langage a été étendu avec des notions orientées objets pour obtenir le langage **C++** [98].

C'est un langage largement utilisé dans l'industrie, à tel point que de nombreuses personnes ont essayé de l'utiliser dans d'autres domaines que la programmation de logiciel. Il a cependant des limitations importantes : il ne dispose pas par exemple de notion de temps, ni de parallélisme. Cela rend difficile son utilisation pour des descriptions de matériel. C'est pour cela que de nouveaux langages dérivés de **C** ou de **C++** voient le jour ajoutant ces notions, soit au langage directement comme c'est le cas pour **SpecC** [45], ou par le biais de bibliothèques comme c'est le cas pour **SystemC** [102].

1.2.4.2 Le langage VHDL

Le langage **VHDL** [8] est un langage de description et de simulation de matériel. Il utilise un modèle d'exécution à événements discrets avec des notions de temps, et un modèle de description structurelle avec instanciation.

Il permet de décrire et de simuler du matériel à plusieurs niveaux d'abstraction : du niveau système, au niveau temps de propagation des portes.

C'est un langage très utilisé dans l'industrie européenne du semi-conducteur pour la simulation, mais aussi pour la synthèse. Dans le cas de la synthèse, seul un sous-ensemble du **VHDL** est utilisable [103].

1.2.4.3 Le langage SDL

Le langage **SDL** [58] est un langage de description de système de haut niveau. Il décrit le système comme un ensemble de modules communiquant au travers de canaux. Le comportement des modules est décrit sous la forme de machines à états finis étendues. Les communications se font par le biais de messages qui encapsulent tout type de donnée, transmis au travers des canaux. Chaque machine à états finis dispose d'une unique file FIFO de taille infinie en entrée pour recevoir les messages.

C'est un langage utilisé dans les télécommunications, notamment pour la description des protocoles.

1.2.5 Conclusion

Il existe de très nombreuses représentations, structurelles ou comportementales, pour décrire des systèmes. Même si nous nous restreignons aux systèmes embarqués spécifiques le choix reste très grand. Dans cette section nous avons présenté quelques-unes de représentations les plus utilisées dans la recherche ou l'industrie pour concevoir de tels systèmes. Dans le cadre de cette thèse, les représentations les plus utiles seront les représentations structurelles pour modéliser dans sa globalité le système, et les représentations de type langage de programmation pour la description du logiciel.

1.3 Conception des systèmes embarqués

Les systèmes embarqués de première génération étaient suffisamment simples pour que leur conception ne requière pas de méthodologie particulière : quelques essais-erreurs pouvaient suffire pour satisfaire aux contraintes. Avec la seconde génération, la complexité est devenue trop importante pour que le conception puisse être menée à bien sans méthode. Un premier type de flot de conception a donc été utilisé, inspiré par les flots de conception pour les systèmes généralistes. Ce flot est présenté dans la première section. La troisième génération remet en question ce type de flot, et de nouvelles méthodes émergent. Elles sont présentées dans la section suivante.

1.3.1 Flots classique de conception des systèmes embarqués

1.3.1.1 Le flot

Le flot classique de conception des systèmes embarqués [60] est représenté sur la figure 1.14. Ce flot part d'une spécification souvent informelle du système. Il distingue immédiatement les parties logicielles des parties matérielles. Ces parties sont développées indépendamment l'une de l'autre par des équipes différentes. À la fin, une équipe d'intégration assemble les parties, ce qui pose souvent des problèmes d'incompatibilité. Cette intégration donne directement un prototype à tester. En cas d'erreur, il peut être nécessaire de recommencer complètement le flot.

Dans un tel flot il peut être difficile de développer complètement le logiciel sans que le matériel soit défini. C'est pour cela que son développement devait souvent attendre que la partie matérielle soit décrite pour être achevé.

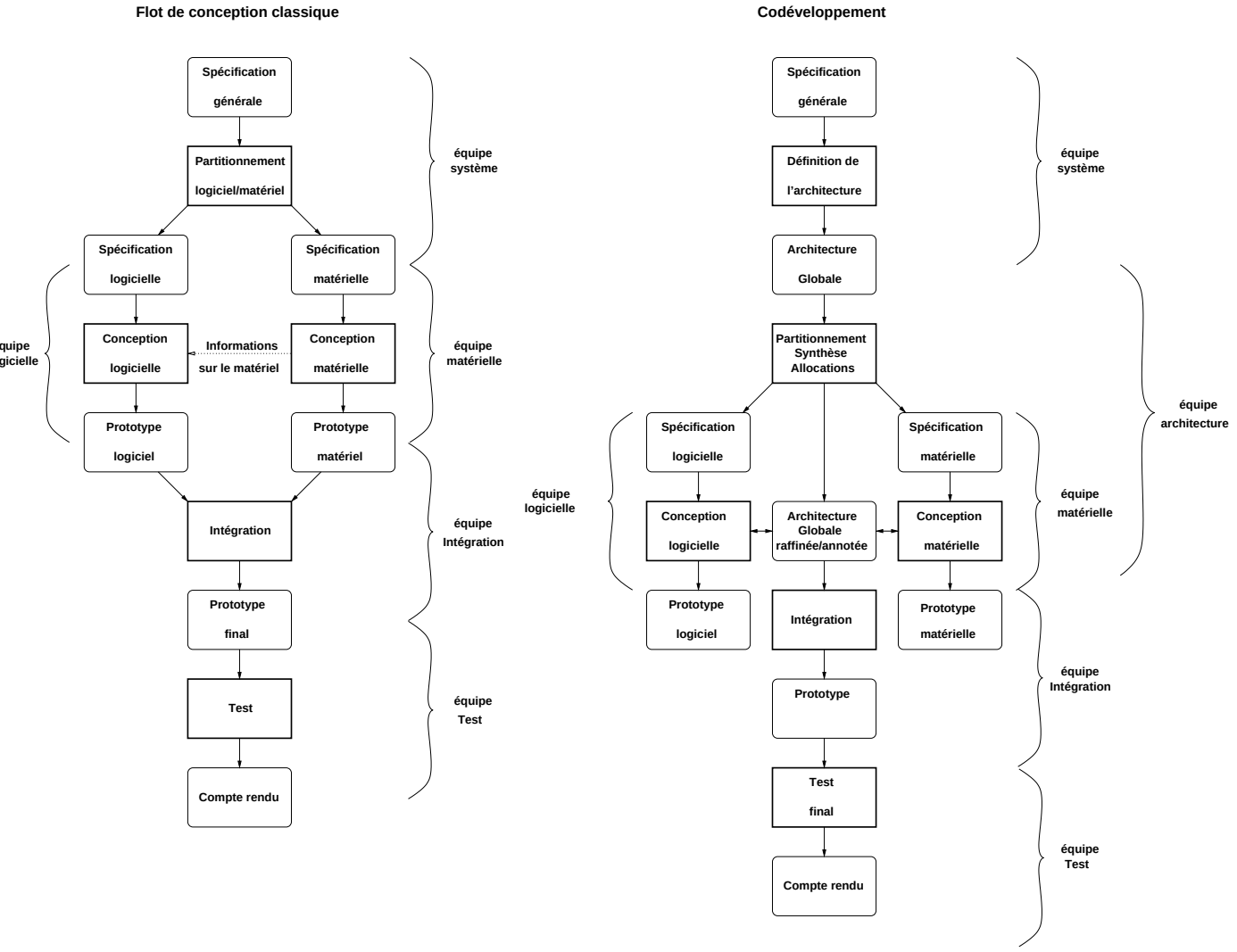

FIG. 1.14 – *Les flots de conception classiques et récents pour les systèmes embarqués*

1.3.1.2 Les limitations du flot classique

Le flot classique de conception de systèmes embarqués possède de nombreuses faiblesses qui le rendent inadapté pour supporter la complexité des systèmes embarqués de troisième génération.

Tout d'abord, il manque une description globale qui accompagnerait la conception du système (logiciel et matériel) du début à la fin. Cette description permettrait à toutes les équipes de conception de bien connaître l'ensemble du système, ce qui éviterait de nombreuses erreurs. Il devrait également être possible d'effectuer des vérifications du système complet à tous les stades de la conception.

La séparation entre le logiciel et le matériel est effectuée trop tôt dans le flot de conception : au stade où elle est effectuée il n'est souvent pas possible de savoir quelle est la meilleure

configuration. À l'inverse, l'intégration des différentes parties est effectuée trop tard dans le flot : il est souvent trop tard pour lever les incompatibilités.

Les faiblesses de ce type de flots font que les temps de développement ne sont plus réalistes pour la conception des systèmes embarqués actuels.

1.3.2 Flots de conception récents

Si le flot de conception classique pouvait suffire pour les première et deuxième générations de systèmes embarqués, la troisième génération, fortement hétérogène et multimaître, est trop complexe pour que les limitations précédemment énoncées soient acceptables. De nouveaux flots sont nécessaires, basés sur de nouvelles méthodes telles que le codéveloppement.

1.3.2.1 Codéveloppement

Le codéveloppement ou *Codesign* [14][63][68][33][44] a pour but de développer conjointement les diverses parties d'un système hétérogène (logiciel, électronique, mécanique, etc). C'est pourquoi une description globale du système est nécessaire.

Deux approches sont possibles pour cette description : la première consiste à décrire toutes les parties dans un langage unique. Ce langage doit alors avoir une sémantique pour chacune des parties. L'avantage est qu'il est plus simple pour les outils et les utilisateurs de gérer un langage unique. L'inconvénient est qu'il est difficile, sinon impossible, de définir un langage qui soit vraiment adapté à toutes les parties du système. La deuxième approche consiste à utiliser plusieurs langages, chacun étant adapté à une partie du système. Un langage de coordination sert à faire le lien entre toutes les descriptions. L'avantage est que cette fois, il est possible d'avoir le langage optimal pour chaque partie. L'inconvénient est que la gestion de tous ces langages est difficile.

Une partie difficile du codéveloppement est le découpage entre le logiciel et le matériel. De nombreuses heuristiques ont étés développées [39][55][31][59][110], etc. Cependant l'approche manuelle prévaut toujours [27].

La figure 1.14 donne un exemple de flot basé sur le codéveloppement en comparaison avec le flot classique : dans le nouveau flot, la séparation entre le logiciel et le matériel est effectuée plus tard, et une nouvelle équipe fait son apparition : l'équipe d'architecture, qui s'occupe du système global et de la coordination entre les équipes.

Le codéveloppement repose aussi sur l'utilisation d'une autre technique permettant la simulation du système aux divers niveaux d'avancement dans la conception. Ces simulations, appelées cosimulations, sont décrites dans la section suivante.

1.3.2.2 Cosimulation

La cosimulation [112][67][86][9] a pour but de simuler conjointement les diverses parties d'un système hétérogène. Cela permet d'effectuer la validation d'un système complet avant le prototype, mais aussi à divers niveaux d'abstraction.

Il existe deux méthodes pour effectuer cette cosimulation (présentées figure 1.15) : la première consiste à traduire les descriptions des diverses parties dans un unique langage pour la simulation. Très souvent, il s'agit d'un langage de programmation tel que le C pour accélérer les simulations. La difficulté est d'être assuré que la traduction et la simulation du langage unique ne changent pas la sémantique des descriptions des diverses parties. La deuxième méthode consiste à conserver les descriptions spécifiques des diverses parties et à exécuter en

parallèle les divers simulateurs. Un programme, appelé bus de consimulation, assure les communications et la synchronisation entre les divers simulateurs. Cette tâche peut s'avérer difficile à effectuer lorsque les modèles de simulation sont différents ; de plus les communications entre les simulateurs peuvent s'avérer coûteuses.

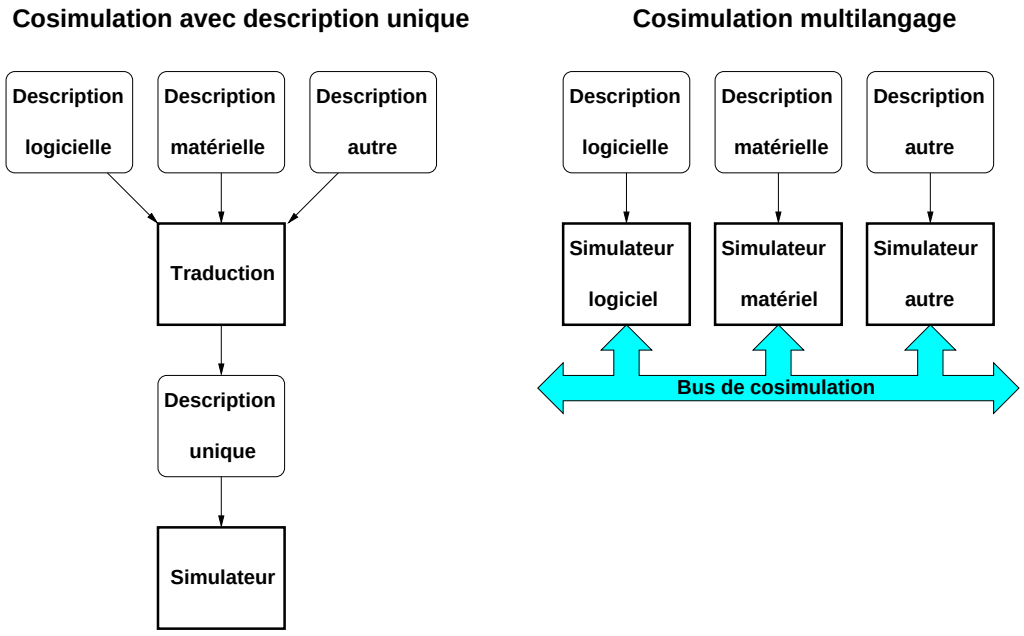

FIG. 1.15 – *Les deux méthodes de cosimulation*

1.4 Le flot de conception développé au sein du groupe SLS

Le travail de cette thèse se déroule dans le cadre d'un projet plus vaste dont le but est de définir un flot[2] et de concevoir les outils permettant de faciliter la conception des systèmes embarqués, et plus précisément des systèmes monopuces. La motivation de ce projet est que les systèmes embarqués sont devenus trop complexes pour être développés avec des méthodes traditionnelles.

L'aide à la conception sera obtenue en élevant le niveau d'abstraction avec lequel les concepteurs travaillent, et en automatisant au maximum les passages aux niveaux d'abstraction inférieurs. Ce projet est centré sur les communications car elles apparaissent comme étant le domaine le moins traité par les méthodologies et les outils actuels. La figure 2 représente ce flot.

1.4.1 Présentation générale du flot

1.4.1.1 Domaine d'application

Le flot a pour but d'aider à la conception des systèmes embarqués spécifiques, et notamment les systèmes sur une puce. Les diverses équipes de conception peuvent utiliser conjointement cet outil (voir la section 1.3.2). Mais il est tout particulièrement destiné à l'équipe d'architecture. L'aide consiste d'une part en l'apport d'une représentation multiniveaux et multilangage de

2. Dorénavant nous utiliserons le terme flot pour «flot de conception».

l'architecture globale. Cette représentation sert de référence pour la conception de toutes les parties du système et aussi pour sa simulation. D'autre part, elle consiste en l'apport d'outils d'automatisation de diverses opérations telles que la génération des interfaces matérielles et logicielles.

Il supporte les architectures hétérogènes multiprocesseurs, multimaîtres et multitâches. Dans ce flot, chaque processeur dispose d'un système d'exploitation et d'un jeu de tâches qui lui sont propres. Chaque composant matériel est encapsulé dans une interface qui adapte ses communications locales aux communications globales de l'architecture.

1.4.1.2 Les restrictions du flot

La conception d'un système embarqué complet commence souvent par une spécification informelle et générale. Il existe des langages tels que UML[87] qui permettent de représenter de manière formelle ce type de spécification, avec un niveau d'abstraction très élevé (le niveau service de la section 1.4.1.3). Le flot proposé n'est pas encore capable d'intégrer ce type de spécifications : il débute plus bas dans l'abstraction, à un niveau où l'architecture globale est définie (le niveau fonctionnel de la section 1.4.1.3).

1.4.1.3 Le modèle de base utilisé dans le flot

La forme intermédiaire utilisée

Le flot utilise une forme intermédiaire [111] pour décrire la spécification quels que soient les niveaux d'abstraction. Cette forme intermédiaire utilise le langage **Colif** [24] qui sera présenté dans la section 4.1.5.1. Ce langage permet de décrire la structure d'un système, à plusieurs niveaux d'abstraction, et en mettant l'accent sur les communications. Les descriptions comportementales sont supposées issues de la description initiale de l'application.

Les objets de la description

Quel que soit le niveau d'abstraction, la description est constituée de modules communicants. Ces modules peuvent représenter des parties matérielles, des parties logicielles ou des éléments de mémorisation. Ils communiquent par l'intermédiaire de ports au travers de canaux.

Chaque objet (module, port ou canal) est décomposé en une interface et un contenu. Le contenu de chaque objet peut être hiérarchique (c'est-à-dire contenir d'autres objets) ou faire référence à un comportement.

La figure 1.16 donne un exemple d'une telle architecture : les modules, les ports et les canaux sont respectivement nommés **M**, **P** et **C** lorsqu'ils sont hiérarchiques et **m**, **p** et **c** lorsqu'ils ne le sont pas.

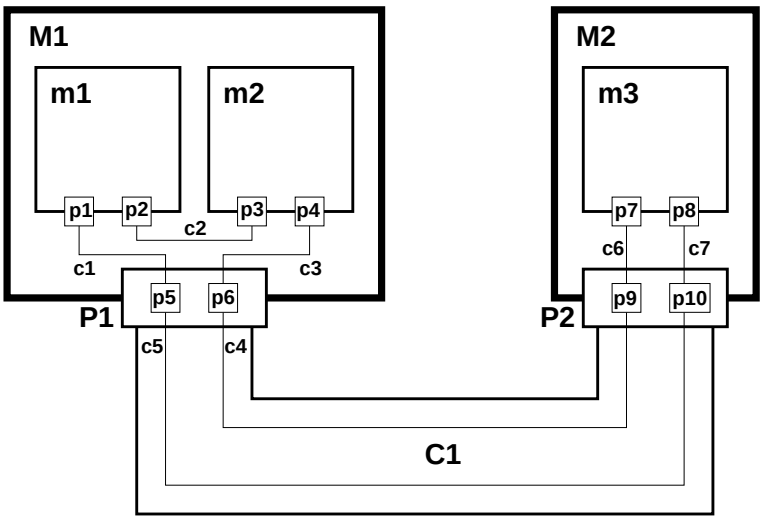

FIG. 1.16 – *Un exemple de la forme intermédiaire utilisée dans le flot du groupe SLS*

Niveau	**Service**	**Fonctionnel**	**Macro-Architecture**	**Micro-Architecture**
Comportement	Objets concurrents	Transactions partiellement ordonnées	Pas de calcul	Cycles
Communications	Requêtes / Services	Passage de messages	Transmission de données / événements	Valeurs de bits
Exemples de langages	CORBA/UML	SDL, SystemC	VHDL, SystemC	VHDL, SystemC

FIG. 1.17 – *Les niveaux d'abstraction utilisés dans le flot*

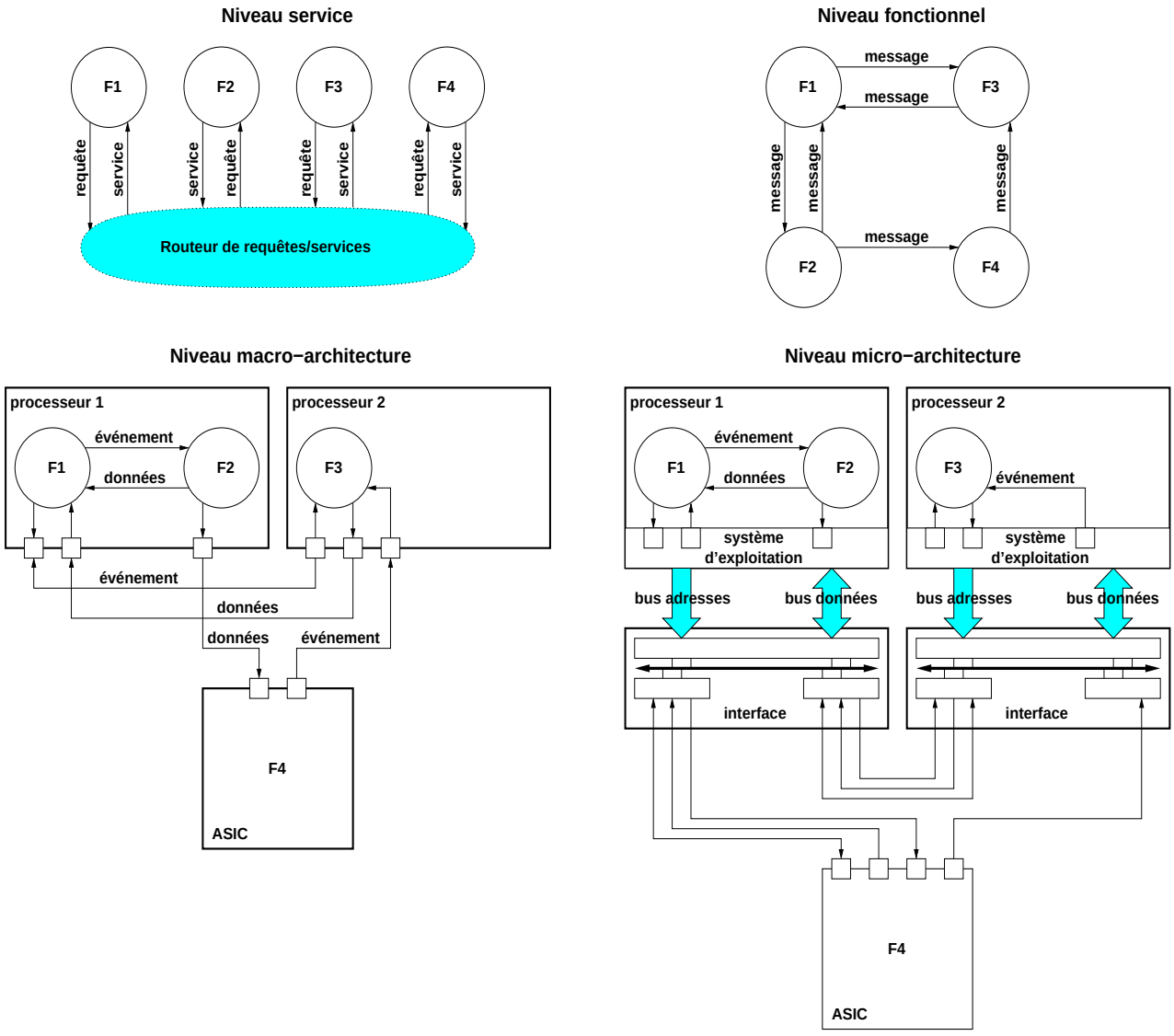

FIG. 1.18 – *Exemple de description pour chaque niveau d'abstraction*

Les niveaux d'abstraction utilisés dans le flot

Le système est représenté sur quatre niveaux d'abstraction différents [85] : le niveau service, le niveau fonctionnel, le niveau macro-architecture et le niveau micro-architecture. La figure 1.17 compare les caractéristiques de ces différents niveaux, et la figure 1.18 donne un exemple pour chaque niveau. Au niveau service, les objets peuvent interagir anonymement par le biais de requêtes et de services. Seules les fonctionnalités sont définies. Au niveau fonctionnel, les connexions entre les objets sont définies. Au niveau macro-architecture, les éléments d'architecture sont définis : les processeurs, les *ASIC*, etc. Au niveau micro-architecture, tous les détails de réalisation sont définis, comme les interfaces de communications des processeurs, ou les systèmes d'exploitation.

Dans l'exemple de la figure 1.18, une application est présentée aux quatre niveaux d'abstraction. Cette application est composée de quatre fonctions (ou tâches[3]) **F1**, **F2**, **F3** et **F4**. Au niveau service elles communiquent anonymement grâce au routeur de requêtes. Au niveau fonctionnel, les connexions entre les tâches sont explicitées, par exemple **F1** envoie et reçoit des messages avec **F2** et **F3** mais **F4** n'envoie des messages qu'à **F3**, et n'en reçoit que de **F2**. Au niveau macro-architecture, l'architecture globale a été décidée : **F1** et **F2** sont sur le processeur 1, **F3** sur le processeur 2 tandis que **F4** est un *ASIC*. Finalement, au niveau micro-architecture, les détails locaux sont aussi explicités : les processeurs disposent de leurs interfaces logicielles (systèmes d'exploitation) et matérielles. Les ports des processeurs et de l'*ASIC* sont réels et non plus ceux de la communication entre les tâches (qui sont cachés dans les interfaces).

Le niveau service n'est pour l'instant pas traité par le flot. Nous commençons par le niveau fonctionnel, pour arriver jusqu'au niveau micro-architecture. Ce dernier est souvent appelé niveau transfert de registres ou *RTL (Register Transfer Level)*.

1.4.1.4 Architecture générale du flot

La figure 1.19 donne une vision simplifiée du flot de conception du groupe SLS. Ce flot débute au niveau fonctionnel, mais après que le partionnement logiciel/matériel ait été décidé. Il se termine au niveau micro-architecture, où une classique étape de compilation et de synthèse logique permet d'obtenir la réalisation finale du système. C'est un flot descendant qui permet cependant de simuler à tous les niveaux et de revenir en arrière à chaque étape. En entrée, le flot attend une description de l'application au niveau fonctionnel. Cette description est traduite dans la forme intermédiaire multiniveaux Colif pour être raffinée au cours de trois étapes :

- La première étape fait passer la description au niveau macro-architecture : elle effectue la synthèse de la communication et les allocations globales de la mémoire.
- La deuxième étape génère un table d'allocation (mémoire et protocoles) : elle effectue les affectations de mémoire et de protocoles, et également les optimisations des accès.
- La dernière étape fait passer la description au niveau micro-architecture : elle effectue la génération des interfaces logicielles et matérielles qui permettent l'assemblage des divers composants ainsi que leurs communications.

3. les tâches sont des modules feuilles

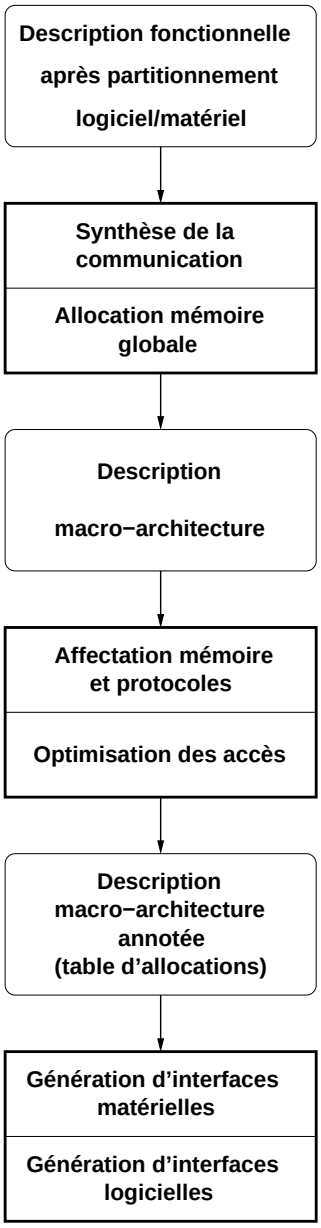

FIG. 1.19 – *Représentation simplifiée du flot de conception pour systèmes monopuces*

1.4.2 Architecture détaillée du flot

Ce flot global est présenté sur la figure 1.20. Il part d'une spécification architecturale et comportementale du système à concevoir. La description est alors raffinée de niveaux d'abstraction en niveaux d'abstractions. En sortie, nous obtenons le code logiciel et matériel réalisant l'application.

1.4.2.1 L'entrée du flot au niveau fonctionnel

En entrée du flot, nous prenons une description dans le langage **SystemC** [102]. Ce langage permet de décrire la structure et le comportement d'un système logiciel et matériel. Il est basé sur le langage C++ étendu avec des bibliothèques permettant la modélisation et la simulation de systèmes logiciels/matériels globalement synchrones, ou asynchrones avec un modèle à événements proches de celui du **VHDL** [8].

Cette description peut être effectuée au niveau fonctionnel, ou aux niveaux inférieurs. Il est aussi possible de combiner les niveaux. Elle donne les informations sur la structure, et éventuellement le comportement. Si certains composants sont fournis sans comportement, ils sont considérés comme des boîtes noires. La figure 1.21 donne un exemple d'une telle description : les modules **vm1** et **vm2** communiquent au travers de ports abstraits. La réalisation de ces communications n'est pas définie. Pour la simulation, une bibliothèque de simulation est nécessaire pour fournir un comportement à ces ports.

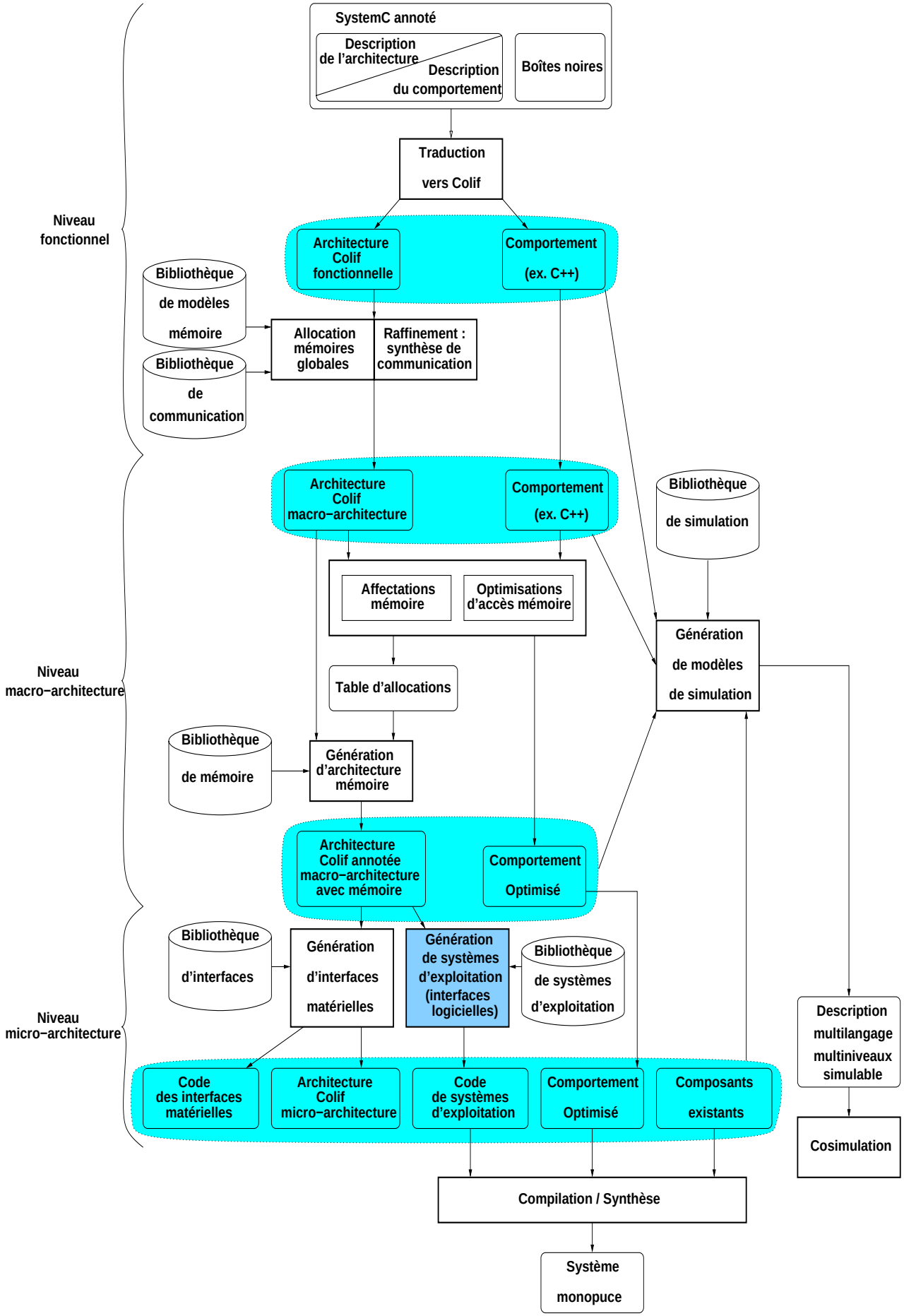

FIG. 1.20 – *Le flot de conception général pour les systèmes monopuces*

```
#include <iostream.h>
#include <stdlib.h>
#include <vadel.h>
#include "appli_types.h"
#include "vm1.h"
#include "vm2.h"



//-----------------------------------------
// definition des canaux virtuels
//
VA_CHANNEL(vc1)
{
  va_ch_mac_pipe<long int>   CH2;

  VA_CCTOR(vc2)
    {
     VA_CEND
    };
};


//-----------------------------------------
// Programme principal
//
int sc_main(int argc, char *argv[])
{
  // Initialisation
  va_init();

  // instancie les canaux virtuels

  vc1 VC1("VC1");

  // instancie les modules virtuels
  vm1 VM1("VM1");
  vm2 VM2("VM2");


  // connecte les ports
  (*VM1.VP1)(VC1);
  (*VM2.VP1)(VC1);


  // Traduction vers COLIF
  sc_start(0);

  return EXIT_SUCCESS;
};
```

FIG. 1.21 – *Exemple de spécification d'entrée du flot*

Cependant il était préférable que nous possédions l'entière maîtrise du flot et donc qu'il ne dépende pas d'un langage d'entrée externe à notre groupe. C'est pourquoi la première étape du flot consiste à convertir la description SystemC en une description Colif.

Remarque : lorsqu'un module représente un processeur, nous considérons qu'il est accompagné par une mémoire locale dans laquelle sont stockés le code et les données locales du logiciel que le processeur doit exécuter. Tout au long de ce mémoire, dès que nous parlerons de processeur, nous supposerons implicitement l'existence de cette mémoire locale. Dans le cas où le processeur est un microcontrôleur, les périphériques intégrés seront eux aussi encapsulés dans l'enveloppe représentant le processeur. La figure 1.22 montre l'architecture représentée par un module processeur.

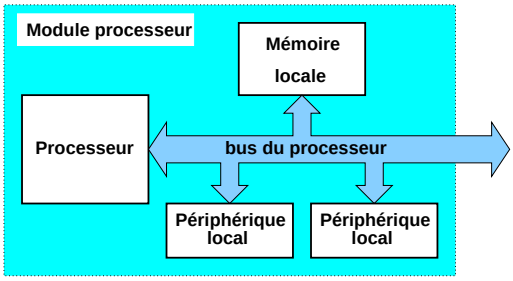

FIG. 1.22 – *Un module processeur*

1.4.2.2 La sortie du flot

1.4.2.3 Étape de traduction vers Colif

Cette étape extrait les informations structurelles de la description d'entrée (SystemC), pour les mettre sous le format Colif qui sera traité tout au long du flot. Les informations comportementales et les informations «boîte noire» sont conservées. Cette étape a été automatisée grâce aux outils développées dans l'équipe par Wander Cesário.

1.4.2.4 Étape d'allocation mémoire et de synthèse

Cette étape permet de passer d'une description du niveau fonctionnel au niveau macro-architecture.

Allocation mémoire

L'allocation mémoire consiste en la définition des blocs mémoires qui vont contenir les données de l'application. Cette étape, présentée dans [80], fait partie de la thèse de Samy Meftali. Dans ce travail, une heuristique a été définie afin de déterminer automatiquement une configuration optimale pour réduire le coût en communication mémoire.

Synthèse de la communication

La synthèse consiste à choisir les protocoles de communication et les éléments de calcul (processeurs, *ASIC*, etc.) qui seront utilisés. Dans l'état actuel du flot, aucun outil ne permet d'effectuer cette synthèse de communication. Cette opération est donc effectuée à la main.

Des résultats de simulation au niveau macro-architecture (voir la section 1.4.2.9) permettent de guider les choix pour les protocoles. À l'avenir, il est prévu d'intégrer une méthodologie et des outils permettant d'automatiser les choix à l'aide d'une bibliothèque de résultats de simulation.

1.4.2.5 Étape d'affectation et d'optimisation de mémoire

Une fois que les blocs mémoire ont été décidés, il est possible de leur assigner les données. Cette opération est effectuée au cours de cette étape en parallèle avec l'optimisation des accès mémoire. Cette étape, présentée dans [79], fait partie des travaux de thèse de Samy Meftali et de Ferid Gharsalli.

1.4.2.6 Étape de génération d'architecture mémoire

La génération d'architecture mémoire, présentée dans [79], fait partie des travaux de thèse de Ferid Gharsalli : les types de mémoires, leurs contrôleurs et leurs interfaces sont générés à partir d'une bibliothèque de mémoire. Le principe de cette génération est celui d'un assemblage de blocs de la bibliothèque. Cet assemblage est guidé par les choix d'allocation et de synthèse (voir la section 1.4.2.4).

1.4.2.7 Étape de génération d'interfaces matérielles

La génération d'interfaces matérielles permet d'interconnecter les divers éléments de calcul : en effet ces éléments ne sont pas tous compatibles entre eux. Ces interfaces permettent aussi de réaliser des protocoles de communication non supportés nativement par les éléments.

Cette étape, décrite dans [77], fait partie des travaux de thèse de Damien Lyonnard. Elle aussi est basée sur un assemblage de blocs à partir d'une bibliothèque. Le modèle d'interface généré au sein du flot est présenté dans la figure 1.23. L'interface est constituée de trois parties : une partie dépendante de l'élément de calcul (processeur), qui fait le pont entre son bus et celui de l'interface. La deuxième partie est le bus de l'interface, et la dernière est l'ensemble des contrôleurs de communication qui réalisent les protocoles. Ce découpage permet de générer aisément des interfaces pour tout type de processeur et tout type de protocole sans que la bibliothèque soit trop grande.

1.4.2.8 Étape de génération de systèmes d'exploitation

Les parties logicielles ne peuvent pas être exécutées directement sur les processeurs : l'exécution concurrente de plusieurs tâches sur un même processeur et les communications entre le logiciel et le matériel doivent être gérées par une couche logicielle appelée système d'exploitation. La figure 1.23 illustre le modèle de ces interfaces logicielles. La génération de systèmes d'exploitation est le sujet de ce mémoire, et sera détaillée tout au long des chapitres suivants.

1.4.2.9 Les étapes de simulation

La génération d'enveloppes de simulation

Le flot permet d'effectuer des simulations du système à tout niveau d'abstraction, et même en mélangeant les niveaux d'abstraction. La technique de base consiste à encapsuler les divers composants à simuler dans des enveloppes qui adaptent le niveau de leurs communications à celui de la simulation globale. La figure 1.24, illustre l'utilisation de ces enveloppes : à l'intérieur de chacune se trouve un composant qui est simulé à un niveau d'abstraction qui peut être inférieur (module **A**) égal ou supérieur (module **B**) au niveau d'abstraction de la simulation. Ces enveloppes permettent aussi d'adapter les divers simulateurs entre eux, comme par exemple un simulateur VHDL (module **B**) et un simulateur SystemC. Cette étape, décrite dans [86], fait partie des travaux de thèse de Gabriela Nicolescu.

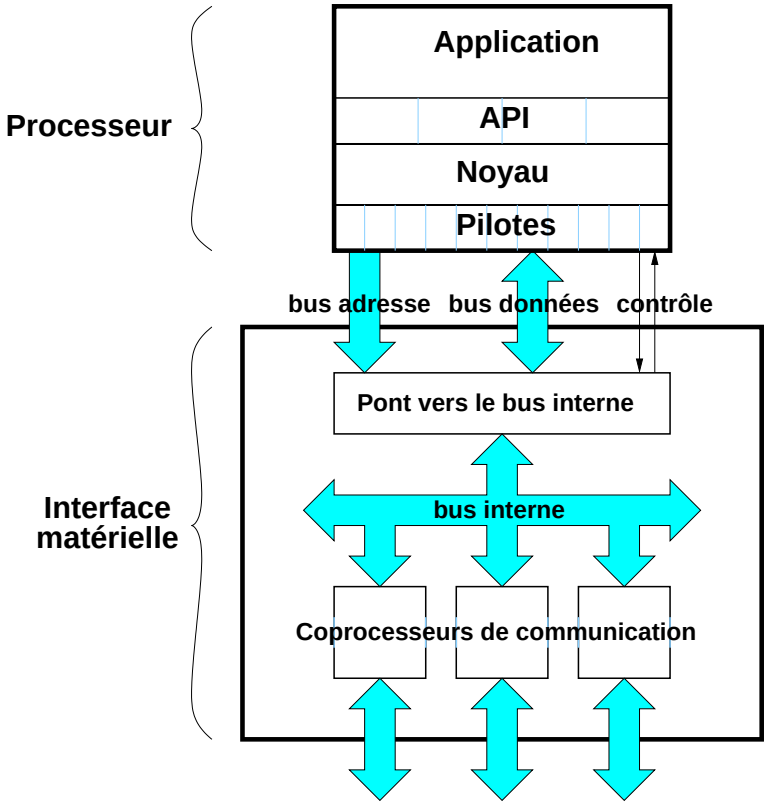

FIG. 1.23 – *Génération d'interfaces matérielles et logicielles*

Un outil qui permet de générer ces enveloppes est basé sur le même principe que la génération d'interface : la bibliothèque contient les trois types de composants précédemment cités, mais le bus interne devient une API (*Application Programming Interface*) de simulation.

La cosimulation

Grâce aux enveloppes, il est possible d'effectuer des validations par cosimulation pour toutes les étapes du flot, ou même sur des composants situés à des étapes différentes. Plus le niveau d'abstraction est élevé, plus la simulation est rapide et plus le niveau est bas et plus la simulation est précise. Cette cosimulation est multiniveaux, mais elle peut être aussi multilangage. La raison de telles cosimulations est que très souvent un langage est adapté pour

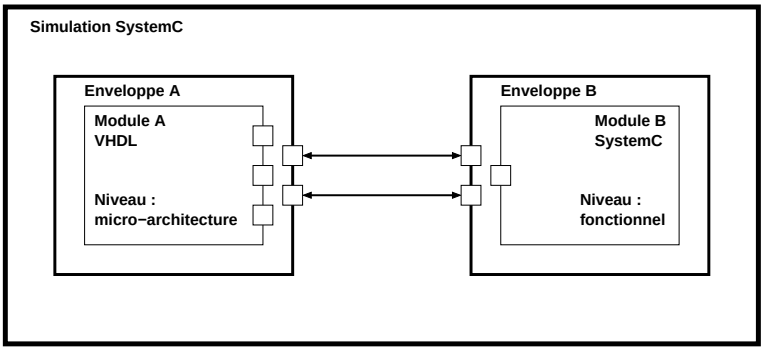

FIG. 1.24 – *Enveloppes de simulation*

la simulation de certaines parties d'un système, mais inadapté pour d'autres. Par exemple les simulateurs **VHDL** [8] sont bons pour les circuits mais pas pour la mécanique qui est mieux modélisée par **Matlab** [53].

Pour effectuer une cosimulation, chaque simulateur est exécuté dans un processus différent. Un processus de contrôle gère l'ensemble des communications et synchronisations entre les simulateurs par le biais de mémoires partagées. Le modèle temporel utilisé est le modèle synchrone : chaque simulateur exécute un pas de calcul, puis toutes les données sont échangées grâce au processus de routage. Une fois que les communications sont achevées, un autre pas de calcul peut être effectué.

Le processus de contrôle est décrit en SystemC. Cela permet de l'utiliser pour décrire des parties matérielles ou logicielles au niveau macro-architecture, sans avoir à utiliser un simulateur externe [4]. Il est généré en même temps que les enveloppes à partir de la bibliothèque de simulation.

1.4.2.10 Utilisation des résultats de simulation

Les simulations pourront servir de base pour mettre en place une méthodologie d'évaluation qui permettra dans un premier temps de guider les choix du concepteur dans l'étape de synthèse, puis d'automatiser cette synthèse avec recherche d'optimum. Pour ce faire, il faudra définir des modèles de performance, et des heuristiques permettant de les composer en même temps que l'on compose les divers modules des systèmes à générer.

1.5 Conclusion

Dans ce chapitre, nous avons précisé le cadre du travail de cette thèse : il s'agit des systèmes embarqués électroniques et informatiques spécifiques, et notamment les systèmes sur une puce (SoC). Trois générations de tels systèmes ont été présentées. Si la conception de la première génération était suffisamment simple pour ne pas requérir de méthode ni d'outil particuliers, la dernière génération, hétérogène, multiprocesseur et multimaître est trop complexe pour être conçue sans de nouvelles approches.

Les méthodes récentes de conception se basent sur le codéveloppement et la cosimulation c'est-à-dire le développement et la simulation conjoints de toutes les parties d'un système. L'équipe SLS du laboratoire TIMA propose elle aussi un flot de conception basé sur ces principes. Ce flot démarre au niveau fonctionnel, c'est-à-dire lorsque les divers composants de l'architecture ont été définis. Il raffine cette description jusqu'au niveau transfert de registres (appelé micro-architecture). Le travail présenté dans ce manuscrit est l'étape de ce flot consacrée à la génération d'interfaces logicielles et au ciblage logiciel. La principale action de cette étape est de générer un ensemble logiciel appelé système d'exploitation. Ce logiciel particulier est complexe et recouvre un domaine très vaste, il est présenté au prochain chapitre.

4. ce qui permet d'avoir un gain notable en vitesse de simulation

Chapitre 2

État de l'art sur les systèmes d'exploitation

Sommaire

Il est courant d'utiliser un système d'exploitation pour gérer plusieurs tâches concurrentes sur un même processeur. C'est une méthode aussi couramment employée dans les systèmes embarqués spécifiques, même s'il existe des solutions alternatives comme nous le verrons au cours de ce chapitre. Cependant, le terme système d'exploitation est un terme très vaste, et ce chapitre va commencer par une introduction générale à ce sujet. Dans la section suivante, leurs caractéristiques pour les systèmes embarqués seront précisées. Enfin, la dernière section sera consacrée à leur intégration dans les flots de conceptions pour systèmes embarqués.

2.1 Introduction sur les systèmes d'exploitation

Dans le monde de l'informatique et de l'électronique le terme de système d'exploitation peut être employé avec des sens différents : il peut par exemple être un ordonnanceur de tâches

[74], ou à l'extrême opposé un environnement complet pour faire fonctionner des programmes [2][108] (sans forcément fournir d'ordonnancement). Il est très souvent logiciel, mais il peut aussi être matériel. Comme dans ce mémoire nous proposons une méthode de génération de systèmes d'exploitation, il convient de préciser ce que nous entendons par ce terme.

Dans cette section, nous présenterons dans un premier temps quelques définitions pour les systèmes d'exploitation, puis nous donnerons quelques propriétés qui leur sont propres. Ensuite nous présenterons les caractéristiques de quelques systèmes d'exploitation bien connus. Pour finir nous présenterons quelques inconvénients liés à l'emploi des systèmes d'exploitation ainsi que des alternatives à leur emploi.

2.1.1 Systèmes d'exploitation : définitions

Dans cette section, nous allons tenter de définir le terme système d'exploitation. Cette section n'a pas pour but de définir de manière absolue les systèmes d'exploitation, mais plutôt d'éviter les confusions avec d'autres définitions relatives au même sujet. Vous pourrez trouver des définitions plus formelles dans [106].

2.1.1.1 Un système d'exploitation en tant qu'abstraction du matériel

Le matériel idéal et le matériel réel

Pour un programme, le matériel idéal aurait des ressources infinies (mémoire, calcul, etc.), et immédiatement disponibles. Tous ses composants disposeraient aussi de la même interface simple.

Le matériel réel n'a hélas pas ces propriétés : aucun composant matériel ne peut réagir instantanément. Les ressources matérielles étant onéreuses, il est souvent nécessaire de les partager entre plusieurs tâches logicielles (c'est notamment le cas pour le processeur et la mémoire). Enfin les différents composants matériels peuvent proposer une très grande variété d'interfaces (pour des raisons de performance, ou tout simplement du fait de leurs fonctionnalités).

Solution apportée par le système d'exploitation

Un système d'exploitation peut être vu comme un matériel abstrait idéal. Il s'agit en fait d'une couche logicielle qui encapsule le matériel et dont le but est de simplifier la conception des applications logicielles. Il peut alors se placer comme unique interlocuteur avec les programmes car il présente les caractéristiques suivantes [106] :

- Il peut cacher aux tâches logicielles les indisponibilités du matériel.
- Il peut offrir des fonctions de gestion de ressources, qui, même si elle ne permettent pas de les rendre infinies, soulagent le programme de cette gestion. Par exemple un ordonnanceur permettra à plusieurs tâches logicielles de partager le processeur.
- Il peut fournir une interface simple identique quel que soit le composant matériel (par exemple UNIX encapsule les accès matériels dans des accès fichiers).

La figure 2.1 représente une application complète allant des programmes logiciels (Tâche 1, Tâche 2 et Tâche 3) à l'architecture matérielle permettant son fonctionnement. Chaque couche ne peut communiquer qu'avec ses voisines, ainsi les tâches logicielles ne peuvent communiquer qu'avec le système d'exploitation.

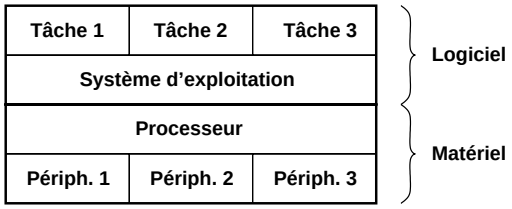

FIG. 2.1 – *Système d'exploitation en tant qu'abstraction du matériel*

2.1.1.2 Un système d'exploitation en tant que gestionnaire de ressources

Les ressources matérielles

Les programmes peuvent être vus comme des entités consommatrices de ressources, ces ressources représentant le matériel [106].

Parmi les ressources matérielles, les premières sont le processeur sur lequel les programmes sont exécutés, et la mémoire dans laquelle ils stockent leurs données. Les autres peuvent être des éléments de calcul, de communication, etc.

La gestion des ressources : le système d'exploitation

Les ressources matérielles ne sont pas toujours directement accessibles ; de plus dans les applications complexes elles sont souvent partagées entre plusieurs programmes : il arrive par exemple souvent que plusieurs programmes doivent se partager le même processeur. Il est donc nécessaire de pouvoir gérer ces ressources.

Le système d'exploitation peut être considéré comme ce gestionnaire de ressources matérielles pour l'application, il peut gérer la ressource processeur grâce aux algorithmes d'ordonnancement de tâches, la ressource mémoire grâce aux fonctions d'allocation et de libération de la mémoire, ainsi que toutes les autres ressources grâce à des gestionnaires de périphériques.

La gestion de ces ressources peut être plus ou moins équitable suivant les besoins de l'application : l'ordonnancement peut avoir des tâches prioritaires, certaines zones mémoire peuvent être réservées pour certaines tâches. Une telle vision du système d'exploitation est illustrée dans la figure 2.2 : dans cette figure le système d'exploitation, associé au matériel est considéré comme un serveur de ressources, et les tâches logicielles sont considérées comme des clients.

2.1.1.3 Élargissement du terme système d'exploitation

système d'exploitation : uniquement logiciel ?

La plupart des systèmes d'exploitation sont logiciels.

Toutefois, une partie du système d'exploitation peut être réalisée en matériel. Dans ce mémoire, nous nous intéressons principalement aux parties logicielles d'un système d'exploitation, mais la méthode présentée est compatible avec la mise en place d'éléments matériels pour le système d'exploitation.

Les interpréteurs de langages évolués

Les processeurs sont des interpréteurs de langage machine. Il existe aussi des interpréteurs de langages plus évolués comme le **LISP** [48] ou le **Java** [101].

Certain des ces interpréteurs fournissent toutes les fonctionnalités d'un système d'exploitation. C'est le cas des machines virtuelles Java. Ils peuvent donc être considérés comme des ensembles système d'exploitation/processeur.

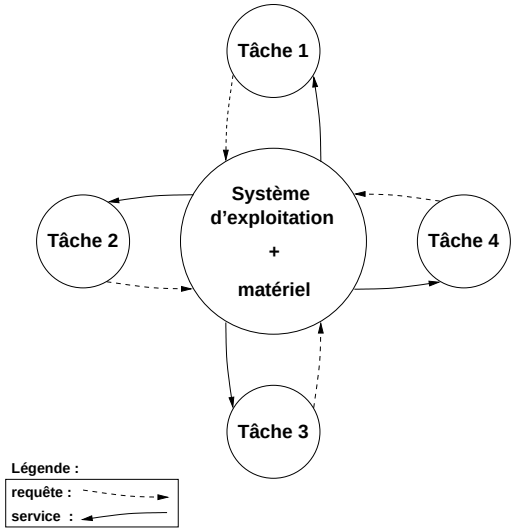

FIG. 2.2 – *Système d'exploitation en tant que gestionnaire de ressources*

2.1.2 Propriétés des systèmes d'exploitation

Dans cette section nous allons préciser les principales propriétés des systèmes d'exploitation : nous présenterons d'abord les modèles d'exécution pour les programmes, puis les modèles temporels, puis les modes de fonctionnement, puis les modèles de mémoire, et enfin, les services proposés aux applications logicielles.

2.1.2.1 Modèles d'exécution pour les programmes

Modèle monotâche

Le modèle d'exécution le plus simple est le modèle monotâche : un seul programme peut s'exécuter à la fois sur le processeur.

Il s'agit d'un modèle très simple, il est donc assez facile à programmer, et il ne consomme que très peu de ressources de calcul et de mémoire. De plus, avec un tel modèle, l'unique programme dispose alors de toutes les ressources, il peut donc être optimal pour une architecture donnée.

Évidement, il n'est pas possible avec un tel modèle d'exécuter simultanément plusieurs programmes sur le même processeur. Cela implique qu'il est difficile d'avoir un bon taux d'utilisation des ressources, par exemple lorsque l'unique programme est en attente d'un périphérique du processeur, les autre ressources restent inutilisées (comme les ressources de calcul du processeur).

Ce modèle d'exécution est intéressant lorsque l'application ne permet pas au processeur d'effectuer plusieurs tâches différentes simultanément.

Avec ce modèle simple il manque aussi la possibilité de réagir aux événements grâce aux interruptions. C'est pour cela que les systèmes d'exploitation monotâches utilisent aussi des routines d'interruption ou *ISR* (*Interrupt Service Routine*) . Ces routines sont des fonctions qui sont automatiquement appelées par le processeur lorsque survient une interruption.

Modèle multitâche

Le modèle d'exécution monotâche n'est pas suffisant lorsqu'on veut faire exécuter en parallèle plusieurs programmes sur le processeur. Dans ce deuxième cas il convient d'utiliser un

modèle d'exécution multitâche.

En général, les processeurs ne peuvent exécuter qu'un seul programme à la fois. Pour réaliser le modèle multitâche, l'exécution parallèle de plusieurs programmes ne peut être que simulée. Pour ce faire, une partie de chaque programme est exécutée à tour de rôle. La figure 2.3 illustre ce fonctionnement : deux tâches (programmes) T1 et T2 doivent fonctionner sur un seul processeur, leur exécution est alors fractionnée et «entrelacée» sur le processeur.

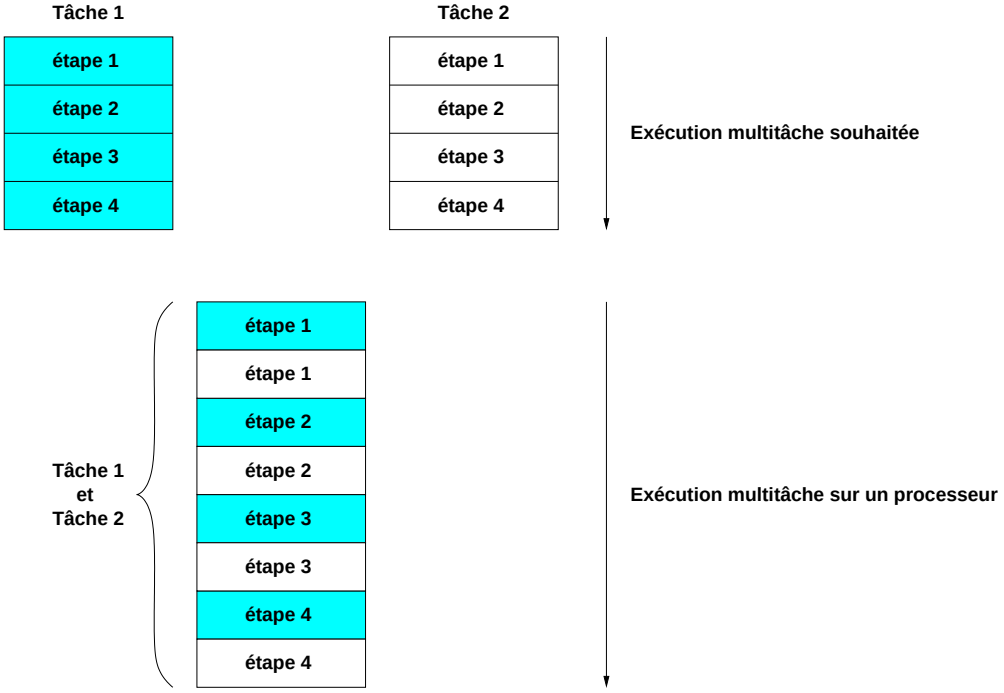

FIG. 2.3 – *Principe du fonctionnement multitâche*

Cet entrelacement d'exécutions entre plusieurs programmes requiert deux mécanismes :

- Un mécanisme pour décider quand le processeur doit interrompre l'exécution d'un programme, et quel programme doit être exécuté à la place. Ce mécanisme, appelé ordonnancement, peut être réalisé de nombreuses manières différentes ; nous en verrons deux grandes familles dans les sections suivantes.
- Un mécanisme permettant au processeur de passer de l'exécution d'un programme à celle d'un autre tout en conservant l'état du premier programme : ce mécanisme s'appelle le changement de contexte.

Modèle multitâche coopératif

Le modèle d'exécution multitâche coopératif permet l'exécution simultanée de plusieurs tâches, mais ce sont les tâches elles mêmes qui décident quand elles rendent la main, auquel cas le système d'exploitation choisit la prochaine tâche qui va s'exécuter suivant un algorithme d'ordonnancement [74].

Un tel modèle permet de gérer plusieurs tâches logicielles, sans pour autant que le système d'exploitation ne ralentisse trop l'application. Il est par exemple possible d'avoir un nombre optimal de changements de contexte[1] pour une application donnée, puisque ce sont les tâches

1. Les changements de contexte sont les passages de l'exécution d'une tâche à une autre, ils sont très coûteux en ressources processeur.

elles-mêmes[2] qui déclenchent ce mécanisme. Le fait que les tâches doivent explicitement indiquer quand elles donnent la main rend cependant leur programmation difficile. D'autant plus qu'une erreur dans une tâche peut avoir des conséquences catastrophiques : par exemple, une tâche ayant la main peut très bien partir dans une boucle infinie, ce qui bloque l'ensemble du système. Il est également difficile d'arbitrer les ressources entre les tâches, et notamment les ressources processeur.

Tout comme les systèmes monotâches, les systèmes multitâches coopératifs ont recours à des routines d'interruption pour prendre en compte les événements extérieurs.

Modèle multitâche préemptif

Le dernier modèle d'exécution que nous allons voir est le modèle multitâche préemptif. Ce modèle permet l'exécution simultanée de plusieurs tâches, mais c'est le système d'exploitation seul qui décide quand la tâche en cours d'exécution doit donner la main.

Ce modèle de fonctionnement est rendu possible par l'utilisation des interruptions : toutes les routines d'interruption font partie du système d'exploitation, et une interruption particulière est réservée pour un temporisateur qui déclenche périodiquement l'algorithme d'ordonnancement entre les tâches.

Avec un tel modèle d'exécution multitâche, les tâches sont assez simple à écrire, car elles n'ont plus à s'occuper des problèmes d'ordonnancement. Ce modèle tolère aussi beaucoup mieux les erreurs dans le code des tâches que les autres modèles, par exemple une tâche qui boucle indéfiniment ne bloquera pas le système pour autant. Enfin, il devient possible d'avoir un arbitrage fin des ressources entre les tâches.

Comme ce modèle demande plus de travail au système d'exploitation, sa programmation est plus difficile, et il consomme aussi plus de ressources (mémoire et processeur). On remarque par exemple que l'algorithme d'ordonnancement déclenche souvent des changements de contexte superflus.

Ce modèle est le seul utilisable pour les systèmes multiutilisateurs. Il n'est pas obligatoire pour de nombreuses applications embarquées, cependant, il reste très employé du fait de ses possibilités pour la gestion des ressources, et de sa robustesse.

2.1.2.2 Modèles temporels

Les modèles temporels utilisés par les systèmes d'exploitation dépendent des contraintes de l'environnement extérieur. Une grande partie des systèmes d'exploitation n'ont pas de contraintes fortes, et peuvent se contenter d'un modèle simple. Les systèmes qui sont soumis à des contraintes temporelles fortes, doivent utiliser un modèle dit «temps-réel».

Modèles simples

Une première catégorie de système n'a aucune notion de temps. C'est le cas de nombreux systèmes d'exploitation monotâches : puisqu'il n'y a qu'une seule tâche, elle peut avoir continuellement la main, et dispose donc du maximum de ressources disponibles pour répondre aux éventuelles contraintes temporelles.

Le modèle à temps partagé consiste en l'utilisation d'une horloge système qui génère périodiquement un événement. Cela offre une mesure du temps, et la possibilité de réagir en fonction de celui-ci. Nous avons vu dans la section 2.1.2.1 que le modèle multitâche préemptif utilisait souvent un temporisateur pour enclencher l'algorithme d'ordonnancement. C'est un des cas d'utilisation du modèle temporel à temps partagé. Plus généralement, le modèle temps

2. le programmeur des tâches connaît l'algorithme de l'application et peut donc savoir quel est le meilleur moment pour passer d'une tâche à une autre.

partagé, permet de réaliser des équilibrages de ressources entre les différentes tâches. Il ne permet cependant pas de respecter des contraintes temporelles fortes, car il n'apporte aucune garantie quant à la date à laquelle une opération va se terminer.

Modèles temps-réel

Dans le monde des systèmes embarqués, le modèle temps-réel est souvent employé. Dans ce modèle, les différentes tâches doivent effectuer leurs actions dans un délai défini au préalable [35].

Ce modèle a pour but d'assurer le respect des délais qui ont été imposés. Pour pouvoir réaliser ce modèle le système doit posséder les propriétés suivantes :

- temps d'exécution borné de chacune de ses actions
- déterminisme
- capacité à ordonner l'exécution des tâches pour que chaque opération soumise à un délai soit achevée à temps

Remarque : la notion de vitesse d'exécution est absente des propriétés requises pour le modèle temps-réel.

Le respect de ces contraintes temps-réel est obtenu par le biais d'algorithmes d'ordonnancement. C'est un problème difficile [113], nous nous contenterons dans le cadre de ce mémoire de faire les remarques suivantes :

1. Les systèmes d'exploitation capables de respecter tous les délais qui leurs sont imposés, sont dits temps-réel durs. Il n'est pas toujours possible d'avoir un tel système, notamment lorsque les délais sont trop serrés. Il existe alors des systèmes qui, lorsqu'ils ne peuvent pas respecter les délais, tentent de «faire au mieux». Il est par exemple possible d'instaurer des priorités d'exécution entre les tâches. Ces priorités peuvent être définies en fonctions de l'«importance» des tâches, ou en fonction de l'urgence. De tels systèmes, plus réalistes sont dits «temps-réel mous».
2. La génération de système d'exploitation, sujet qui nous préoccupe, n'est pas remis en cause par les problèmes temps-réel, car, comme nous le verrons plus tard, le flot de génération étant basé sur l'assemblage et l'expansion de macros, il supporte n'importe quel code pour le système d'exploitation.

2.1.2.3 Modes de fonctionnement

Niveaux de droits

Lorsque plusieurs tâches s'exécutent concurremment sur un processeur, il peut être intéressant de pouvoir limiter leur action de telle sorte qu'une tâche ne puisse pas troubler le déroulement des autre tâches.

Pour résoudre ce problème, les processeurs possèdent souvent plusieurs niveaux d'utilisation, ayant accès à plus ou moins d'instructions, et plus ou moins d'adresses mémoire. Classiquement il y a deux modes :

1. le mode utilisateur, ne pouvant pas exécuter certaines instructions. Généralement, les tâches sont exécutées dans ce mode.
2. le mode système, ou superutilisateur, pouvant exécuter toutes les instructions. Généralement, les fonctions du système d'exploitation sont exécutées dans ce mode.

De nombreux processeurs disposent de bien plus de niveaux d'utilisation comme c'est le cas pour le processeur **ARM7** [12].
Au niveau du système d'exploitation, un utilisateur est représenté par une série de droits,

qui peuvent être par exemple : des droits d'exécution de certaines tâches, d'accès à certaines ressources, ou de contrôle sur l'ordonnancement. Les systèmes multiutilisateurs comme **UNIX** [21] ont une gestion très fine des droits des utilisateurs.

Priorité

Nous avons vu que sur un processeur, l'exécution concurrente de plusieurs tâches ne peut être que simulée. Dans certaines applications, notamment les applications temps-réel, certaines tâches, ou certaines fonctions sont plus prioritaires que d'autres. Il est possible de gérer ces priorités au niveau de l'ordonnanceur en lui donnant la possibilité de favoriser l'exécution de certaines tâches, ou de certaines fonctions. Deux type de priorités peuvent être distinguées :

- la priorité «absolue» : la tâche la plus prioritaire qui n'est pas endormie est toujours exécutée.
- la priorité «de charge» : les tâche prioritaire occupe plus souvent le processeur que les tâches non prioritaires.

Remarques :

- d'autres systèmes de priorités existent, comme celui d'UNIX [21], ou les processus utilisateur qui viennent d'être lancés sont prioritaires, et ou leur priorité décroît au cours de leur utilisation.
- dans le cas de la priorité absolue se pose le problème très connu de l'inversion de priorité, qui peut se produire lorsqu'une tâches prioritaire veut accéder à une ressource bloquée par une tâche non prioritaire.

2.1.2.4 Modèles de mémoire pour un système d'exploitation

Mémoire unique

Ce modèle de mémoire consiste à partager un unique espace mémoire entre toutes les tâches. Les tâches sont souvent appelées dans ce cas des processus allégés ou *threads*. Un tel modèle de mémoire est simple à mettre en œuvre et ne nécessite aucun mécanisme matériel particulier pour fonctionner ; il est utilisable pour tout type de processeur. Le fait que les tâches coexistent dans le même espace mémoire permet de mettre en place des communications simples par variables globales sans l'intervention du système d'exploitation. Mais cela implique aussi que les différentes tâches peuvent se perturber mutuellement, une tâche défectueuse peut par exemple modifier une donnée d'une autre tâche. Un autre inconvénient d'un tel modèle est que, pour coexister, les différentes tâches doivent avoir des adresses différentes pour leur code et pour leurs données ; elles doivent donc pouvoir fonctionner quelles que soient ces adresses. La figure 2.4 illustre ce problème : deux tâches ayant le même code (écrit dans un langage d'assemblage simplifié) doivent coexister dans un même espace mémoire : (a) illustre le cas où le code n'est pas translatable[3] ce qui implique des conflits au niveau des variables et des branchements, (b) illustre le cas où le code est translatable[4].

Mémoire virtuelle

Ce modèle de mémoire consiste à fournir pour chaque tâche un espace de mémoire différent. Du point de vue de la tâche (souvent appelée processus), tout se passe comme si «elle était seule au monde». Avec un tel modèle les tâches ne peuvent pas se perturber mutuellement, et leur code n'a pas besoin d'être translatable. En effet, comme pour chacune des tâches l'espace

3. c'est-à-dire qu'il ne fonctionne qu'avec un unique jeu d'adresses

4. c'est-à-dire qu'à l'édition de liens et au chargement du programme, les adresses du code peuvent être changées sans altérer le fonctionnement

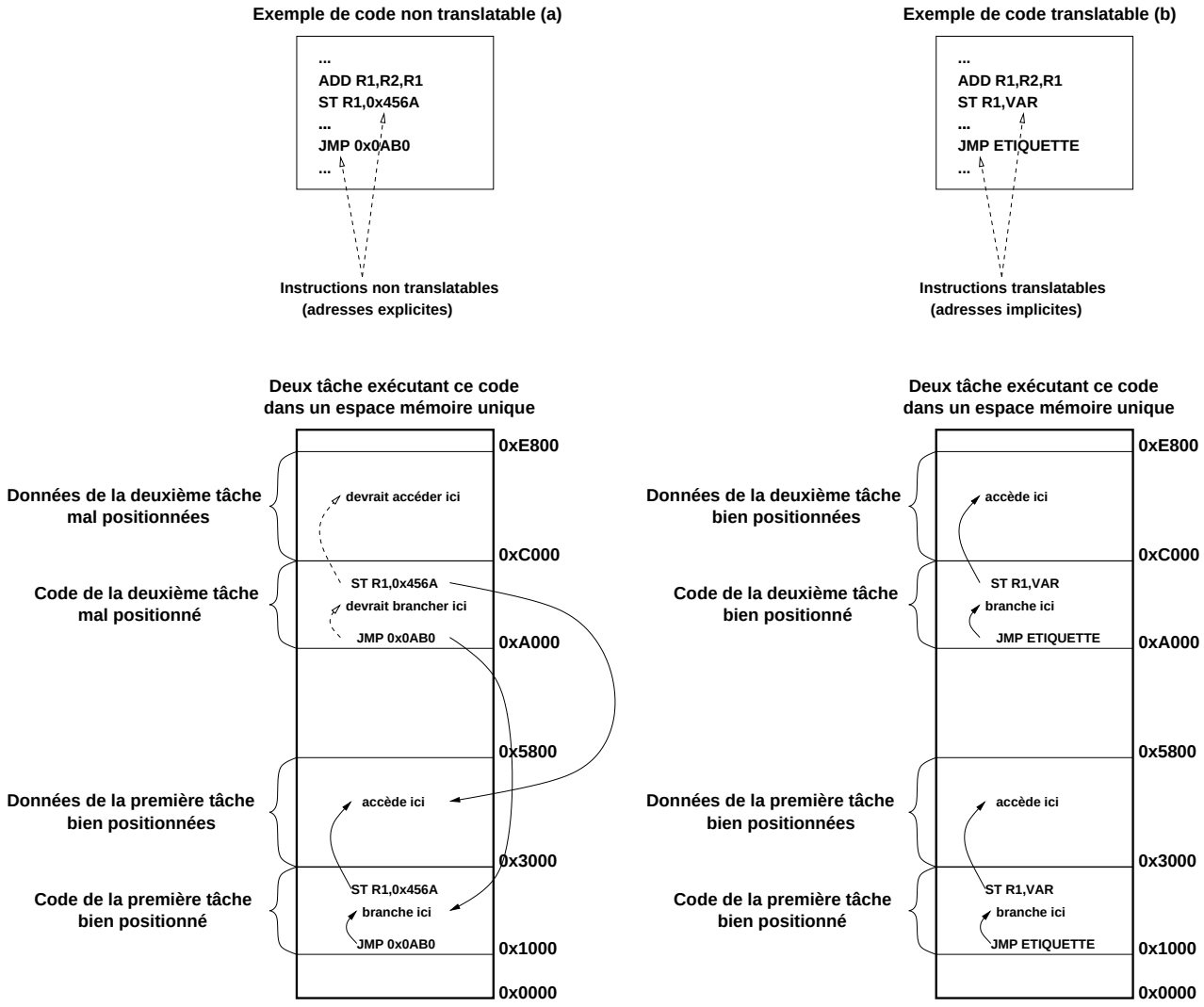

FIG. 2.4 – *Illustration sur le code relogeable*

mémoire apparaît complètement vierge, elles peuvent librement utiliser toutes les adresses. De tels mécanismes sont par contre complexes à mettre en œuvre : nous ne les étudierons pas dans le cadre de ce mémoire, pour davantage d'informations, voir [105]. Pour pouvoir les utiliser, il faut que le processeur dispose d'un système de traduction d'adresses, appelé unité de gestion de mémoire ou *MMU (Memory Management Unit)*. La figure 2.5 illustre le fonctionnement d'une unité de gestion de mémoire : les tâches utilisent des adresses correspondant à leur espace mémoire propre (appelées adresses virtuelles) qui sont converties en adresses physiques avant d'être transmises à la mémoire.

Mémoire partagée

Nous avons vu que le modèle de mémoire virtuelle rend complètement indépendants les espaces mémoire des différentes tâches. Pourtant il est souvent utile que deux tâches puissent partager une partie de leur espace mémoire, notamment pour réduire les coûts en communications. Dans ce but, un modèle de mémoire partagée peut se superposer à celui de mémoire virtuelle : il consiste pour le système d'exploitation à mettre en commun des zones de mé-

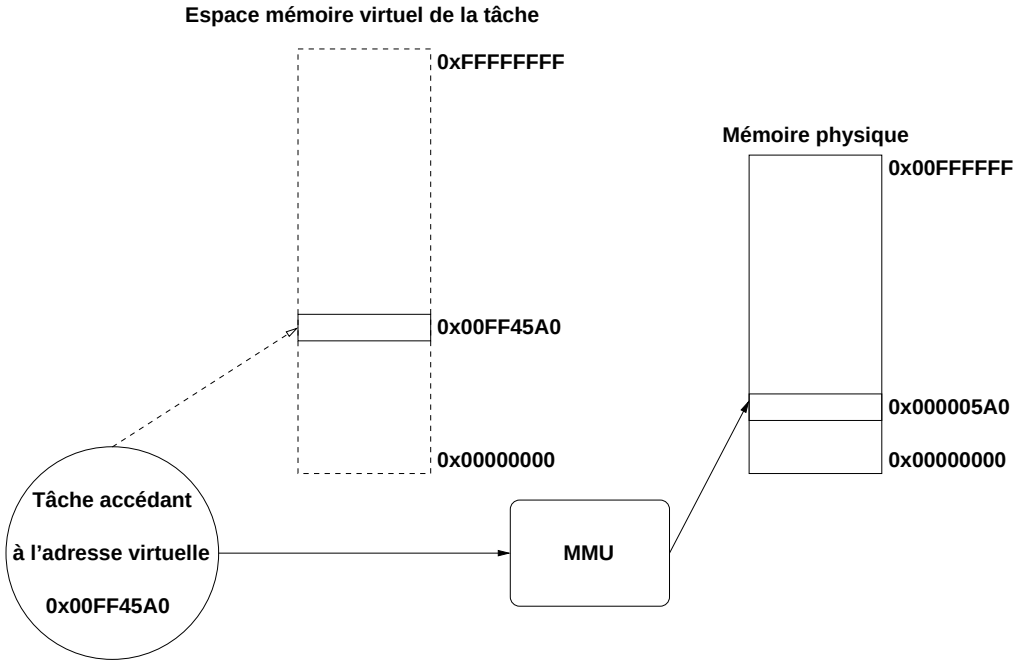

FIG. 2.5 – *Principe de fonctionnement d'un MMU*

moire de tâches différentes[5]. La figure 2.6 donne un exemple de mémoire partagée entre deux tâches dans un modèle de mémoire virtuelle. Sa mise en œuvre est simple : il suffit de faire correspondre à la même adresse physique une partie de chaque espace mémoire.

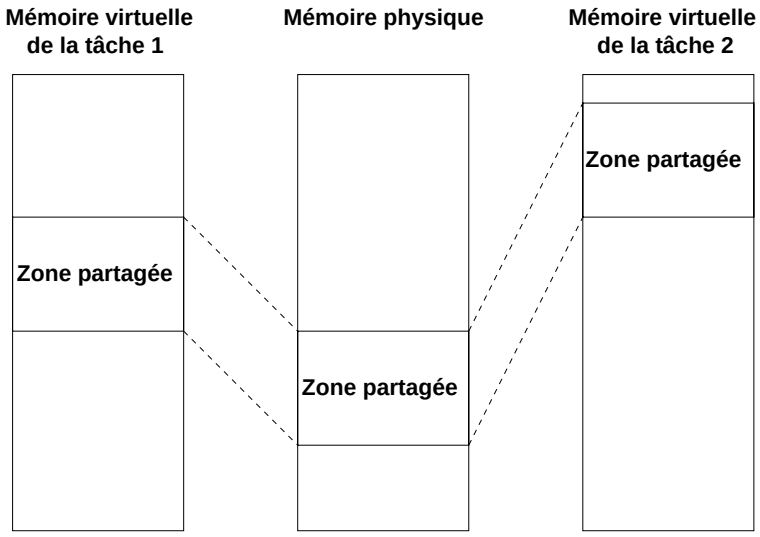

FIG. 2.6 – *Un segment de mémoire partagé entre deux tâches*

5. Ce mécanisme est notamment fourni par les *IPC* d'UNIX

2.1.2.5 Les services rendus par un système d'exploitation

Un système d'exploitation peut proposer des services aux tâches dont il permet l'exécution. Ces services peuvent être par exemple :

- la réalisation de communications entre tâches sur un même processeur ou sur plusieurs processeurs
- la gestion de périphériques
- la possibilité de les endormir ou de les réveiller suivant certaines conditions

Le mécanisme utilisé pour permettre aux tâches d'accéder à ces services est celui des appels système : lorsqu'une tâche utilise un de ces appels, une interruption logicielle est générée. Une telle interruption fait passer le processeur en mode système pour l'exécution de la fonction réalisant ce service. Une fois que le service est rendu, la tâche peut reprendre la main et en utiliser le résultat : par exemple traiter la donnée qu'elle avait demandé de récupérer.

2.1.3 Quelques exemples de systèmes d'exploitation

Dans les sections précédentes, nous avons défini les grandes lignes des systèmes d'exploitation.

Nous allons maintenant présenter les caractéristiques de quelques systèmes d'exploitation bien connus : DOS, Windows 3.x et MacOS, UNIX et Windows NT, et QNX.

2.1.3.1 DOS[108]

Caractéristiques de DOS

Ce système d'exploitation pour les PC ne peut exécuter qu'un seul programme principal à la fois. Il supporte tout de même les programmes «résidents», qui sont en fait des *ISR*. Il n'a pas de notion d'utilisateur, et le programme principal n'a aucune restriction. De même le modèle de mémoire est basique : mémoire commune sans aucune protection.

Les services proposés par DOS

Le système DOS propose peu de services comparativement aux autres systèmes d'exploitation. De plus, ces services sont très fortement liés à l'architecture des PC. Mais il offre plus de liberté aux utilisateurs qui peuvent notamment accéder directement au matériel.

Les services proposés sont :

- gestion d'entrées/sorties : principalement clavier et disque (la gestion vidéo n'est prise en compte que dans le cas des modes textes) Les accès disques ne peuvent pas être effectués en parallèle avec l'exécution du programme principal
- gestion de fichiers
- chargement/exécution/déchargement de programmes. Un seul programme à la fois peut être géré
- gestion de bas niveau de la mémoire non directement accessible du fait de l'architecture des PC initiaux

2.1.3.2 Windows 3.x[88]

Caractéristiques de Windows 3.x

Windows 3.x est un système d'exploitation multitâche coopératif, monoutilisateur, et sans aucune notion temps-réel. Il utilise un modèle de mémoire virtuelle, de plus un certain nombre

de tâches fonctionnent comme des routines d'interruption, indépendamment des autres tâches, comme par exemple la gestion de la souris.

Remarque : Windows 3.x n'est en fait un système d'exploitation que s'il et associé à MS-DOS qui représente la partie bas niveau du système complet.

Les services proposés par Windows 3.x

Ces systèmes proposent un plus grand nombre de services que DOS, notamment, une gestion de la totalité des périphériques, et une gestion de la mémoire virtuelle. Mais ils ne proposent aucun service pour gérer plusieurs utilisateurs, et très peu pour assurer la sécurité du système (par exemple les attributs des fichiers sont réduits à des droit de lecture, d'écriture et de visibilité, qui peuvent être ignorés par l'utilisateur).

2.1.3.3 UNIX[21]

Caractéristiques d'UNIX

Il existe de nombreux systèmes UNIX, mais leurs différences sont essentiellement au niveau des implémentations.

Unix est un système prévu pour être utilisé par de nombreux utilisateurs simultanément. Il possède notamment les caractéristiques suivantes :

- C'est un système multitâche préemptif.
- C'est un système multiutilisateurs avec deux niveaux de privilège : utilisateur et superutilisateur. Il permet aussi de donner quelques droits superutilisateur à certains utilisateurs, ou à certains programmes.
- Il utilise un modèle temporel temps partagé.
- La mémoire est protégée, et chaque tâche évolue dans un espace mémoire virtuel différent.

Les services proposés par UNIX

Les services proposés par les systèmes UNIX sont très nombreux : il offrent évidemment tous les services de gestion des périphériques, mais aussi une gestion poussée des utilisateurs, et diverses méthodes de communication entre les tâches (*pipe*, *IPC*, *socket*, etc.). Il offre aussi des fonctionnalités d'arbitrage entre les ressources, comme par exemple les ressources processeur avec la possibilité d'assigner des priorités aux tâches. Il ne propose cependant pas de mécanismes pour respecter des délais, et ne peut pas non plus garantir que les temps d'exécution de services restent bornés. Il n'est donc pas adapté pour résoudre des problèmes temps-réel. Il existe cependant des versions modifiées qui peuvent convenir pour ce type de contraintes [90].

2.1.3.4 Windows NT[95]

Windows NT possède la plupart des caractéristiques des systèmes UNIX, et propose aussi le même type de services, avec les même limitations concernant le temps-réel. La gestion multiutilisateurs est cependant beaucoup moins développée que dans le cas d'UNIX.

2.1.3.5 QNX[89]

Jusqu'à présent nous n'avons vu que des systèmes d'exploitation pour ordinateur ou micro-ordinateur. QNX est quant à lui un système d'exploitation pour système embarqué. Il possède toutes les caractéristiques d'un système tel qu'UNIX, sauf le modèle temporel qui est temps-réel, et l'absence de gestion multiutilisateurs (hormis les deux modes «utilisateur» et «superutilisateur»).

En fait les principales différences viennent des types de services fournis, comme nous le verrons dans le chapitre 2.2.

2.1.4 Remarques sur les systèmes d'exploitation multiprocesseurs

Jusqu'à présent nous ne nous sommes intéressés qu'aux systèmes d'exploitations fonctionnant sur des machines monoprocesseur. Cependant, il existe des machines contenant plusieurs processeurs : c'est notamment le cas pour les calculateurs, mais aussi de plus en plus pour les systèmes électroniques embarqués.

De nombreuses recherches ont été effectuées sur les architectures et les systèmes multiprocesseurs homogènes (c'est-à-dire contenant des processeurs identiques) [107][52]. Il en ressort que les deux principales difficultés à surmonter sont les coûts en communication et le logiciel.

Les systèmes d'exploitation multiprocesseurs les plus utilisés sont les systèmes qui équipent les serveurs, tels que les systèmes UNIX. Ces systèmes sont prévus pour des architectures multiprocesseurs homogènes, et permettent une utilisation efficace du parallélisme en distribuant les tâches complètes sur les divers processeurs. Pour les systèmes embarqués, ce type de système d'exploitation global pour plusieurs processeurs peut également être utilisé. Son inconvénient est qu'il est difficile de réaliser un tel système efficace lorsque l'architecture comprend des processeurs différents.

Une autre approche, qui se rencontre notamment dans les réseaux d'ordinateurs, consiste à avoir un système d'exploitation par processeur (ou ordinateur). Cette approche est plus coûteuse et plus rigide (les tâches sont statiquement affectées à un processeur), mais elle permet d'utiliser efficacement les hétérogénéités présentes dans les systèmes embarqués. Des projets tels que **ITRON** [104] utilisent ce modèle. Dans le cadre de cette thèse, c'est aussi ce modèle qui est utilisé.

2.2 Les systèmes d'exploitation dans les systèmes embarqués

Dans la section précédente, nous avons présenté les systèmes d'exploitation en général, et notamment pour les ordinateurs. Dans le monde des systèmes embarqués, le logiciel a une part de plus en plus importante ; les systèmes d'exploitation deviennent donc essentiels dans ce domaine.

Cette section présente les systèmes d'exploitation dans le cas particulier des systèmes embarqués. Dans une première sous-section nous présenterons les fonctionnalités requises pour le logiciel dans les systèmes embarqués, puis dans la deuxième sous-section les contraintes, et les degrés de liberté. Enfin, nous résumerons les avantages et inconvénients de l'utilisation de systèmes d'exploitation dans les systèmes embarqués, et nous présenterons quelques solutions alternatives.

2.2.1 Fonctionnalités requises pour le logiciel dans les systèmes embarqués

Dans cette section nous allons présenter quelles sont les fonctionnalités requises pour les systèmes embarqués. Nous verrons d'abord les fonctionnalités communes avec les systèmes d'exploitation généraux, puis nous verrons les fonctionnalités spécifiques aux systèmes embarqués : pour les communications, pour le temps, et pour les pilotes de périphériques.

2.2.1.1 Fonctionnalités communes avec les systèmes d'exploitation généraux

Les systèmes d'exploitation embarqués possèdent de nombreuses fonctionnalités communes avec les systèmes d'exploitation généraux. Ils doivent par exemple pouvoir gérer une ou plusieurs tâches et les ressources matérielles.

Ces fonctionnalités sont cependant à moduler suivant les besoins spécifiques d'un système embarqué : par exemple une gestion multitâche n'est pas nécessaire si une seule tâche est exécutée par processeur. De plus, elles doivent respecter des contraintes particulières pour les systèmes embarqués qui peuvent notablement changer leur implémentation comme nous le verrons dans la section 2.2.2.

2.2.1.2 Fonctionnalités de communication spécifique

Dans les systèmes embarqués spécifiques, et notamment dans les systèmes monopuces, l'architecture est dédiée à l'application pour optimiser les performances et le coût. Cela implique que les architectures de tels systèmes sont très variées. Cette variété se répercute directement sur les communications : tout d'abord parce qu'elles aussi sont optimisées pour l'application, mais aussi parce que les divers composants n'utilisent que rarement les mêmes types de communications.

Ainsi les communications peuvent être point à point ou multipoints comme le montre la figure 2.7. Cette figure présente les deux types de communications, le premier requiert plus de connexions et donc plus de surface, tandis que le deuxième peut être un goulet d'étranglement et donc un facteur ralentissant. Elles peuvent être implémentées avec ou sans mémorisation intermédiaire. La mémorisation intermédiaire permet de désynchroniser deux blocs, sans pour autant forcer les blocs à s'attendre mutuellement pour échanger des données. Cette mémorisation intermédiaire peut elle-même être gérée de plusieurs manières différentes : par exemple cela peut être un système de mémoire partagée (déjà présentée dans la section 2.1.2.4), ou cela peut être un système *FIFO*, illustré dans la figure 2.8, qui montre aussi les avantages de la mémorisation intermédiaire pour désynchroniser deux blocs.

Une caractéristique importante des communications est la définition des protocoles : ils sont très nombreux suivant les architectures, les données à transiter et les contraintes associées (par exemple : CAN [20], ou même TCP/IP [97]). Ces communications peuvent être réalisées en faisant plus ou moins intervenir le logiciel ou le matériel, suivant les compromis choisis entre la performance et la souplesse. De plus, au cours de la conception, ou même après la réalisation, la frontière entre le logiciel et le matériel n'est pas fixe.

Ces divers cas se retrouvent souvent combinés dans la même architecture.

2.2.1.3 Fonctionnalités temporelles

Tout comme les systèmes d'exploitation multiutilisateurs tel qu'UNIX, les systèmes embarqués ont des contraintes temporelles fortes. Cependant, ces contraintes n'ont pas la même nature [36] :

- Pour les systèmes multiutilisateurs il est important de ne pas bloquer longtemps une tâche. Le modèle temporel employé est celui du temps partagé, le but étant d'assurer une certaine équité entre les tâches.
- Pour les systèmes d'exploitation embarqués, il est important de respecter des délais, même s'il est nécessaire de bloquer des tâches pendant une longue durée. Le modèle temporel employé est le modèle temps-réel (mou ou dur suivant les cas, voir la section 2.1.2.2).

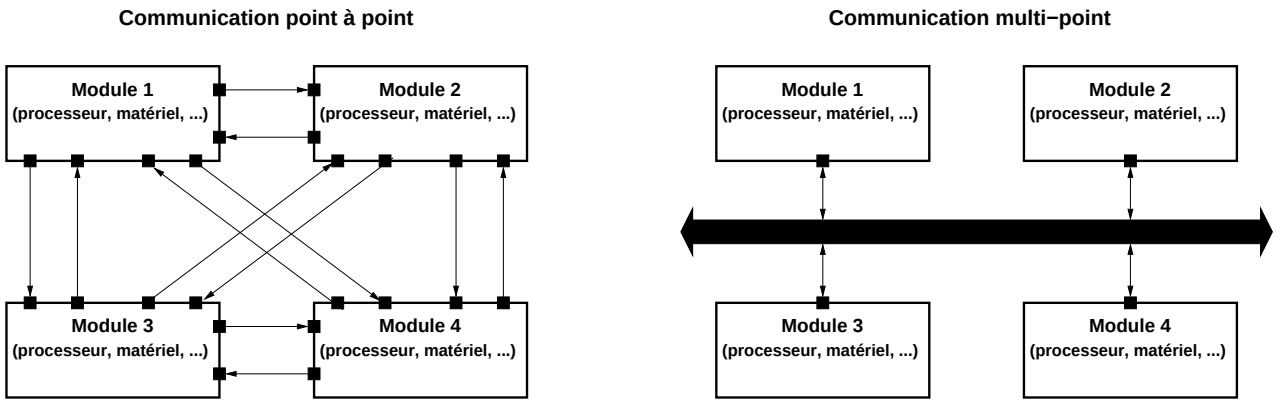

FIG. 2.7 – *Communication point à point et communication multipoint*

2.2.1.4 Pilotes de périphériques

Avec les systèmes embarqués spécifiques il faut souvent trouver des compromis entre le respect des contraintes temporelles, les performances, la consommation et la surface. Les pilotes de périphériques ne sont pas les mêmes suivant les compromis choisis, ce qui augmente d'autant leur nombre.

Enfin les systèmes embarqués devenant très complexes, il est fréquent que plusieurs processeurs fonctionnent en concurrence. Cette concurrence doit elle aussi être gérée par des pilotes.

2.2.1.5 Conséquence des fonctionnalités requises sur les systèmes d'exploitations embarqués

Nous avons vu que les fonctionnalités requises pour les systèmes d'exploitations embarqués étaient d'une grande variété, notamment pour les communications. Cette variété peut se retrouver sur une même puce, voir pour un même système d'exploitation. Il est donc nécessaire que ce dernier puisse supporter cette variété, et il doit donc disposer de très nombreuses parties spécifiques. C'est un obstacle à l'idée de standardisation générale des systèmes d'exploitation embarqués : en effet, à moins d'avoir un jeu de fonctionnalités disproportionné capable de fournir des fonctions optimales pour chaque cas, il est souvent nécessaire d'ajouter des fonctions spécifiques au système pour qu'il puisse fonctionner avec une architecture particulière.

2.2.2 Contraintes imposées par les systèmes embarqués pour le logiciel

Dans cette section, nous allons présenter les contraintes spécifiques liées aux systèmes embarqués [70]. Nous verrons dans un premier temps les contraintes purement matérielles (surface et consommation). Ensuite nous verrons les contraintes temporelles, en mémoire, et en vitesse d'exécution.

2.2.2.1 Contraintes en surface et consommation

Deux contraintes spécifiques aux systèmes embarqués sont la surface [120] et la consommation [32]. Le logiciel influe aussi sur ces paramètres, et il peut être par exemple intéressant de savoir s'il vaut mieux implémenter une fonctionnalité en logiciel plutôt qu'en matériel pour la consommation ou la surface.

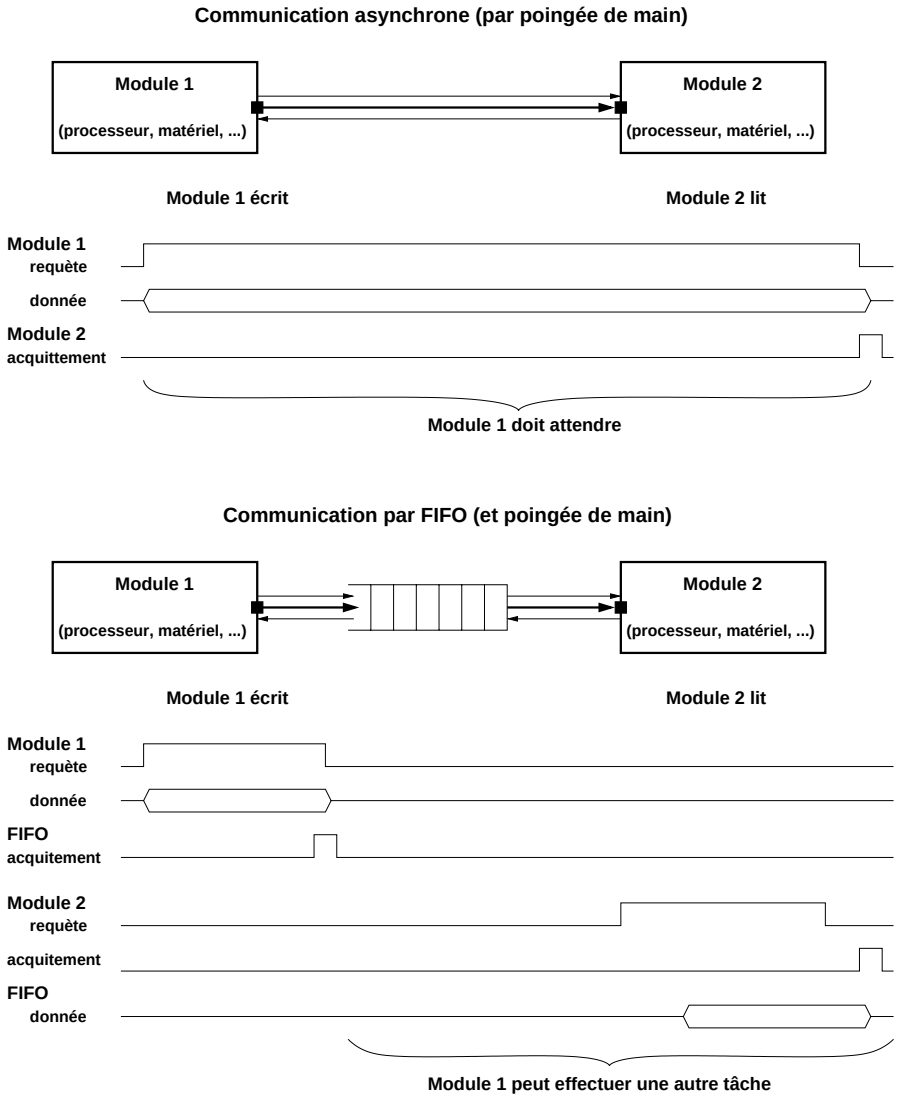

FIG. 2.8 – *Files d'attentes FIFO pour désynchroniser deux blocs sans les forcer à s'attendre mutuellement*

Il est possible de dégager du logiciel quelques paramètres influençant la taille et la consommation :

- La surface pour le logiciel est celle de la mémoire nécessaire pour le stocker et l'exécuter. Ainsi plus le code du logiciel est important ou plus les besoins en mémoire à l'exécution seront importants et plus la surface nécessaire sera importante[6].
- La consommation dépend du taux d'activité des processeurs : plus ils effectuent de calculs, plus ils consomment[7]. De meilleurs algorithmes peuvent réduire les calculs, et donc réduire la consommation (voir [32][75]).
- La consommation dépend aussi du nombre d'accès mémoire. Or les mémoires qui utilisent le moins de surface sont les mémoire dynamiques [65] qui doivent être fréquemment

6. Le processeur prend lui-même de la surface sur le circuit, elle doit être prise en compte lors du choix du processeur. Par contre il est impossible de la changer au niveau du logiciel, c'est pourquoi nous n'en tiendrons pas compte.

7. Les processeurs pour systèmes embarqués disposent souvent de mode basse consommation lorsqu'ils n'ont pas d'activité.

rafraîchies, ce qui provoque une grande consommation. Il y a donc un compromis à trouver entre surface et consommation dans le cas de la mémoire.

2.2.2.2 Contraintes temporelles

Les performances

Lorsqu'il est question de contraintes temporelles, il est souvent sous-entendu performances. Pourtant, la notion de performance diffère entre les systèmes d'exploitation généraux et les systèmes d'exploitation embarqués :

- Les performances des systèmes d'exploitation généraux sont en général évaluées en moyenne car les écarts momentanés importent peu.
- Les performances des systèmes d'exploitation embarqués sont évaluées pour les pires cas des parties citriques : quand le système doit réagir vite, il doit le faire dans tous les cas de figures ; par contre dans les cas ou il n'a pas de délai à respecter, le temps qu'il met n'a pas d'importance.

Ces contraintes temporelles peuvent devenir vitales comme par exemple dans le cas d'un système de freinage ABS.

Contraintes temps-réel

Nous avons vu dans la section précédente, qu'il y a des cas où les contraintes temporelles pour les systèmes embarqués prennent la forme de délais qu'il est impératif de respecter : elles sont appelées contraintes temps-réel, et pour pouvoir y répondre des systèmes dits «temps-réel» sont nécessaires (voir la section 2.1.2.2).

2.2.2.3 Contraintes en mémoire

Nous avons vu dans la section 2.2.2.1, que dans les systèmes embarqués, la mémoire disponible pour le code et pour les données pouvait être limitée. Cette contrainte se répercute sur le système d'exploitation puisque son code et ses données ajoutés à ceux de l'application doivent pouvoir tenir dans la mémoire disponible. Il est de plus préférable de disposer d'un maximum de mémoire pour l'application, le système d'exploitation doit donc être le plus petit possible.

Cette contrainte en mémoire est si forte dans le domaine des systèmes embarqués, qu'il est fréquent de ne pouvoir disposer que de quelques kilo octets de mémoire pour le système d'exploitation.

2.2.2.4 Les erreurs

Les systèmes embarqués occupant des fonctions critiques disposent d'une dernière contrainte : il ne peuvent pas être en panne, même en cas d'erreur.

Cette dernière contrainte conduit à des validations beaucoup plus strictes, et donc beaucoup plus longues des applications embarquées, et aussi des systèmes d'exploitation embarqués. Des mécanismes de tolérance aux fautes peuvent aussi être ajoutés au système, ou à l'application, pour pouvoir continuer à fonctionner malgré les erreurs.

2.2.2.5 L'aspect multiprocesseur hétérogène

Les architectures embarqués récentes contiennent souvent plusieurs processeurs différents, chacun étant spécialisé pour un domaine : par exemple un processeur général pour le contrôle

et un processeur de traitement du signal pour certains calculs. Avec de telles architectures, l'utilisation d'un système d'exploitation unique pour gérer l'ensemble des processeurs est problématique : en effet, une fois compilé pour un processeur, le logiciel ne peut pas être exécuté sur une autre. Une solution possible est d'utiliser une machine virtuelle au-dessus des processeurs pour pouvoir exécuter tout le logiciel dans un langage d'assemblage unique. Cette solution est peu performante, et il est préférable d'envisager un autre méthode : utiliser un système d'exploitation par processeur. Cette seconde méthode apporte cependant des contraintes supplémentaires : tout d'abord les contraintes en surface deviennent plus fortes, car plusieurs systèmes d'exploitation sont susceptibles de consommer plus de mémoire qu'un seul ; ensuite il est peut être difficile de maintenir une cohérence globale avec plusieurs systèmes. Enfin avec une telle architecture, les tâches ne peuvent pas passer d'un processeur à un autre : il faut décider avant la compilation de la conception à quel processeur chacune est allouée.

2.2.3 Les degrés de liberté pour le logiciel dans les systèmes embarqués

Jusqu'à présent nous avons vu les difficultés supplémentaires liées aux systèmes d'exploitation embarqués par rapport aux systèmes d'exploitation généraux. Dans cette section nous allons voir les points qui sont plus simples à traiter pour les systèmes d'exploitation embarqués : la gestion des utilisateurs, les évolutions du logiciel, et le nombre de tâches.

2.2.3.1 Gestion utilisateurs simple

En général, les systèmes embarqués n'ont pas à gérer plusieurs utilisateurs en même temps. Il n'est donc plus nécessaire de gérer la sécurité ni le temps partagé entre les utilisateurs.

2.2.3.2 Évolution du logiciel lente

Nous entendons par évolution du logiciel lente le fait que les programmes sont plus rarement chargés ou déchargés dans les systèmes embarqués que dans les systèmes généraux : en général les systèmes embarqués disposent d'un nombre restreint de programmes, soit disponibles en ROM, soit chargés au démarrage du système. Au contraire les systèmes d'exploitation généraux (pour station de travail), peuvent à tout moment charger ou décharger une multitude de programmes présents sur leurs disques.

Cette évolution logicielle lente limite le risque du chargement d'un programme «indésirable» qui pourrait provoquer des erreurs, ou déséquilibrer l'accès aux ressources. Le nombre d'allocations et de libérations en mémoire est aussi réduit, ce qui donne plus de liberté dans les mécanismes de gestion de la mémoire.

2.2.4 Exemples de systèmes embarqués généralistes

Il existe un très grand nombre de systèmes embarqués. Nous allons ici présenter rapidement deux catégories de systèmes d'exploitation généralistes.

2.2.4.1 Les systèmes d'exploitation embarqués propriétaires

Les systèmes de cette catégorie les plus connus sont : **VxWorks** [116], **NucleusPLUS** [4], **QNX** [89], **pSOS** [56], **Windows CE**[84].

Ce sont des systèmes d'exploitation généralistes qui présentent souvent de bonnes caractéristiques temps-réel et en mémoire. Ils manquent cependant souvent de fonctionnalités spécifiques. Il est aussi difficile de les adapter à de nouvelles architectures du fait de leur nature propriétaire qui empêche leur modification.

2.2.4.2 Les extensions temps-réel

Cette catégorie comprend les systèmes d'exploitation obtenus après adaptations aux contraintes temps-réel de systèmes d'exploitation ouverts existants. Il existe ainsi Linux temps-réel (dont le plus connu est **eCos** [90]) ou mach[22] temps-réel.

Ces systèmes, n'étant pas initialement prévus pour l'embarqué, ont souvent de moins bonnes caractéristiques que les systèmes d'exploitation embarqués propriétaires. Notamment, ils sont souvent non déterministes et plus lents. Ils disposent par contre de beaucoup plus de fonctionnalités spécifiques que les précédents, et ils sont simples à étendre du fait de leur nature ouverte.

2.2.5 Avantages et inconvénients des systèmes d'exploitation pour les systèmes embarqués

Tout au long de ce chapitre, nous avons présenté les systèmes d'exploitation en général puis dans le cas particulier des systèmes embarqués. Dans cette section nous allons essayer de donner les avantages et inconvénients de l'utilisation d'un système d'exploitation pour les systèmes embarqués.

2.2.5.1 Avantages des systèmes d'exploitations pour les systèmes embarqués

Programmation simplifiée des applications

Le système d'exploitation gère lui-même le matériel et propose aux applications des fonctions d'accès de haut niveau. Le travail du programmeur d'applications est donc soulagé de la programmation des accès au matériel, travail difficile, fastidieux et source de nombreuses erreurs.

Cet avantage serait encore plus important si tous les systèmes d'exploitation offraient une même interface très simple. Ce n'est malheureusement pas le cas : les impératifs de performance empêchent souvent l'utilisation d'interfaces génériques abstraites, et la multitude des systèmes d'exploitation et des architectures sont des freins à l'uniformité des interfaces.

Utilisation des spécificités des processeurs

Les systèmes d'exploitation, spécialement programmés pour le processeur sur lequel ils vont s'exécuter, peuvent tirer parties de ses spécificités :

- Le mécanisme d'interruption permet d'interrompre le fonctionnement séquentiel du programme suite à un événement extérieur. Ces interruptions ne sont en général pas prises en compte dans les modèles logiciels, de plus elles sont très variables d'un processeur à un autre. Un système d'exploitation est capable de les gérer comme nous l'avons vu dans la section 2.1.2.1.
- Des instructions de réduction de consommation sont proposées par de nombreux processeurs pour systèmes embarqués. Il y a par exemple des fonctions de mise en veille du processeur jusqu'au prochain événement[8].

8. manifesté par une interruption

- Des instructions de synchronisation ou d'exclusion mutuelle (par exemple l'instruction «Test And Set») servent pour l'utilisation de mémoires partagées entre plusieurs processeurs.
- Des instructions permettent de contrôler le fonctionnement du processeur, comme les instructions de gestion de cache (et qui permettent des optimisations de performances ou de consommation).

2.2.5.2 Inconvénients des systèmes d'exploitations pour les systèmes embarqués

Les systèmes d'exploitation consomment de la mémoire

Les systèmes d'exploitation sont spécifiques au processeur sur lequel ils s'exécutent, par contre, ils restent très généraux pour l'application qu'ils doivent gérer. En effet, ils sont prévus pour exécuter n'importe quel type d'application et ils doivent donc proposer des services suffisamment généraux pour être utilisables par toutes.

La généralité du système d'exploitation vis à vis de l'application fait qu'il est souvent plus volumineux que nécessaire. C'est un défaut important dans le monde des systèmes embarqués où la mémoire est limitée.

Les systèmes d'exploitation modulaires tentent de résoudre ce problème en mettant leurs fonctionnalités sous la forme de modules optionnels, qui ne seront effectivement chargés dans la mémoire que s'ils sont utilisés. Cependant, ces modules restent eux-mêmes généraux, à moins d'avoir une bibliothèque de modules contenant tous les types de modules spécifiques possibles, ce qui n'est guère réaliste.

Les systèmes d'exploitation consomment des ressources processeur

Comme nous l'avons vu dans la section précédente, les systèmes d'exploitation sont très généraux pour les applications qu'ils doivent exécuter. Cette généralité se paye en terme de mémoire consommée, mais elle peut se payer aussi en terme de vitesse d'exécution : par exemple les synchronisations utiliseront toujours des mécanismes de sémaphores complets, alors que dans de nombreux cas un simple verrou suffit.

La vitesse du système d'exploitation est aussi limitée par l'ordonnancement dynamique des tâches [94] qui demande du temps aussi bien pour la décision que pour le passage d'une tâche à l'autre.

Les systèmes d'exploitation peuvent être non déterministes

Nous avons vu que dans les systèmes embarqués, des contraintes temps-réel pouvaient imposer que le fonctionnement soit déterministe. Ce déterminisme n'est pas toujours aisé à obtenir avec les systèmes d'exploitation qui sont des programmes à exécution complexe. En fait, il est souvent impossible de savoir avant utilisation si une application basée sur un système d'exploitation va respecter des délais ou non.

2.2.5.3 Solutions alternatives aux systèmes d'exploitation

Nous avons vu précédemment les avantages et inconvénients de l'utilisation des systèmes d'exploitation pour les application embarquées. Il existe aussi des solutions sans système d'exploitation. Nous allons ici présenter les deux grand types de solutions alternatives.

Utilisation d'un programme unique par processeur

La solution la plus simple consiste à écrire un seul programme qui fonctionnera directement pour le processeur. Elle présente l'avantage de pouvoir prendre en compte les spécificités du processeur et de l'application, elle n'est par contre utilisable que si l'application et le processeur

sont simples. Il est notamment difficile d'utiliser cette méthode dans le cas d'applications multitâches.

Cette méthode a été la première employée dans la conception des systèmes embarqués, et elle est encore couramment utilisée dans le cas des processeurs de traitement du signal par exemple.

Génération d'un exécutif statique

Une autre solution consiste en la génération d'un unique programme monotâche séquentiel à partir d'une description de plus haut niveau, éventuellement multitâche, souvent basée sur des ensembles de machines à états. Cette solution nécessite un ordonnancement statique de l'application.

Cette approche est notamment utilisée dans les travaux de [71][50][27][37]. Avec cette méthode, il n'y a plus d'ordonnancement dynamique des tâches, qui était un gros consommateur de ressources processeur et un facteur d'indéterminisme. De plus, le code d'entrée reste aussi simple à écrire que dans le cas de l'utilisation d'un système d'exploitation multitâche. Il est également possible d'obtenir des équivalences formelles entre la description d'entrée et le code généré. Enfin, du fait du déterminisme, il est possible d'avoir des estimations fiables des temps d'exécution. La validation de l'application finale est donc plus simple que pour un système d'exploitation.

Cette méthode présente cependant des inconvénients qui limitent son champ d'utilisation. Premièrement le code généré est souvent un peu plus lent et plus volumineux que celui qui aurait été obtenu par un codage manuel. Cela vient essentiellement de l'ordonnancement statique, qui ajoute de très nombreux branchements dans le code, et qui rend difficile l'intégration des appels de fonctions. Cette relative lenteur du code généré peu cependant se comparer favorablement avec le temps perdu par les changements de contexte dans le cas des systèmes d'exploitation. Un autre problème avec cette méthode est que les mécanismes de synchronisation entre tâches sont limités à des attentes actives : en effet même s'il y a plusieurs tâches en entrée, au final il n'y a qu'un seul programme, qui ne peut pas être bloqué [9] pour une tâche, car alors les autres tâches le sont aussi. Ce mécanisme d'attente active peut être très coûteux en ressources processeur, de plus il n'offre pas la possibilité de réagir «instantanément» à un événement puisque un événement ne peut être pris en compte que s'il a été contrôlé au préalable. La figure 2.9 illustre ce problème : une tâche doit attendre l'arrivée d'une valeur dans un registre, dans le premier cas une attente active est utilisée ce qui oblige la tâche à aller lire périodiquement l'état du registre, dans le deuxième c'est une interruption qui est utilisée.

Cette méthode convient très bien pour les applications de type flot de données, comme celles que l'on peut rencontrer dans le traitement du signal [49]. Par contre dès que l'apparition des événements n'est pas prévisible l'ordonnancement statique est un handicap.

2.3 Intégration des systèmes d'exploitation dans les flots de conception

Pendant longtemps les systèmes embarqués étaient très simples, tant au point de vue du matériel, qu'au point de vue logiciel. Il était alors souvent superflu d'avoir un système d'exploitation : les parties logicielles étaient tellement simples, et les processeurs utilisés tellement petits qu'elles étaient écrites directement en assembleur. Ce n'est plus le cas de nos jours : l'utilisation d'un système d'exploitation (ou autre méthode fonctionnellement équivalente, voir la

9. C'est-à-dire qu'il n'est plus exécuté.

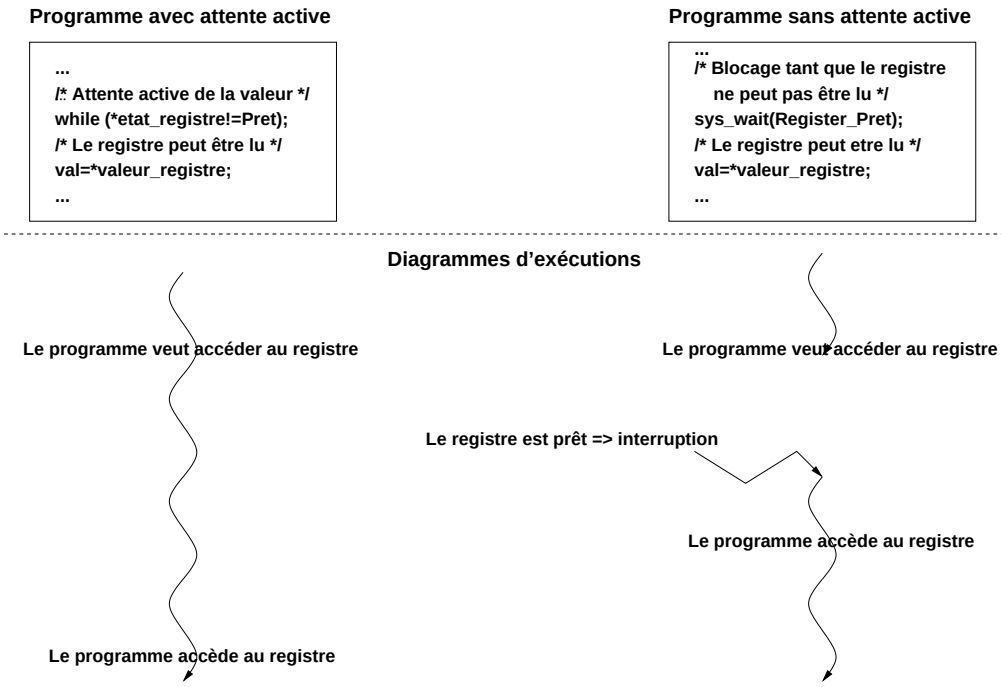

FIG. 2.9 – *Inconvénient de la synchronisation par attente active*

section 2.2.5.3) est devenue une nécessité, les applications quant à elles sont devenues trop complexes pour être totalement écrites en assembleur.

Le but de ce chapitre est de présenter les différents flots de conception existants. Dans une première section nous présenterons les méthodes classiquement employées dans l'industrie, ensuite nous présenterons des méthodes plus élaborées, qui commencent à être utilisées industriellement, enfin nous présenterons quelques approches récentes pour résoudre les problèmes dus à l'utilisation des systèmes d'exploitation.

2.3.1 Méthode classique : système d'exploitation fixe et écriture à la main du code des applications adapté au système

Dans cette section, nous allons étudier le flot de conception classique pour les applications dans les systèmes embarqués [113]. Nous en donnerons les justifications, puis les inconvénients.

2.3.1.1 Présentation du flot

Dans le développement logiciel pour les ordinateurs, les systèmes d'exploitation sont traditionnellement des programmes monolithiques, c'est-à-dire qu'ils forment un bloc qui ne peut être modifié. Les applications sont développées à la main, et spécifiques au système d'exploitation et à l'architecture cible.

On retrouve la même chose pour les systèmes embarqués classiques, et le flot de conception peut alors être représenté comme sur la figure 2.10 : le programmeur dispose de spécifications de l'application à développer ainsi que d'un manuel utilisateur pour le système d'exploitation et pour l'architecture matérielle cible, et il produit à la main le code de l'application adapté. Ensuite l'exécutable final est obtenu par compilation et édition de liens.

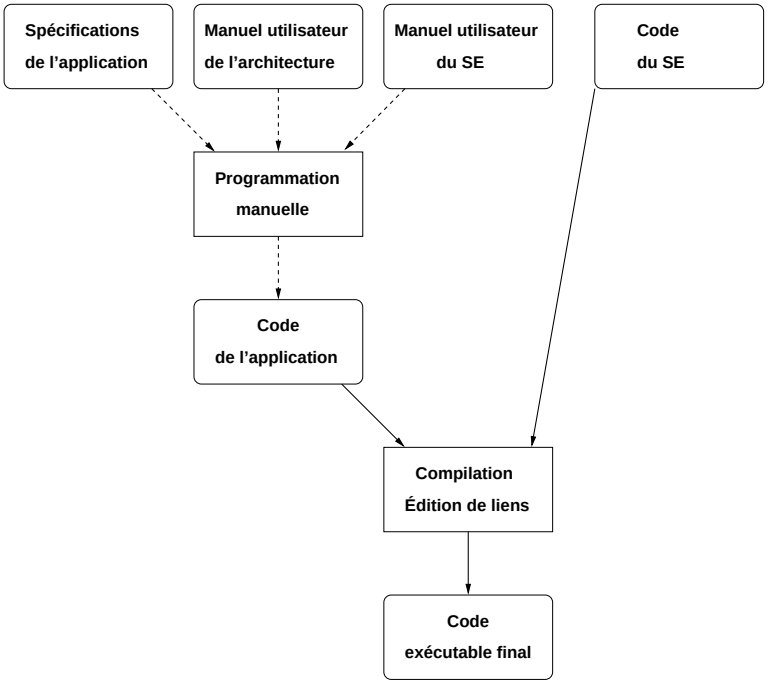

FIG. 2.10 – *Le flot de conception classique pour le logiciel embarqué*

2.3.1.2 Les architectures visées par ce flot classique

Ce flot, identique à celui utilisé pour le développement des applications pour ordinateurs, peut se justifier par le fait que les architectures classiques des systèmes embarqués sont semblables à celles des micro-ordinateurs. La figure 2.11 représente ce type d'architectures : elles sont basées sur un processeur central, qui est maître de l'ensemble du système, et d'une série d'autres composants périphériques, éventuellement programmables, mais tous répondant aux ordres du processeur.

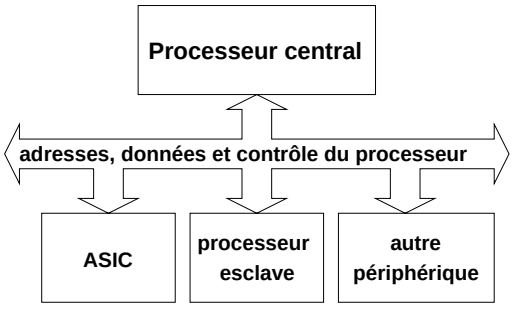

FIG. 2.11 – *Un exemple d'architecture classique pour système embarqué*

Lorsque le processeur central est répandu, il existe en général un ou plusieurs systèmes d'exploitation compatibles avec l'architecture. En outre, la simplicité de l'architecture limite le nombre de cas à traiter par le système d'exploitation. Dans ces conditions, un flot plus élaboré n'est pas nécessaire.

2.3.1.3 Les inconvénients et limitations de ce flot

Ce flot classique est toujours couramment utilisé pour les systèmes embarqués, notamment du fait qu'il reste l'approche la plus naturelle pour développer de telles applications. Cependant, il présente un certain nombre d'inconvénients qui le rendent de plus en plus obsolète pour répondre aux besoins récents.

Tout d'abord, le système d'exploitation est monolithique, et peut difficilement être adapté pour une architecture particulière. Il est dès lors difficile d'avoir un logiciel minimal, et inversement si une fonctionnalité est manquante dans le système, il est difficile de l'ajouter. C'est un gros handicap pour les systèmes embarqués, ou l'architecture peut beaucoup changer au cours de la conception. Ensuite, le concepteur doit effectuer un travail fastidieux qui est facteur d'erreurs. Il doit notamment lire et interpréter la spécification de l'application, mais aussi les descriptions du processeur et de l'architecture. Ces documentations sont souvent redondantes, et n'offrent que rarement les informations sous une forme pertinente (c'est-à-dire utilisable sans que le concepteur ait à les recouper avec d'autres : par exemple, pour accéder un périphérique le concepteur devra combiner les informations concernant l'adressage du processeur avec celles concernant celui du périphérique). Ensuite, le concepteur doit programmer l'application, configurer, et éventuellement porter le système, avec tous les paramètres qu'il aura pu interpréter d'après les spécifications.

De nos jours, la complexité requise pour les systèmes embarqués est très importante, et reste en croissance exponentielle. Les inconvénients suscités empêchent l'exploration d'un grand espace de solutions, ce qui veut dire qu'il devient impossible de déterminer en des temps raisonnables une implémentation optimale pour un système embarqué récent. Cette complexité requise pour les systèmes embarqués implique aussi des architectures matérielles plus complexes, notamment avec plusieurs composants maîtres. Le flot classique en devient d'autant plus complexe et difficile à appliquer en des temps raisonnables.

2.3.2 De nos jours : configuration des systèmes d'exploitation

De nos jours, le flot classique présenté précédemment commence à montrer ses limites. Un nouveau flot, un peu plus raffiné, a été utilisé ces dernières années pour pallier les inconvénients de l'ancien. Il repose essentiellement sur l'ajout de flexibilité au niveau du système d'exploitation.

Dans cette section, nous allons présenter ce flot avec des exemples industriels, en montrant d'abord les systèmes d'exploitation modulaires, puis en présentant une approche permettant de simplifier l'adaptation de ces systèmes.

2.3.2.1 Les systèmes d'exploitation modulaires

La qualification de modulaire (*scalable* en anglais) est assez vague, et on trouve dans les produits industriels de nombreuses manières d'apporter cette propriété. Nous allons étudier chaque catégorie.

L'architecture cible

Le premier domaine où il est intéressant de fournir un système d'exploitation modulaire est celui de l'architecture cible, et notamment du processeur cible. Dans ce domaine, deux solutions prévalent :

1. Soit le système est fourni sans les sources, auquel cas il est proposé une version du système par processeur cible. C'est le cas de **VxWorks** [116], **Virtuoso** [41], **C EXECUTIVE**

[62], **Chorus OS** [100], **QNX** [89], **EYRX** [43]. Cette situation ne favorise pas une exploration avec des processeurs différents, de plus il est impossible d'utiliser le système avec un processeur qui n'a pas été prévu par le fournisseur.

2. Soit le système est fourni avec les sources, auquel cas le changement de processeur revient à changer quelques fichiers spécifiques et à les recompiler, il n'est donc pas nécessaire d'avoir plusieurs versions du système. C'est le cas pour **LynxOS** [76], **eCos** [90], **NucleusPLUS** [4], **RTXC** [40], **MicroC/OS-II** [81]. En outre, dans le cas où le système n'a pas été prévu pour un processeur donné, il est toujours possible de l'adapter (avec plus ou moins de facilité suivant la portabilité du système et la qualité de ses sources).

La deuxième solution est plus flexible que la première, mais elle peut être plus difficile à supporter par l'éditeur du système d'exploitation car l'architecture cible est moins figée.

Pour les autres parties de l'architecture, il est fourni en générale des règles pour écrire la couche intermédiaire permettant d'adapter le système d'exploitation. Cette couche est appelée *BSP* (*Board Support Package*). Il est aussi habituel de permettre l'ajout de pilotes (*drivers* en anglais) au système pour pourvoir gérer des types de périphériques non pris en compte.

Sélection de services

Avec la plupart des systèmes modulaires, il est possible de choisir les services qui seront fournis. Cela permet d'éliminer les services superflus.

Cette sélection de services est rendue possible principalement par deux méthodes :

1. l'utilisation d'une bibliothèque de modules compilés (codes objets) qui peuvent être liés pour obtenir le système final. C'est le cas de **VxWorks** [116], **QNX** [89], **Chorus OS** [100], **Virtuoso** [41], **C EXECUTIVE** [62] et **EYRX** [43]
2. la présence de sources que l'on peut choisir d'inclure ou non dans le flot de compilation. C'est le cas de **LynxOS** [76], **NucleusPLUS** [4], et **RTXC** [40]

Un paramètre important est la finesse du grain pour ce découpage en services : plus le grain est fin, plus le système résultant sera petit, mais plus il sera difficile de l'obtenir (des outils peuvent faciliter le choix de services à grain fin, voir la section 4.2.6). Nous pouvons constater que pour l'ensemble des systèmes d'exploitation modulaires les services fournis restent très généraux [49] (par exemple boîte aux lettres), ce qui a pour conséquence un grain de service assez grossier. Des services plus précis auraient permis un grain beaucoup plus fin, mais cela aurait fortement augmenté leur nombre dans les bibliothèques.

Configuration

Certains systèmes modulaires proposent en plus du choix de services à inclure la possibilité de les configurer. Cette configuration est effectuée par le biais de macros (basées sur le préprocesseur du langage C [96]), ce qui permet de définir les paramètres de configuration. Parmi ce type de système, nous avons : **LynxOS** [76], **VxWorks** [116], **eCos** [90], **RTXC** [40], **C EXECUTIVE** [62], **EYRX** [43].

Ces macros apportent plus de finesse pour la génération du système final. Elles restent cependant limitées à la définition de paramètres et à l'inhibition de certaines parties du code.

2.3.2.2 Les constructeurs de système d'exploitation

Les constructeurs de système d'exploitation sont en fait des interfaces utilisateur donnant accès à un ensemble d'outils permettant la configuration, la compilation, et le débogage d'application avec utilisation d'un système d'exploitation modulaire.

Ces interfaces sont généralement liées à un système d'exploitation donné et encapsulent la modularité de ces systèmes pour en faciliter l'accès à l'utilisateur.

Un bon exemple de constructeur de système d'exploitation est **Tornado** [117] pour le système **VxWorks** [116] de Windriver. Il permet à l'utilisateur de réaliser facilement les opérations suivantes :

- choix des services du système d'exploitation : un système de dépendances entre services aide au choix interactif des services souhaités pour l'application.
- configuration du système d'exploitation : les macros de paramétrage peuvent être modifiées par l'intermédiaire de l'interface.
- compilation, choix de la cible : si l'utilisateur dispose de plusieurs versions du système pour des cibles différentes, il peut effectuer les choix à partir de l'interface, et les fichiers de compilation (*makefile*) seront générés. La compilation est elle-même contrôlable à partir de l'interface.
- interface avec la cible : il est possible de transférer l'exécutable sur la carte cible, et de l'exécuter avec débogage.
- extension : l'interface peut être étendue par l'ajout de plugiciels (*plug-in*).

2.3.2.3 Les inconvénients et limitations de ces flots

L'utilisation de systèmes d'exploitation modulaires se généralise dans le monde des systèmes embarqués. En outre, l'existence d'intégrés aidant à la construction du système et au développement de l'application est un critère qui devient important dans le choix du système d'exploitation.

Cependant, ce type de flot présente toujours des inconvénients et des limitations qui rendent toujours très difficile le ciblage logiciel pour les systèmes embarqués complexes.

Trop de détails à gérer par le concepteur

Si le système d'exploitation propose des fonctions performantes, elles sont la plupart du temps spécifiques à l'architecture cible, et c'est au concepteur d'adapter l'application de haut niveau pour utiliser ces fonctions. L'adaptation se limite à quelques changements au niveau des appels système, mais cela reste une perte de temps et une source d'erreurs. Par exemple de nombreux appels système utilisent des identificateurs pour représenter les ressources concernées : se tromper d'identificateur est une erreur fréquente, qui peut être très difficile à repérer.

Le concepteur doit aussi effectuer lui-même le choix de tous les services requis pour que son application fonctionne. Certains constructeur de systèmes d'exploitation proposent un système de dépendance pour guider le concepteur dans ses choix : il sait ainsi, lorsqu'une fonctionnalité du système est choisie, quelles sont les autres fonctionnalités requises. Toutefois, cela n'élimine pas tous les cas d'erreur : l'utilisateur peut oublier de sélectionner une fonctionnalité, il peut aussi choisir la mauvaise fonctionnalité.

Enfin c'est au concepteur d'insérer tous les paramètres de spécialisation (comme les informations d'allocations : adresse, taille ou disponibilité des ressources, etc.). C'est une tâche qui est fastidieuse et source de nombreuses erreurs. Il convient donc de limiter au maximum le nombre de paramètres que le concepteur doit manipuler, or comme nous le verrons dans la section suivante, les flots basés sur des systèmes d'exploitation modulaires ne le permettent pas.

Architecture cible inadaptée

Ces flots sont toujours basés sur le schéma du processeur maître unique dans l'architecture, modèle qui n'est plus suffisant (voir la section 2.3.1.3). Il manque par exemple des fonctionnalités de communication entre tâches sur plusieurs processeurs.

Manque de souplesse

Les systèmes sont modulaires, mais le grain reste gros : chaque module apporte une fonctionnalité complète assez figée, ceci du fait que le code n'est pas fourni, ou que les macros permettant la configuration des modules manquent de pouvoir d'expression. Les macros sont en général basées sur le préprocesseur du langage C, qui n'est pas complet, comme nous le verrons dans la section 4.1.6.2. Cette limitation réduit notamment la factorisation possible des paramètres. Cette limitation réduit aussi les possibilités d'extension d'un constructeur de système d'exploitation : si un paramètre nécessite des calculs importants, l'outil de construction devra être modifié.

Une autre limitation importante est que les dépendances entre les divers modules sont directes. C'est-à-dire lorsqu'un élément a besoin d'une fonctionnalité, alors ce besoin est matérialisé par une dépendance avec un autre élément. Il n'est donc pas possible (sans conflit), d'avoir plusieurs éléments fournissant la même fonctionnalité. Par exemple il n'est pas possible d'offrir plusieurs choix d'implémentations qui pourraient présenter des performances différentes suivant les architectures cibles. En conséquence, il n'est pas possible avec ce système de dépendance d'effectuer une exploration de configuration de système d'exploitation pour trouver un optimum. Cette limitation rend aussi l'extension du système d'exploitation difficile : pour ajouter un élément il faut parcourir tous les éléments existants pour savoir quelles sont les dépendances à ajouter.

2.3.3 Approches récentes

Récemment de nombreuses approches ont vu le jour pour intégrer de manière plus efficace le système d'exploitation dans les flot de conceptions pour systèmes embarqués. Certaines approches se concentrent sur l'optimisation de l'utilisation des système d'exploitation, tandis que d'autres ont pour objectif de faciliter la conception du logiciel pour systèmes d'exploitation.

Comme nous le verrons dans cette section, de nombreuses approches sont complémentaires avec le travail de cette thèse, et il serait intéressant de les utiliser conjointement dans un flot.

2.3.3.1 Combiner génération de code séquentiel et utilisation d'un système d'exploitation

Présentation de l'approche

Cette approche est présentée dans [26]. Elle part d'une description de l'application logicielle représentée sous la forme de réseaux de processus, et se propose de générer le code sous la forme d'un minimum de tâches concurrentes, c'est-à-dire avec un maximum de code séquentiel. Un système d'exploitation classique est alors utilisé pour gérer les tâches.

L'objectif de cette approche est de réduire le nombre de changements de contexte en réduisant le nombre de tâches. L'idée étant que le problème de l'ordonnancement des tâches n'a pas de solution générale, il est préférable de limiter au maximum son utilisation.

Réalisation de l'approche

Dans un premier temps, un réseau de Petri unique est généré à partir de la spécification fournie sous la forme d'un réseau de processus. Cette opération est effectuée en deux temps :

1. compilation : chaque processus de la spécification est compilé en un réseau de Petri. Le résultat est donc un ensemble de réseaux de Petri.
2. édition de liens : les réseaux de Petri générés sont combinés en un seul en assemblant chaque paire de places de deux réseaux correspondant à une communication.

L'étape suivante consiste en la synthèse logicielle. Cette opération est elle aussi effectuée en deux temps :

1. ordonnancement : des graphes de franchissement sont construits. Chacun de ces graphes correspond à un ordonnancement.
2. génération de code : un programme séquentiel est généré pour chaque graphe de franchissement.

Lien avec le travail de cette thèse

Cette approche permet de produire un code d'application limitant au maximum le recours au système d'exploitation. Cependant, elle ne le supprime pas complètement, et rien n'est proposé pour faciliter sa génération.

Cette approche est donc complémentaire de celle proposée dans ce mémoire : elle peut être appliquée avant le ciblage avec génération de système d'exploitation, ce qui permettrait d'avoir automatiquement le logiciel et le système d'exploitation ciblés très performant en terme d'ordonnancement.

2.3.3.2 Systèmes d'exploitation abstraits

Présentation de l'approche

Cette approche présentée dans [30], propose une représentation à haut niveau du système d'exploitation.

L'objectif de cette représentation est de permettre la prise en compte des problèmes dynamiques, et de les simuler dès la description de haut niveau du système embarqué, ce qui n'était pas possible avec les méthodes classiques (basées par exemple sur SystemC).

Réalisation de l'approche

Le modèle employé pour l'application est celui de processus communicants gérés par le système d'exploitation de haut niveau. Ces processus communiquent par l'intermédiaire de ports en effectuant des appels à des fonctions de communication.

Le système d'exploitation est modélisé par une bibliothèque d'exécution fonctionnant sur station de développement et gérant les processus modélisant l'application et leur fournissant les fonctions de communication. Ensuite le code de l'application peut être raffiné en remplaçant les fonctions de communication par les appels au système d'exploitation final correspondants.

Lien avec le travail de cette thèse

Cette approche vise à prendre en compte le système d'exploitation à un niveau d'abstraction élevé à l'aide d'une bibliothèque d'exécution. Cette bibliothèque d'exécution peut tout à fait être considérée comme un système d'exploitation à part entière fonctionnant sur le processeur «station de travail». Il est donc tout à fait envisageable de générer ce système d'exploitation et de cibler l'application pour ce dernier. Ce travail peut donc être combiné avec celui présenté dans cette thèse pour pouvoir générer des systèmes d'exploitation pour les simulations de haut niveau, ou pour les réalisations de bas niveau.

2.3.3.3 Spécialisation de système d'exploitation

Présentation de l'approche

Cette approche présentée dans [5] propose de partir d'un système d'exploitation complet et de spécialiser ses services en fonction de l'architecture cible.

L'objectif de cette approche est d'obtenir un système d'exploitation adapté et optimisé à l'architecture cible en partant d'un système plus général.

Réalisation de l'approche

Cette approche suppose un comportement quasi-statique d'un système d'exploitation dans une application embarquée quelconque. Elle propose alors de fournir un certain nombre de services généraux pour le système d'exploitation qui sont spécialisés et optimisés en fonction de l'architecture cible.

La spécialisation consiste en une configuration fine du code du système d'exploitation avec comme paramètres d'entrée les informations sur l'architecture cible. Le grain de cette configuration peut être beaucoup plus fin que celui des systèmes d'exploitation modulaires, puisque même les algorithmes peuvent être configurés.

Lien avec le travail de cette thèse

Cette approche est assez proche de celle que nous présentons dans cette thèse. La différence est que nous ne supposons pas le comportement quasi-statique des systèmes d'exploitation et que cette thèse se focalise d'abord sur le choix du minimum de services (les services nécessaires et suffisants) avant d'effectuer toute spécialisation. Cela permet de proposer un plus grand choix de services tout en conservant un système de taille réduite.

2.3.3.4 Génération d'exécutif (travail de T. Grandpierre)

Présentation de l'approche

Cette approche présentée dans [49] propose de générer automatiquement l'exécutif distribué temps-réel implémentant une application sur une architecture parallèle. La description de l'application donnée en entrée est un graphe flot de données qui a été automatiquement ordonnancé. La partie système d'exploitation est donc réduite aux fonctions de communication et de synchronisation.

L'objectif de cette approche est de cibler automatiquement une application flot de données sur une architecture multiprocesseur, avec une seule tâche par processeur pour s'abstraire des indéterminismes de l'ordonnancement dynamique.

Réalisation de l'approche

La première étape est un ordonnancement de graphes flot de données où un ordre partiel est instauré entre certaines opérations indépendantes pour éviter les interblocages. Les méthodes d'ordonnancement statique dépassent le cadre de cette thèse, pour plus de détails à ce sujet vous pourrez consulter [94].

Le choix des services de communication et de synchronisation est effectué en fonction des opérations effectuées par l'application.

La génération du code du système d'exploitation, c'est-à-dire des fonctions de communication et de synchronisation, est effectuée à l'aide du programme d'expansion de macro m4 [3]. Ce programme traite un langage de macro à fort pouvoir d'expression, mais basé uniquement sur le principe de substitutions récursives. Sa puissance d'expression permet d'obtenir une très grande finesse de génération, comme par exemple la modification des algorithmes du code généré en fonction des paramètres de l'architecture cible, ce que ne permet pas le préprocesseur du langage C[10] habituellement utilisé (voir la section 2.3.2.1).

10. cpp (*C PreProcessor*)

Lien avec le travail de cette thèse

La dernière partie de ce travail est très similaire à ce que l'on se propose de faire (voir la section 4.2.7). Cependant, l'étape initiale est très différente car elle prend en entrée des descriptions de type flot de données, alors que nous prenons en entrée des descriptions structurelles (voir la section 4.2.5). De plus, le flot interdit l'utilisation d'un ordonnancement dynamique ce qui réduit les types d'applications et d'architectures possibles.

Plus généralement ce travail propose une solution complètement déterministe de génération d'exécutif, mais se restreint au cadre des applications flot de données uniquement.

2.4 Conclusion

Le terme système d'exploitation recouvre un domaine très vaste. Dans ce chapitre les systèmes d'exploitation ont été présentés suivant plusieurs points de vues. Ils ont été vus comme une abstraction du matériel et comme des gestionnaires de ressources. Puis leurs propriétés ont été explicitées.

Les systèmes d'exploitation pour les systèmes embarqués en sont une catégorie particulière. Ils doivent notamment pouvoir fournir une très grande variété de fonctionnalités spécifiques aux architectures, et ils doivent répondre à des contraintes très fortes en terme de consommation de surface et surtout de temps. Les architectures embarquées récentes, sont souvent composées de processeurs de types différents. Du fait de cette hétérogénéité, chacun doit disposer d'un système d'exploitation, ce qui peut être coûteux.

L'utilisation de systèmes d'exploitation dans les systèmes embarqués est coûteuse, et pose parfois des problèmes de performance, et de déterminisme. Il existe des solutions alternatives à leur utilisation, et notamment la génération d'exécutif statique. C'est une très bonne solution dans le cas des application de type flot de données, par contre l'ordonnancement statique présente des faiblesses lorsqu'il s'agit de gérer des événements.

Ces inconvénients peuvent cependant être réduits grâce à de nouvelles méthodes de conception qui permettent de sélectionner et de configurer les diverses parties du système d'exploitation. Récemment certaines approches tentent de réduire au maximum voir de supprimer l'ordonnancement dynamique, tandis que d'autres tentent de les prendre en compte le plus tôt possible dans les flots de conception.

Ce chapitre a mis en évidence l'intérêt de l'utilisation d'un système d'exploitation notamment grâce à l'abstraction du matériel qu'il permet. Cependant, les systèmes d'exploitation eux-mêmes dépendent très fortement de l'architecture cible. Or pour les systèmes embarqués spécifiques les architectures cibles sont rarement identiques. Une étape est donc nécessaire pour adapter l'ensemble constitué des systèmes d'exploitations et de l'application. Cette étape qui peut être nommée étape de ciblage logiciel est étudiée au chapitre suivant.

Chapitre 3

De la compilation au ciblage logiciel

Sommaire

Dans la conception de logiciel, de très nombreux travaux ont été et sont effectués sur la compilation de descriptions de haut niveau vers les langages machines (interprétés par les processeurs). Les habitudes de programmation pour les ordinateurs portent à croire que la compilation et l'édition de liens sont les seuls traitements à apporter au logiciel pour pouvoir le faire fonctionner sur une architecture cible. Le but de ce chapitre est de montrer que dans le cas des systèmes embarqués, une autre étape est nécessaire avant la compilation : elle adapte le logiciel à l'architecture cible spécifique et sera nommée dans ce manuscrit le ciblage logiciel. La première section rappelle les principes de la compilation, ce qui permet dans la deuxième section de faire la différence avec le ciblage logiciel. La dernière section s'intéressera au ciblage logiciel dans le cas des systèmes embarqués spécifiques.

3.1 Introduction sur la compilation

La compilation est un sujet récurent en informatique comme en électronique : en effet pour simplifier les descriptions, il est préférable d'utiliser des langages de haut niveau, proche d'un langage naturel. Cependant, ces langages ne peuvent en général pas être utilisés directement : il est nécessaire de les traduire dans des formes moins sympathiques, mais correspondant

directement à la réalité (dans le cas de l'électronique), ou directement utilisable par les outils ou les machines (dans le cas de l'électronique et de l'informatique). Cette section est une introduction à ce mécanisme de traduction appelé compilation.

3.1.1 Définitions de la compilation

La compilation [7] est la traduction automatique d'une description écrite dans un langage vers un autre langage. Très souvent, ce terme est restreint à la traduction d'un langage de programmation de haut niveau tel que le C vers le langage machine du processeur qui devra exécuter le programme.

Le flot de compilation est décomposé en plusieurs étapes (passes) consécutives :

- l'analyse lexicale qui produit une suite d'identificateurs correspondants aux instructions du programme d'entrée. Ces identificateurs sont appelés unité lexicales ou *token*.
- l'analyse syntaxique qui produit un arbre dont chaque nœud décrit une opération élémentaire, et dont les arcs indiquent les dépendances de contrôle.
- l'analyse sémantique qui enrichit cet arbre pour obtenir un arbre dit «décoré». Il s'agit d'une étape de synthèse.
- la génération de code qui parcourt l'arbre «décoré» en générant le code du langage cible correspondant à chaque noeud.

Les trois premières étapes ne dépendent que du langage d'entrée et sont appelées le frontal du compilateur. La dernière étape ne dépend que du langage cible et est appelée le dorsal du compilateur.

Remarques :

- Très souvent il y a un étape supplémentaire entre l'analyse sémantique et la génération de code : l'étape d'optimisation indépendante du langage cible. Cette étape se rencontre notamment avec les compilateurs vers les langages d'assemblages.
- L'étape de génération de code peut être très complexe lorsque des optimisations spécifiques au langage cible sont entreprises.
- Un abus de langage souvent commis est d'inclure l'édition de liens dans le flot de compilation vers le langage machine d'un processeur. L'édition de liens consiste à associer une adresse à chaque objet d'un programme en langage machine pour pouvoir le charger en mémoire et l'utiliser conjointement avec d'autres programmes.

3.1.2 Exemple de la compilation du langage C vers le langage d'assemblage

Un flot de compilation typique pour le langage C est représenté dans la figure 3.1. Il s'agit d'un flot séquentiel, le résultat de chaque étape étant l'entrée de l'étape suivante. Les trois premières étapes sont le frontal du compilateur, elles prennent en entrée le programme C, et restent inchangées quel que soit le processeur cible. La dernière étape est le dorsal du compilateur qui produit en sortie le programme en langage d'assemblage. Cette étape dépend du processeur cible.

La figure 3.2 donne un exemple simplifié de l'arbre obtenu après l'étape d'analyse sémantique. Il s'agit d'un arbre correspondant à une addition entre une variable (**idf**) et une valeur numérique (**cst**). Une telle représentation peut être vue comme une spécification pour la génération de code : les éléments sont les noeuds, et la structure est donnée par les arcs. Indépendamment des optimisations, le générateur de code parcourt l'arbre, et pour chaque

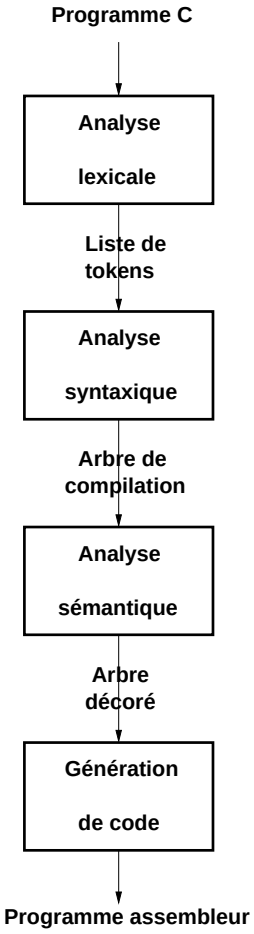

FIG. 3.1 – *Les étapes de la compilation d'un programme en langage C*

noeud produit le code assembleur correspondant. Lorsque le compilateur vise plusieurs processeurs (comme c'est le cas pour **gcc**), ces parties de code assembleur sont classées dans des bibliothèques spécifiques au processeurs. Elles associent chaque type de noeud à des suites d'instructions assembleur génériques pouvant être adaptées (en changeant les adresses ou les registres utilisés par exemple).

3.2 Le ciblage logiciel en général

Dans ce mémoire nous présentons un flot de ciblage avec génération de système d'exploitation. Dans cette section nous nous intéresserons au ciblage logiciel de manière plus générale,

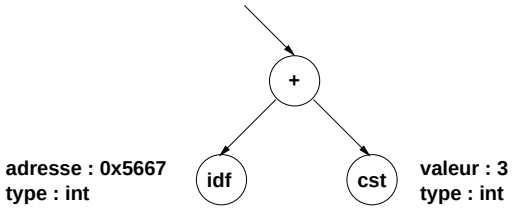

FIG. 3.2 – *Exemple d'arbre obtenu après analyse sémantique*

c'est-à-dire sans se soucier des architectures spécifiques, et sans l'introduction ou non de systèmes d'exploitation dans le flot. Dans un premier temps nous donnons quelques définitions concernant le ciblage logiciel, et notamment la nuance entre ciblage et compilation, puis nous nous intéresserons aux flots de ciblage avec les représentations utilisées et les étapes à suivre.

3.2.1 Notion d'exécutable

Le logiciel peut se présenter sous plusieurs formes, dans plusieurs langages [93]. Cependant, le processeur sur lequel il s'exécute ne comprend en général qu'un langage basé sur des suites de 0 et de 1, on parle de langage machine. Un programme est dit exécutable, lorsqu'il se présente sous cette forme, c'est-à-dire qu'il est «compris» par le processeur, et donc qu'il peut être exécuté [10].

On parle parfois de langage exécutable pour les langages qui peuvent être traduits en un langage exécutable (par exemple le C), par opposition à ceux qui ne le permettent pas (par exemple UML).

Il existe de très nombreux processeurs, et ils ne sont en général pas compatibles entre eux. En particulier, ils ne «comprennent» pas le même langage machine [119]. Dès lors la notion d'exécutable n'est pas générale : un programme est sous une forme exécutable pour un processeur donné.

De nos jours il existe aussi ce qu'on appelle des machines virtuelles, qui sont des couches logicielles ajoutées aux processeurs, et comprenant le même langage quel que soit le processeur sur lequel elles sont installées. L'exemple le plus connu est le langage Java qui s'exécute sur une machine virtuelle Java [101]. Dans ce cas, un programme exécutable pour la machine virtuelle, l'est indépendamment du processeur. L'avantage est qu'un même programme peut fonctionner quel que soit le processeur. L'inconvénient est que ces machines sont en général très lentes.

Dans ce mémoire, nous considérons qu'un exécutable est la forme binaire d'un programme directement utilisable par un processeur.

3.2.2 Définition du ciblage logiciel

Le ciblage logiciel consiste à produire une description exécutable (pour une architecture matérielle donnée) d'une application logicielle, à partir d'une ou plusieurs descriptions de haut niveau de cette dernière, ne contenant pas de détails d'implémentation [47].

3.2.3 Ciblage logiciel idéal

Le but d'un flot de ciblage est de permettre au programmeur de développer son application logicielle à haut niveau, sans se préoccuper des détails d'implémentation spécifiques à l'architecture. Il est aussi souhaitable que ce flot ne soit pas trop contraignant quant au codage de l'application. Enfin, il faudrait que le programmeur puisse remonter assez facilement du code ciblé au code, pour faciliter le travail de vérification, et correction des erreurs.

Idéalement, un flot de ciblage pourrait cibler n'importe quelle application (quel que soit son style de programmation), pour n'importe quelle architecture cible. De plus, le code ciblé par un tel flot contiendrait sans aucune modification le code initial, c'est-à-dire que l'opération de ciblage aura consisté uniquement à ajouter le code nécessaire pour rendre la description initiale compatible avec l'architecture.

3.3 Représentations pour le ciblage logiciel

3.3.1 Représentations initiales pour le ciblage logiciel

Comme tout flot dont le but final est la génération de code les flots de ciblage utilisent au minimum deux types de représentations : une représentation d'entrée, et une représentation de sortie.

3.3.1.1 La représentation d'entrée

La représentation d'entrée est une spécification du système. Cette spécification peut contenir des informations structurelles, des informations comportementales, mais aussi des paramètres qui précisent les détails des éléments à générer. Ces paramètres peuvent être présents sous la forme d'annotations associées aux éléments de la spécification.

La représentation d'entrée n'a pas besoin d'être exécutable. Au contraire, les informations nécessaires pour l'exécution ou la simulation peuvent interférer avec le ciblage. Il peut être par exemple suffisant pour la description comportementale de donner des listes de fonctionnalités requises, sans donner leurs réalisations.

Cette représentation est souvent une forme intermédiaire, produite par une étape de synthèse. Elle n'a donc pas forcément de format humainement accessible : elle peut par exemple être limitée à une structure de données interne à un outil de ciblage.

3.3.1.2 La représentation de sortie

Le résultat de la génération est donné dans la représentation de sortie. Étant la cible de la génération, il n'y a pas de contraintes particulière sur cette représentation.

Elle décrit le système donné en entrée, mais avec tous les détails d'implémentation obtenus par le générateur de code.

3.3.1.3 Représentation supplémentaire, nécessaire pour le fonctionnement du ciblage

Le ciblage consiste souvent à produire et assembler des parties de code élémentaires (décrite dans la représentation de sortie) correspondant à certains éléments donnés dans la spécification d'entrée. L'ordre dans lequel les parties de code élémentaires doivent être assemblées est déduit de l'ordre dans lequel sont présentés les éléments du système, c'est à dire les informations structurelles de la spécification (voir la section 3.3.1.1). La production des parties de code élémentaires est dirigée par les paramètres associés aux éléments de la spécification.

Le cibleur de code doit donc être capable de produire les parties élémentaires de code pour chaque type d'élément de la représentation d'entrée. La méthode couramment employée est de ranger toutes les parties de code élémentaire possibles dans une bibliothèque. Le problème se réduit alors à la recherche dans la bibliothèque des portions de code correspondant aux éléments de la spécification. Pour le résoudre il convient d'avoir une représentation pour décrire le contenu de cette bibliothèque. Cette représentation doit être relationnelle pour donner les correspondances entre les éléments de spécification et les portions de code finales. Elle doit être aussi comportementale pour la description des portions de code.

Si la représentation de sortie[1] est suffisamment souple pour pouvoir décrire les parties élémentaires de code, leur paramétrage et leur assemblage, elle peut être utilisée également

1. c'est-à-dire celle utilisée pour décrire le code généré

pour la description de la partie comportementale de la bibliothèque. Si ce n'est pas le cas, il convient alors d'encapsuler les parties de code final dans une autre représentation plus flexible.

3.3.2 Les étapes du ciblage logiciel

Nous allons présenter dans cette section les étapes à suivre lors du ciblage logiciel. Nous procéderons par ordre chronologique, en partant des descriptions de l'application et de l'architecture matérielle cible, jusqu'à la production des exécutables finaux.

Le but de cette section n'est pas de présenter une méthodologie de ciblage, mais plutôt de montrer le type d'opérations, et la chronologie, qu'il sera généralement nécessaire de suivre. Nous nous contenterons donc ici de présentations sommaires des étapes du ciblage, sachant que les détails les concernant varient suivant la méthodologie employée.

La figure 3.3 présente les diverses étapes d'un flot de ciblage. Ces étapes sont présentées dans les sections suivantes.

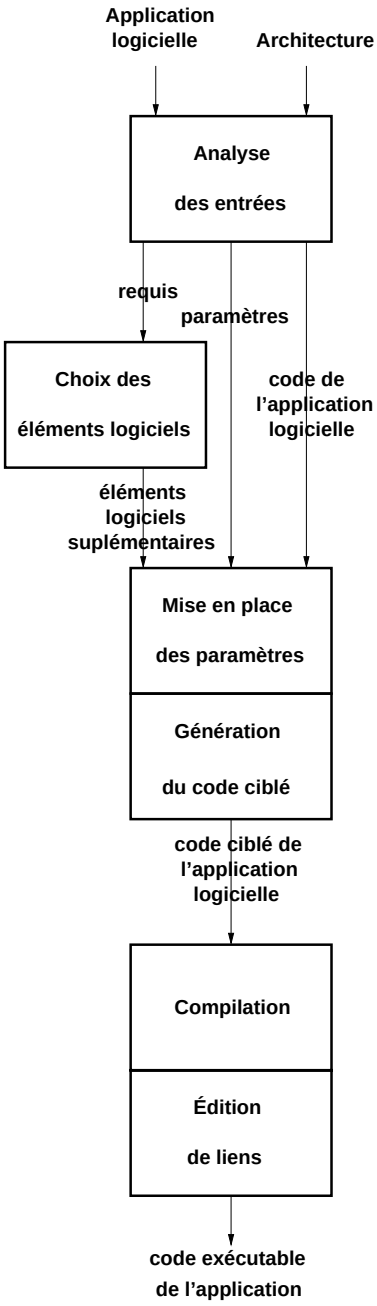

FIG. 3.3 – *Les étapes du ciblage logiciel*

3.3.2.1 Analyse des entrées : application et architecture matérielle

Cette étape consiste en l'acquisition des informations nécessaires au ciblage. Nous allons voir les types d'information requis pour effectuer un ciblage, et comment nous pouvons y accéder.

Les informations nécessaires pour le ciblage

Les informations nécessaires pour le ciblage sont tout d'abord les services requis : dans cette catégorie, se trouvent toutes les informations concernant les fonctionnalités logicielles à ajouter. Parmi celles-ci se trouvent les fonctionnalités de communication, les fonctionnalités du système d'exploitation, et les gestionnaires de périphériques.

D'autres informations requises prennent la forme de paramètres : dans cette catégorie, se trouvent toutes les valeurs qui permettront de configurer les éléments logiciels. Ces paramètres peuvent avoir les origines suivantes :

- les informations sur le matériel : quel type de processeur, quel type de périphériques, etc.
- les allocations, c'est-à-dire les adresses mémoire des divers éléments (logiciels ou matériels).
- les contraintes imposées par l'environnement. Il peut s'agir de contraintes temporelles (délais à respecter, débit, etc.), de contraintes de surface (quantité de mémoire limitée), de contraintes de consommation ou d'autres contraintes plus spécifiques (liées par exemple à du matériel particulier). Ces contraintes peuvent elles aussi avoir une influence sur le choix des services.

Accès aux informations

Ces informations sont en règle générale dispersées dans plusieurs endroits, et sous plusieurs formes :

- directement dans la spécification, sous la forme d'annotations
- déductibles à partir de la structure de la spécification
- déductibles à partir d'autres informations de la spécification
- données en langage naturel

Il est préférable de rassembler ces informations avant de commencer toute opération de ciblage.

3.3.2.2 Choix des éléments logiciels à ajouter

Dans cette étape sont sélectionnés les divers éléments logiciels à ajouter pour le ciblage. Ces éléments sont à choisir en fonction de leurs fonctionnalités, sachant qu'au final il faut que toutes les fonctionnalités requises par le système soient fournies (voir la section 3.3.2.1).

Les éléments logiciels peuvent avoir plusieurs formes. La forme la plus courante consiste en l'ensemble des fonctions présentes dans le système d'exploitation choisi [30]. Dans ces conditions, l'étape de choix des éléments logiciels consiste simplement au choix du système d'exploitation qui fournira le plus de fonctionnalités requises par l'architecture, les autres devant être développées. L'inconvénient est évidement qu'il y a souvent de très nombreuses fonctionnalités inutiles apportées par le système choisi [47]. Certains systèmes d'exploitation peuvent être paramétrés en fonction des besoins (voir la section 2.3.2.1). Dans ce cas l'étape de choix consiste au choix du système qui fournira le plus de fonctionnalités requises par l'architecture, et à l'inhibition de ses fonctionnalités superflues. Il reste toujours nécessaire de développer les fonctionnalités manquantes. Des flots de ciblage plus élaborés se basent sur des bibliothèques. L'étape de choix consiste alors au choix des bons éléments dans la bibliothèque.

S'il manque des fonctionnalités, il faut les développer, cependant elles peuvent être ajoutées à la bibliothèque pour pouvoir être réutilisées dans d'autres conditions.

3.3.2.3 Mise en place des paramètres et production du code logiciel exécutable

Dans cette étape le code exécutable est généré. Pour cela il faut introduire les paramètres précédemment définis (voir la section 3.3.2.1), et éventuellement adapter le code de l'application à l'environnement utilisé (système d'exploitation et architecture). Là encore les méthodes varient suivant le code des éléments logiciels. Parfois le code est présent sous la forme d'un simple exécutable (voir la section 2.3.1). Dans ces conditions, les paramètres ne peuvent être mis en place qu'au niveau de l'application, qui doit être réécrite en fonction de l'environnement, avec par exemple l'utilisation des appels système fournis par le système d'exploitation. Une autre conséquence est que de nombreux paramètres (comme les contraintes sur l'ordonnancement) ne peuvent pas être introduits. Parfois le code est présent sous la forme de sources paramétrables (voir la section 2.3.2.1).

3.3.2.4 Compilation et édition de liens

La dernière étape consiste en la compilation et en l'édition de liens pour la production du code final. Avec cette étape, le flot rejoint les flots standards de conception.

3.4 Le ciblage logiciel pour les architectures spécifiques

Cette section présente les difficultés particulière du ciblage logiciel dans le cas des systèmes embarqués spécifiques. Pour atténuer ces difficultés, la faisabilité de flots de ciblage automatiques est étudiée. Cette section propose enfin l'intégration des systèmes d'exploitations dans le flot de ciblage.

3.4.1 Difficultés du ciblage logiciel

Nous présentons dans cette section les difficultés que présente le ciblage logiciel. Ces difficultés sont de deux ordres : la variété des domaines, et la quantité d'opérations à effectuer.

3.4.1.1 La variété des architectures cibles

La variété des processeurs

Il existe de très nombreux processeurs, qui ne sont pas compatibles entre eux et qui ne comprennent pas le même langage. En première analyse, il pourrait sembler que seul le compilateur soit concerné par cette diversité de processeurs, et qu'il suffit de choisir le compilateur correspondant au processeur pour s'affranchir des incompatibilités.

En fait, cette première vue est inexacte : le type du processeur a une influence non négligeable sur le code de haut niveau du logiciel, et notamment dans le cas des applications embarquées. Pour voir pourquoi, nous allons montrer quelques caractéristiques d'un processeur qui peuvent conduire à des adaptations difficiles voire impossibles à la compilation :

- les types des données : les processeurs possèdent en général un type de données privilégié, correspondant à la taille de leurs registres [118]. Pour qu'une application soit performante, il est préférable d'utiliser ce type au maximum pour les variables. Évidement, suivant le processeur ce type n'est pas le même : cela peut être entier sur 8 bits,

16 bits, 32 bits, ou d'autres types plus exotiques. Une solution apportée par le langage C est de nommer ce type privilégié le type int, auquel cas, c'est bien le compilateur qui décide du type à prendre [66]. Mais cela pose toujours des problèmes lorsqu'il s'agit de communiquer ces données : le type doit correspondre à l'émission et à la réception, et si les deux n'ont pas le même type de base il faut en choisir un, et l'imposer aux deux. On se retrouve donc toujours dans une situation ou il faut changer de type suivant le processeur et son environnement.

- la représentation des données : les données sont représentées en mémoire par des paquets de bits. Cependant, il existe de nombreuses manières pour interpréter ces bits (par exemple en complément à deux, en mantisse et exposant, etc.), et il est aussi possible des les interpréter en gros-boutistes *big endian* ou petit-boutistes *little endian*. Il est important de noter que pour un même type il peut exister plusieurs représentations possibles.
- les instructions : chaque type de processeur dispose d'un jeu d'instructions qui lui est propre. C'est au compilateur de produire le code compréhensible par le processeur, et en fait n'importe quel programme de haut niveau doit pouvoir être traduit par ce dernier. Cependant, une description de haut niveau générale ne prend pas en compte les spécificités d'un processeur, notamment en termes de performance. Il est parfois préférable de changer l'algorithme pour un processeur donné. En outre, un certain nombre d'instructions ne peuvent pas être représentées à haut niveau : c'est le cas par exemple des instructions de contrôle de la consommation que possèdent de nombreux processeurs utilisés dans les systèmes embarqués [119]. Dans ces deux cas le compilateur ne peut pas agir.
- les interruptions : la plupart des processeurs disposent d'un mécanisme d'interruptions, qui permet de générer des événements. Ces mécanisme, sont toujours très spécifiques à chaque processeur, et peuvent être d'une grande complexité. Cependant, ils sont très rarement pris en compte par le compilateur, car ils ne font pas partie du modèle de programmation classique (séquentiel continu).

Comme on le voit les différences entre les processeurs sont de plusieurs ordres, et ne peuvent être prises en compte par les seuls compilateurs.

La variété des communications

Suivant l'architecture cible, des modes de communication très différents peuvent être utilisés. Il peut y avoir des différences d'un point de vue topologique : communications point à point ou non, hiérarchique ou non, à base de barres croisées ou non, etc. Les différences peuvent être d'un point de vue protocolaire : avec ou sans connexion, avec ou sans poignée de main, avec ou sans rafale, etc. Enfin elles peuvent être d'un point de vue matériel : avec ou sans mémoire, avec ou sans accès direct à la mémoire (*DMA*), etc.

Toutes ces caractéristiques peuvent se combiner, ce qui fait un très grand nombre de cas possibles, pour chacun desquels un ciblage différent sera nécessaire.

La variété des circuits connectés au processeur

Suivant l'architecture cible, des circuits différents peuvent être utilisés, et même s'ils peuvent avoir des fonctionnalités identiques, ils risquent d'avoir des modes de fonctionnement et de communication différents. Cela implique que chaque circuit induira des détails logiciels différents, et donc change les opérations de ciblage à effectuer.

Logiciel abstrait

Le but du ciblage est d'adapter le logiciel aux architectures cibles. Ce logiciel est donc très général, et utilise des primitives abstraites pour interagir avec l'extérieur. Nous avons vu dans les sections précédentes que les éléments d'architecture étaient très variés. Pour pouvoir les relier avec le logiciel, il convient de fournir au concepteur du logiciel les primitives abstraites correspondante.

Pour simplifier et clarifier la programmation, il convient que le nombre de primitives soit le plus faible possible. Dès lors, une partie du travail de ciblage sera d'adapter ces quelques primitives à la diversité des architectures cibles.

Conséquences sur la variété des architectures cibles

Comme nous venons de le voir, le nombre d'architectures cibles possibles pour une même application au départ est énorme. Évidement de très nombreuses configurations ne sont pas valables, mais il est souvent difficile à priori de savoir si une architecture sera valable ou non (notamment du point de vue temporel). Dès lors, il arrive souvent que le choix d'une architecture soit effectué au terme d'une suite d'essais sur différentes architectures [64][14].

Dans ces conditions, il apparaît important que le ciblage soit le plus rapide possible, et demande le moins d'efforts possible. Ainsi, il sera possible d'obtenir plus rapidement une réalisation finale valide, et il sera aussi possible de tester un plus grand nombre d'architectures, ce qui donne plus de chances pour atteindre un optimum.

3.4.1.2 La diversité des connaissances liées au ciblage logiciel

Le ciblage requiert des domaines de compétence très divers ; nous allons ici en donner une liste des principaux, et expliquer en quoi ils sont nécessaires.

Connaissances sur les processeurs

Il est important de bien connaître le processeur cible lorsqu'on veut adapter le logiciel qui doit être exécuté dessus. Il faut notamment connaître :

- les «dimensions» du processeur : la taille de ses registres, la taille de ses bus adresse et donnée, pour savoir quels types de données utiliser.
- les modes de communication du processeur : utilise-t-il une mémoire commune données/programme (Von Neumann) ou distincte (Harvard), quel protocoles utilise-t-il pour lire/écrire les données[2], quel est son système d'interruption[3].
- le langage d'assemblage du processeur : pour pouvoir écrire toutes les parties du code spécifiques au processeur, comme la gestion des interruptions, ou le changement de contexte.
- les performances du processeur suivant les cas (vitesse d'exécution, temps de réaction aux interruptions, etc.) : pour pouvoir choisir les meilleurs algorithmes et réalisations.
- l'architecture du processeur (pipeline, RISC ou CISC, etc.) : cela aidera aussi au choix des algorithmes et des réalisations.

Connaissances sur les communications

Une partie du ciblage consiste en la réalisation des communications entre le logiciel à cibler et le reste. Pour ce travail il est nécessaire d'avoir des connaissances en communication, et plus précisément :

- protocole : il faut connaître leurs caractéristiques.

2. Certain protocoles nécessitent une programmation particulière (par exemple le 8051 [57]).

3. Chaque système se programme de manière différente, et la plupart du temps en langage d'assemblage uniquement.

- langage de description de communication : très souvent les communications sont décrites dans un langage qui leur est propre (par exemple **SDL** [58]), qu'il faut donc savoir interpréter.
- méthode de réalisation des communications : il existe un certain nombre de méthodes pour réaliser des communications, certaines complètement logicielles, certaines faisant appel à du matériel (par exemple un DMA).

Connaissances en description d'application

Connaître l'application permet de faire de meilleurs choix lors du ciblage. Cela implique de connaître également :

- le langage dans lequel l'application est décrite : très souvent c'est du **C** ou du **C++** [66][98], cependant d'autres langages peuvent être utilisés comme **Ada** [6], **SDL** [58], Esterel [17] etc.
- les algorithmes habituellement utilisés dans une application donnée, pour savoir quelles sont les implémentations optimales
- les paramètres de ces applications : nombre de données à traiter, nombre de tâches possibles, etc.

Connaissances en architecture

Le ciblage sur une architecture implique la compréhension de cette dernière, il est donc nécessaire de connaître :

- les langages de description d'architectures (**VHDL** [8], **SystemC** [102], etc.)
- les principes utilisés dans les architectures comme la modularité, l'instanciation, etc.
- les différents types d'architectures, et les conséquences sur l'implémentation [13]

Connaissances en électronique

Bien qu'il s'agisse de ciblage logiciel, la cible est une architecture matérielle comprenant des éléments électroniques. Si de vastes connaissances en électronique ne sont pas nécessaires, il est important d'avoir des notions sur :

- les bases de l'électronique numérique (portes), sachant que le processeur cible peut être connecté à un peu de logique
- quelques circuits électroniques comme les DMA, les arbitres de bus, et autres contrôleurs
- les bases de l'électronique analogique pour être capable de prendre en compte les problèmes liés à la consommation

Connaissances sur les systèmes d'exploitation

Une bonne part du ciblage consiste dans le choix et l'adaptation du ou des systèmes d'exploitation qui seront utilisés. Il est donc important de bien connaître les principes liés aux systèmes d'exploitation.

3.4.1.3 Nombreuses opérations, et nombreuses erreurs

Une bonne part du ciblage consiste en une fastidieuse suite d'opérations élémentaires (construire de nombreuses tables de correspondances, choisir et assembler de nombreux composants logiciels, etc.). Toutes ces opérations sont sources de nombreuses erreurs de frappe difficiles à détecter. De plus, elles sont à recommencer complètement à chaque fois que l'architecture cible change [47].

3.4.2 Faisabilité de l'automatisation du ciblage logiciel

Dans cette section, nous allons voir dans quelle mesure il est possible d'automatiser certaines opérations dans le ciblage logiciel. Le but est d'accélérer le processus de ciblage et de limiter les erreurs, ce qui permettra une exploration plus large des diverses solutions possibles pour réaliser une application.

Pour ce faire nous allons essayer de localiser la redondance dans les opérations de ciblage, puis nous analyserons les types de choix à effectuer, et nous déterminerons ce qui est systématique dans les flot de ciblage. Enfin nous conclurons en précisant ce qui serait possible d'automatiser.

La redondance des opérations de ciblage

Il y a redondance dans les informations : très souvent une même information se trouve présente dans plusieurs endroits dans les entrées du flot de ciblage ; par exemple le format de données dans un protocole de communication peut se trouver dans la description de l'architecture et dans les spécifications ou même le code de l'application à cibler. Il y a aussi redondance dans les fonctionnalités des éléments logiciels : il est fréquent que deux éléments logiciels aient des fonctionnalités communes, ce qui fait que s'ils sont tous les deux sélectionnés pour le ciblage, il y aura du code inutile.

Cette redondance est source de confusion lors d'un ciblage manuel, mais elle peut être un avantage pour l'automatisation, car elle peut permettre certaines factorisations.

Le problème des choix à effectuer lors d'un ciblage

Nous avons vu précédemment qu'il fallait faire un certain nombre de choix lors du ciblage : il faut par exemple choisir des éléments logiciels en fonction des fonctionnalités requises, des contraintes et des performances. Il faut aussi choisir les valeurs de paramètres de configuration des éléments logiciels (par exemple la taille des mémoires tampons), en fonction des contraintes et des performances.

Si le choix en fonction des fonctionnalités requises peut être effectué automatiquement par simple gestion de dépendances (comme par exemple dans Tornado de Windriver [117]), le choix de fonctions de contrainte et de critères de performance (vitesse, taille, ou consommation) est plus délicat. Dans ce dernier cas il est nécessaire d'avoir une évaluation des conséquences des choix sur les performances, ce qui peut être d'autant plus difficile à obtenir si les éléments logiciels possèdent de nombreux paramètres. Dans tous les cas le résultat est rarement certain, et il est donc préférable de laisser à l'utilisateur la possibilité d'influencer ces choix.

Ce qui est systématique dans le flot de ciblage

Pour automatiser le flot de ciblage, il est intéressant de repérer les actions systématiques, c'est-à-dire les actions qui ne demandent pas d'intelligence.

Dans le cas du ciblage, les actions suivantes sont directement systématiques :

- l'acquisition des informations : cette action consiste juste en la lecture des entrées du flot avec extraction des informations. À partir du moment où les entrées sont bien définies (par exemple des fichiers sous un format connu), les techniques de base d'analyse syntaxique peuvent être utilisées.
- la mise en place des paramètres : en supposant que les divers éléments logiciels pour le ciblage sont décrits suivant un formalisme connu, et que les paramètres sont directement accessibles, cela revient à effectuer des opérations d'expansion de macros.
- la compilation et l'édition de liens : les compilateurs et éditeurs de liens existent déjà et sont automatiques, il suffit juste d'automatiser leur appel dans le bon ordre.

Il existe aussi d'autres actions, qui peuvent devenir en partie systématiques :

- le choix des éléments logiciels nécessaires : si tous les choix ne sont pas systématiques, un certain nombre peuvent l'être, comme la détection de tous les éléments ayant une certaine fonctionnalité, et le choix des éléments compatibles avec une certaine architecture matérielle. Cela suppose que le flot de ciblage dispose de descriptions des fonctionnalités de ces éléments ainsi que de leurs compatibilités.
- l'adaptation du code de l'application peut elle aussi être en partie systématique [49] : cela concerne notamment les appels de communications qui peuvent être remplacés par les bons appels au système d'exploitation.

Conclusion sur l'automatisation du ciblage logiciel

Nous avons vu dans cette section que si l'automatisation complète du ciblage n'est pas encore réaliste, une bonne part du flot peut l'être avec seulement quelques interventions humaines lorsqu'il s'agit d'effectuer les choix concernant les performances.

Avec une automatisation partielle, il y aura gain de temps : il est toujours plus rapide de générer et d'adapter du code automatiquement plutôt qu'à la main (hors des considérations d'optimisation, qui reste un problème NP-complet). Il y aura aussi un gain au niveau des erreurs : toutes les opérations répétitives et sources d'erreurs pour un homme peuvent être prise en charge par l'outil de ciblage automatique. Il peut être aussi possible de prouver des équivalences entre l'entrée et la sortie du flot de ciblage, auquel cas nous avons la certitude de la validité du résultat. Enfin même s'il est toujours recommandé d'avoir une bonne compréhension des concepts liés au ciblage, il n'est plus nécessaire au concepteur de connaître tous les détails d'implémentation.

3.4.3 Les systèmes d'exploitation dans le ciblage logiciel

3.4.3.1 Système d'exploitation : cible et source logicielle pour le ciblage

Les systèmes d'exploitation ont un statut particulier dans les flots de ciblage car ils peuvent être à la fois vus comme une partie logicielle à cibler pour une architecture matérielle et comme paramètre d'un ciblage pour faire fonctionner une application pour cette architecture.

Un flot de ciblage complet devrait être capable de prendre en compte ces deux vues du système d'exploitation : il devrait être capable de générer ou de choisir le système d'exploitation avec les bons paramètres pour faire fonctionner l'application, et il devrait être capable de cibler ce même système d'exploitation pour l'architecture matérielle.

3.4.3.2 Apport de l'utilisation d'un système d'exploitation pour le ciblage logiciel

Nous avons vu dans la section 2.1.1.1 qu'un système d'exploitation peut être vu comme une abstraction du matériel, qui soulage le programmeur de la programmation des détails spécifiques à l'architecture cible. L'utilisation d'un système d'exploitation peut donc grandement simplifier le ciblage logiciel. Il faut par contre que, pour chaque architecture, il existe un système d'exploitation capable de la gérer. Nous avons vu que ce n'était pas le cas : il est très souvent nécessaire d'adapter un système d'exploitation plus ou moins générique à une architecture donnée.

Chaque système d'exploitation fournit une API pour l'application logicielle. C'est cette API qui représente les contraintes pour la programmation de l'application. L'avantage est qu'une API ne fournit pas de contraintes pour le style de programmation, comme cela peut être le cas pour les logiciels de synthèse de haut niveau, mais seulement des contraintes pour

les points de communication et de synchronisation, qui doivent être des fonctions du système d'exploitation[4].

Ces fonctions système peuvent être très générales, mais alors elles manquent d'efficacité, ou elles peuvent être plus spécifiques à l'architecture cible, ou au système d'exploitation. Dès lors, si l'application logicielle doit être ciblée pour une autre architecture, ou si un autre système d'exploitation doit être utilisé, il faut changer son code. Ce dernier point n'est pas souhaitable. Nous verrons dans la section 4.2.7.5 comment nous pouvons l'éviter en utilisant le polymorphisme de la programmation orientée objet.

3.5 Conclusion

Dans ce chapitre le concept de ciblage logiciel a été introduit : il s'agit de l'adaptation du logiciel à l'architecture cible. C'est une étape obligatoire dans les flots de conception, mais elle est souvent négligée par rapport à la seule compilation. Pourtant, le ciblage logiciel est difficile, fastidieux, et sujet à de nombreuses erreurs.

Cette étape peut être grandement automatisée, et l'inclusion du système d'exploitation simplifie ce travail. Dès lors le concepteur du logiciel n'a plus qu'à se concentrer sur les fonctionnalités, les détails d'implémentation étant en grande partie automatiquement résolus par le flot de ciblage.

L'automatisation de cette étape avec l'intégration du système d'exploitation est le but de cette thèse. La solution proposée est présentée au chapitre suivant.

4. des appels systèmes en fait

Chapitre 4

Ciblage automatique avec génération de systèmes d'exploitation

Ce chapitre montre en détail le flot de ciblage logiciel qui a été développé au cours de cette thèse. Ce flot propose un outil qui effectue automatiquement la génération des systèmes d'exploitation ciblé en prenant en entrée une description de haut niveau de l'application et de l'architecture. Cette génération utilise une bibliothèque de composants de systèmes d'exploitation qui sont adaptés et combinés pour obtenir le logiciel final. La première section présente les représentations utilisées dans ce flot, et la deuxième détaille le fonctionnement des diverses étapes.

4.1 Les représentations pour le flot de ciblage

La méthode de ciblage logiciel de cette thèse s'appuie sur plusieurs représentations pour modéliser, cibler et générer des système d'exploitations spécifiques. Ces représentations, étudiées dans cette section, permettent de décrire la spécification d'entrée et la bibliothèque de systèmes d'exploitation.

4.1.1 Organisation générale des représentations utilisées pour la génération de systèmes d'exploitation

La méthodologie de génération de système d'exploitation utilise trois représentations en entrée pour générer le code du système d'exploitation. Ces représentations, et leurs interactions sont présentées figure 4.1 :

- La spécification d'entrée utilise le langage **Colif** [24]. Cette spécification décrit la topologie de l'architecture cible logicielle et matérielle. Elle sera présentée dans la section 4.1.5.1.
- La bibliothèque de système d'exploitation fournit les éléments qui, paramétrés et assemblés, produiront le code du système d'exploitation final. Cette bibliothèque utilise deux représentations : une représentation relationnelle décrite dans le langage **Lidel** (voir la section 4.1.3), donnant les informations permettant de retrouver et d'identifier les divers éléments, et une représentation comportementale[1] qui donne le code de ces éléments sous la forme de macros (voir la section 4.1.6.2).

1. Cette représentation peut être décrite dans n'importe quel langage.

– La représentation utilisée pour le code généré dépend de celle choisie dans la représentation comportementale des éléments de bibliothèque. La flexibilité du flot permet de prendre n'importe quel langage de programmation pour cette représentation, et même d'utiliser plusieurs langages différents dans la même bibliothèque.

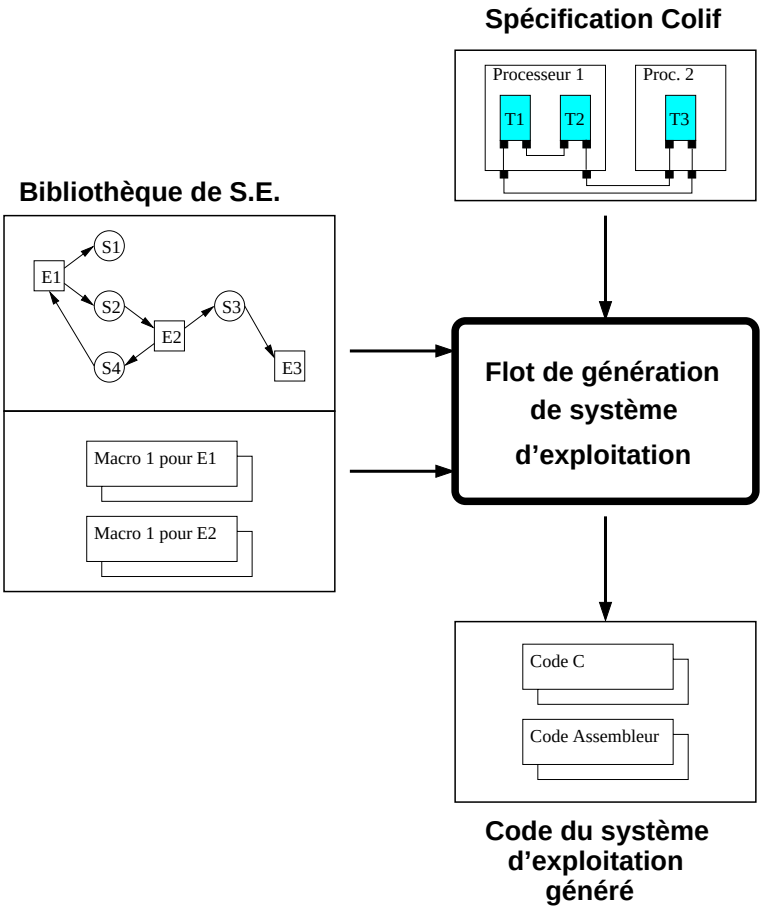

FIG. 4.1 – *Les représentations pour le ciblage avec génération de système d'exploitation*

Dans le flot de génération, l'utilisateur ne doit fournir que la spécification Colif. Dans le cas où il manque des éléments dans la bibliothèque, il enrichira celle-ci avant de pouvoir effectuer la génération.

4.1.2 Construction des langages Lidel et Colif

Le but de cette section est de présenter le langage **Middle** qui a servi à définir le langage de description de bibliothèque de système d'exploitation **Lidel**, et le langage de spécification **Colif** 4.1.5.1. L'organisation de ces langages est présenté figure 4.2.

4.1.2.1 Motivations pour introduire le langage Middle

Les langages **Lidel** et **Colif** sont tout deux des langages de description structurelle. Dans les deux cas, ces langages peuvent être vus comme des ensembles de données complexes.

Plutôt que de définir deux langages indépendants, nous avons décidé d'utiliser un méta-langage de description de données complexes. Les caractéristiques souhaitées étaient :

– avoir un seul analyseur pour tout langage de description non comportementale. Cet

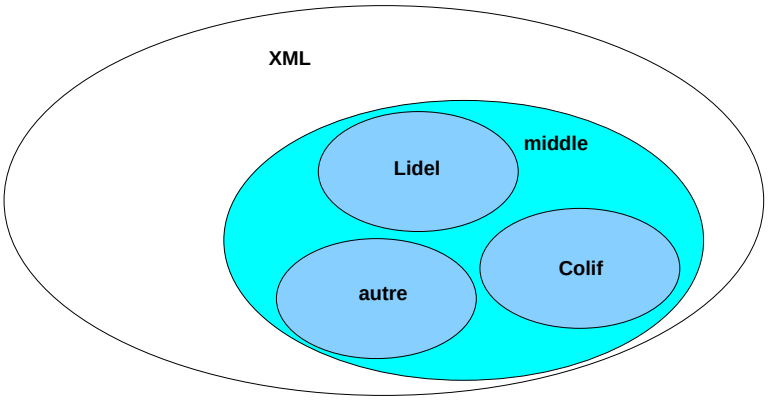

FIG. 4.2 – *Langages XML, Middle, Lidel et Colif*

analyseur doit pouvoir détecter les erreurs lexicales, syntaxiques et sémantiques liées à **Middle** mais aussi le maximum d'erreurs sémantiques des langages dérivés.

- disposer d'une API générique pour manipuler tous ces langages
- pouvoir générer facilement une API spécifique pour chaque langage dérivé de *Middle*
- utiliser une représentation standard reconnue

4.1.2.2 Utilisation de XML

Pour concilier toutes les caractéristiques citées dans la section précédente, il a d'abord été prévu d'utiliser le langage **XML** [115]. C'est un langage qui a été standardisé par le W3C, et qui connaît un grand succès dans le monde de l'informatique grâce à sa souplesse, sa clarté et à sa facilité à être analysé.

C'est un métalangage basé sur le concept de balise : toute information est délimitée par une balise ouvrante et une balise fermante. Les balises **XML** peuvent avoir des attributs et peuvent être hiérarchiques, c'est-à-dire qu'il peut y avoir d'autres balises entre deux balises. Par contre, il n'est pas possible de croiser deux balises (voir la figure 4.4). La figure 4.3 donne un exemple de description **XML** : la première ligne indique la version de **XML** utilisée et le type de caractères utilisés dans le fichier. La deuxième ligne indique le type de document (définitions de balises particulières). Ensuite la description commence, la première balise (**<middle...>...</middle>**) délimite la description **Middle**.

Pour obtenir un langage dérivé de **XML**, il faut définir des balises particulières, et préciser les contraintes qui leur sont associées. Ces contraintes peuvent être à propos de leurs attributs, ou à propos de la hiérarchie. Ces informations sont à donner dans un fichier à part nommé **dtd**. Un fichier **XML** valide ne peut utiliser que les balise décrite dans sa **dtd** et doit en respecter les contrainte..

Ce langage possède deux caractéristiques intéressantes pour le but fixé : c'est un métalangage, qui grâce au système des **dtd** n'a besoin que d'un seul analyseur quel que soit le langage dérivé, et c'est un standard recommandé pour la représentation de données.

XML est cependant plus une notation qu'un langage : en particulier il ne dispose pas de sémantique. **XML** ne peut donc pas être utilisé tel quel pour décrire **Lidel** ou **Colif**.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE middle SYSTEM "middle.dtd">
<middle language="Colif" version="1.0">
    <typedef name="PARAMETER" structure="set">
        <typefield name="name">
            <typeref base="str"/>
        </typefield>
        <typefield name="value">
            <typeref base="str"/>
            <value access="value" data="!"/>
        </typefield>
</middle>
```

FIG. 4.3 – *Exemple de description xml*

```
<balise1>
    <balise2>
</balise1>
    </balise2>
```

FIG. 4.4 – *Exemple de balises croisées : invalide en xml*

4.1.2.3 Le langage Middle

Le langage **Middle** est un langage dérivé de la notation **XML**, permettant de définir des structures de données complexes avec une sémantique associée.

Middle est lui-même un métalangage à partir duquel sont dérivés **Lidel** et **Colif**. La figure 4.2 représente la hiérarchie de ces langages.

Structure d'une description Middle

Middle est constitué de deux parties :

- une partie de description des types de données et leurs relations autorisées. Cette partie sert à définir le langage dérivé.
- une partie de définition des objets. La syntaxe et la sémantique de ces objets doit correspondre aux types définis dans la première partie. Cette partie peut être vue comme étant écrite dans le langage dérivé défini précédemment.

La définition d'un type consiste en l'association d'un nom (représentant le nouveau type défini) et de la description d'un nouveau type. La description du nouveau type est réalisée à l'aide de constructeurs de type. Pour chaque type de base il existe un constructeur de type. Les constructeurs de types peuvent se composer entre eux, ce qui permet de définir des types complexes : par exemple lorsque l'on utilise le constructeur d'arrangement, il faut définir un autre type pour chacun de ses champs.

La définition d'une donnée consiste en l'association d'un nom (représentant la nouvelle donnée définie), d'un type, et du contenu de la donnée. Pour que la définition soit valide, il faut que le contenu corresponde bien au type, et que toutes données incluses dans ce contenu soient elles aussi valides. Cette contrainte sur la validité de données complexes permet de vérifier la cohérence du contenu des descriptions, et non pas seulement leur forme comme c'est

le cas pour **XML**.

Les types de Middle

Les données **Middle** peuvent être des types suivants :

- type de base : nombre, chaîne de caractères, ou vide
- liste : suite ordonnée de données. Dans une liste, les données doivent avoir le même type. Une liste peut ne contenir aucune donnée.
- arrangement : structure composée de champs, chacun contenant une donnée. Chaque champ possède un type, de telle sorte qu'il n'est pas possible d'y mettre n'importe quelle donnée. Un champ peut être initialisé ; dès lors il n'est pas nécessaire à la déclaration de la donnée de lui attribuer une valeur. Par contre si un champ n'est pas initialisé, il faut lui attribuer une valeur.
- union : structure composée de champs comme l'arrangement, mais dont un seul doit être utilisé à la déclaration de la donnée
- référence : lien vers une donnée existante. Les références sont elles aussi typées. Une référence doit obligatoirement désigner une donnée existante[2].
- publique : définition d'une donnée interne à une autre donnée, visible depuis l'extérieur. Elle possède toutes les caractéristiques d'une donnée globale : elle possède un nom, un type, une valeur et elle peut être référencée par une autre donnée.

L'analyseur Middle

L'analyseur fonctionne en deux passes : dans une première passe, il effectue l'analyse lexicale et syntaxique du fichier **XML**, détecte les erreurs **XML** et les violations de la **dtd**. Durant cette analyse, il construit l'arbre syntaxique **Middle** correspondant. Ensuite, il analyse la sémantique cet arbre pour détecter les erreurs **Middle**, et pour réduire l'arbre sous une forme canonique.

4.1.3 La bibliothèque de système d'exploitation

La bibliothèque contient toutes les parties de code génériques qui peuvent être spécialisées et assemblées pour obtenir un système d'exploitation complet, ainsi que toutes les informations qui définissent ces portions de code et leur environnement d'utilisation. Dans cette section, nous allons nous intéresser à ces informations (décrites dans le langage **Lidel**).

4.1.3.1 Concepts de base pour la bibliothèque

Lidel est basé sur trois concepts qui donnent chacun une vue différente du système d'exploitation : l'**élément**, le **service** et l'**implémentation**. Un **élément** représente une partie de système d'exploitation de sorte que la structure d'un système d'exploitation complet peut être modélisée par un ensemble d'**éléments** liés. Un **service** représente quant à lui une fonctionnalité du système d'exploitation, de telle sorte que le comportement fonctionnel d'un système d'exploitation complet peut être modélisé par un ensemble de **services**. Une **implémentation**, enfin, représente une réalisation particulière d'une partie du comportement précis du système d'exploitation, de telle sorte que le code d'un système d'exploitation complet peut être vu comme un assemblage d'implémentations paramétrées. Une implémentation possède

2. Il existe la donnée vide, qui peut elle aussi être référencée.

un domaine de validité : elle peut être compatible avec certaines architectures matérielles (notamment le processeur), et incompatible avec d'autres. C'est aux **implémentations** que les portions de code génériques de systèmes d'exploitation sont associées.

Ces trois concepts ne sont pas indépendants : il y a des relations entre les **éléments** et les **services**, les **éléments** et les **implémentations**, et les **services** et les **implémentations**.

Relations entre les éléments et les services

Chaque **élément** peut fournir des **services**, et requérir d'autres **services**. Cela induit des relations de dépendance indirectes entre les **éléments** et les **services** : un **élément** ne requiert jamais un autre **élément** en particulier, mais il peut requérir un **élément** fournissant un **service** en particulier. Tout **élément** fournissant ce **service** peut convenir.

Ces relations de dépendance peuvent être présentées sous la forme d'un graphe orienté. Chaque nœud représente soit un **élément** soit un **service**. Chaque arc orienté relie soit un nœud **élément** à un nœud **service** pour modéliser que l'**élément** requiert le **service**, soit un nœud **service** à un nœud **élément** pour modéliser que l'**élément** fournit le **service**. Un tel graphe est représenté figure 4.5.

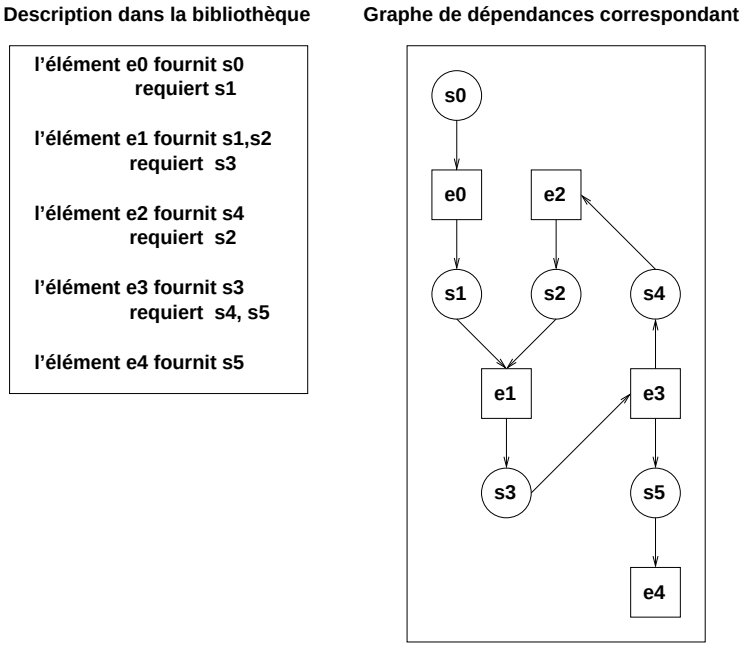

FIG. 4.5 – *Relations de dépendance entre éléments et services*

L'analyse et le parcours de ce graphe seront décrits dans la section 4.2.6.

Relations entre les éléments et les implémentations

Chaque **élément** possède une ou plusieurs **implémentations**.

Les **implémentations** d'un **élément** sont organisées suivant un arbre hiérarchique, les fils d'une **implémentation** étant toujours compatibles avec moins d'architectures que leur père. Pour décrire complètement un **élément** compatible avec une architecture donnée, il faut prendre les sources associées à chaque **implémentation** traversée et parcourir l'arbre en profondeur avec comme critère de choix la compatibilité avec les **processeurs** et **médias**. S'il est impossible d'atteindre une feuille, alors l'**implémentation** est invalide pour l'architecture visée. Si toutes les **implémentations** d'un **élément** sont invalides, alors l'**élément** est lui

aussi invalide. Un exemple est donné figure 4.6 : les **implémentations** en gras sont celles compatibles avec le processeur ARM7.

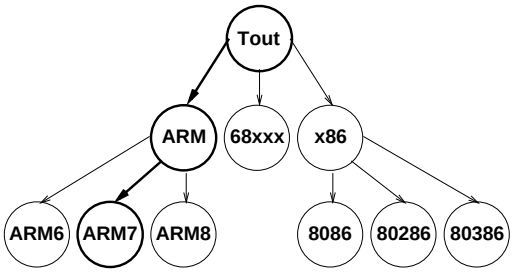

FIG. 4.6 – *Arbre d'implémentation d'un élément compatible avec le processeur ARM7*

Relations entre les services et les implémentations

Il est possible de définir des parties d'**implémentation** incluses uniquement si un **service** fournit par l'**élément** correspondant est requis par un autre **élément**. Cela permet, lorsqu'un **élément** fourni un **service** non requis, de ne pas mettre le code correspondant dans le système d'exploitation généré. La figure 4.7 montre un exemple où un **élément e1** fournit deux **services s0 et s1** : **s0** est requis par un autre **élément e0** (supposé nécessaire), et **s1** n'est pas requis.

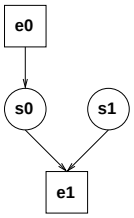

FIG. 4.7 – *Exemple d'élément fournissant un service requis et un service non requis*

4.1.3.2 Détails sur les objets de la bibliothèque

Nous allons maintenant donner les détails des divers objets de la bibliothèque :

L'élément

Un **élément** représente une partie du système d'exploitation. Un élément contient les informations suivantes :

- un nom unique permettant de l'identifier
- une liste de **services** fournis par l'élément
- une liste de **services** requis par l'élément
- un arbre d'**implémentations** de l'élément

Le service

Un **service** représente une fonctionnalité du système d'exploitation. Il s'agit d'une chaîne de caractères. Une sémantique et des règles de construction ont été définies dans le cadre de la thèse. Elles sont présentées section 4.1.4.1.

L'implémentation

Une **implémentation** représente une réalisation possible d'un **élément**. Une **implémentation** contient les informations suivantes :

- une liste de **sources** composant l'**implémentation**
- une liste de **processeurs** avec lesquels l'**implémentation** est compatible
- une liste de **médias** qui indiquent des choix d'**implémentation**
- une liste d'**implémentations** filles, qui donnent une structure d'arbre aux **implémentations** d'un élément (voir la section 4.1.3.1 pour plus d'information à ce sujet)

Le code source

Un **code source** contient les informations suivantes :

- un **langage**, indiquant dans quel langage de programmation le code généré correspondant sera écrit
- une liste de fichiers de macro-code source (liste de chaînes de caractères)
- une liste de fichiers de macro-entête (liste de chaînes de caractères)
- une liste de fichiers de macro à utiliser pour générer les **éléments** qui ont besoin d'un **service** fourni par l'**élément** correspondant à la **source**
- une liste de **fonctions**, décrivant les fonctions d'interface fournies par la source : ces fonctions peuvent être des appels système ou des fonctions d'interruption (*ISR*)

Le média

Un *média* symbolise un choix d'**implémentation** pour l'architecture matérielle ou logicielle. Par exemple, une communication entièrement logicielle utilisera le **média Soft**, tandis qu'une communication par registre matériel utilisera le média **Register**. Il permet de choisir entre plusieurs **élément** ou **implémentations** proposant les mêmes services. Si nous reprenons l'exemple, une communication de base peut requérir les services **Get** pour lire et **Put** pour écrire. Tous les éléments réalisant ces communications fournirons ces services, le choix de celui qui sera utilisé sera déterminé par des choix sur l'architecture : dans notre cas, avec quel périphérique (et donc quel pilote) la communication est-elle réalisée.

Dans la bibliothèque, les **médias** sont des chaînes de caractères. Leur contenu et leur sémantique sont laissés au choix du programmeur de la bibliothèque, cependant une sémantique et des règles de construction ont été définies dans le cadre de la thèse, et sont présentées section 4.1.4.2.

La fonction

Une **fonction** décrit une fonction d'une source pouvant être appelée par le système d'exploitation généré ou par l'application. Sa description contient les informations suivantes :

- le nom de la **fonction**, simple chaîne de caractères
- le type de la **fonction** : ce type peut être fonction appelée par un appel de fonction classique, fonction système (appelée par une trappe système) et fonction d'interruption (appelée suite à une interruption)
- une liste de **services** fournis par la fonction

Le processeur

Un **processeur** représente un processeur ou une famille de processeurs sur lesquels pourra s'exécuter le système d'exploitation généré. Il contient les informations suivantes :

- le nom du processeur, simple chaîne de caractères
- une liste de **compilateurs**

- une liste d'**éditeurs de liens**
- une liste de **convertisseurs**
- une liste de **processeurs** fils utilisée dans le cas des familles hiérarchiques de processeurs (voir la section 4.1.3.1)

Le langage

Un **langage** représente un langage ou une famille de langages de programmation avec lesquels peut être écrite une **source**. Il contient les informations suivantes :

- le nom du langage, simple chaîne de caractères
- une liste de **compilateurs** compilant le langage
- Une liste de **langages** fils utilisée dans le cas des familles hiérarchiques de langages

Le compilateur

Un **compilateur** représente un compilateur pouvant compiler des **sources** dans un **langage** pour un ou plusieurs **processeurs**. Sa description contient les informations suivantes :

- le nom du **compilateur**, simple chaîne de caractères
- une liste de **langages** supportés
- une liste de **processeurs** supportés
- Diverses informations sur les options à utiliser pour effectuer la compilation

L'éditeur de liens

Un **éditeur de liens** représente un éditeur de liens pouvant lier des programmes compilés pour un ou plusieurs **processeurs**. Sa description contient les informations suivantes :

- le nom de l'**éditeur de liens**, simple chaîne de caractères
- une liste de **processeurs** supportés
- Diverses informations sur les options à utiliser pour effectuer l'édition de liens

Le convertisseur

Un **convertisseur** représente un programme pouvant convertir un programme d'un format binaire en un autre. Ce type d'utilitaire est intéressant car il arrive souvent que le format binaire obtenu après compilation et édition de liens ne soit pas le même que celui supporté par un simulateur de processeur, ou un chargeur de ROM. Sa description contient les informations suivantes :

- le nom du **convertisseur**, simple chaîne de caractères
- une liste de **processeurs** supportés
- des informations sur les formats supportés en entrée et en sortie
- diverses informations sur les options à utiliser pour effectuer l'édition de liens

4.1.3.3 Utilisation des objets de la bibliothèque

Les **éléments**, les **services**, les **implémentations**, les **médias** et les **processeurs** sont utilisés pour la sélection des parties qui seront assemblées pour former le système d'exploitation final. La section 4.2.6 du chapitre sur l'outil de génération de système d'exploitation précisera comment cette sélection est effectuée.

Les **services**, les **implémentations**, les **processeurs** les **sources**, les **fonctions** et les **langages** sont utilisés pour l'adaptation à l'architecture et l'assemblage des parties génériques du système d'exploitation. La section 4.2.7 du chapitre sur l'outil de génération de système d'exploitation précisera comment cette adaptation et cet assemblage sont effectués.

Les **processeurs**, les **compilateurs**, les **éditeurs de liens** et les **convertisseurs** sont utilisés pour la production du code binaire final du système d'exploitation et de l'application logicielle prêt à être chargé sur les processeurs de l'architecture cible.

4.1.4 Choix d'organisation et de sémantique pour la bibliothèque

La structure et les définitions de la bibliothèque de système d'exploitation laissent une liberté totale au programmeur de la bibliothèque sur la sémantique liée à certains objets (les **services** et les **médias**), et sur le découpage en **éléments** et **implémentations** du système d'exploitation. Nous allons décrire dans cette section les choix que nous avons effectués pour utiliser la bibliothèque.

4.1.4.1 Hiérarchie des services

Les **services** sont des chaînes de caractères représentant des fonctionnalités du système d'exploitation. Nous avons décidé d'organiser les **services** sous la forme d'arbres, les nœuds étant des familles de **services**, et les feuilles étant des **services** finaux. À chaque service peut correspondre des instructions ou des variables, ou toute autre composition des deux dispersées dans le code d'une **implémentation**. Pour obtenir une flexibilité maximale de la bibliothèque, et pour réduire la taille des **éléments** qui la composent, nous avons décidé de définir un **service** pour chaque fonctionnalité élémentaire. Par exemple, une fonctionnalité de communication sera au minimum définie par deux **services** : un **service** représentant la fonctionnalité de communication du point de vue de l'application logicielle (c'est-à-dire à haut niveau), et un **service** la représentant du point de vue du pilote de communication (c'est-à-dire à bas niveau).

Un **service** est représenté comme la suite de ses pères séparés par des caractères «/», comme le seraient des fichiers et des répertoires sous UNIX.

Les différentes familles de **services** choisies sont les suivantes :

- **API** : regroupe tous les **services** représentant l'API du système d'exploitation utilisable par l'application logicielle. Cette famille contient trois sous familles :
  - **IO** : regroupe tous les **services** liés aux communications utilisables par l'application logicielle (comme le tube (ou *pipe*) de **POSIX** [46])
  - **Synchronization** : regroupe tous les **services** liés aux synchronisations utilisables par l'application logicielle (comme les sémaphores)
  - **Other** : regroupe les **services** de haut niveau n'entrant pas dans les familles précédentes
- **Kernel** : regroupe tous les **services** concernant le noyau du système d'exploitation. Elle contient les sous-familles suivantes :
  - **Kernel/Boot** : regroupe tous les **services** liés au démarrage du système d'exploitation. Elle contient le **service Kernel/Boot** (du même nom que la famille).
  - **Kernel/Cxt** : regroupe tous les **services** liés à la gestion des contextes associés aux tâches. Les **éléments** fournissant ces **services** sont toujours spécifiques au processeur cible.
  - **Kernel/Schedule** : regroupe tous les **services** liés à l'ordonnancement des tâches (par exemple le tourniquet, ou les priorités mais aussi la mise en sommeil ou le réveil d'une tâche).

- **Kernel/Task** : regroupe tous les **services** liés à la gestion de tâches. En pratique, elle fait le lien entre les autres sous-familles de la famille Kernel, et contient les tables de tâches.

- **Data** : regroupe tous les **services** liés aux structures de données (comme des FIFO)
- **Interrupt** : regroupe tous les **services** liés aux interruptions (par exemple la gestion des fonctions d'interruption, ou des appels système)
- **Synchronization** : regroupe tous les **services**, liés aux mécanismes de synchronisation internes au système d'exploitation
- **Driver** : regroupe tous les services représentant des gestionnaires de périphériques. Elle peut posséder de nombreuses sous-familles, dont **Driver/IO** pour les services liés aux communications de bas niveau.

Ces services peuvent aussi être vus sous la forme d'une pile protocolaire, allant de l'application logicielle jusqu'au matériel. Chaque couche ne peut interagir qu'avec ses voisines. La figure 4.8 représente ces piles de services.

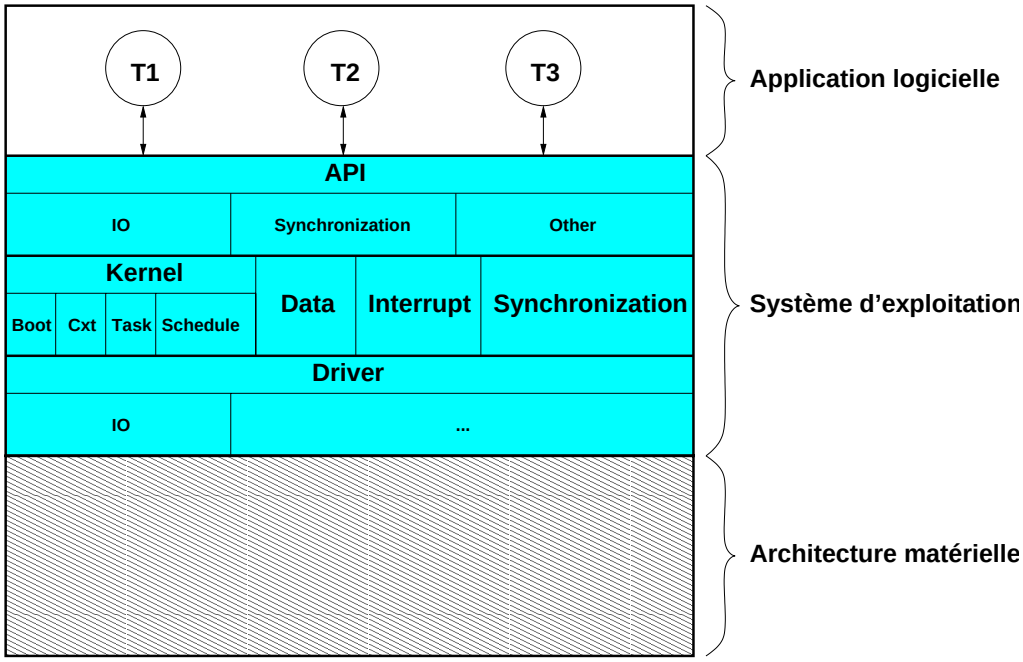

FIG. 4.8 – *Pile de services allant de l'application logicielle au matériel*

4.1.4.2 Hiérarchie des médias

Les **médias** sont des chaînes de caractères représentant les composants de l'architecture utilisés par le système d'exploitation. Ils sont organisé sous la forme d'arbres hiérarchiques comme pour le cas des **services**. De plus, ces chaînes de caractères sont aussi des références vers le description **Colif** (voir la section 4.1.5.1).

Un **média** est représenté comme la suite de ses pères séparés par des caractères «/», comme le seraient des fichiers et des répertoires sous UNIX.

4.1.4.3 Choix pour la représentation du code des éléments

Le but est de décrire le comportement des divers éléments qui composeront le système d'exploitation. Une bonne partie du code d'un système d'exploitation peut être écrite avec un langage de programmation de haut niveau quelconque (cependant le résultat doit être efficace). Le langage le plus utilisé dans le domaine est le C, pour son efficacité, et pour sa bonne gestion des pointeurs (beaucoup utilisés, notamment dans les communications avec l'extérieur du processeur).

Cependant, certaines parties du système d'exploitation, comme le changement de contexte entre deux tâches, sont trop spécifiques au processeur pour pouvoir être décrites dans un langage de haut niveau ; le langage d'assemblage du processeur doit alors être utilisé.

La génération du code du système d'exploitation consiste en l'assemblage et en le paramétrage de parties de codes. Cela nécessite de pouvoir modifier ce code, ce qui est difficile. C'est pour cela que ces parties de code sont encaspulées dans des macros. Dès lors, la génération revient à appeler le programme d'expansion de macros avec les bons paramètres.

4.1.4.4 Exemple de bibliothèque de système d'exploitation

Dans cette section, nous présentons en exemple une version simplifiée de la bibliothèque de système d'exploitation utilisée pour générer un système d'exploitation pour une application industrielle qui sera présentée dans la section 5.1. La bibliothèque complète sera présenté dans le chapitre 5.

Services et éléments de la bibliothèque

L'ensemble des éléments et services de cette bibliothèque est présenté dans la figure 4.9.

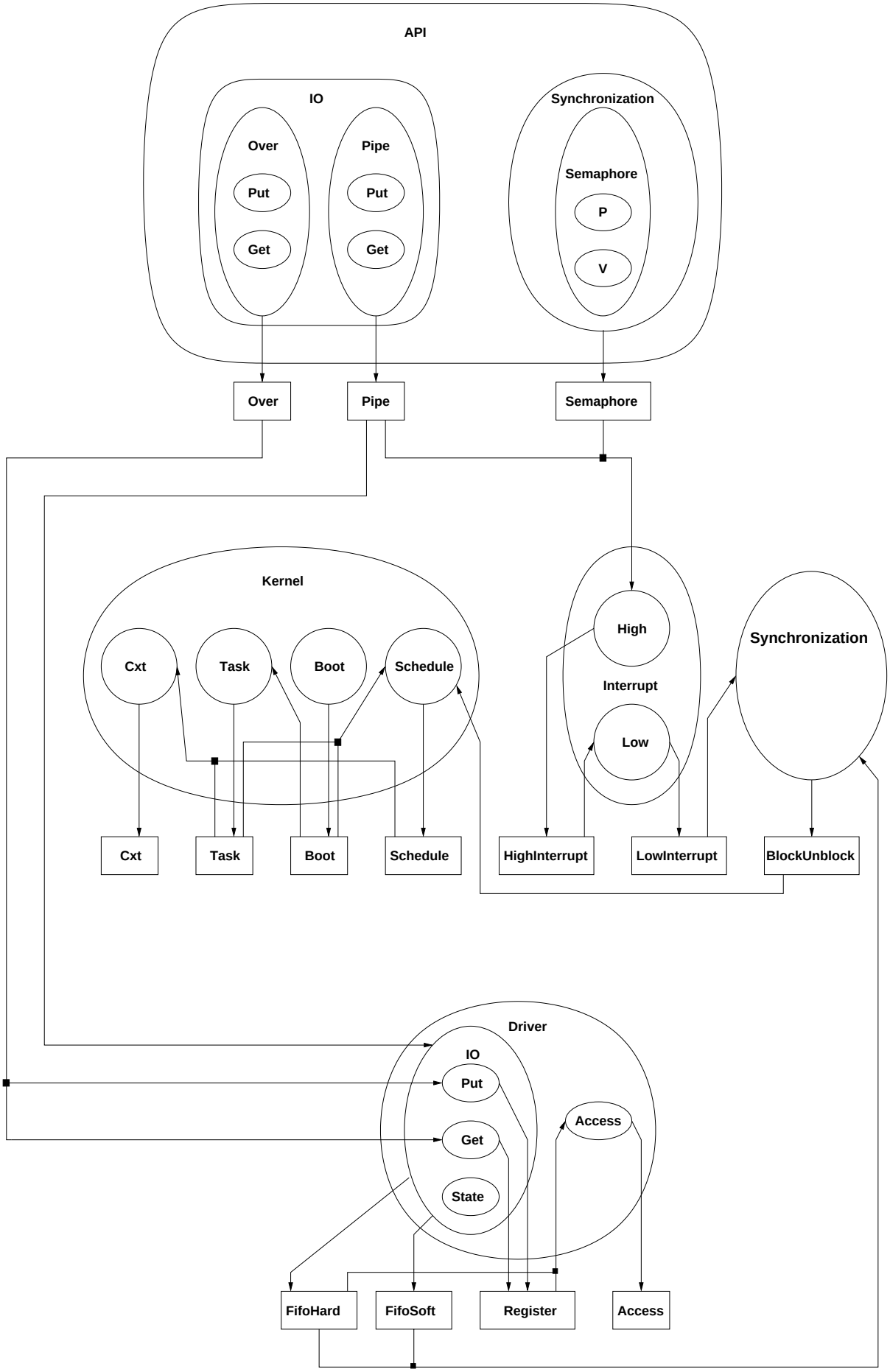

FIG. 4.9 – *Exemple d'une bibliothèque de système d'exploitation*

Dans cette figure, les **éléments** sont symbolisés par des rectangles, et les **services** par des formes arrondies. La hiérarchie des **services** est représentée par des formes arrondies imbriquées : ainsi, au niveau des **services** d'**API**, il faut par exemple lire **API/Pipe/Put**, pour le **service** noté **Put** dans le **service** noté **Pipe**. Chaque flèche allant d'un **service** vers un **élément**, indique que le **service** est fourni par l'**élément**. Chaque flèche allant d'un **élément** vers un **service** indique que l'élément requiert le service.

Pour alléger les notations, nous avons pris les conventions suivantes :

- Lorsque plusieurs flèches ont la même origine, ou arrivent à la même destination, elles peuvent être connectées entre elles (la connexion est symbolisée par un petit carré noir)
- Lorsqu'une flèche part d'un ensemble de services, ou le rejoint, elle est équivalente à plusieurs flèches rejoignant chaque sous-sous-service. Par exemple la flèche allant du **service Pipe** à l'**élément Pipe** veut dire que l'**élément** fournit les **services API/Pipe/Put** et **API/Pipe/Get**.

Les détails sur le contenu de cette bibliothèque seront précisés dans la section 5.2.2.

Caractéristiques des éléments de la bibliothèque

Les **éléments** fournissant les **services API** sont les seuls accessibles par l'application logicielle. Ce sont des éléments de haut niveau, possédant donc une seule **implémentation** indépendante du matériel, écrite en C, c'est-à-dire compatible avec tout type d'architecture cible. Ces **éléments** fournissent des **fonctions** de type système : ces appels système sont les seules fonctions du système d'exploitation utilisables par l'application logicielle.

Les **éléments** fournissant les services **Kernel** décrivent le noyau du système d'exploitation. **Task** et **Schedule** sont indépendants du **processeur**, et disposent donc d'une seule **implémentation** écrite en C. **Cxt** (qui représente la gestion du contexte des tâches) est entièrement dépendant du processeur. Il dispose de plusieurs **implémentations**, compatibles avec divers **processeurs**, souvent écrites dans le langage d'assemblage correspondant. **Boot**, qui représente l'initialisation du système d'exploitation, présente des parties dépendantes et des parties indépendantes du processeur. Il dispose donc d'une hiérarchie d'**implémentations**, la racine étant indépendant du **processeur**, et écrite en langage C. Les fils sont compatibles avec divers **processeurs**, et souvent écrits en langage d'assemblage.

Les **éléments** fournissant les services **Interrupt** décrivent le système de gestion des interruptions du système d'exploitation. Nous adoptons un système d'interruptions virtuelles qui encapsule les possibilités du processeur. Ainsi l'**implémentation** de l'**élément HighInterrupt** est indépendante du **processeur**, mais celles de **LowInterrupt** ne l'est pas.

L'**élément** fournissant les **services Synchronization** possède une unique **implémentation** indépendante du processeur.

Les **éléments** fournissant les **services Driver** ont des **implémentations** dépendantes du **processeur** et utilisent un **média**. Plus précisément, **FifoSoft**, **FifoHard** et **Register** disposent chacun d'une **implémentation** indépendante du processeur (décrite en langage C), mais utilisant un **média** particulier. Et l'**élément Access**, qui encapsule les entrées/sorties du processeur est quant à lui dépendant de la famille du processeur.

Autres informations

Les **langages** décrits sont : **C**, **C++** et **Assembly**.

Les **processeurs** décrits dans la bibliothèque sont : **ARM7**, **68000** et **UNIX**. Le processeur **UNIX** et une simulation de processeurs grâce aux primitives d'UNIX, et sera présenté dans la section 5.3.1.5.

Pour chaque **processeur**, un éditeur de liens a été défini, et pour chaque couple **pro-**

cesseur/langage un compilateur a été décrit. Il est donc possible d'utiliser C, C++, ou un langage d'assemblage pour chaque **processeur**.

4.1.5 La spécification d'entrée du flot de ciblage

4.1.5.1 Colif

Colif[24] est un langage de description d'architecture logicielle/matérielle dérivé de **Middle**, faisant clairement la distinction entre la structure et le comportement. C'est un langage qui permet des descriptions hétérogènes, et notamment des descriptions qui combinent plusieurs niveaux d'abstraction.

Les concepts de base de Colif

Comme toute description structurelle, Colif propose les trois concepts de module (nommé **module**), de port (nommé **port**) et de canal (nommé **net**). Chacun de ces objets peut avoir des attributs, dont un nom, un type et un niveau d'abstraction.

Chacun de ces objets est décomposé en deux parties : une interface (nommée **entity**) et un contenu (nommé **content**). L'**entity** a un type prédéfini qui caractérise l'objet, et d'autres informations qui varient selon que c'est un **module**, un **port** ou un **net**. Le **content** contient soit une référence à un comportement (nommé **behavior**) externe à Colif, soit des instances, ce qui permet d'introduire la hiérarchie[3]. Le contenu peut aussi être caché s'il n'est pas connu, ou inintéressant pour la description.

La figure 4.10 donne un exemple graphique d'une description Colif. Les **modules** sont représentés par des rectangles s'ils sont hiérarchiques, des rectangles noirs si leur contenu est caché, et des cercles s'il s'agit de modules feuilles avec un comportement connu. Les **ports** sont représentés par des carrés, et les **net** par des réseaux de droites orientées.

Les objets Colif

Le **module** représente un composant matériel ou logiciel. Son **entity** donne son type et les **ports** par lesquels il communique. Son **content** peut être caché, contenir des références à un comportement, ou contenir un sous-système : des **modules**, des **ports** et des **nets**.

Le **port** représente un point de communication pour un **module**. Son **content** a deux parties : une partie vers l'intérieur du **module**, et une partie vers l'extérieur du **module**. Ces deux parties qui sont indépendantes, peuvent être cachées, contenir des références à un comportement, ou contenir d'autres **ports**. Ces **ports** hiérarchiques découpés en deux parties interne et externe, permettent tout type de description hétérogène [86].

Le **net** représente la connexion entre plusieurs **ports**. Le **content** peut être caché, contenir des références à un comportement, ou contenir d'autres **net**.

Le mécanisme d'instanciation de Colif

Le mécanisme d'instanciation de Colif est utilisé pour pouvoir réutiliser des objets Colif pour en construire d'autres. Notamment, la hiérarchie et la connexion des canaux n'est obtenue que par instanciation.

Il a été défini deux niveaux d'instanciation : l'instanciation de définition (nommée **decl**), qui permet de définir des objets hiérarchiques et réutilisable, et l'instanciation d'utilisation (nommée **instance**), qui permet d'instancier effectivement un objet Colif.

Ainsi tout **module** contient dans son interface des **portdecl**. Un **module** hiérarchique,

3. Les **module** mais aussi les **port** et les **net** peuvent être hiérarchiques.

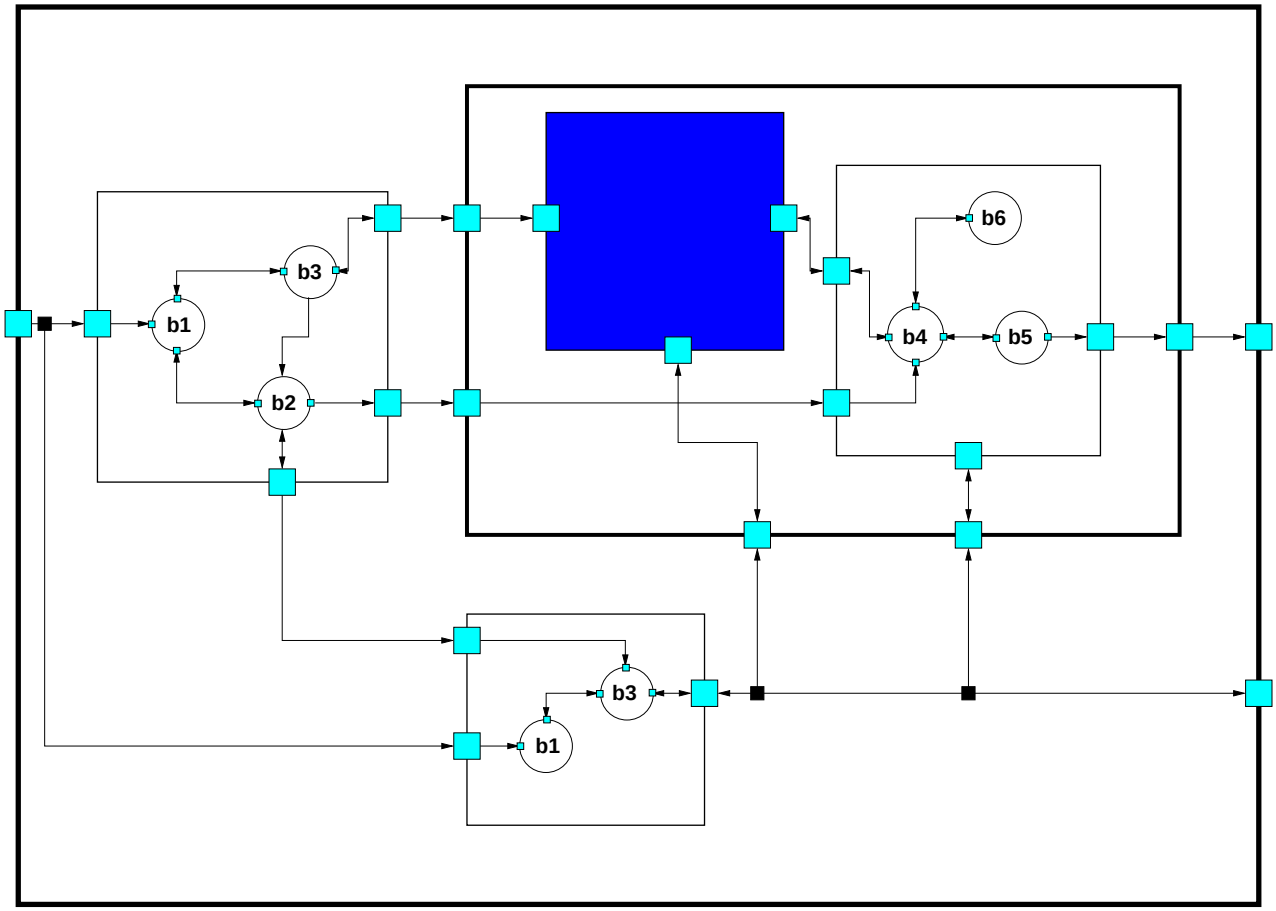

FIG. 4.10 – *Exemple d'une description Colif*

contient des **moduledecl**, et des **netdecl**. De même un **port** hiérarchique contient des **portdecl**. Enfin tout **netdecl** relie des **portdecl**.

Si nous instancions un **module**, nous obtenons une hiérarchie de **moduleinstance**, **portinstance** et **netinstance**.

4.1.5.2 Interprétation de Colif pour le ciblage logiciel

Dans cette section, nous allons indiquer quelles sont les informations nécessaires, et comment elles peuvent être représentées pour permettre la génération d'un système d'exploitation à partir de la bibliothèque présentée dans la section précédente.

Les parties de la spécification d'entrée qui nous intéresse sont celles qui concerne le logiciel. Ainsi, nous nous concentrerons sur deux type de modules Colif particuliers : les **modules processeurs**, et les **modules tâches**. Les **modules processeurs** sont des modules hiérarchiques qui possèdent des attributs décrivant les processeurs qu'ils modélisent et qui contiennent des **modules tâches**. Ces derniers, sont des **modules** feuilles (non hiérarchiques) qui possèdent des attributs décrivant les tâches qu'ils modélisent, et dont le contenu est un lien vers leur comportement.

Informations requises

Le premier type d'informations requises pour la génération de système d'exploitation est l'ensemble des fonctionnalités initiales requises pour ce dernier. Il doit être possible de déduire

de ces dernières les **services** de bases requis (c'est-à-dire d'**API** et les caractéristiques de l'ordonnanceur, voir la section 4.1.4.1). Il faut aussi connaître les composants matériels de l'architecture : quels sont les processeurs utilisés, quels sont les périphériques accessibles par le système d'exploitation. Enfin il faut connaître la structure logicielle et ses paramètres (adresses, types des données, tailles des tampons, etc.) pour savoir comment spécialiser et assembler les diverses parties du système d'exploitation.

Représentation à partir d'une description architecturale

La description architecturale doit permettre de déduire les services initiaux pour la génération du système d'exploitation ainsi que tous les paramètres qui permettront cette génération. Nous rappelons que nous utilisons Colif pour cette représentation, les informations suivantes sont donc disponibles en tant qu'attributs des objets :

- les informations sur les processeurs
- les informations sur les priorités et contraintes mémoire des tâches
- les informations d'allocations des divers éléments matériels

Il reste à définir les services initiaux, et l'assemblage des divers élément logiciels, c'est-à-dire quel élément sera combiné avec quel autre élément, et combien de fois[4]. Pour pouvoir interpréter ces informations à partir de Colif, des règles d'interprétation ont été définies pour un sous-ensemble de Colif. Ce sous-ensemble est défini par les contraintes suivantes :

1. Les ports des tâches ne possèdent pas de vue interne, et ne peuvent référencer que des services de communication de haut niveau (directement utilisables par les tâches).
2. Les processeurs possèdent un unique port hiérarchique. Le côté externe de ce port n'est pas pris en compte, par contre les sous-ports du côté interne doivent faire référence à des services pilotes de périphériques.
3. Les nets servent à définir les allocations logicielles.

Les règles d'interprétation seront définies dans la section 4.2.5 du chapitre consacré à l'outil de génération de système d'exploitation.

La figure 4.11 donne un exemple de description architecturale.

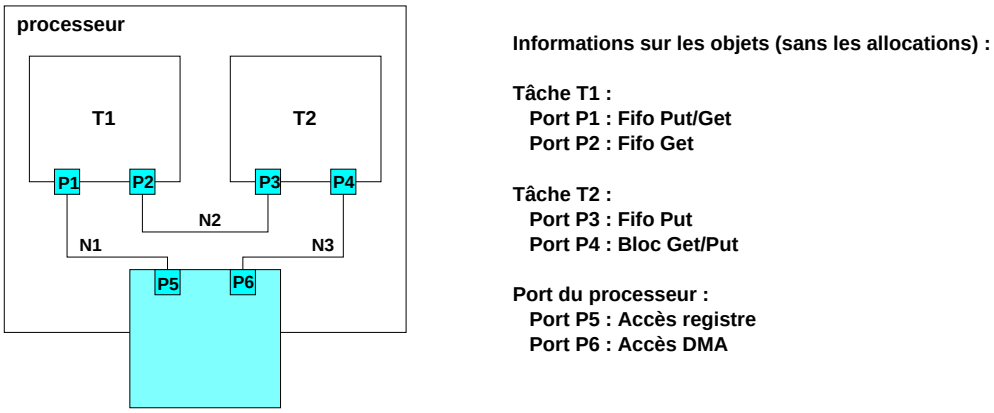

FIG. 4.11 – *Un premier exemple de description architecturale pour la génération de système d'exploitation*

4. Nous rappelons qu'il est préférable de découper au maximum les éléments du système d'exploitation, pour avoir avec une bibliothèque concise un maximum de possibilités).

4.1.6 Les macros qui encapsulent le code du système d'exploitation

4.1.6.1 Intérêt de la macro génération

Le code d'un système d'exploitation est en général écrit dans deux langages : un langage de haut niveau tel que le C pour décrire les parties générales, et un langage d'assemblage pour décrire les parties spécifiques au processeur. Certains compilateurs C permettent d'inclure du code en langage d'assemblage dans les sources C, cependant, ce code est modifié au moment de la compilation, ce qui peut le rendre instable lorsqu'il effectue des opérations critiques telles que le changement de mode de fonctionnement du processeur. Les parties de code du système d'exploitation doivent donc pouvoir être écrit dans n'importe quel langage. Elles doivent aussi être suffisamment génériques pour pouvoir être adaptées à l'architecture cible et être assemblées.

Pour que le flot de ciblage soit vraiment intéressant, il convient que la spécification d'entrée soit plus simple à écrire que la réalisation finale. Il faudrait notamment que les paramètres inscrits dans cette spécification soient canoniques[5], et dans l'esprit de la spécification plutôt que dans celui du système d'exploitation qui sera généré. Le but est de limiter au maximum les possibilités d'erreurs humaines, en réduisant la redondance et le nombre de concepts à manipuler dans la spécification. Pour répondre à cette contrainte, il convient donc de pouvoir calculer tous les paramètres de génération de système d'exploitation, qui contiennent beaucoup de redondance et qui sont basés sur des concepts différents.

Cette génération de paramètres pourrait être effectuée au niveau de l'outil de génération, mais il est préférable de choisir une solution plus flexible : faire effectuer ce calcul par l'expanseur de macros. Nous verrons dans la partie **CodeGenerator** les avantages de ce choix.

Enfin pour des raisons pratiques, il serait préférable que ce langage et le programme d'expansion de macros correspondant soient facilement intégrables dans l'outil de génération.

4.1.6.2 Le langage d'expansion de macro choisi

La section précédente impose que le langage de macros soit indépendant du langage à générer. Cette première caractéristiques est remplie par tous les langages de macros qui génèrent du texte sans en interpréter le contenu. Cela empêche par contre l'utilisation des gabarits ou *templates* de C++, ou de **Ada** [6].

Il faut aussi être sûr qu'il soit toujours possible de décrire un assemblage, ou un calcul de paramètres à partir des macros. Il est donc nécessaire que le langage de macros soit un langage complet. De nombreux langages de macros, tels que celui du préprocesseur C ou C++, ne sont pas complets, et sont donc à proscrire.

Le langage de macros **m4** [3] pourrait très bien convenir, mais nous lui avons préféré **Rive** pour sa facilité à être intégré dans d'autres outils, et pour sa possibilité de gérer aussi bien les itérations que les récursions.

4.1.6.3 Principales caractéristiques du langage de macro Rive

Nous allons ici donner un rapide aperçu des possibilités de Rive. Pour plus de détails vous pourrez consulter sa documentation fournie en annexe.

Le langage de macro Rive prend en entrée un fichier texte contenant à la fois des macros et du texte non traité, et éventuellement des paramètres d'expansion, et produit en sortie un fichier texte avec toutes les macros expansées.

5. C'est-à-dire minimum et suffisants.

Les macros sont vues comme des données, et peuvent avoir deux types : le type chaîne de caractères (interprété différemment suivant les situations), et le type tableau.

Il est possible de définir des variables, et de les modifier ultérieurement. Chaque variable peut être globale, ou locale. Elle peuvent contenir une donnée, ou une fonction avec des paramètres[6].

Il est possible faire des calculs arithmétiques et logiques, et des comparaisons avec des nombres entiers ou flottants. Les structures de contrôle possibles sont quant à elles : les boucles de type «pour», «tant que» et «jusqu'à», et les tests de type «si», et «selon».

Rive dispose de nombreuses fonctions prédéfinies, dont par exemple celles permettant l'interaction avec l'utilisateur : par exemple il est possible d'interrompre l'expansion pour demander un paramètre particulier.

4.2 La génération automatique de système d'exploitation

Nous avons vu dans le chapitre précédent les diverses méthodes actuellement employées pour cibler du logiciel dans les systèmes embarqués. Nous avons vu qu'elles présentaient l'inconvénient d'être soit trop longues (intervention humaine pour des opérations systématiques), soit pas assez générales (orienté flot de données), soit produisant un résultat peu efficace.

Dans ce chapitre nous allons présenter le flot mis au point pour la génération automatique de système d'exploitation. Dans une première section nous en donnerons une vue générale, ensuite nous détaillerons les entrées les sorties et chaque étape du flot. La section suivante précise les originalités du flot. Finalement nous préciserons les méthodes à employer pour ajouter de nouvelles fonctionnalités dans la bibliothèque du système d'exploitation.

4.2.1 Architecture globale du flot de ciblage avec génération de systèmes d'exploitation

Dans cette section nous présentons le flot de ciblage avec génération de système d'exploitation qui a été conçu dans la cadre de cette thèse.

4.2.1.1 Principe de la génération de systèmes d'exploitation

Le flot de ciblage avec génération de systèmes d'exploitation produit le code final des systèmes d'exploitation en adaptant et assemblant des parties de code contenues dans une bibliothèque de système d'exploitation (décrite dans le langage Lidel, voir la section 4.1.3).

L'ensemble de parties de codes qui sont sélectionnées pour générer le système d'exploitation final est une sorte de fermeture transitive de services initiaux qui sont déduits à partir de la spécification. Les services initiaux sont ceux correspondant aux fonctions d'API utilisées par les tâches de l'application logicielle.

4.2.1.2 Présentation générale du flot de ciblage avec génération de systèmes d'exploitation

Le flot est représenté dans la figure 4.12. Il prend en entrée une description annotée de l'architecture matérielle et logicielle, et une bibliothèque de composants pour le système d'ex-

6. Les fonctions peuvent être récursives

ploitation[7]. Il produit en sortie le code source du système d'exploitation généré, et celui de l'application adaptés au système d'exploitation et à l'architecture cible.

7. la bibliothèque à déjà été décrite au chapitre 4.1.3 de la partie 3

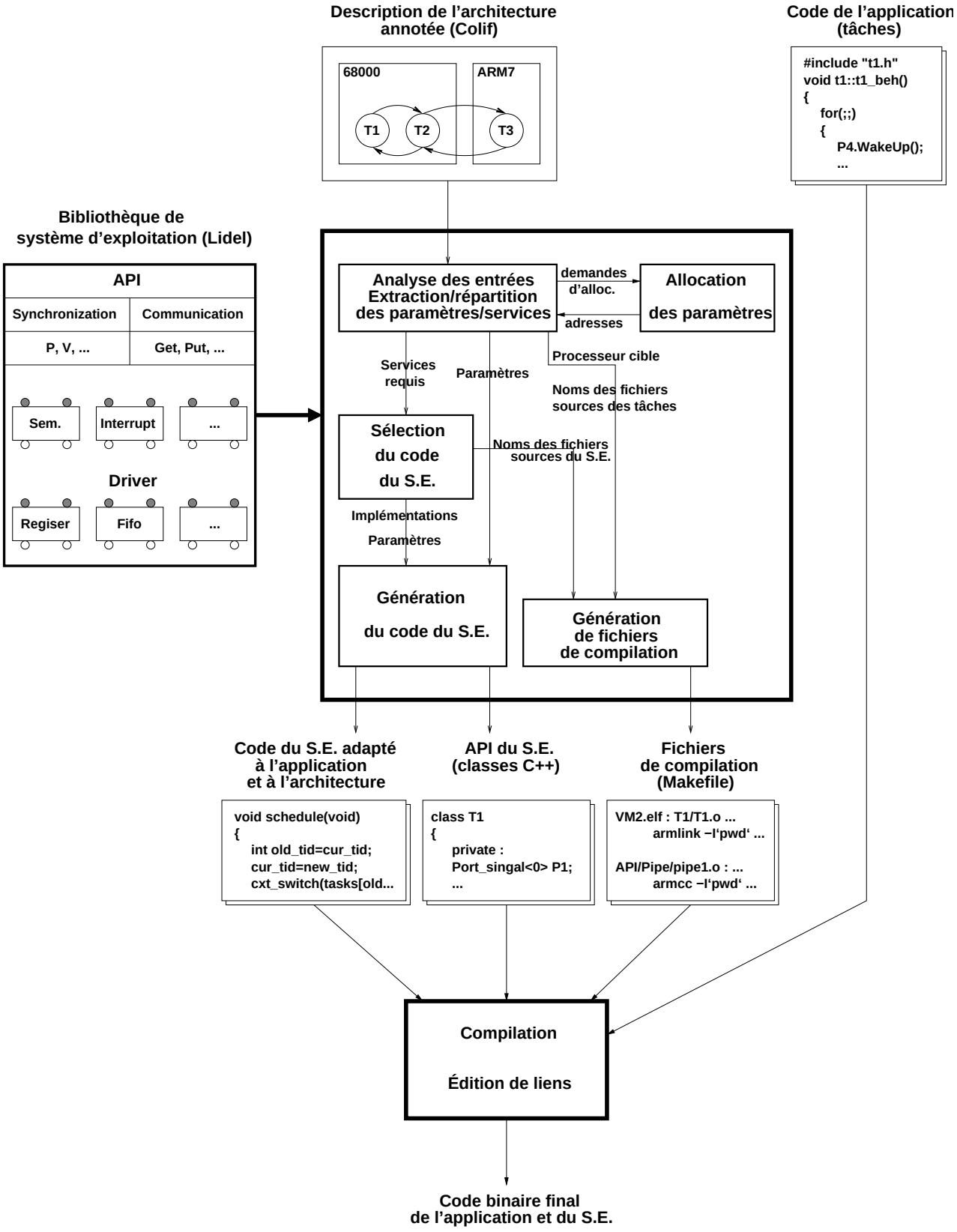

FIG. 4.12 – *Le flot de ciblage avec génération de système d'exploitation*

Les étapes du flot sont les suivantes :

1. analyse des entrées et extraction des paramètres : cette étape recherche les informations pour le ciblage contenues dans les entrées utilisateur (description de l'architecture et de l'application, allocations) et les redistribue aux autres étapes
2. allocation des paramètres locaux
3. sélection du code du système d'exploitation : cette étape recherche dans la bibliothèque tous les éléments qui constitueront le système d'exploitation final en suivant des règles de dépendances entre les éléments
4. génération du code du système d'exploitation : cette étape génère le code final du système d'exploitation adapté à l'architecture et à l'application, en effectuant des expansions de macros
5. génération des fichiers de compilations : il s'agit simplement de la génération de *makefiles* [1]

4.2.2 Description détaillée des entrées du flot

Dans cette section nous détaillons les diverses entrées du flot. Ces entrées sont de deux ordres : les entrées de l'utilisateur, et la bibliothèque du système d'exploitation.

4.2.2.1 Description de l'architecture matérielle et logicielle

Présentation générale de la description

Cette description est présentée sous la forme d'un fichier Colif. Elle représente l'architecture globale du système sous la forme de modules hiérarchiques interconnectés. Les tâches logicielles sont représentées sous la forme de modules fils des modules de type processeur. Des détails sur ce type de représentation sont donnés dans la section 4.1.5.1. Dans le flot général du groupe SLS, le modèle Colif provient d'un autre langage tel que systemC (voir la section 1.4.2.1).

Position et nature des informations intéressantes pour le ciblage

Les informations contenues dans cette description sont de plusieurs types :

- les informations topologiques : il s'agit du nombre de processeur, du nombre de tâches logicielles, quels processeurs exécutent quelles tâches, par quels port les tâches communiquent-elles, etc. Ces informations proviennent de la hiérarchie des modules de la description.
- les informations sur l'API : il s'agit de l'API du système utilisée par les tâches. Ces informations font parties des paramètres des ports des tâches.
- les caractéristiques des processeurs : les types de processeur (68000, ARM7, etc.), les ressources locales (voir la section 4.2.2.2), etc. Ces informations font partie des paramètres des modules processeurs.
- les médias utilisés pour les interactions : ces informations sont situées au niveau des ports des processeurs et des canaux de communication
- les paramètres d'allocation pour les tâches et les divers éléments du système d'exploitation. Dans ces paramètres se trouvent les adresses, les tailles, mais aussi les types de donnée ou les priorités. Ces paramètres peuvent être sous la forme de valeurs fixées, ou sous la forme de demandes d'allocation (voir la section 4.2.2.2). Ces informations sont données comme annotations de la description initiale suite à l'étape d'affectation globale de la mémoire (voir la section 1.4.2.5).

4.2.2.2 Les allocations de ressources locales au processeur

Les ressources locales au processeur sont ses mémoires locales et ses interruptions. Suivant le système d'exploitation et le processeur, ces ressources peuvent avoir des formes différentes : par exemple même si le processeur ne dispose que d'une interruption matérielle, le système d'exploitation peut fournir un multiplexeur pour obtenir plusieurs interruptions matérielles.

Si les allocations de ressources globales de l'architecture doivent être déjà définies lorsque le flot de ciblage débute, les allocations de ressources locales au processeur peuvent par contre être effectuées pendant la phase de génération de système d'exploitation.

La description d'entrée du flot ne doit alors contenir que la définition des ressources du processeur (mémoires et interruptions), et les demandes de ressources. Par exemple, pour une communication entre tâches, si le concepteur veut utiliser un tampon de taille bien définie, il devra demander de réserver la mémoire nécessaire pour le stocker.

Les ressources du processeur

Les ressources du processeur sont définies par des paramètres du module processeurs. Elles ont un nom, un type d'objet alloué (type de donnée, interruption), une origine et une taille maximale en nombre d'objets. La figure 4.13 donne deux exemples de définition de ressource pour un processeur : la première définit une zone mémoire de taille 65536 **long int**[8] dont la première adresse est 0, et la deuxième un ensemble de 256 d'interruptions dont la première est l'interruption 0.

```
Exemples de définition d'une zone mémoire (MEM) :
ALLOCATION = MEM=long int:0:65535

Exemples de définition d'un ensemble d'interruptions (IT) :
ALLOCATION = IT=interrupt:0:255
```

FIG. 4.13 – *Exemple de définition de ressource locales à un processeur dans une spécification Colif*

Demande de ressource

Dans un processeur, chaque objet (port, canal ou tâche) peut utiliser des ressources. Trois cas peuvent se présenter :

- Soit les ressources utilisées sont globales, et sont donc déjà été allouées : il suffit alors de préciser quels sont les références des ressources utilisées (adresse pour les mémoires, et numéro pour les interruptions).
- Soit les ressources sont locales au processeur, et ne sont pas allouées : il faut alors indiquer quelle ressource et quelle taille doivent être allouées. La ressource est à choisir parmi les ressources définies au niveau du processeur (voir la section précédente).
- Soit les ressources sont locales au processeur, et sont déjà allouées : il faut alors indiquer quelle ressource du processeur est concernée, la taille allouée et la référence (adresse ou interruption).

Ces trois cas sont représentés dans la figure 4.14 : pour le premier cas, la valeur du paramètre **ADRESSE_REGISTRE** est l'adresse du registre (c'est-à-dire la référence à la ressource globale). Pour le deuxième cas la valeur du paramètre **ADRESSE_TAMPON**

8. C'est-à-dire dans le cas d'un processeur 32 bits 262144 octets.

indique la demande de 256 ressources pour la mémoire **MEM1** du processeur et celle du paramètre **IT_TAMPON** indique la demande d'une ressource pour les interruptions **IT_SPACE** du processeur. Pour le dernier cas, la valeur du paramètre **ADRESSE_SHM** indique l'utilisation de 32 ressources pour la mémoire **MEM1** du processeur à partir de l'adresse 0x4500.

La section 4.2.9 précisera les algorithmes d'allocation utilisés.

```
Exemple de ressource globale déjà allouée :
ADRESSE_REGISTRE = 0xF001500

Exemple de ressource locale au processeur à allouer :
ADRESSE_TAMPON  = (256):MEM1
IT_TAMPON       = (1):ITSPACE

Exemple de ressource locale au processeur déjà allouée :
ADRESSE_SHM = 0x4500:(32):MEM1
```

FIG. 4.14 – *Exemple de paramètres d'allocation pour des ports de processeur*

4.2.2.3 Description du comportement de l'application logicielle

L'application logicielle se présente sous la forme de plusieurs tâches concurrentes. À chaque tâche doit correspondre un code de haut niveau. Les interactions avec l'extérieur sont encapsulées dans des accès aux ports de communication de chaque tâche qui sont décrits dans la spécification Colif. Cette encapsulation a pour conséquence que le comportement d'une tâche est écrit de la même manière, que la tâche communique ou non avec l'extérieur du processeur. Dès lors, le modèle de programmation du comportement des tâches est celui représenté par la figure 4.15.

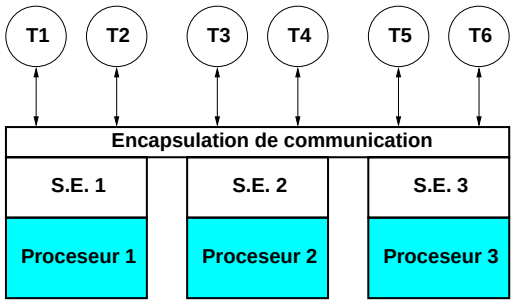

FIG. 4.15 – *Abstraction de l'architecture pour les tâches*

Cette description de l'application peut être faite dans n'importe quel langage permettant de s'abstraire de détails d'implémentation. Comme le flot global utilise actuellement SystemC (voir la section 1.4.2 de la première partie), elles sont donc naturellement décrites dans ce langage.

4.2.2.4 La bibliothèque de systèmes d'exploitation

La bibliothèque a déjà été présentée à la section 4.1.3.

4.2.3 Description des sorties du flot

Le flot produit en sortie le code du système d'exploitation adapté à l'architecture et à l'application. Ce code est constitué de plusieurs fichiers pouvant être de divers langages comme le C le C++ ou les langages d'assemblage.

Le flot produit aussi les entêtes permettant d'adapter le code de l'application au système d'exploitation généré et à l'architecture, sans que l'utilisateur n'ait rien à changer. Comme le flot global se base sur une description SystemC de l'application, les entêtes générées sont décrites en C++. Cependant, il est possible d'utiliser d'autres langage pour la description de l'application, sans pour autant devoir modifier l'outil : il sera seulement nécessaire d'ajouter des implémentations dans la bibliothèque.

Le flot produit les fichiers de compilation : ce sont les fichiers qui permettent l'automatisation de la compilation. Les fichiers produits sont dans le format *Makefile*, compréhensible par le programme d'automatisation de commandes make [1].

Enfin le flot produit en sortie des comptes rendus sur les opérations effectuées :

- rappel des actions effectuées durant le ciblage
- les erreurs survenues durant le ciblage
- des informations sur les caractéristiques du système d'exploitation généré (pour plus de détails sur ces informations, voir la section 4.2.5).

4.2.4 Enchaînement détaillé des étapes du flot

L'enchaînement et les données circulant dans le flot de génération sont présentés dans la figure 4.16.

La première étape à être exécutée est l'étape d'analyse d'architecture. Elle prend en entrée la spécification Colif de l'application, et en déduit les paramètres de génération, qui sont répartis en unités de générations. Une unité de génération représente la réalisation complète d'une fonction d'un système d'exploitation, par exemple la communication entre deux tâches d'un processeur, ou la communication entre une tâche et un périphérique. Le module d'allocation est quant à lui invoqué par cette étape pour chaque paramètre à allouer.

La seconde étape, qui est exécutée pour chaque unité de génération, est l'étape de sélection de code : elle détermine la fermeture transitive des services requis par l'unité de génération traitée. Cette fermeture transitive est l'ensemble des éléments qui réaliseront la fonction associée à l'unité de génération.

La troisième étape, qui est exécutée pour chaque élément distinct précédemment déterminé, est l'étape de génération de code : elle génère le code de chaque élément, en y associant les paramètres de toutes les unités de génération où il est apparu.

La dernière étape est la génération des fichiers de compilation. Elle utilise les propriétés des compilateurs et les noms des fichiers des systèmes générés et des tâches de l'application.

Remarque : comme on applique la sélection de code sur les unités de génération de manière séparée, il est possible de générer du code pour réaliser un même service utilisé dans des contextes différents.

4.2.5 Analyse de l'architecture

L'étape d'anlyse de l'architecture est la première étape du flot, son but est de fournir les informations requises par les autres étapes sous un format directement utilisable. L'entrée

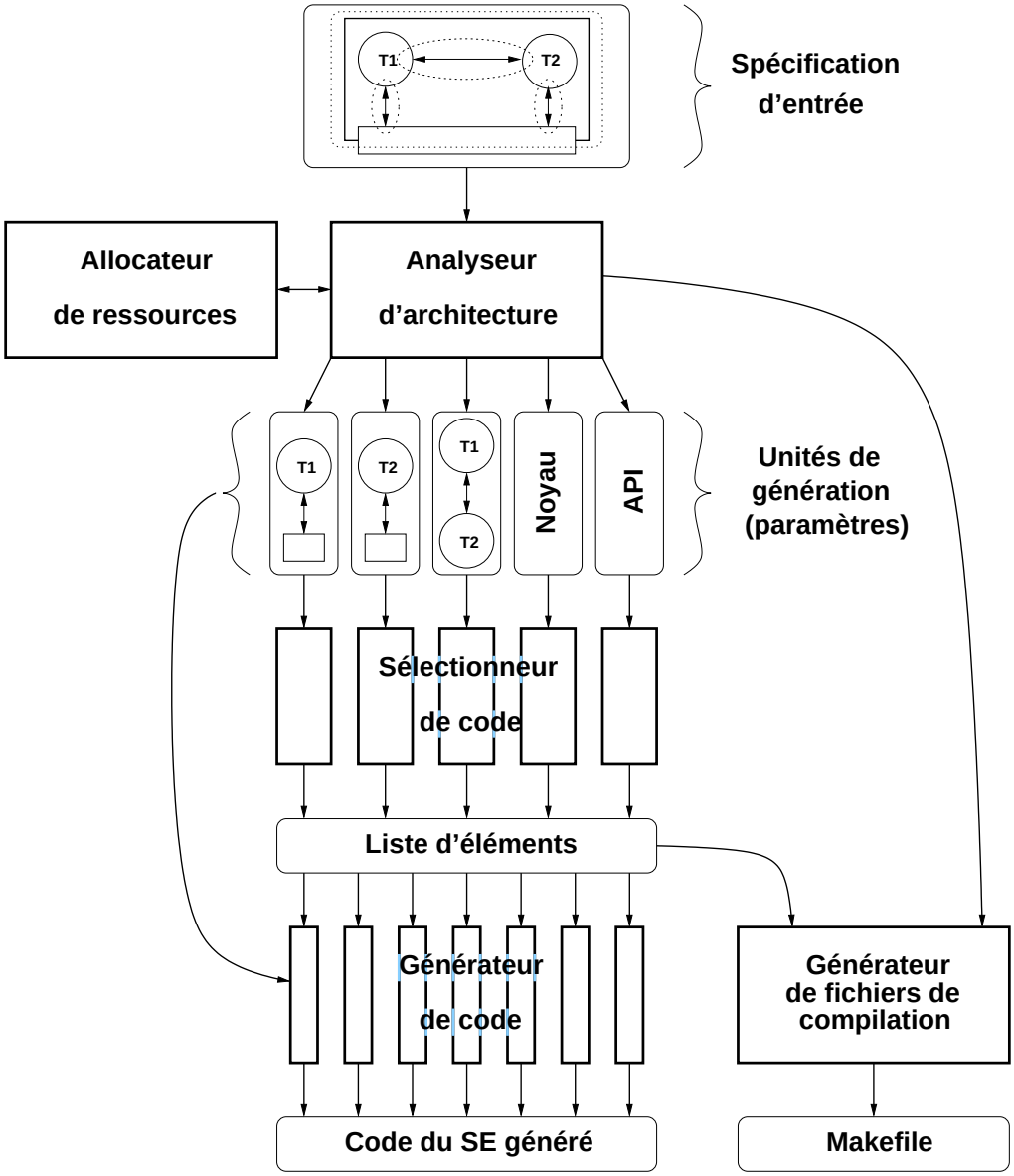

FIG. 4.16 – *L'enchaînement des étapes de génération*

concernée par cette analyse est la description de l'architecture matérielle et logicielle annotée. La bibliothèque et la description comportementale de l'application logicielle fournissant directement les informations utiles, elles n'ont pas besoin d'être traitées durant cette étape.

Remarques :

- Cette partie est la seule qui soit en contact avec la spécification d'entrée. Ce serait donc la seule à changer s'il y avait une remise en question des parties antérieures au flot de ciblage.
- La spécification d'entrée est à fournir par l'utilisateur, où par un éventuel outil de synthèse qui l'aurait généré. Notamment, tous les paramètres nécessaires à la générations et décrit dans la section 4.2.5.1 sont supposés être présents.

4.2.5.1 Interprétation de la spécification d'entrée

Nous avons précisé dans la section 3.3.2.1 les informations utiles pour la génération de systèmes d'exploitation. Ces informations doivent être interprétées pour obtenir le système d'exploitation correct.

Problèmes pour la spécification d'entrée

La spécification d'entrée donne des informations directes sur les tâches, et sur l'architecture matérielle globale. Il n'en est pas de même pour le système d'exploitation. En effet ce dernier est implicite à cette description : les paramètres du système d'exploitation font partie des paramètres d'implémentation dont le concepteur veut s'abstraire.

La spécification ne contient notamment pas d'information directe sur le nombre et la taille des variables et des fonctions que le système d'exploitation doit utiliser. Il convient donc de définir une sémantique de traduction des informations abstraites et architecturales de la spécification vers les informations définissant les systèmes d'exploitation à générer.

Sémantique associée aux objets de la spécification

Pour la génération de systèmes d'exploitation, nous ne nous intéressons qu'aux modules processeurs et à leur contenu.

Les ports des tâches sont vus comme les points d'accès utilisés par les tâches pour interagir avec leur environnement. Les paramètres associés à ces ports sont seulement les services d'API du système d'exploitation. Un port peut proposer plusieurs services, par exemple un accès à une mémoire partagée et des services de sémaphores pour garder cet accès.

Les ports internes du processeur sont vus comme les pilotes de périphériques du système d'exploitation. Les paramètres qui leurs sont associés décrivent le média, c'est-à-dire le périphérique qu'ils utilisent, et les informations d'allocation, c'est-à-dire les adresses, les interruptions et les dimensions du périphérique (taille des mémoires par exemple).

Les canaux de communication représentent les dépendances d'interaction. Ces canaux peuvent être laissés sans aucun paramètre, auquel cas ils correspondent à une jonction directe entre ports. Sinon ils peuvent avoir des paramètres précisant le média utilisé[9], et les paramètres précisant les ressources utilisées. Ces paramètres (adresses, taille et interruption) peuvent avoir trois formes :

- paramètres déjà allouées comme c'était le cas pour les ports du processeur
- paramètres en demande d'allocation : c'est alors à l'outil de définir leurs adresses ou leurs interruptions (voir la section 4.2.9)
- paramètres implicites : certaines fonctionnalités du système d'exploitation ne présentent pas de structure de données requérant des allocations du point de vue de l'utilisateur. C'est le cas par exemple des sémaphores : l'implémentation d'un sémaphore requiert une file d'attente dont les propriétés concernent peu l'utilisateur. Les paramètres de ce type sont calculés aux cours de la génération de code (voir la section 4.2.7).

Remarques :

- Lorsqu'un canal n'a pas de média, un média **Soft** lui est automatiquement associé. Ce type média est associé à toutes les implémentations décrivant une interaction logicielle.
- Les allocations des ressources utilisées au niveau des ports des processeurs sont normalement définies avant le flot de ciblage : il peut sembler étrange que ce soit le logiciel qui définisse les adresses de l'architecture matérielle.

9. C'est-à-dire l'implémentation logicielle utilisée pour matérialiser l'interaction

Les modules internes au module processeur doivent être des tâches (des modules feuilles). Elles représentent l'application logicielle décomposée et plusieurs tâches concurrentes. Les paramètres qui leurs sont associés précisent comment elles doivent être ordonnancées (type, priorité, etc.). Ils précisent aussi ou se trouve la description du comportement des tâches.

Règles d'interprétation pour les connexions

La figure 4.17 recense toutes les configurations acceptées pour la spécification. Le cas **(a)** représente une interaction point à point entre les deux tâches **T1** et **T2**. L'interprétation est immédiate : les tâches **T1** et **T2** interagissent respectivement par les ports **P1** et **P2** grâce aux services associés à ces ports. Cette interaction est réalisée et instanciée de manière unique par le média associé au canal **C1**, et caractérisée par ses informations d'allocation. Par exemple, s'il s'agit d'une communication par FIFO, alors, quel que soit le service utilisé par chacune des tâches, la structure de données réalisant la FIFO sera unique.

Le cas **(b)** représente une interaction multipoint entre les trois tâches **T1**, **T2** et **T3**. Comme dans le cas précédent il peut y avoir plusieurs services d'accès, mais l'implémentation de l'interaction est unique, matérialisée par le canal **C1**.

Le cas **(c)** représente une interaction entre une tâche et un pilote de périphérique (matérialisé par le port **P2**). Si le canal intermédiaire **C1** a des informations sur un média et sur des allocations, alors il est considéré comme un intermédiaire entre la tâche et le pilote. S'il n'a aucune information, alors la tâche est considérée comme accédant directement au pilote. Supposons par exemple, que le pilote encapsule un simple accès à un registre de donnée d'un périphérique. Si **C1** est vierge, alors **T1** écrira directement dans le registre (par l'intermédiaire de l'API et du pilote). Si **C1** contient des informations décrivant un protocole FIFO, alors, **T1** accédera à la structure de données réalisant le protocole, et non plus au registre.

Le cas **(d)** représente une interaction entre plusieurs tâches, et un unique pilote (matérialisé par le port **P2**). Comme dans le cas précédent, si le canal **C1** ne contient aucune information, alors les deux tâches accèdent directement au pilote. Sinon, un intermédiaire logiciel (matérialisé par le canal) est utilisé. Comme dans le cas **(b)**, la présence d'un seul canal indique que l'instance de l'intermédiaire logiciel est unique. De même la présence d'un seul port de processeur indique que l'instance du pilote est unique, quel que soit le nombre de tâches en interaction.

Les cas présentés dans la figure 4.18 sont ambigus : en effet les tâches en accédant à un seul port ont le choix entre deux pilotes différents (ports de processeurs). Pour lever l'ambiguïté chaque configuration donnée à gauche, peut être transformée en la configuration correspondante donnée à droite. Dans les deux cas la modification consiste à dupliquer les ports des tâches et les canaux reliés à plusieurs ports de processeur. D'autres modifications, avec des sens différents auraient été possibles. En effet les situations représentées sont ambiguës car elle ne précisent pas suivant quel arbitrage les pilotes correspondant aux ports sont accédés.

Les ports hiérarchiques au niveau du processeur sont interprétés comme des hiérarchies de pilotes de périphériques. Ils peuvent par exemple servir à résoudre le problème présenté dans la figure 4.18, où une tâche doit accéder avec un seul port à deux pilotes de périphériques : le port hiérarchiquement supérieur représente alors un média d'arbitrage (voir la figure 4.19).

4.2.5.2 Décomposition de la spécification en unités de génération

Pour pouvoir traiter correctement les paramètres de génération ils sont groupés par unités de générations qui seront par la suite traitées indépendamment, avant d'être rassemblées pour la génération finale (voir les section 4.2.6 et 4.2.7). Il existe trois types d'unités de génération :

1. l'unité de génération représentant le noyau du système d'exploitation : elle est associée

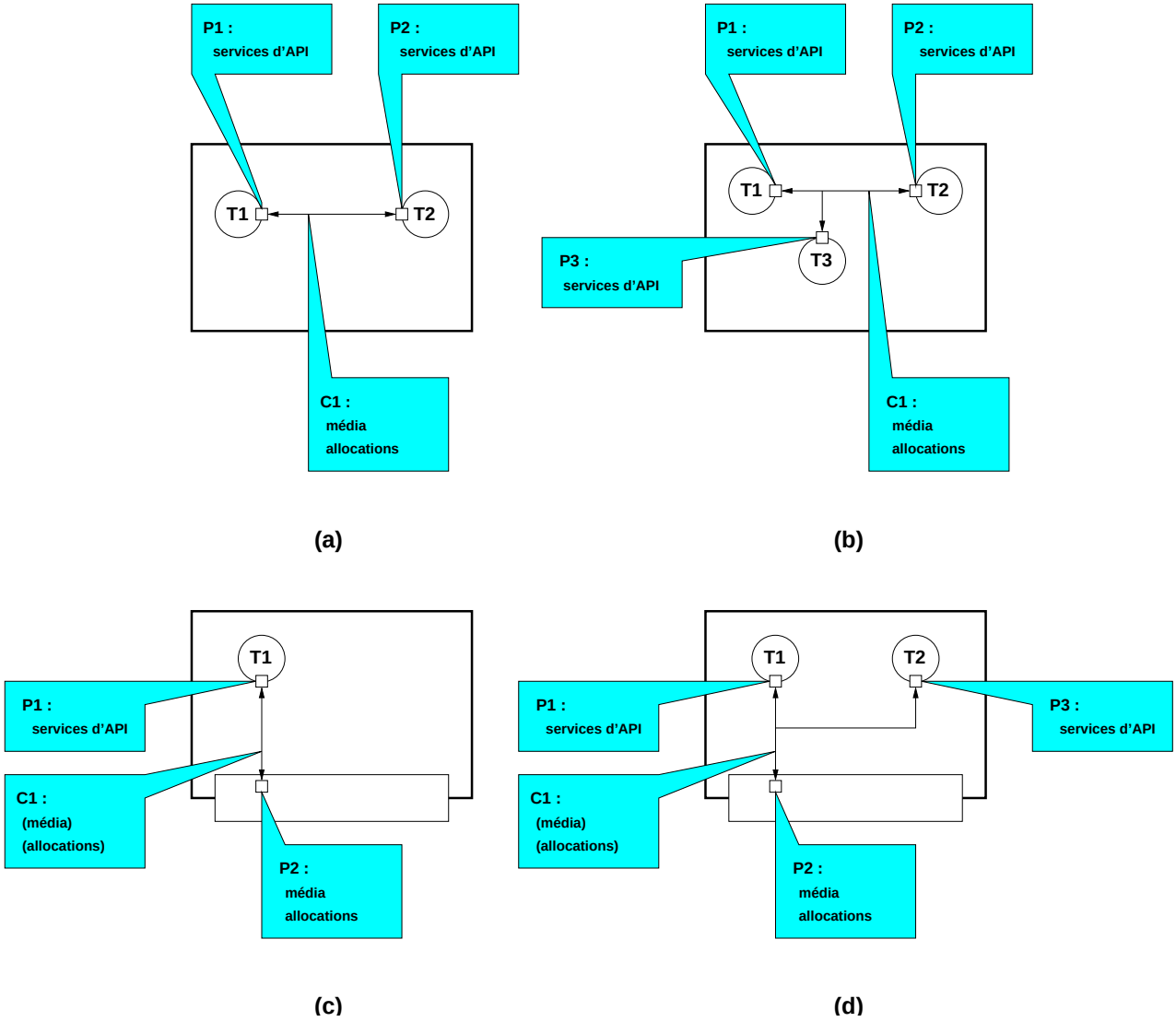

FIG. 4.17 – *Les interactions correctement interprétées*

au module du processeur et aux modules des tâches. Il peut y avoir plusieurs unités d'allocation de ce type, par exemple dans le cas ou il y aurait plusieurs noyaux indépendants.

2. l'unité de génération représentant les communications du système d'exploitation : à chaque canal est donc associé une telle unité de génération. Les paramètres regroupés par cette unité sont ceux du canal, mais aussi ceux des ports connectés à ce canal.

3. l'unité de génération représentant l'ensemble des fonctionnalités globales du système d'exploitation (interruptions, API, etc.) : Elle est associée à l'ensemble des objets de la spécifications, et contient les paramètres groupés comme les unités d'allocation précédentes. Il n'y a qu'une seule unité d'allocation de ce type.

La figure 4.20 donne un exemple de décomposition en unités de générations : la spécification de droite est décomposée en quatre unités de génération.

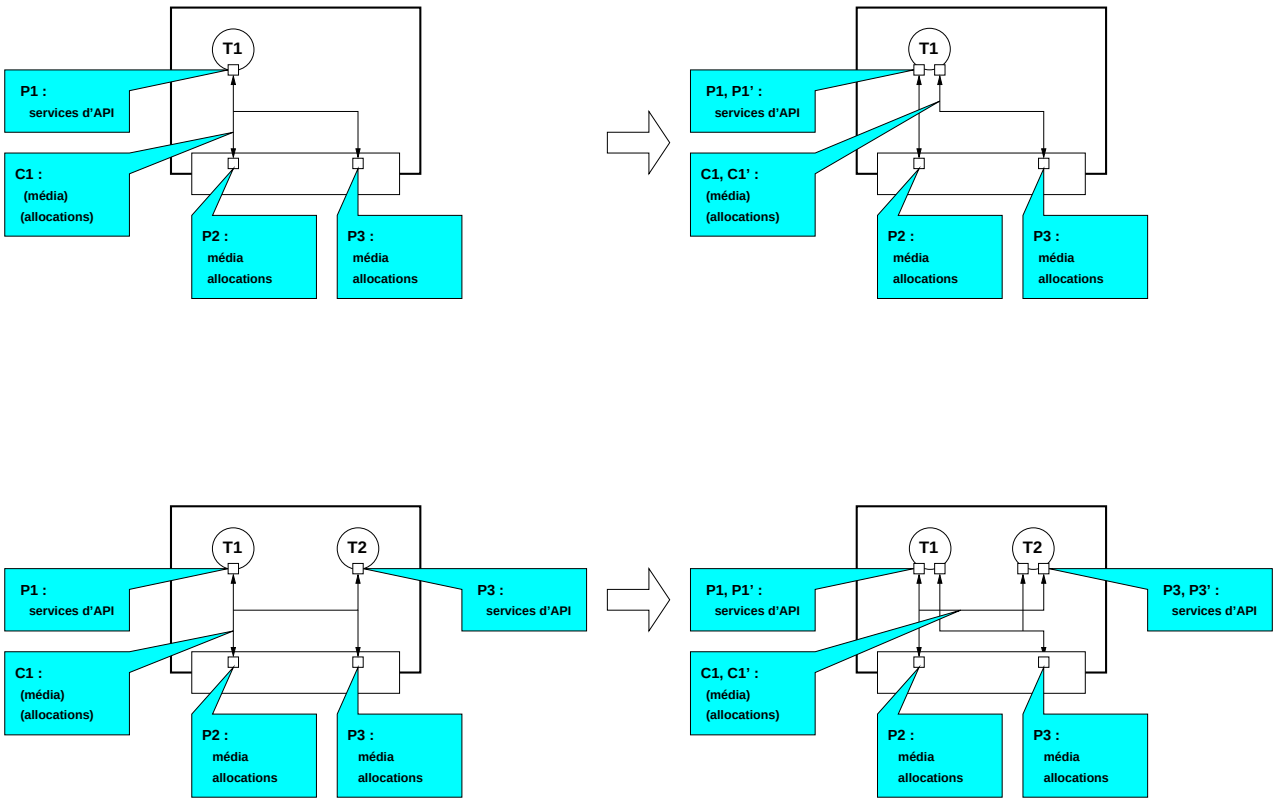

FIG. 4.18 – *Les interactions transformées pour être interprétées*

4.2.5.3 Informations requises pour l'étape de sélection de code du système d'exploitation

Le sélectionneur de code a besoin avant tout des services d'API requis par les tâches. Ensuite, pour raffiner la sélection, il peut avoir besoin des médias utilisés, et des types de processeurs.

4.2.5.4 Informations requises pour l'étape de génération du code du système d'exploitation

Le générateur du code du système d'exploitation a besoin de connaître les éléments du systèmes d'exploitation à générer ainsi que tous les paramètres qui vont spécialiser cette génération pour l'architecture et l'application. Ces paramètres sont rassemblés sous le terme de paramètres d'allocation, et peuvent être des adresses, des tailles, des types de donnée, etc.

4.2.5.5 Informations requises pour l'étape de génération des fichiers de compilation

Le générateur des fichiers de compilation a besoin de la liste des fichiers à compiler et des types des processeurs cibles. Le premier type d'information est fourni par l'étape de sélection de code et par l'étape d'adaptation du code de l'application et sera précisé dans la section 4.2.6. Le second type est juste une chaîne de caractères représentant le type de processeur.

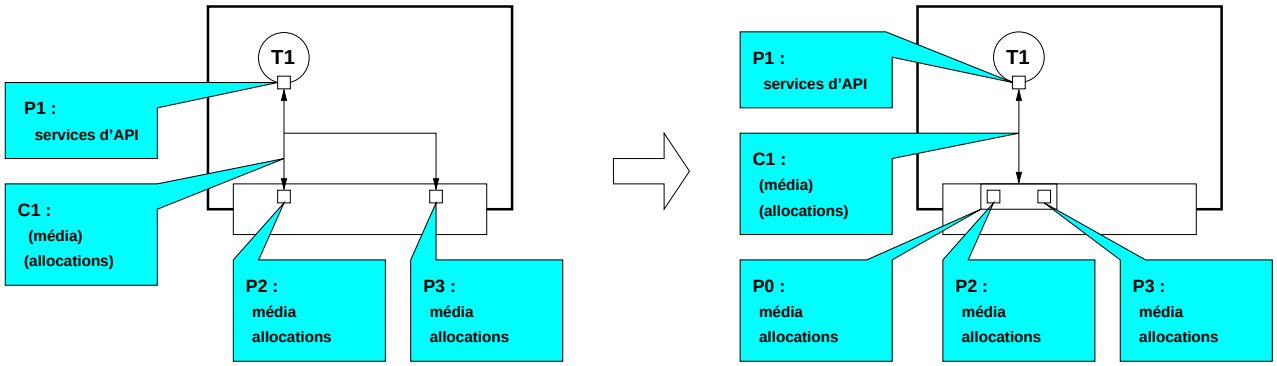

FIG. 4.19 – *Utilisation d'un port hiérarchique pour relier plusieurs pilotes à un seul port de tâche*

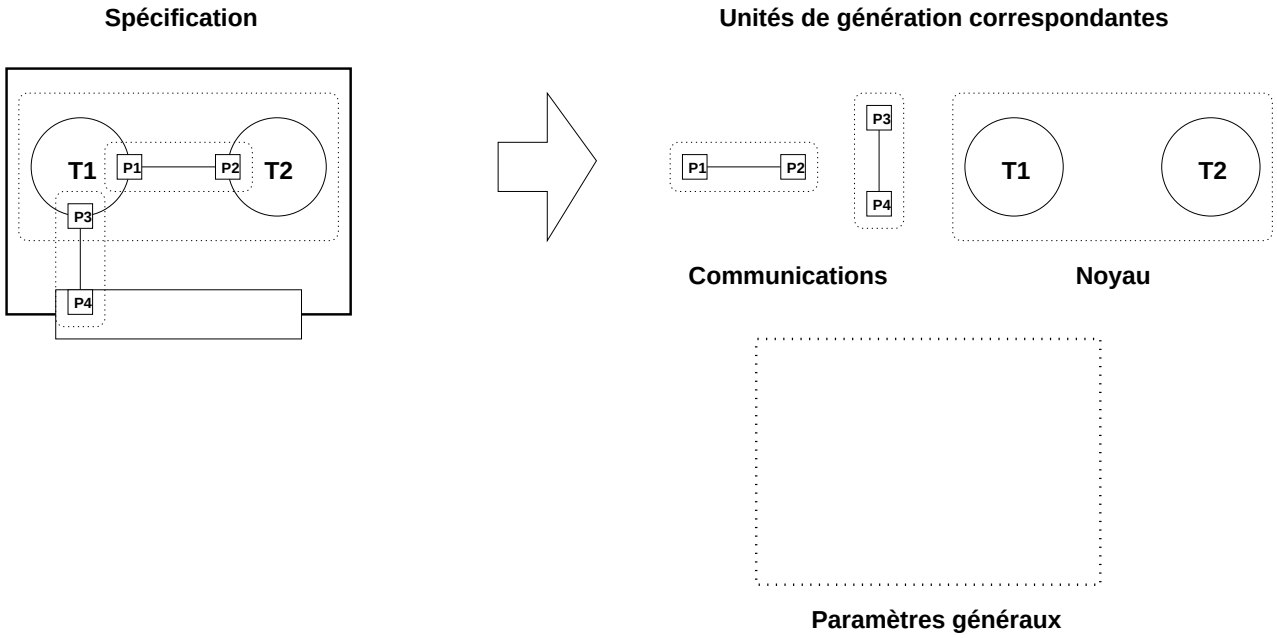

FIG. 4.20 – *Décomposition en unités de génération d'une spécification*

4.2.6 Sélection des éléments du système d'exploitation

La sélection de code prend en entrée des listes de services d'API et de médias associés aux diverses unités de génération produites par l'étape d'analyse de la spécification d'entrée[10]. Cette sélection est exécutée une fois par unité de génération et divisée en deux étapes.

Remarque : dans cette section, toutes les figures représenteront les éléments par des carrés et les services par des cercles.

4.2.6.1 Première étape de sélection

Cette étape calcule la fermeture transitive des services d'API d'une unité de génération : elle construit le graphe de tous les éléments et services valides pour l'architecture en dépendance

10. Nous rappelons que les services d'API se trouvent au niveau des ports de tâches et que les média se trouvent au niveau des canaux et des ports du processeur

directe ou indirecte avec les services initiaux[11]. Un élément est considéré comme valide s'il est compatible avec le processeur et s'il n'utilise pas de média non présent dans l'unité de génération[12].

L'algorithme

L'algorithme de la première passe est basé sur une création récursive de nœuds. Un système de marquage permet de sortir des cycles et d'éliminer les éléments ou services invalides. Il est constitué de deux procédures : la première traite les éléments et la deuxième traite les services. L'algorithme est amorcé en appelant la procédure de traitement de services pour chaque service d'API associé à l'unité de génération en cours de traitement. La figure 4.21 donne des versions simplifiées des deux procédures de l'algorithme.

11. Les relations dépendances sont défini par les services requis et fournis par chaque élément
12. Si une unité de génération ne défini pas de média alors tout type de média est accepté.

```
Notations pour les algorithmes :
 Nx : le noeud x ; S  : service ; E  : élément ; M  : média ; R  : récursivité

Procédure de création d'un noeud service :

création_noeud_service(S : service, R : entier) : noeud
   - si S est marqué :
      - sortir de la procédure en retournant le noeud correspondant à S
     sinon
      - marquer S : en cours
      - créer le noeud NS
      - pour tous les élément E fournissant S faire :
         - NE=création_noeud_élément(E,R+1)
         - si NE est valide :
            - ajouter NE aux fils de NS
   fin si
        fin pour
      - si aucun élément fournissant S n'est valide
         - marquer S : invalide
         - détruire NS
         - pour chaque élément et service de recursivite > R faire :
            - détruire le noeud
            - enlever la marque
   fin pour
         - sortir de la procédure en retournant NULL
        sinon
         - marquer S : traité (R)
         - sortir de la procédure en retournant NS
        fin si
     fin si

Procédure de création d'un noeud élément :

création_noeud_élément(E : élément, R : entier) : noeud
   - si E est marqué :
      - sortir de la procédure en retourant le noeud NE
     sinon
      - marquer E : en cours
      - si E est incompatible avec l'architecture :
         - sortir de la procédure en retournant NULL
        sinon
         - creer le noeud NE
         - pour tous les services S requis par E faire :
            - NS=création_noeud_service(S,R+1)
            - si NS est invalide :
               - marquer E : invalide
               - detruire le NE
               - pour chaque élément et service de recursivite > R faire :
                  - détruire le noeud
                  - enlever la marque
                 fin pour
               - sortir de la procédure en retournant NULL
              sinon :
```

Structure de données générée par l'étape

L'étape génère un graphe orienté ayant deux types de nœuds :

1. les nœuds services : ils représentent des services requis. Ils ont pour fils des nœuds éléments représentant des éléments pouvant apporter les services.
2. les nœuds éléments : ils représentent les éléments pouvant apporter des services requis et compatible avec l'architecture. Ils ont pour fils les services requis par les éléments.

Remarque : Nous disons qu'un nœud B est le fils d'un nœud A s'il existe un arc orienté allant de A vers B. A est alors le père de B. La figure 4.22 illustre ce cas.

FIG. 4.22 – *Père et fils dans un graphe de sélection*

FIG. 4.23 – *Deux arcs de même sens allant de A vers B*

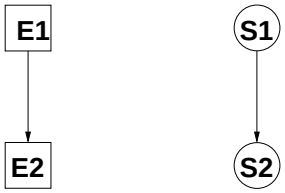

FIG. 4.24 – *Deux arcs allant d'un élément vers un autre élément, et d'un service vers un autre service*

Le graphe généré présente les propriétés suivantes :

1. Ce graphe peut être cyclique : cela arrive lorsqu'il y a des interdépendances entre les éléments.
2. Il ne peut pas y avoir deux arcs dans la même direction reliant deux nœuds : en effet un élément ne peut fournir ni requérir un service donné qu'une seule fois. L'exemple donné dans la figure 4.23 n'est donc pas possible.
3. Il ne peut y avoir d'arc entre deux nœuds éléments ou deux nœuds services : en effet il n'y a pas de relation directe entre les éléments, ni entre le services. Les exemples donnés dans la figure 4.24 ne sont donc pas possibles.
4. Il existe des nœuds services n'ayant pas de père : il s'agit des nœuds correspondant aux services d'API ; ils seront appelés racines du graphe. Tous les autres nœuds services possèdent au moins un père.

5. Ce graphe ne contient que les services et éléments compatibles avec l'architecture. Ainsi ne peuvent être présents dans l'arbre :
   - les éléments incompatibles avec l'architecture (voir le premier paragraphe de la section)
   - les services qu'aucun élément compatible avec l'architecture ne peut fournir
   - les éléments dont aucun service requis ne peuvent être présent dans l'arbre du fait de la deuxième condition
6. Pour les nœuds éléments tous les fils sont obligatoires : il faut que tous les services requis soient fournis par des éléments valides pour que l'élément soit lui-même valide et donc présent dans le graphe.
7. Pour les nœuds services les fils sont facultatifs : il suffit que le service soit fourni par un élément valide pour être valide et donc présent dans le graphe.

Un exemple de graphe complet est donné dans la figure 4.25. Cette figure représente la bibliothèque de la figure 4.9 après la première étape de sélection appliquée à une unité de génération correspondant à la communication par tube logiciel (**API/IO/Pipe**) entre les deux tâches **T1** et **T2** situées sur un même processeur. Remarques :

- En plus de l'élément de communication de haut niveau **Pipe** et des éléments pilotes de périphériques (**Driver**), les éléments du noyau, d'interruptions et de synchronisation générale sont aussi sélectionnés.
- Les deux éléments pilotes de périphériques **FifoSoft** et **FifoHard** sont sélectionnés alors qu'un seul est nécessaire. Cette redondance sera supprimée durant la prochaine étape.

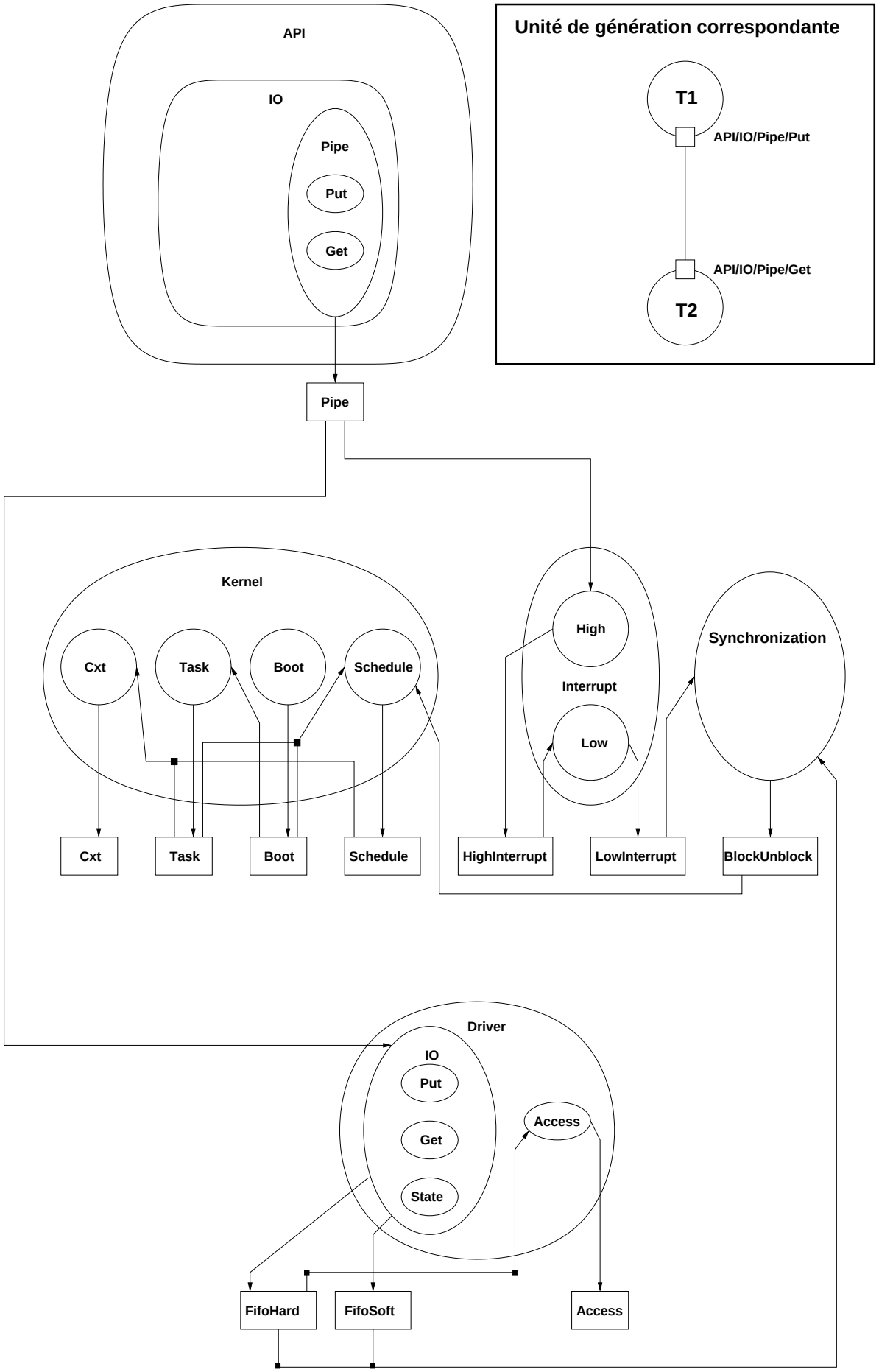

FIG. 4.25 – *La bibliothèque après application de la première étape de sélection pour une unité de génération*

Choix des implémentations

C'est aussi durant cette étape que les implémentations sont choisies : au moment ou la compatibilité d'un élément avec l'architecture est déterminée, son arbre d'implémentations est parcouru en profondeur en sélectionnant à chaque implémentation le fils qui est compatible avec l'architecture. Au final, une liste d'implémentations compatibles avec l'architecture est obtenue pour être associée à l'élément. Dans l'exemple de la figure 4.26 le graphe (a) représente l'arbre d'implémentation d'un élément. Avec un tel arbre, l'élément est compatible avec les processeurs ARM6, ARM7, ARM8, 8086, 80286 et 80386. Le graphe (b) représente les implémentations de l'élément qui seront retenues lorsque le processeur ARM7 est utilisé : leur enchaînement correspond au parcours en profondeur de l'arbre de l'implémentation compatible avec «tout» vers l'implémentation compatible avec ARM7.

Remarque : le code de l'élément est réparti dans toutes ses implémentations, ainsi, lorsqu'une chaîne d'implémentation est sélectionnée pour sa compatibilité avec l'architecture, le code de chacune est nécessaire pour former le code complet de l'élément.

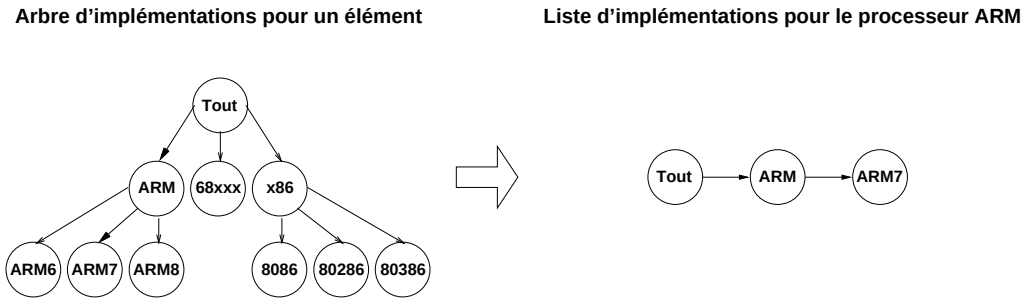

FIG. 4.26 – *Construction de la liste d'implémentation d'un élément pour le processeur ARM7*

4.2.6.2 Deuxième étape de sélection

Fonction de l'étape

La deuxième étape de sélection est appliquée sur le graphe généré par le premier algorithme, et les médias associés à l'unité de génération traitée. Elle construit la liste des éléments nécessaires et suffisants pour fournir tous les services racines en utilisant les médias demandés, c'est-à-dire qu'elle enlève les éléments et services non inutiles pour faire fonctionner l'application sur l'architecture.

l'algorithme

L'algorithme de cette étape parcourt récursivement le graphe précédemment créé en partant de ses racines. Les cycles et les éléments ou services invalides, sont de nouveau traités grâces au procédé de marquage. Pour chaque racine cet algorithme va tout d'abord rechercher un élément dont une implémentation fonctionne avec l'un des médias de l'unité de génération. S'il en trouve un, il va éliminer du graphe tous les services et élément partant de cette racine, conduisant à d'autres médias (pour ce faire il supprime les arcs les concernant ce qui évite de supprimer un élément ou un service correctement utilisé à partir d'un autre racine).

L'algorithme utilise deux fonctions mutuellement récursives ayant la même forme que ceux présentés pour le premier algorithme. La figure 4.27 donne une version simplifiée de cet algorithme.

Un exemple de graphe complet est donné dans la figure 4.28. Cette figure représente la bibliothèque de la figure 4.9 après la deuxième étape de sélection. Désormais il n'y a plus

d'élément redondant : l'élément **FifoHard** a disparu puisque la communication est logicielle. Comme cet élément était le seul à requérir le service **Access**, ce dernier disparaît aussi ainsi que l'élément correspondant.

```
Procédure de recherche d'un média au travers d'un service :

recherche_média_service(NS : noeud service, M : média) : état
   - soit T un tableau d'états

   - si NS est marqué :
      - sortir de la procédure en retournant état de NS
     sinon
      - marquer NS : traité
      - pour tout fils NE de NS faire :
         - T[NE]=recherche_média_élément(NE,M)
         - si T[NE]=vers_média :
            - sortir de la boucle
          fin si
        fin pour
      - si état=vers_média :
         - pour tout fils NE de NS faire :
            - si T[NE]=non_vers_média :
               - enlever NE des fils de NS
              fin si
           fin pour
         - sortir de la procédure en retournant vers_média
        sinon
         - sortir de la procédure en retournant non_vers_média
        fin si
     fin si

Procédure de recherche d'un média au travers d'un élément :

recherche_média_élément(NE : noeud élément, M : média) : état

   - si NE est marqué :
      - sortir de la procédure en retournant état de NE
     sinon
      - marquer NE : traité
      - pour tout fils NS de NE faire :
         - si recherche_média_élément(NS,M)=vers_média :
            - sortir de la procédure en retournant vers_média
           fin si
        fin pour
      - sortir de la procédure en retournant non_vers_média
     fin si
```

FIG. 4.27 – *Deuxième algorithme de sélection*

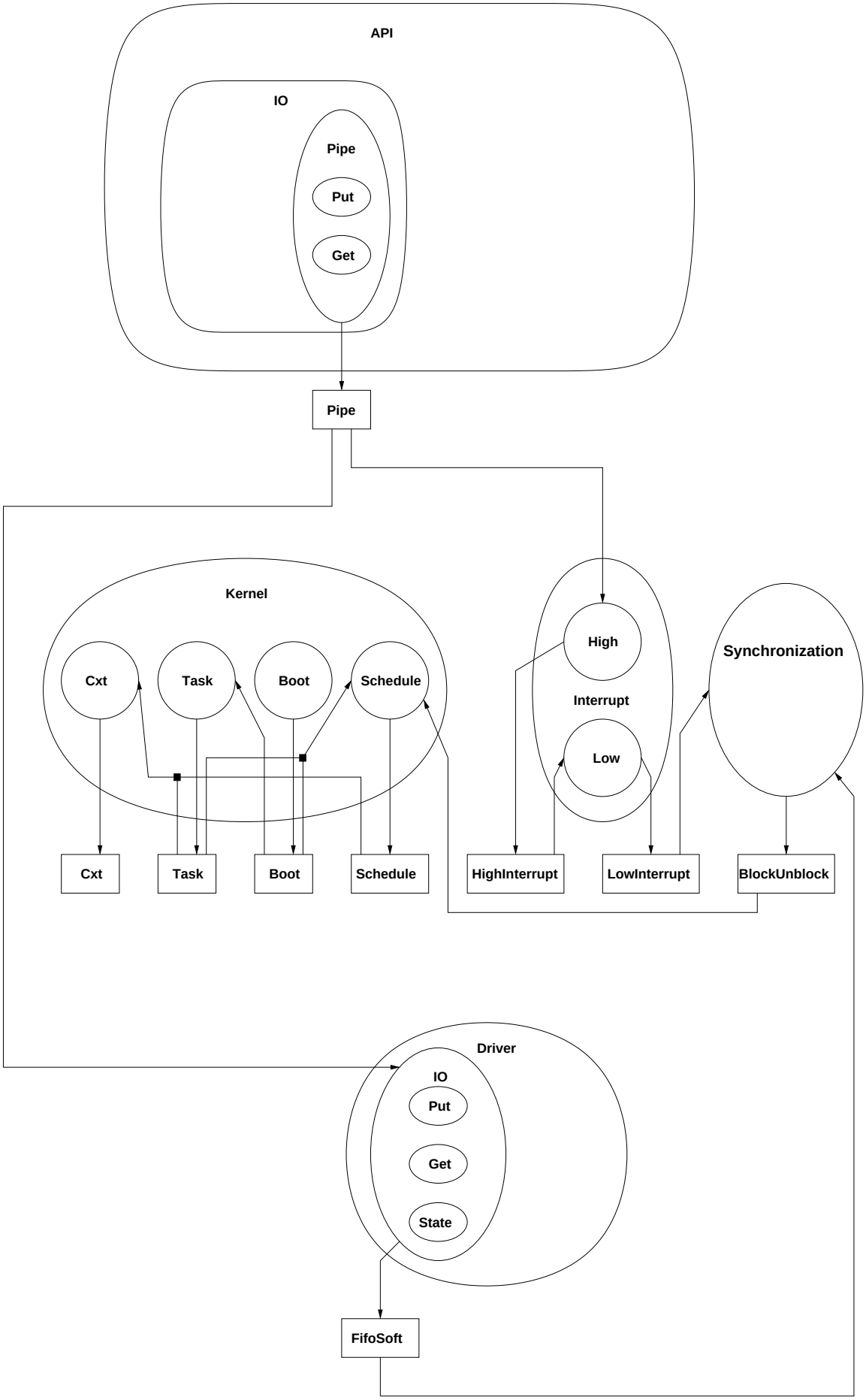

FIG. 4.28 – *La bibliothèque après application de la deuxième étape de sélection pour une unité de génération*

4.2.6.3 Choix de l'utilisateur dans la sélection des éléments

Les algorithmes de sélection présentés fournissent tous les ensembles d'éléments qui fonctionnent pour l'architecture donnée en utilisant les médias indiqués. Il peut rester plusieurs éléments pour fournir chaque service : il faudrait n'en conserver qu'un seul par service.

C'est alors à l'utilisateur de choisir quel est l'élément qu'il désire conserver pour chaque service, sachant que fonctionnellement, toutes les solutions sont valides. Dans des travaux avenir, ce choix pourra être effectué automatiquement avec des algorithmes d'optimisations sous contraintes.

4.2.6.4 Services inutiles pour un élément

Chaque élément peut fournir plusieurs services. Il est fréquent que tous les services fournis par un élément ne soient pas requis après la sélection de code. Les parties du code de l'élément qui sont liés à ces services inutiles pourraient donc être inhibées. Pour ce faire les services effectivement requis sont ajoutés en paramètres associés aux éléments.

La figure 4.29 en donne un exemple : dans le graphe, l'élément **E2** fournit les services **S1**, **S2** et **S3**, mais seuls le services **S1** est requis. Il sera donc ajouté en paramètre associé à E1 que le service **S1** doit être fourni.

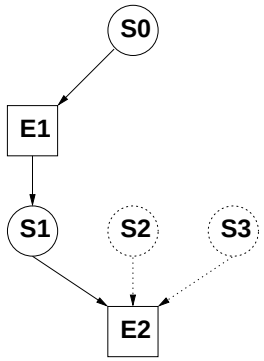

FIG. 4.29 – *Services qu'un élément doit fournir après sélection*

4.2.7 Génération du code du système d'exploitation

Dans cette section nous allons expliquer le fonctionnement du générateur de code du système d'exploitation. Nous indiquerons d'abord comment sont traitées les données fournies par l'étape d'analyse et l'étape de sélection de code. Ensuite nous présenterons le mécanisme d'expansion utilisé pour la génération. Enfin nous verrons plus précisément comment ce mécanisme a été utilisé pour la génération.

4.2.7.1 Traitement des informations d'entrée provenant des étapes d'analyse d'architecture et de sélection de code

Les informations arrivant en entrée du générateur de code sont les listes d'éléments provenant de la sélection de code, et les paramètres associés à chaque unité de génération provenant de l'analyse de l'architecture.

Pour chaque élément précédemment sélectionné, le générateur de code construit une liste de paramètres. Cette liste est construite en y ajoutant tous les paramètres de toutes les unités

de génération qui contiennent l'élément. Un même élément peut se trouver dans plusieurs unités, sa liste peut donc contenir plusieurs jeux de paramètres. Il ne s'agit pas d'un conflit : cela signifie simplement que cet élément est utilisé plusieurs fois. Pour être traités, ces jeux de paramètres multiples sont mis sous la forme de tableaux. Par exemple si un élément réalise une file d'attente, il peut être utilisé par plusieurs canaux de communication avec des adresses différentes. Les paramètres indiquant ces adresses seront donc logés dans un tableau, qui sera traité à l'expansion comme la présence de plusieurs structures de données représentant des files différentes.

4.2.7.2 Expansion des macros

Pour chaque élément sélectionné le générateur de code va faire appel au programme d'expansion de macros avec les paramètres associé à l'élément. Par exemple, pour un élément réalisant une mémoire partagée dont les sources macros sont **shm.c.gen** et **shm.h.gen**, et dont les paramètres sont la table des adresses des mémoires **ADRESSE** et la table des tailles des mémoires **TAILLE**, les commandes suivantes pourront être exécutée :

```
rivemp -DADRESSE=0x4000 -DTAILLE=0x100 shm.c.gen shm.c
rivemp -DADRESSE=0x4000 -DTAILLE=0x100 shm.h.gen shm.h
```

Dans cet exemple, deux fichiers seront générés : **shm.c** et **shm.h**. Il défissent une mémoire partagée d'adresse 0x4000 et de taille 0x100.

Remarque : nous avons vu qu'un élément peut être sélectionné plusieurs fois par processeur[13]. Cependant, il ne sera traité qu'une seule fois dans cette étape.

4.2.7.3 Adaptation des éléments

L'adaptation des éléments consiste en deux opérations : le paramétrage pour définir l'élément final spécifique à l'architecture, et l'optimisation pour obtenir l'élément le plus petit et le plus efficace possible.

Traitement des paramètres d'allocation : paramétrage

Le code des éléments est paramétré avec les paramètres d'allocations. Si nous reprenons l'exemple précédent de la mémoire partagée, les paramètres **ADRESSE** et **TAILLE** sont remplacés par leur valeur dans la macro, ce qui définit la mémoire partagée (voir la figure 4.30.

```
# Code de la macro shm.c.gen
unsigned char* shm=@ADRESSE;
unsigned long int shm_size=@TAILLE;


# Résultat après expansion
unsigned char* shm=0x4000;
unsigned long int shm_size=0x100;
```

FIG. 4.30 – *Paramétrage du code d'un élément simple*

13. mais au plus une fois par unité de génération

Nous avons vu dans la section 4.2.7.1 que lorsqu'un élément était utilisé dans plusieurs unités de générations, les paramètres de ces diverses unités étaient placés dans des tableaux de paramètres.

Lorsqu'un paramètre est sous la forme d'un tableau, cela veut dire que des parties du code de l'élément doivent être dupliquées. Si le paramètre permet la construction d'une structure de données, alors la structure devra être instanciée autant de fois qu'il y a de cases dans le tableau. Si le paramètre sert à adapter une partie du comportement de l'élément, alors ce comportement devra lui aussi être dupliqué. La figure 4.31 donne un exemple d'utilisation des tableaux de paramètres pour la définition de la structure de données utilisée pour les communications par tube (*pipe*).

```
typedef struct
{
   int* status;
   int* data;
   int signal;
} pipe_type;

pipe_type pipes[@SIZEOF{DATA}]=
{
@{
        DEFINE N=SIZEOF{DATA}-1 ENDDEFINE
        FOR I FROM 0 TO N-1 DO
        "    {"STATUS[I]","DATA[I]","SIGNAL[I]"},"
        ENDFOR
        "    {"STATUS[N]","DATA[N]","SIGNAL[N]"}"
}@
};

# Résultat après expansion
typedef struct
{
   int* status;
   int* data;
   int signal;
} pipe_type;

pipe_type pipes[3]=
{
   {0,1,5},{2,3,6},{4,5,7}
};
```

FIG. 4.31 – *Structure de données pour un tube (pipe)*

Liens avec les services : optimisation

Un élément fournit un certain nombre de services. Il est cependant possible que tous les services fournis par l'élément ne soient pas nécessaires.

Pour permettre l'inhibition des services inutiles fournis par l'élément, nous transmettons en paramètres les noms des services effectivement utilisés pour chaque élément. Il suffit alors de tester dans les macros l'existence d'un service donné pour changer le code. Par exemple une fonction peut être présente si et seulement si le service auquel elle correspond est requis, mais il est possible d'effectuer des changement plus élaborés comme par exemple la simplification de l'algorithme d'ordonnancement si les priorités ne sont pas utilisées. La figure 4.32 donne un exemple de l'utilisation des services requis dans une macro : elle représente le code d'une macro pour l'écriture d'un bloc en mémoire. La figure 4.33 donne le résultat de cette macro après expension dans le cas ou le service **LOCKED_SHM** est requis, et la figure 4.34 dans la cas ou le service n'est pas requis.

```
void shm_bloc_write(int shmid, void* buf, size_t size)
{
        int i;
        void* adr=shmaddr(shmid);
@{
        IF (ISDEFINED{LOCKED_SHM}) DO
                "P(shmid);"
        ENDIF
}@
        while(i)
        {
                *adr++=*buf++;
                i--;
        }
@{
        IF (ISDEFINED{LOCKED_SHM}) DO
                "V(shmid);"
        ENDIF
}@
}
```

FIG. 4.32 – *Utilisation d'un service dans une macro*

```
void shm_bloc_write(int shmid,void* buf,size_t size)
{
    int i;
    void* adr=shmaddr(shmid);

    P(shmid);
    while(i)
    {
        *adr++=*buf++;
        i--;
    }
    V(shmid);

}
```

FIG. 4.33 – *Expansion de la macro de la figure 4.32 si le service* ***LOCKED_SHM*** *est requis*

```
void shm_bloc_write(int shmid,void* buf,size_t size)
{
    int i;
    void* adr=shmaddr(shmid);

    while(i)
    {
        *adr++=*buf++;
        i--;
    }

}
```

FIG. 4.34 – *Expansion de la macro de la figure 4.32 si le service* ***LOCKED_SHM*** *n'est pas requis*

4.2.7.4 Assemblage des éléments

Chaque élément fournit et requiert des fonctionnalités (services). L'assemblage des éléments d'un système d'exploitation généré consiste en la concrétisation sous forme de code de ces relations. Par exemple un élément de communication en tube **Pipe** peut requérir les services de pilote **Get** et **Put** pour lire et écrire une donnée. Si l'élément **LockedRegister** fournit ces services, l'assemblage de ces deux éléments consistera à avoir dans les sources de **Pipe** le moyen d'utiliser les services **Get** et **Put** de l'élément **LockedRegister**.

Le modèle utilisé pour matérialiser ces relations est assez courant : il s'agit de l'appel de procédure. Dans l'exemple, cela signifie que l'élément **LockedRegister** aura des procédures **Get** et **Put** que l'élément **Pipe** pourra appeler.

Remarques :

- Ce modèle fonctionne aussi pour les service ne correspondant pas à des fonctions mais plutôt des variables : les appels de procédures correspondant sont simplement des fonctions sans paramètres.
- Pour les services comme un type d'ordonnancement pour le noyau, il n'est pas nécessaire de définir des appels de procédure car les effets de tels services sont implicites au système d'exploitation généré.

Utilisation des macro-entêtes

La plupart des langages de programmation permettent l'utilisation du modèle des appels de procédures. Cependant, un tel mécanisme est peu efficace s'il est trop souvent utilisé[14]. De plus, cela imposerait d'utiliser les mêmes procédures pour tout élément fournissant le même service, ce qui manquerait de souplesse. Cela provoquerait aussi des conflits de noms si plusieurs éléments sélectionnés fournissent les mêmes services. Par exemple, deux pilotes de périphériques différents tels que des FIFO ou des tampons peuvent fournir les même services (**Get** et **Put**) et peuvent être utilisés par le même élément de haut niveau (comme le tube).

Pour résoudre ce problème, les appels de procédures sont réalisés par l'intermédiaire des macros. À chaque élément est associé un fichier de macros contenant les prototypes de tous les appels de procédures. Dès lors, la contrainte sur les noms des prototypes est reportée sur les noms des macros, qui ne sont plus visibles lorsque le code est généré. Un exemple de ces macros est présenté dans la figure 4.35. Elles réalisent un appel de procédure pour l'accès à une donnée. L'argument **mode** de la macro-procédure **GET** de l'élément **LowRegister** permet de l'adapter en fonction de l'élément qui va l'utiliser (ici l'élément **Pipe**). Pour l'exemple nous avons mis deux modes : par identificateur et par adresse.

Remarque : le paramètre **ARG** peut être un tableau ce qui permet d'avoir un nombre quelconque d'arguments pour la macro **GET**, son prototype n'a donc pas de restriction sur le nombre d'arguments.

14. Or il est encouragé dans notre flot de décomposer le plus possible la bibliothèque, ce qui veut dire que l'appel de procédure est beaucoup utilisé.

```
# Définition d'un appel de procédure macro (élément LowRegister) :
@{
DEFINE {MODE,ARG} GET =
    CASE MODE
        WHEN "ID"    DO "(*address["ARG"])"
        WHEN "ADDR" DO "(*"ARG")"
        WHEN OTHERS DO OUTPUT{mode" n'est pas reconnu pour GET"}
    ENDCASE
ENDDEFINE
}@

# Utilisation de l'appel procédure macro (élément Pipe) :
int PipeGet(int id)
{
    LockedRegister p=pipes[id];
    if (!@GET{"ID","p.ready"})
       @WAIT{"p.signal"};
    return @GET{"ID","p.data"};
}

# Code généré :
int PipeGet(int id)
{
    LockedRegister r=regs[id];
    if (!(*address[r.ready]))
          wait(r.signal);
    return (*address[r.data]);
}
```

FIG. 4.35 – *Prototype macro d'un appel de procédure*

Inclusion des fichiers d'entêtes

Le code final du système d'exploitation est, comme tout programme complexe, composé de plusieurs fichiers (modules). Un module peut utiliser des fonctions ou des variables globales fournies par d'autres modules. Pour pouvoir compiler un tel module, il faut faire référence à ces fonctions et variables externes. Par exemple dans le cas du langage C, ces objets externes au module sont déclarés avec le mot clé **extern**. Il est courant de rassembler ces déclarations dans des fichiers d'entêtes qui sont inclus dans le fichier source du module.

Dans la bibliothèque chaque élément dispose en plus de ses fichiers de source de fichiers d'entête contenant ces déclarations. Par exemple pour le langage C, les fichiers sources seront les fichiers d'extension **.c** et les fichiers d'entête seront les fichiers d'extension **.h**. Il suffit alors à la génération du code d'un d'élément d'inclure les fichiers d'entête de tous les éléments dont il dépend. Cependant, un élément peut être assembler avec des éléments différents suivant les architectures. Par exemple, un élément fournissant une communication par tube pourra être assemblé avec différents pilotes de périphériques suivant le type matériel qui est utilisé.

L'inclusion des fichiers d'entête ne peut donc être effectuée que par macro. Les noms des fichiers d'entête qu'un élément utilise sont passés en paramètres de l'expanseur, et sont traités

par une macro comme celle présentée dans la figure 4.36. Cette macro parcourt le tableau des fichiers d'entête **HEADERS** et génère pour chacune une ligne de la forme :

```
#include "entête"
```

```
@{
  IF ISDEFINED{HEADERS} DO
     FOR i FROM 0 TO SIZEOF{HEADERS} DO
        "#include \"HEADERS[i]\"\n"
     ENDFOR
  ENDIF
}@
```

FIG. 4.36 – *Macro d'inclusion d'entêtes*

4.2.7.5 Adaptation de l'application logicielle

Les tâches de la description comportementale de l'application communiquent et se synchronisent par l'intermédiaire de ports abstraits. Chaque port supporte des opérations abstraites du matériel, telle que l'écriture dans un tube, ou la prise d'un sémaphore. Ces opérations font parties de l'API de haut niveau proposées au programmeur de l'application. Cependant, elles ne correspondent pas directement aux appels système spécifiques à l'architecture du système d'exploitation généré. Pour des raisons d'efficacité, il peut être par exemple préférable qu'il y ait deux appels système différents pour l'écriture dans un tube selon qu'elle soit réalisée en logiciel ou en matériel. Les appels système utilisent aussi souvent des identificateurs de ressources qui ne sont pas indiqués dans l'architecture.

Une petite couche logicielle est donc encore nécessaire pour adapter les accès aux ports aux appels système. Cette adaptation ne doit pas diminuer les performances. L'utilisation de la surcharge des méthodes des langages orientés objets est une solution intéressante : les ports sont des objets dont les méthodes sont les fonctions d'accès. Suivant les ports, une même méthode correspondra à un appel système différent.

Remarque : l'utilisation de la surcharge revient à modifier implicitement le code des tâches puisque c'est le compilateur qui effectue cette modification.

Le langage C++ a été utilisé pour cette adaptation, mais cette méthode est généralisable a tous les langages permettant d'effectuer de la surcharge. Si un langage utilisé ne le permet pas, il deviendra nécessaire de modifier explicitement le code de l'application.

Les modifications nécessaires pour faire fonctionner l'application

L'application doit tout d'abord être modifiée lors de toutes ses interactions avec l'extérieur. En pratique cela revient à replacer les appels à des fonctions de communication et de synchronisation générales, par les appels système correspondants.

Les types utilisés par l'application sont des types abstraits, il doivent être remplacés par les types correspondant à l'architecture et au système d'exploitation généré.

Génération des fichiers d'entête

Un élément particulier permet la génération des fichiers d'entête. Cet élément contient une macro pour construire les classes des tâches, et une macro pour construire les classes des ports. Pour chaque tâche et pour chaque port de tâche différent une classe est créée.

La classe d'une tâche contient un lien vers le comportement de la tâche (une méthode), et l'instanciation des ports de la tâche. La figure 4.37 donne un exemple de macro générant une telle classe. La classe générée est celle de la tâche **T1** de la figure 4.20. Cette tâche possède deux ports de communication par tube : **P1** permet la communication avec la tâche **T2** et **P3** permet la communication avec l'extérieur du processeur. Dans la macro, le paramètre **id** est l'identificateur de la tâche défini lors de l'analyse de la spécification. Les macros **TASKNAME**, **TASKPORT** et **PORTNAME** sont des tableaux contenant les noms et identificateurs des tâches et des ports.

```
# Macro permettant de construire la classe d'une tâche
@{
DEFINE {id} TASKCLASS=
    "class " TASKNAME[id] "_class"
    "\n{\n    private :\n"
    FOR i FROM (0) TO (SIZEOF{TASKPORT[id]}-1) DO
        "    " PORTCLASSNAME{TASKPORT[id][i]} " "
        PORTNAME[TASKPORT[id][i]] ";\n"
    ENDFOR
    "    public :\n        void " TASKNAME[id] "();\n"
    "};\n"
ENDDEFINE
}@

# Exemple de classe générée
class T1_class
{
    private :
        Port_Pipe_FifoSoft_class P1;
        Port_Pipe_FifoHard_class P3;
    public :
        void T1();
};
```

FIG. 4.37 – *Exemple de génération de la classe d'une tâche*

La classe d'un port contient une liste de méthodes correspondant aux services d'API fournis par le port : chaque méthode est en fait un appel à la fonction du système d'exploitation correspondante. La figure 4.38 donne un exemple de macro générant une telle classe. Elle présente aussi la macro **PORTCLASSNAME** utilisé dans l'exemple précédent, et qui permet de déterminer le nom de la classe d'un port. La classe générée est celle du port **P1** de la tâche **T1**. Ce port encapsule une communication par tube, et fournit les méthodes **Put** et **Get** qui permettent de lire et d'écrire dans le tube. Dans l'exemple, le nom de la classe est généré en concaténant le nom de l'élément fournissant les services d'API à celui du média utilisé[15]. Le mécanisme est identique pour les méthodes en remplaçant le nom de l'élément par celui du service. Comme précédemment cette macro utilise des tableaux de noms et d'identificateurs.

15. Lorsque plusieurs éléments fournissent des services d'API pour un port, ils sont tous concaténés dans le nom de la classe.

La macro **SERVICEPROTOTYPE** permet de construire le prototype de la méthode correspondant au service qui fait partie de la spécification de l'API fournie au programmeur des tâches. Les macros **SYSNAME** et **SYSARGUMENT** permettent de construire le nom et les arguments de l'appel système, et sont fournies par l'élément correspondant.

```
# Macro permettant de construire le nom d'une classe de port
@{
DEFINE {id} PORTCLASSNAME=
    FOR i FROM (0) TO (SIZEOF{PORTELEMENT{id}}-1) DO
        PORTELEMENT{id}[i] "_"
    ENDFOR
    PORTMEDIA{id} "class"
ENDDEFINE
@}

# Macro permettant de construire une classe de port
@{
DEFINE {id} PORTCLASS=
    "class " PORTCLASSNAME{id} "_class"
    "\n{\n    private :\n"
    FOR i FROM (0) TO (SIZEOF{PORTSERVICE[id]}-1) DO
        SERVICEPROTOTYPE{PORTSERVICE[i]}
        "        \n{\n"
        "        return " SYSNAME{PORTSERVICE[i],PORTMEDIA{i}}
        "(" SYSARGUMENT{SERVICEPROTOTYPE{PORTSERVICE[i]}}
        ");\n        }\n"
    ENDFOR
    "};\n"
ENDDEFINE
}@

# Exemple de classe générée
class Port_Pipe_FifoSoft_class
{
    private :
          Pipe_Put_FifoSoft(int val)
          {
             return sysPipe_Put_FifoSoft(val);
          };

          Pipe_Get_FifoSoft()
          {
             return sysPipe_Get_FifoSoft();
          };
};
```

FIG. 4.38 – *Exemple de génération de la classe d'un port*

4.2.8 Génération des fichiers de compilation

Le but de l'étape de génération des fichiers de compilation est de produire les fichiers Makefile qui permettront l'automatisation de la compilation. Dans cette section, nous décrirons d'abord comment les dépendances entres fichiers sont crées, puis nous nous pencherons sur le problème du choix des outils pour la compilation, enfin nous indiquerons comment les Makefiles sont générés.

4.2.8.1 Création des dépendances de compilation entre les sources

Le but est de compiler complètement tous les fichiers et de les lier en un exécutable par processeur. Dès lors les dépendances entre fichiers sont assez simples : pour chaque processeur, l'exécutable doit dépendre de tous les fichiers objets concernant ce processeur. Du fait des possibilités de make (voir [1] pour plus d'informations), il suffit de ces seules dépendances, et éventuellement de règles de compilations pour les langages non reconnus par make, pour que la compilation complète puisse être effectuée.

4.2.8.2 Choix des outils pour la compilation

Suite à la génération nous avons des fichiers écrits dans divers langages, que nous voulons compiler pour divers processeurs. Nous supposons disposer de tous les outils nécessaires, cependant il convient de choisir les bons.

Les outils nécessaires pour obtenir l'exécutable final sont :

1. le compilateur : c'est l'outil qui produit les fichiers objets à partir des sources. Il y a un compilateur par langage et par processeur.
2. l'éditeur de liens : c'est l'outil qui lie tous les fichiers objets en un exécutable. Il y a un éditeur de lien par processeur, en supposant que le format objet est unique, sinon il peut y en avoir plusieurs.
3. le convertisseur : il existe plusieurs formats pour les fichiers objet ou les fichiers exécutables, pour pouvoir les générer il est intéressant de pouvoir les convertir en des formats reconnus par les outils dont nous disposons. C'est à cela que servent ces programmes de conversion comme par exemple **objcopy**.

La bibliothèque de systèmes d'exploitation permet aussi la description de tous ces outils, ainsi que leurs liens avec les processeurs et les langages (voir la section 4.1.3). Il suffit donc de suivre ces liens pour déterminer automatiquement quels sont les outils à utiliser dans la compilation. Par exemple si un fichier source est un fichier C à compiler pour le processeur ARM7, le générateur de fichier de compilation recherchera parmi tous les compilateur qui admettent le C comme entrée, celui qui peut produire du code pour ARM7, et indiquera dans ce cas qu'il faut utiliser *armcc*. Une fois compilés, les fichiers objets obtenus[16] doivent être assemblés par un éditeur de lien pour obtenir l'exécutable final. Dans notre exemple, le générateur déterminera que l'éditeur de liens à utiliser est *armlink*, qui produit un exécutable au format elf. Supposons maintenant que le simulateur de ARM7 utilisé ne comprenne que le format coff, le générateur cherchera parmi les convertisseur celui qui pourra convertir le elf en coff, et indiquera qu'il faut utiliser *objcopy –out-target=coff*.

16. La compilation d'un fichier source produit un fichier binaire non exécutable appelé fichier objet.

4.2.8.3 Création des *makefile*

Un *makefile* est créé par processeur. Les étapes de sa création sont les suivantes :

1. définition des variables : ces variables sont les définitions des outils (par exemple *CC=armcc* pour un compilateur c pour ARM), et la liste des fichiers objets souhaités
2. définition de règles de compilations non reconnues par make, comme par exemple le passage du format coff au format ihex
3. définition de la règle pour obtenir l'exécutable finale

Un exemple de Makefile généré pour l'exemple étudié au cours de cette partie est donné 4.39. Le compilateur C est **armcc**, le compilateur C++ est **armcpp**, l'assembleur est **aemasm** et l'éditeur de liens est **armlink**. Le format de binaire souhaité est le format **coff**, celui produit après édition de liens est **elf**, il faut donc faire la conversion.

```
.SUFFIXES: .oelf.elf.coff $(SUFFIXES)

CC=armcc
CXX=armcpp
AS=armasm
LD=armlink
CONV=objdump

CFLAG=-I‘pwd‘
CXXFLAG=-I‘pwd‘
ASFLAG=-I‘pwd‘
LDFLAG=-first armboot.oelf

OBJS=     Kernel/armboot.oelf Kernel/armcxt.oelf Kernel/task.oelf \
          Kernel/schedule.oelf Synchronization/blocunbloc.oelf \
          Interrupt/low_interrupt.oelf Interrupt/high_interrupt.oelf \
          API/IO/Pipe/pipe_lockedregiter.oelf API/IO/Pipe/pipe_soft.oelf \
          T1/T1.oelf T2/T2.oelf T3/T3.oelf

VM1.coff : VM1.elf

VM1.elf : $(OBJS)
          $(LD) -first armboot.o -o VM1.elf $(OBJS)

.elf.coff :
          $(CONV) $*.elf $*.coff

.c.oelf :
          $(CC) $(CFLAGS) -o $*.oelf -c $*.c
```

FIG. 4.39 – *Exemple de fichier makefile généré*

4.2.9 L'allocateur de ressources

L'allocateur de ressources sert à définir les paramètres locaux d'allocation non définis dans la spécification (voir la section 4.2.5).

Les ressources qui peuvent être allouées sont la mémoire et les interruptions. La mémoire est une ressource non partageable : lorsqu'une adresse est allouée pour une donnée, elle ne peut pas être utilisée pour une autre[17]. Les interruptions quand à elles sont des ressources partageables : plusieurs traitants peuvent partager la même interruption[18].

4.2.9.1 Allocation de mémoire

Nous avons vu dans la section 4.2.2.2, qu'une mémoire locale au processeur pouvait faire l'objet de demandes d'allocation, ou d'allocations forcées (c'est-à-dire que les adresses réservées sont imposées dans la spécification).

L'allocateur de mémoire traite dans un premier temps les allocations forcées : pour ce faire il divise la mémoire en une succession de blocs (zones d'adresses contiguës), soit entièrement libres, soit entièrement occupés par les adresses imposées par la spécification. C'est la première étape représentée dans la figure 4.40 : les zones fortement grisées sont les allocations forcées.

Ensuite, les demandes d'allocations sont traitées. L'algorithme utilisé est un simple algorithme paresseux : il parcours tous les blocs libres, pour trouver le plus petit qui puisse contenir la demande. La taille de ce bloc est alors diminuée de celle de la demande d'allocation. C'est la deuxième étape représentée dans la figure 4.40 : les zones faiblement grisées sont les demandes d'allocation.

4.2.9.2 Allocation d'interruptions

De même que la mémoire, les interruptions locales au processeur peuvent faire l'objet de demandes d'allocation, ou d'allocations forcées (c'est-à-dire que les adresses réservées sont imposées dans la spécification).

Les interruptions sont utilisées individuellement et une même interruption peut être allouée plusieurs fois. Cependant, pour des raisons de performance, il est préférable de les allouer le moins de fois possibles. La dernière différence est qu'en général les espaces d'interruptions sont beaucoup plus petits que les espaces mémoire.

Pour éviter d'allouer trop souvent la même interruption, l'algorithme précédent peut être utilisé, en permettant toute fois d'allouer plusieurs fois les mêmes interruptions : dès lors, le choix des interruptions pour les demandes d'allocations commence en priorité par les interruptions les moins allouées.

4.2.10 Extension de la bibliothèque de système d'exploitation

Nous avons vu que grâce aux macros, il était possible de d'avoir des éléments génériques ce qui réduisait fortement la taille de la bibliothèque.

Dans cette section nous donnons des indications pour l'ajout efficace d'éléments dans la bibliothèque : comment ajouter des éléments comment ajouter un processeur. Nous introduirons ensuite des pistes pour augmenter la flexibilité de la bibliothèque et diminuer sa taille.

17. S'il y avait des allocations dynamiques, ce ne serait pas le cas.
18. Il suffit qu'ils soient appelés successivement lorsque l'interruption apparaît.

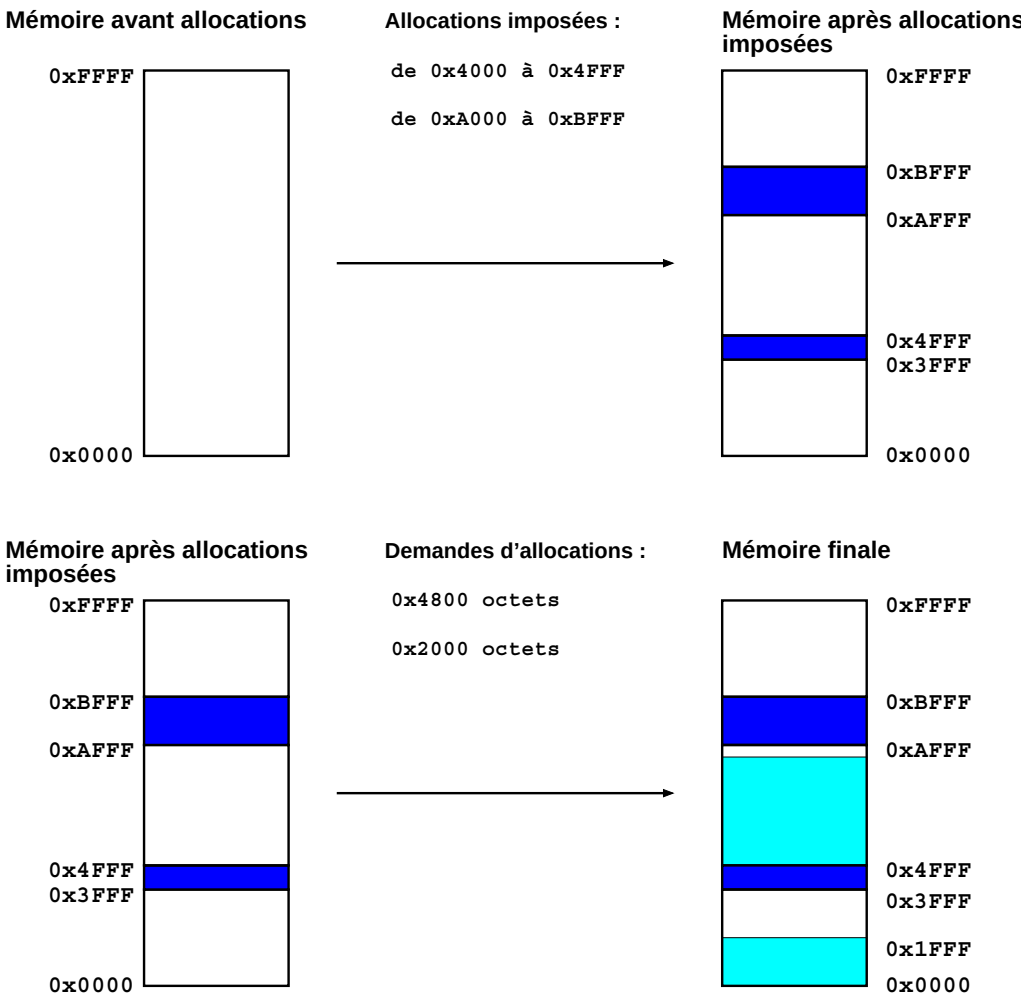

FIG. 4.40 – *Exemple d'allocation mémoire*

4.2.10.1 Découpage du code en éléments

Nous allons ici supposer que nous voulons inclure de nouvelles fonctionnalités au système d'exploitation. Nous supposerons aussi disposer des sources de ces fonctionnalités.

Méthode générale pour ajouter un élément

Pour ajouter un élément il faut tout d'abord déterminer :

1. les services qu'il fournit : pour cela il suffit de traduire ses fonctionnalités en services. Si les services n'existent pas déjà il faut les ajouter.
2. les services dont il a besoin : ceux-ci peuvent être déduits de fonctionnalités qu'il requiert et qu'il ne possède pas, ou de fonctionnalités qu'il requiert et qu'il possède, mais qui sont déjà présentes dans la bibliothèque
3. les compatibilités avec le processeur et les matériels particulier ce qui permet de définir les diverses implémentations de l'élément

Une fois que tous ces paramètres sont déterminés, l'élément peut être créé, en récrivant le code pour chaque chaîne d'implémentation.

Règles de découpage

Nous avons vu dans la section précédente comment ajouter un élément de façon générale sans se soucier de la pertinence de cet ajout. En fait lorsque l'on dispose d'un jeu de fonctionnalités à ajouter, il est possible de les découper en plusieurs éléments. Il faut alors trouver un compromis entre deux qualités opposées :

1. avoir une bibliothèque flexible et de taille réduite impose d'avoir les éléments les plus nombreux et les plus petit possibles
2. pouvoir maintenir aisément la bibliothèque impose d'avoir les éléments les moins nombreux possible et les plus grands

Un bon compromis peut être atteint en suivant les règles suivantes :

1. Un élément ne devrait pas fournir de services de types différents (il faudrait par exemple éviter de mélanger des services de communication avec des services d'ordonnancement).
2. Il devrait être possible d'inhiber chaque service d'un élément.
3. Un élément spécifique pour un processeur ou un matériel ne doit fournir que des services de bas niveau.

La première règle défini la taille d'un élément, et les deuxième et troisième règles permettent d'augmenter la flexibilité.

4.2.10.2 Ajout d'un processeur ou d'un matériel

L'ajout d'un processeur ou d'un matériel requiert de repérer tous les éléments avec qui il peut avoir un rapport, pour pouvoir ajouter ensuite les implémentations correspondantes. Cette recherche peut être longue et fastidieuse, mais elle pourra être automatisée.

4.3 Conclusion

Le flot de ciblage logiciel présenté par ce chapitre rempli le but initialement fixé : effectuer automatiquement l'adaptation du logiciel pour une architecture spécifique en générant les systèmes d'exploitation correspondants. La description initiale distingue bien la description structurelle (décrite dans le langage Colif) de la description comportementale (décrite pour le moment en SystemC). C'est la description structurelle qui guide la génération, la description comportementale doit quant à elle utiliser le modèle d'API abstrait proposé par le flot.

C'est la bibliothèque du système d'exploitation (décrite dans le langage Lidel) qui définit les systèmes d'exploitations, mais aussi le modèle d'API que l'utilisateur devra utiliser. Cette bibliothèque est très flexible, et facilement extensible. En outre, le fait de pouvoir combiner divers éléments fait que, pour une bibliothèque de taille et de complexité modeste, le nombre de systèmes d'exploitation qu'il est possible de générer est très grand.

La grande flexibilité du flot, implique aussi qu'il est difficile d'estimer quelles seront les performances réelles, et quelle sera la qualité des systèmes d'exploitation générés. Pour y apporter quelques éléments de réponses, le chapitre suivant présente un exemple d'utilisation de ce flot pour une application non triviale.

Chapitre 5

Application du flot de ciblage logiciel

Sommaire

Le flot de ciblage présenté dans cette thèse a été utilisé pour quelques applications, dont notamment une partie de modem VDSL. Ce chapitre montre l'utilisation du flot pour cette application. Plus précisément, la première section présente l'application ainsi que les objectifs d'une telle démonstration. Pour mener à bien ce projet, il a fallu développer les bibliothèques nécessaires à la génération et à la simulation de l'application. C'est notamment le cas pour la bibliothèque de systèmes d'exploitation, qui est présentée dans la deuxième section. Enfin la dernière section donne les résultats et conclusions de cette expérience.

5.1 Description d'une application : un *framer VDSL*

Le VDSL *(Very-high-data-rate DSL)* est une technique de communication utilisant les lignes téléphoniques. Elle fait partie des techniques *xDSL (Digital Subscriber Line)*. Cette technique est encore au stade du prototype, et de nombreuses entreprises proposent leur propre version du protocole VDSL.

STMicroelectronics en partenariat avec Telia ont eux-mêmes proposé une version du protocole VDSL appelé *Zipper-DMT*. Cette version utilise le codage *DMT (Discret Multi-Tone)* qui découpe la bande de fréquence initiale en sous-bandes de transmission simultanée. Le *Zipper* découpe la bande de fréquence en 2048 sous-bandes qui peuvent être allouées dynamiquement, et par logiciel, à différents utilisateurs. Un prototype de modem utilisant cette technique a été développé. C'est une partie de ce modem que nous allons étudier ici.

5.1.1 Démonstration de l'utilisation du flot de l'équipe SLS sur une application VDSL

5.1.1.1 Contexte

Le modem VDSL d'origine

L'architecture du modem est constituée d'*ASIC* et de *FPGA* pour le traitement du signal fixe, d'un processeur spécifique pour le traitement du signal configurable, et d'un processeur généraliste pour le contrôle du modem et l'interface avec le PC hôte. Pour gérer les nombreuses tâches du processeur de contrôle, un système d'exploitation commercial avait été choisi. L'architecture de la partie numérique de ce modem est présentée dans la figure 5.1.

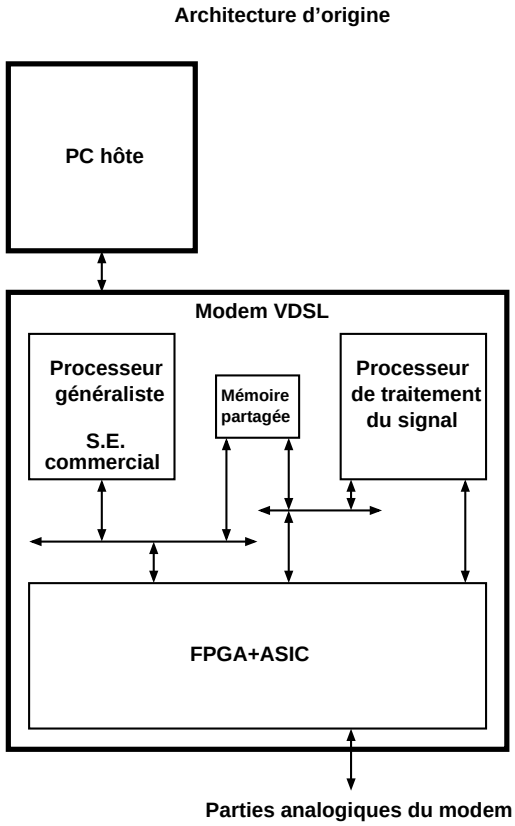

FIG. 5.1 – *La partie numérique du modem VDSL*

Modifications à apporter au modem

Le modem existe sous la forme d'un prototype, et a été testé avec succès. Cependant, son architecture doit être changée : en effet la partie ASIC est trop importante pour permettre l'adaptation rapide du modem si le protocole était changé. De plus, le système d'exploitation choisi n'est plus supporté, il doit donc être remplacé. Il a donc été proposé de montrer l'utilisation du flot de conception pour remplacer une partie des ASIC par un autre processeur, et pour générer les systèmes d'exploitation. La figure 5.2 illustre les changements attendus sur l'architecture du modem. Le travail se limite à la partie grisée de la figure, c'est-à-dire la partie «*framer*» du VDSL.

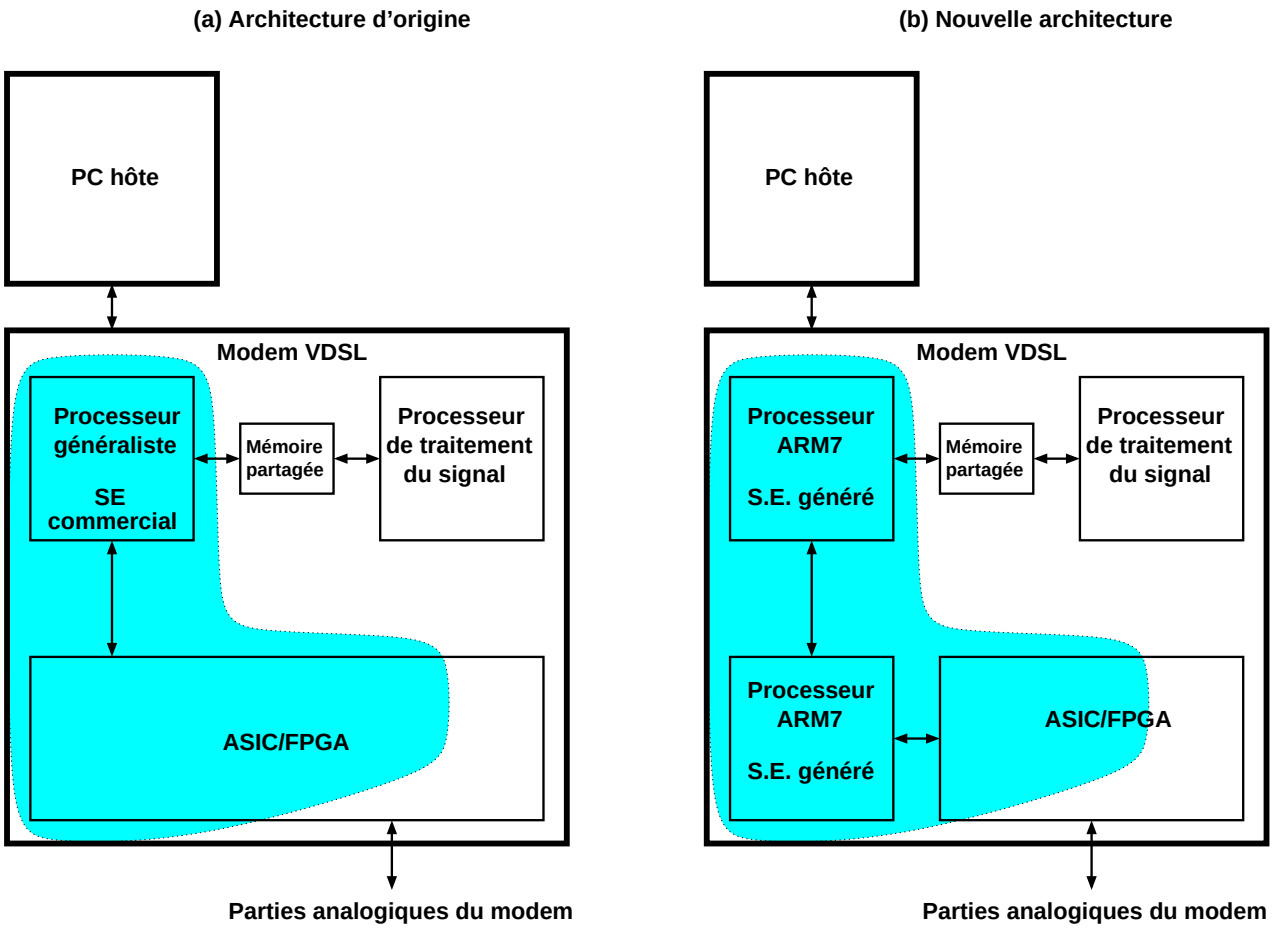

FIG. 5.2 – *La démonstration VDSL*

5.1.1.2 But de la démonstration

Pour le groupe SLS

Le but de la démonstration n'est pas développer un nouveau prototype de modem VDSL : c'est un travail qui demande beaucoup de ressources en hommes et en matériel, et pour lequel nous n'avons pas les compétences. L'application VDSL n'est là que pour servir de cadre à l'expérimentation de notre flot : elle donne des contraintes et une complexité que notre flot doit être capable de supporter.

Les objectifs à atteindre sont multiples :

- utiliser le flot de bout en bout dans le but de montrer sa validité, mais aussi dans le but de faire le lien entre les diverses étapes. En effet, ce flot étant récent, il n'avait jamais été complètement appliqué : les divers outils du flot n'étaient donc pas tous compatibles. Cette démonstration a permis de commencer les adaptations.
- montrer la flexibilité et la rapidité du flot : le fait de changer de processeur ou de protocole de communication ne doit demander qu'un minimum de travail et de temps
- tester le bon fonctionnement des divers outils
- enrichir les diverses bibliothèques avec des composants réalistes issus d'une application réelle

Pour la génération de systèmes d'exploitation

Dans le cas de la génération de systèmes d'exploitation, des objectifs supplémentaires viennent s'ajouter à ceux précédemment cités.

En premier lieu, il fallait montrer que la qualité du système d'exploitation généré se compare favorablement à celle d'un système d'exploitation commercial : les performances doivent être comparables, et la taille bien plus réduite. Il doit aussi permettre une séparation plus claire entre le système d'exploitation et l'application logicielle. En effet les systèmes d'exploitation embarqués classiques proposent rarement toutes les fonctionnalités nécessaires pour une application donnée. Ces fonctionnalités sont donc ajoutées sous la forme de tâches supplémentaires ou de routines d'interruptions, ce qui rend le code de l'application confus et potentiellement instable. La souplesse de la bibliothèque du système d'exploitation doit permettre d'ajouter facilement de nouvelles fonctionnalités aux systèmes d'exploitation sans avoir à les transférer dans le code de l'application.

Enfin il fallait montrer la capacité de l'outil de génération à supporter des points souvent négligés par d'autres outils tels que :

- la possibilité d'utiliser plusieurs fois un même élément dans des conditions différentes. Dans de nombreux outils, cette source d'ambiguïté n'est pas levée, ce qui interdit l'utilisation multiple d'un même élément.
- la possibilité de gérer des communications multipoints
- la possibilité de réduire au maximum la taille de la bibliothèque proportionnellement au nombre de possibilités qu'elle offre

5.1.2 Spécification de l'application

5.1.2.1 Construction de la spécification

La spécification VDSL disponible comprenait la description de l'architecture, une description informelle des différentes tâches logicielles, et la spécification en C++ du *framer*.

Cet ensemble de spécifications n'était pas directement utilisable dans le flot. Nous les avons donc traduites en une spécification SystemC compatible avec notre flot. Le *framer* a été découpé en deux parties : une partie devant être remplacée par un processeur ARM7, et l'autre conservée en tant qu'*ASIC*. Le processeur d'origine a lui-même été remplacé par un processeur ARM7. Enfin, la partie logicielle d'origine a été redéfinie en ne conservant que les parties contrôlant le *framer*.

5.1.2.2 La spécification que nous avons utilisé dans le flot

La figure 5.3 donne l'architecture globale d'application que nous avons étudié. Elle est constituée de trois modules : **M1** est le processeur de contrôle et d'interface entre le *framer* et le reste du modem, c'est lui qui remplace le processeur d'origine. **M2** est le processeur de configuration du flot de données du *framer*, il remplace une partie des *ASIC* du circuit. Enfin, **M3** est l'*ASIC* du flot de données du *framer*.

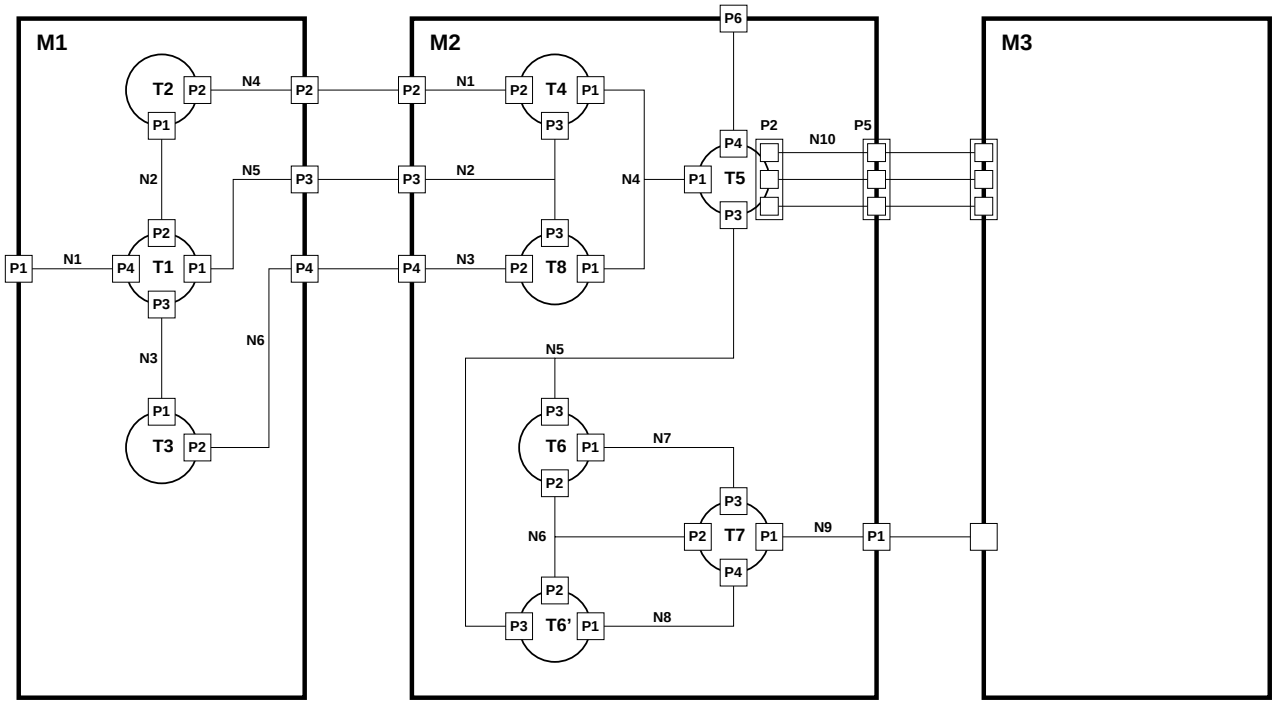

FIG. 5.3 – *L'application VDSL*

Objets	Paramètres
M1	Processeur : ARM7
	Allocation : IT=interrupt:0:256
	Allocation : MEM=char:0:65536
P1	Média : interrupt
	Interruption : 0:(IT)
P2	Média : LockedRegister
	Interruption : 1:(IT)
	type donnée : long int
	Addresse donnée : 0xF00000
	Addresse état : 0xF00004
P3	Média : LockedRegister
	Interruption : 2:(IT)
	type donnée : long int
	Addresse donnée : 0xF00008
	Addresse état : 0xF0000C
P4	Média : LockedRegister
	Interruption : 3:(IT)
	type donnée : long int
	Addresse donnée : 0xF00010
	Addresse état : 0xF00014
N2	Interruption : (1):IT
N3	Interruption : (1):IT
T1	Priorité : 0
	Sources : M1/T1.cpp
P1	Service : API/IO/Pipe/Get
P2	Service : API/Synchronization/Signal/Notify
P3	Service : API/Synchronization/Signal/Notify
P4	Service : API/Synchronization/Signal/Wait
T2	Priorité : 0
	Sources : M1/T2.cpp
P1	API/Synchronization/Signal/Wait
P2	API/IO/Pipe/Put
T3	Priorité : 0
	Sources : M1/T3.cpp
P1	API/Synchronization/Signal/Wait
P2	API/IO/Pipe/Put

FIG. 5.4 – *Paramètres de l'application VDSL pour le module M1*

Objets	Paramètres
M2	Processeur : ARM7 Allocation : IT=interrupt:0:256 Allocation : MEM=char:0:65536
P1	Média : Buffer Interruption : 0:(IT) type donnée : long int Taille : 64 Addresse donnée : 0xF00000 Addresse état : 0xF00004
P2	Média : LockedRegister Interruption : 1:(IT) type donnée : long int Addresse donnée : 0xF00008 Addresse état : 0xF0000C
P3	Média : LockedRegister Interruption : 2:(IT) type donnée : long int Addresse donnée : 0xF00010 Addresse état : 0xF00014
P4	Média : LockedRegister Interruption : 3:(IT) type donnée : long int Addresse donnée : 0xF00018 Addresse état : 0xF0001C
P5	Média : Register type donnée : long int Addresse donnée : 0xF00020
P6	Média : LockedRegister type donnée : long int Addresse donnée : 0xF00024
N4	type donnée : long int Taille : 16 Addresse donnée : (64):MEM
N5	type donnée : long int Taille : 1 Addresse donnée : (4):MEM
N6	type donnée : long int Taille : 64 Addresse donnée : (256):MEM Interruption : (1):IT
N7	Interruption : (1):IT
N8	Interruption : (1):IT
T4	Priorité : 0 Sources : M2/T4.cpp
P1	Service : API/IO/Pipe/Put
P2	Service : API/IO/Pipe/Get
P3	Service : API/IO/Pipe/Put
T5	Priorité : 0 Sources : M2/T5.cpp
P1	Service : API/IO/Pipe/Get
P2	Service : API/IO/Over/Put
P3	Service : API/IO/Memory/Attach
P4	Service : API/Synchronization/Timer/SetVal Service : API/Synchronization/Timer/Wait
T6, T6'	Priorité : 0 Sources : M2/T6.cpp
P1	Service : API/Synchronization/Signal/Wait
P2	Service : API/IO/Memory/Attach Service : API/Synchronization/Semaphore/P
P3	Service : API/IO/Memory/Attach
T7	Priorité : 0 Sources : M2/T7.cpp
P1	Service : API/IO/Pipe/Put
P2	Service : API/IO/Memory/Attach Service : API/Synchronization/Semaphore/V
P3	Service : API/Synchronization/Signal/Notify
P4	Service : API/Synchronization/Signal/Notify
T8	Priorité : 0 Sources : M2/T8.cpp
P1	Service : API/IO/Pipe/Put
P2	Service : API/IO/Pipe/Put
P3	Service : API/IO/Pipe/Get

FIG. 5.5 – *Paramètres de l'application VDSL pour le module M2*

Les figures 5.4 et 5.5 précisent les paramètres des différents objets de la spécification concernant la génération de systèmes d'exploitation :

- Au niveau des processeurs nous retrouvons leur type et la définition des ressources locales. Par exemple dans la figure 5.4, le module **M1** est un processeur de type ARM7, dont les ressources en mémoire sont de 65536 octets, et dont les ressources en interruption (virtuelles) sont de 256 numéros.
- Au niveau des tâches nous retrouvons leur priorité et un lien vers leur code source. Par exemple dans la figure 5.4, la tâche **T1** a la priorité 0 et a pour source le fichier **M1/T1.cpp**. Dans cette figure, toutes les tâches ont la priorité 0, ce qui revient à dire qu'il n'y a pas de priorité.
- Au niveau des ports des tâches nous retrouvons les services de haut niveau utilisés par les tâches. Par exemple dans la figure 5.4 le port **P1** de la tâche **T1** requiert le service **API/IO/Pipe/Get**.
- Au niveau des ports des processeurs nous retrouvons les médias utilisés et les paramètres d'allocations associés (adresses, types des données, etc.). Par exemple dans la figure 5.4, le port **P1** du module **M1** utilise le média **LockedRegister** aux adresses 0xF00000 et 0xF00004, avec l'interruption numéro 1.
- Au niveau des canaux, nous retrouvons les paramètres d'allocations si nécessaires. Par dans la figure 5.4 le canal **N2** utilise l'interruption numéro 1 pour la synchronisation du tube entre **T2** et **T3**.

5.1.2.3 Les protocoles utilisés

Au niveau des tâches

Les tâches communiquent et se synchronisent en utilisant divers protocoles que nous allons décrire.

Le premier est le protocole libre (**Over**) : il permet d'écrire ou lire directement dans une variable globale au système. Il est surtout utile pour accéder directement aux registres d'un périphérique quelconque. Dans le cas de l'application, il est utilisé pour accéder aux registres de configuration du flot de données. Dans la figure 5.3, ce protocole est utilisé pour le port **P2** de la tâche **T5**.

Le deuxième est le tube (**Pipe**) : il permet à un nombre quelconque de tâches de s'échanger des données de manière sûre en FIFO. Lorsqu'une tâche essaie d'écrire dans un tube, elle est bloquée jusqu'à ce que la ressource concernée[1] puisse accepter cette écriture. Il en est de même en lecture : une tâche est bloquée tant qu'il n'y a aucune donnée dans la ressource. Dans la figure 5.3, les ports de tâche des connexions **N4**, **N5** et **N6** du module **M1** sont du type **Pipe**.

Le troisième est la mémoire (**Memory**) : il s'agit de la définition d'une de zone mémoire accessible par une tâche. Pour pouvoir y accéder, la tâche doit «attacher» cette zone de mémoire, ce qui lui permet ensuite d'y accéder en tableau ou pointeur, comme n'importe quelle autre variable. Cette mémoire peut être partagée entre plusieurs tâches, mais aucune garantie n'est apportée quant à la cohérence du contenu d'une mémoire partagée : si certains accès sont critiques, il convient d'utiliser en plus un autre protocole tel que les sémaphores pour en garantir l'accès exclusif. Dans la figure 5.3, les ports de tâche des connexions **N6** et **N5** du module **M2** sont du type **Memory**.

Le quatrième est le signal (**Signal**) : une tâche peut attendre ou émettre un signal. Lorsqu'une tâche est en attente d'un signal, elle est bloquée jusqu'à ce que le signal soit émis par

1. La ressource concernée dépend de la réalisation de bas niveau du tube.

une autre tâche. L'émission d'un signal n'est pas bloquante et il est perdu si aucune tâche ne l'attend. Dans la figure 5.3 les ports de tâches des connexion **N1**, **N2** et **N3** du module **M1** sont du type **Signal**.

Le cinquième est le sémaphore (**Semaphore**) : il s'agit du protocole de synchronisation décrit par Dijkstra [34]. Dans la figure 5.3, les ports de tâche de la connexion **N6** sont aussi du type **Semaphore** en plus d'être du type **Memory**.

Enfin le dernier est le temporisateur (**Timer**) : il s'agit d'un protocole qui permet à une tâche d'être bloquée pendant un temps programmé à l'avance. Le port **P4** de la tâche **T5** est du type **Timer**.

Au niveau des processeurs

Les processeurs communiquent entre eux et avec le flot de données par l'intermédiaire de plusieurs protocoles. Chacun est réalisé par un contrôleur de communication qui dispose de sa propre interface avec le logiciel. Ces sont ces interfaces avec le logiciel qui nous intéressent dans la génération des systèmes d'exploitation, puisque ce sont elles qui permettent de choisir les pilotes de périphériques.

La première interface, est celle de type registre (**Register**) : elle permet d'accéder simplement à un registre par le biais de son adresse sur le bus du processeur. C'est l'interface qui est utilisée pour accéder aux registres de configuration du flot de données. Dans la figure 5.3, le port **P5** du module **M2** est du type **Register**.

La deuxième est celle de type registre à verrou (**LockedRegister**) : cette interface est composée d'un registre de donnée dans lequel le logiciel peut lire ou écrire des données, un registre d'état qui indique s'il est possible d'accéder en lecture ou en écriture au registre de donnée, et d'une interruption qui indique un changement d'état. Cette interface convient bien à des communications matérielles de type FIFO par exemple. Dans la figure 5.3, les ports **P2**, **P3** et **P4** des modules **M1** et **M2** sont du type **LockedRegister**.

La troisième est celle de type tampon (**Buffer**) : cette interface est composée d'un registre de donnée et d'une interruption qui indique quand le tampon doit être rempli. Lorsque cette interruption intervient, le logiciel doit envoyer un nombre fixe de données par l'intermédiaire du registre. Dans la figure 5.3, le port **P1** du module **M2** est du type **Buffer**.

La dernière est celle de type interruption (**Interrupt**) : il s'agit tout simplement d'une interruption qui est utilisée pour activer une action logicielle quelconque. Dans la figure 5.3, le port **P1** du module **M1** est du type **Interrupt**.

5.2 La bibliothèque de système d'exploitation pour l'application

5.2.1 Les choix effectués pour la construction de la bibliothèque

Nous avons vu dans la section 4.1.3 que la bibliothèque offrait de nombreuses libertés quant à la définition des éléments et des services. Même en s'imposant les règles de la section 4.1.4 il reste encore de nombreuses orientations possibles pour définir la bibliothèque.

Remarque : les orientations pour la définition de la bibliothèque n'ont aucune influence sur l'outil. En effet c'est au niveau des dépendances et du code des macros qu'elles prennent effet.

5.2.1.1 Les langages utilisés pour le code des éléments : le langage C et les langages d'assemblage

Choix des langages

Il est naturel de nos jours d'utiliser des langages de programmation de haut niveau, et notamment les langages orientés objet, pour décrire le logiciel. Cependant, les langages de haut niveau ne permettent pas de décrire certaines parties des systèmes d'exploitation telles que les changements de contexte, ou le traitement des interruptions. Une partie du code d'un système d'exploitation doit donc impérativement être décrite en langage d'assemblage.

Il reste cependant un grande partie du système d'exploitation qui peut être décrite dans un langage de haut niveau. Pour cette partie, nous avons utilisé le langage C qui est couramment utilisé dans ce domaine.

La spécification d'entrée du flot est décrite en SystemC qui est un langage basé sur le langage C++. Nous avons pourtant écarté ce langage car il pose des problèmes à l'édition de liens comme nous le verrons dans la section 5.3.1.4. De plus, même si les compilateurs ont fait beaucoup de progrès, un programme écrit en C++ est souvent moins efficace qu'un programme équivalent écrit en C. Enfin l'approche orienté objet simplifie le travail d'un programmeur d'application de haut niveau en cachant l'implémentation des données et du code dans des structures abstraites (les objets). Mais cet avantage peut devenir un inconvénient lorsqu'il s'agit de décrire du logiciel de très bas niveau, comme les pilotes de périphériques ou l'implémentation doit être parfaitement connue pour pouvoir interfacer correctement avec les périphériques.

Proportions du C et de l'assembleur

C'est donc le langage C et le langage d'assemblage qui ont été choisis pour décrire le code des éléments de la bibliothèque. Il reste à déterminer sous quelles proportions nous allons utiliser ces langages.

Pour cette première bibliothèque, nous avons utilisé le moins possible le langage d'assemblage : son utilisation a été réduite à certaines parties de l'initialisation du système, au changement de contexte et à l'entrée dans les traitants d'interruption. ce choix facilite l'ajout d'un nouveau processeur, puisqu'il limite le nombre d'éléments à modifier[2]. Cependant, cela ne permet pas de réaliser un système d'exploitation optimal en taille et en performance. En effet, malgré les progrès effectués en compilation, certaines parties restent plus efficaces si elles sont écrites directement en langage d'assemblage.

Il est prévu, au fur et à mesure de l'extension de la bibliothèque, d'ajouter aux éléments actuellement écrits en C des implémentations en langage d'assemblage. Cela permettra d'avoir le choix entre un code C général ou un code en langage d'assemblage spécifique à certains processeurs.

Remarque : il est possible d'écrire des parties de code en langage d'assemblage à l'intérieur d'un programme en langage C (grâce à la directive **asm**). Cette possibilité n'a cependant pas été utilisée car, pour permettre la compilation du programme complet, le code assembleur est souvent modifié par le compilateur. Or certaines parties critiques, telles que le changement de contexte, ne doivent pas être changées pour pouvoir fonctionner correctement.

5.2.1.2 Découpage en éléments

Nous avons vu dans la section 4.1.4.1 qu'il était souhaitable de découper toute fonctionnalité générale en au moins deux éléments (un de haut niveau, et un de bas niveau) ce qui

2. Nous rappelons que le langage C est indépendant du processeur contrairement aux langages d'assemblage.

permet de décliner cette fonctionnalité sur divers média à moindres frais.

Nous avons poussé cette recommandation au maximum dans la bibliothèque de telle sorte qu'aucune fonctionnalité n'est complètement définie par un élément. Les avantages de ce choix sont que la taille de la bibliothèque est très réduite comparée à ses possibilités (voir la section 5.2.2) et que l'extension de la bibliothèque est peu coûteuse : il suffit souvent d'ajouter un service d'API de haut niveau pour qu'il soit directement décliné sur divers médias (voir la section 5.3.1.2). Et inversement, le simple ajout d'un pilote est suffisant pour permettre à de nombreux service d'API d'accéder à de nouveaux périphériques. L'inconvénient est que les relations entre les divers éléments de la bibliothèque deviennent très complexes comme en témoigne la figure 5.6 ; de plus, il devient nécessaire d'ajouter des éléments ne servant que de lien entre plusieurs fonctionnalités comme c'est le cas pour **interrupt**.

5.2.1.3 L'interface entre les tâches de l'application et le système d'exploitation

Nous parlons ici des fonctions ou appels système fournis par le système d'exploitation, et non pas des méthodes qui encapsulent ces dernières pour adapter le code des tâches sans avoir à les modifier (voir la section 4.2.7.5).

Appels système ou appels de procédure

Le système d'exploitation peut offrir l'accès à ses services par l'intermédiaire de simples appels de procédure ou par des appels système. Ces derniers provoquent des interruptions logicielles qui font basculer le processeur en mode système, ce qui permet notamment de masquer les interruptions lors des traitements critiques.

Dans la bibliothèque, toute l'interface avec le système d'exploitation est réalisée par des appels système. Cela permet une séparation claire entre l'application et le système d'exploitation ; par contre, les appels système ajoutent des délais importants dans les réponses de l'application.

Masquage des interruptions

Dans un système d'exploitation, lors d'une section critique, il est conseillé de masquer le moins d'interruptions possible et le moins longtemps possible.

En pratique, il n'est pas aisé de déterminer pour chaque cas quelles sont les interruptions à masquer. Pour simplifier la définition de la bibliothèque, les interruptions sont systématiquement masquées lors d'un appel système.

Généralité de l'interface entre l'application et le système d'exploitation

Comme nous l'avons vu dans la section 5.1.2.2, les systèmes d'exploitation des deux processeurs doivent générer l'accès à de très nombreuses ressources. De plus, les points d'accès au système d'exploitation (matérialisés par les ports des tâches) sont eux aussi très nombreux. L'interface du système d'exploitation[3] doit permettre d'accéder sans ambiguïté à ces ressources.

La méthode la plus simple pour le faire est de générer un appel système pour chaque service de chaque port de tâche. Cette solution est aisément intégrable dans la bibliothèque, mais elle apporte beaucoup de redondance dans le code et accroît d'autant sa taille. La solution opposée consiste à avoir un appel système par service de haut niveau, et de passer en paramètre les identificateurs permettant de désigner la ressource et le port concerné. Avec cette dernière solution, le code des appels système devient compliqué, surtout lorsqu'un même service de haut niveau utilise des ressources de types différents (comme c'est le cas pour le

3. C'est-à-dire l'ensemble des appels système proposés.

tube dans l'application VDSL). Cette complexité de l'appel système peut fortement réduire ses performances.

Un compromis entre les deux précédentes approches a été préféré lors de la définition de la bibliothèque : un appel système est généré par service de haut niveau couplé à un type de ressource. Ainsi le code d'un appel système reste simple et rapide, sans qu'il y ait de redondance puisque le code est différent suivant le type de ressource.

5.2.2 Contenu de la bibliothèque

La bibliothèque contient des éléments pour générer un système d'exploitation complet pouvant faire tourner l'application VDSL. Les processeurs supportés sont le processeur ARM7, le processeur 68000 et le pseudoprocesseur UNIX (voir la section 5.3.1.5).

5.2.2.1 Relations entre les éléments et les services

Les relations entre les éléments et les services de la bibliothèque sont données dans la figure 5.6. Cette figure est complexe. Nous allons décrire rapidement les éléments qui la composent au cours des sections suivantes.

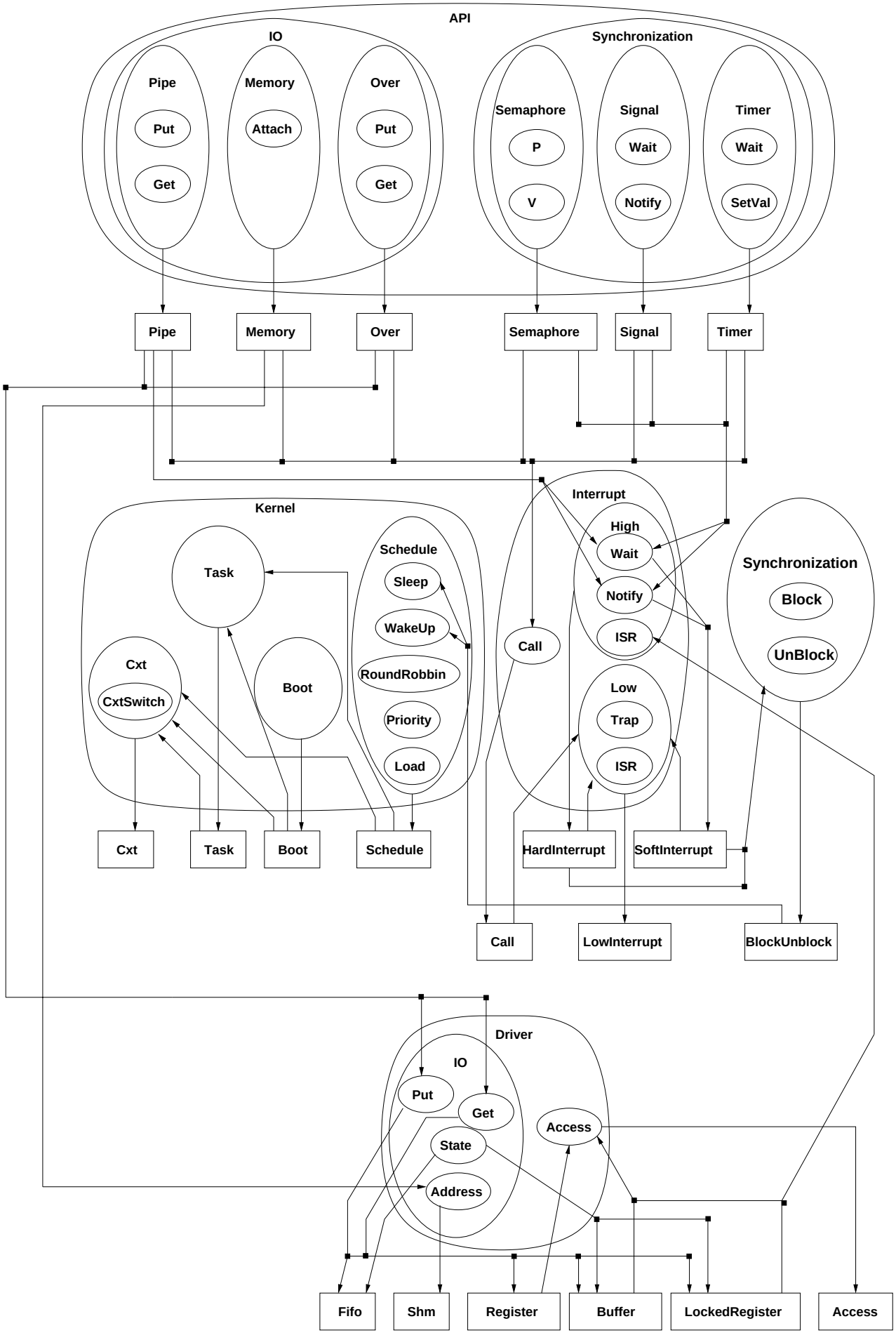

FIG. 5.6 – *La bibliothèque de système d'exploitation pour l'application VDSL*

5.2.2.2 Les éléments fournissant les services d'API

Ces éléments permettent la génération des appels système utilisables par l'application. Pour fonctionner il doivent être assemblés avec des éléments fournissant les services de **Driver** (qui permettent d'accéder au ressources de bas niveau) et des éléments fournissant les services de **Interrupt** (qui permettent la gestion des événements).

l'élément «Pipe»

Il fournit les services d'API réalisant une communication par tube. Cet élément permet la génération de deux types d'appels système : **Put** et **Get** pour écrire et lire des données atomiques dans le tube.

L'élément «Memory»

Il fournit le service d'API permettant «d'attacher» un zone mémoire à une tâche.

L'élément «Over»

Il fournit les services d'API permettant d'accéder directement à une ressource telle qu'un registre. Cet élément permet la génération de deux types d'appels système : **Put** et **Get** pour écrire ou lire directement dans la ressource.

L'élément «Semaphore»

Il fournit les services d'API réalisant des synchronisations par sémaphores. Cet élément permet la génération de deux types d'appels système : **P** et **V** pour effectuer les opérations de même nom sur un sémaphore.

L'élément «Signal»

Il fournit les services d'API réalisant des synchronisations par signaux. Cet élément permet la génération de deux types d'appels système : **Wait** pour attendre un signal et **Notify** pour en générer un.

5.2.2.3 Les éléments fournissant les services de «Kernel»

Ces éléments permettent la génération du noyau du système d'exploitation, c'est-à-dire l'ensemble des fonctionnalités permettant de gérer les tâches. Ces éléments ont de nombreuses dépendances entre eux, mais ils ne dépendent d'aucune autre famille de services.

L'élément «Cxt»

Il fournit la fonction de changement de contexte qui permet de passer de l'exécution d'une tâche à l'exécution d'une une autre tâche. Son code dépend entièrement du processeur cible, c'est pourquoi il dispose d'une implémentation différente par processeur.

L'élément «Task»

Il fournit le squelette de la structure de données qui décrit une tâche. La structure de données finale est obtenue en complétant ce squelette avec les informations propres au processeur (élément **Cxt**) et à l'algorithme d'ordonnancement (élément **Schedule**).

L'élément «Boot»

Il fournit le code d'initialisation du processeur et du système d'exploitation. Une partie de cet élément dépend du processeur. Il dispose donc d'un arbre d'implémentations partant d'un code général pour l'initialisation du système d'exploitation, et allant à un code spécifique au processeur pour son initialisation.

L'élément «Schedule»

Il fournit l'algorithme d'ordonnancement des tâches. C'est un élément très flexible qui permet la construction de divers algorithmes d'ordonnancement. Il peut fournir l'algorithme du tourniquet avec la possibilité de pondérer différemment l'utilisation du processeur pour chaque tâche. Il peut aussi fournir un algorithme à base de priorités (statiques ou dynamiques). Enfin ces algorithmes peuvent être combinés.

5.2.2.4 Les éléments fournissant les services de «Interrupt»

Ces éléments permettent de générer le gestionnaire d'interruptions pour le système d'exploitation. Pour s'abstraire des spécificités du processeur vis à vis des interruptions, un système d'interruptions virtuelles a été mis en place : il fournit un nombre arbitraire d'interruptions virtuelles qui encapsulent les interruptions réelles (matérielles ou logicielles). Ces éléments ont besoin des primitives de synchronisation de base du système fournies par l'élément **BlockUnBlock**.

L'élément «Call»

Il encapsule les interruptions logicielles du type «appel système».

L'élément «LowInterrupt»

Il encapsule les interruptions matérielles : à chaque fois que l'une d'entre elles apparaît, la routine d'interruption générée par cet élément va lire le numéro de l'interruption virtuelle souhaitée sur le bus donnée du processeur.

Les éléments «HardInterrupt» et «SoftInterrupt»

Ils fournissent l'interface de haut niveau pour ces interruptions virtuelles : **Wait** et **Notify** permettent d'en attendre ou d'en générer une. **HardInterrupt** permet aussi la déclaration de routines d'interruption virtuelles.

5.2.2.5 L'élément fournissant les services de «Synchronization»

Cet élément fournit deux primitives de base et des files d'attente permettant de construire toutes les synchronisation dans le système. La primitive **Block** met la tâche courante dans l'état endormi et la place dans une file d'attente. La primitive **UnBlock** réveille une des tâches endormie dans une file d'attente. Pour fonctionner, cet élément a besoin des services de **Kernel** permettant d'endormir et de réveiller des tâches.

5.2.2.6 Les éléments fournissant les services de «Driver» (pilotes)

Ces éléments permettent de générer des pilotes de périphériques. Ce sont eux qui font l'interface entre le logiciel et le matériel. Ils peuvent utiliser des routines d'interruption, et donc requérir les éléments fournissant les services de **Interrupt**.

L'élément «Fifo»

Il réalise un tampon logiciel fonctionnant en FIFO. Il ne gère aucun périphérique ; cependant sa fonctionnalité et son niveau étant identique à ceux des pilotes de périphériques il a été placé dans cette catégorie.

L'élément «Shm»

Il réalise une zone de mémoire locale partagée entre plusieurs tâches. C'est un élément très simple, qui fournit simplement la plage d'adresse qui représente la mémoire partagée.

L'élément «Register»

Il permet l'accès à des registres. Il contient des tables d'adresse de registres, et des macros permettant d'y accéder. Cet élément est indépendant du processeur quel que soit son mode d'accès aux périphériques. En effet il utilise le service **Access** qui est chargé de générer le code permettant ce type d'accès.

Les éléments «LockedRegister» et «WaitRegister»

Ils permettent l'accès aux périphériques de type **LokedRegister** et **Buffer** (voir la section 5.1.2.3). Ils contiennent des tables d'adresse de registres donnée et état, ainsi que des routines d'interruption appelées au moment d'un changement d'état. Comme l'élément **Register**, ces éléments utilisent le service **Access**.

L'élément «Access»

Il fournit les macros permettant d'accéder à un périphérique d'un processeur. Si le processeur place ses entrées/sorties en mémoire, cet accès se réduit à l'utilisation d'un pointeur. Sinon, il faut utiliser des instructions particulières du processeur.

5.3 Résultats

5.3.1 Évaluation du flot de génération de système d'exploitation

5.3.1.1 La spécification d'entrée

Langage de la spécification d'entrée

La démonstration a mis en avant l'intérêt de ne pas décrire la spécification dans le langage Colif: en effet c'est un langage qui a été prévu pour être aisément analysé et modifié par les outils du flot. Il est fastidieux de décrire une spécification dans ce langage. Au contraire SystemC, avec sa syntaxe provenant du C++, est accessible pour un homme, surtout si ce dernier a une bonne expérience en programmation C ou C++.

Les premières spécifications ont été écrites directement en Colif, puis, pour l'application VDSL, un traducteur SystemC vers Colif a été développé. Le gain en productivité est important : une spécification telle que celle de l'application VDSL fait plus de 1 Mo en Colif, tandis que l'équivalent en SystemC ne fait plus que 23 Ko.

Un inconvénient commun entre SystemC et Colif est la détection des erreurs d'architecture[4]. Pour détecter ces erreurs, un outil graphique de visualisation d'architecture Colif a donc été développé dans le cadre de la démonstration.

Paramètres de la spécification d'entrée

Nous avons indiqué dans la section 5.1.2.2 les paramètres concernant la génération de système d'exploitation. Il y en a beaucoup d'autres pour la génération des interfaces matérielles et pour la génération des modèles de simulation. Bien que le flot permettent de s'affranchir de nombreux détails d'implémentation il reste beaucoup trop de paramètres à définir manuellement. La démonstration a mis en relief ce problème puisque une grande partie des erreurs de spécification étaient des erreurs au niveau des paramètres. Ces erreurs ne sont souvent détectées qu'au cours de la génération.

Ce problème au niveau des paramètres montre que l'étape de synthèse de la communication (qui doit produire ces paramètres comme l'indique la section 1.4.2.4) est nécessaire pour que le flot soit complet.

4. C'est-à-dire les mauvais placements ou interconnexions des modules, des ports et des nets.

5.3.1.2 La bibliothèque de système d'exploitation

Contenu de la bibliothèque

La bibliothèque a été complété pour satisfaire l'application VDSL. C'est pourquoi, elle n'a manqué d'aucun élément pour permettre la génération des systèmes d'exploitation de la démonstration.

Bien que la bibliothèque ait été spécialement prévue pour un application particulière, la plupart des éléments de la bibliothèque sont suffisamment généraux pour qu'elle puisse être utilisée pour d'autres applications.

De plus, les dépendances indirectes et l'utilisation d'un langage de macros complet ont permis de factoriser fortement la bibliothèque. C'est ainsi qu'avec 21 éléments, la bibliothèque permet de générer 37 fonctionnalités différentes (29 appels système et 8 algorithmes d'ordonnancement différents) indépendamment du processeur cible. En outre, le code de chaque élément dépasse rarement la centaine de ligne. Enfin ces diverses fonctionnalités peuvent toutes cohabiter dans un même système d'exploitation (il est même possible d'avoir plusieurs ordonnanceurs de tâches).

Pour le moment la bibliothèque ne contient que peu de pilotes de périphériques de peu API. Cela signifie que le rapport entre nombre d'éléments et nombre de fonctionnalités possible sera bien meilleur à mesure que la bibliothèque sera complétée.

Les systèmes d'exploitation générés à partir de la bibliothèque

Les figure 5.7 et 5.8 représentent les éléments sélectionnés et assemblés à partir de la bibliothèque pour les deux processeur. Dans ces figures, les rectangles tangents représentent un assemblage d'éléments, tandis que les formes arrondies représentent des liens par appels des fonctions C[5]. Le haut de la figure correspond à l'interface avec l'application tandis que le bas correspond au noyau. Ces figures montrent par exemple qu'un élément d'API tel que **Pipe** est dupliqué pour être assemblé avec chaque pilote de périphérique utilisé pour ce type de protocole.

5. ces fonctions correspondent aux services indiqués dans la forme

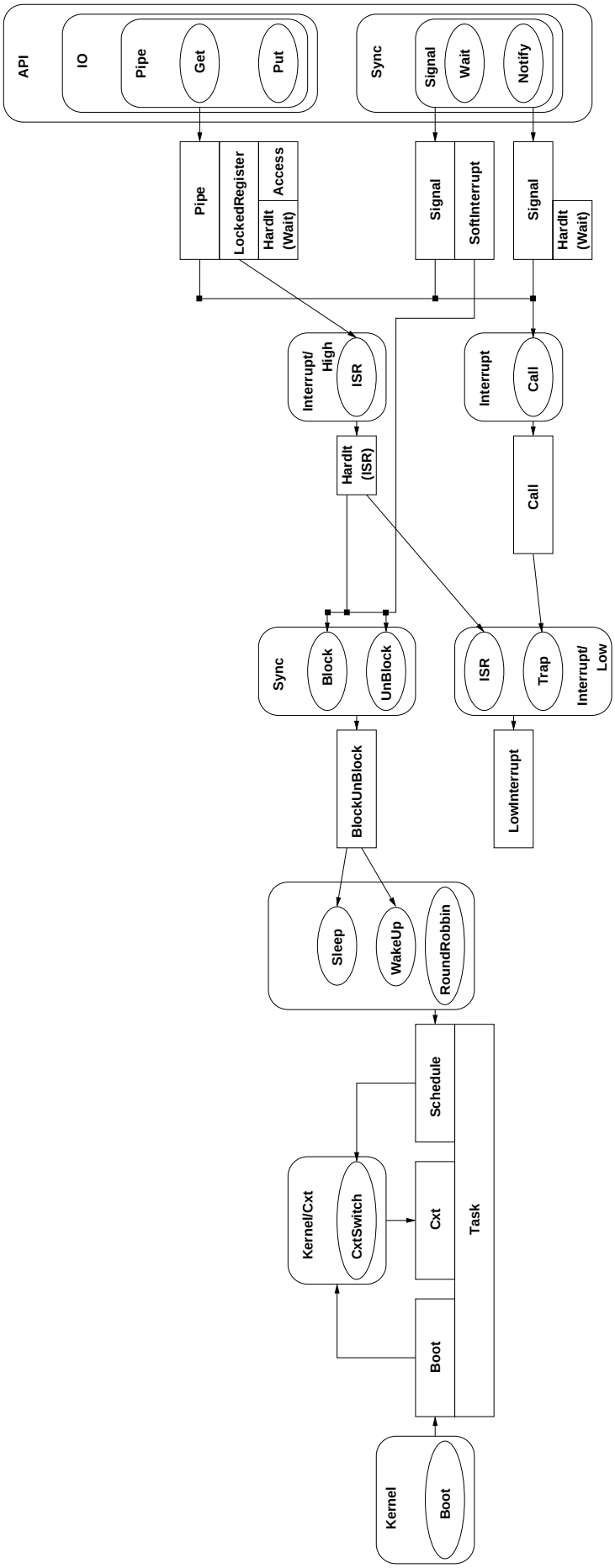

FIG. 5.7 – *Le système d'exploitation généré pour le module M1*

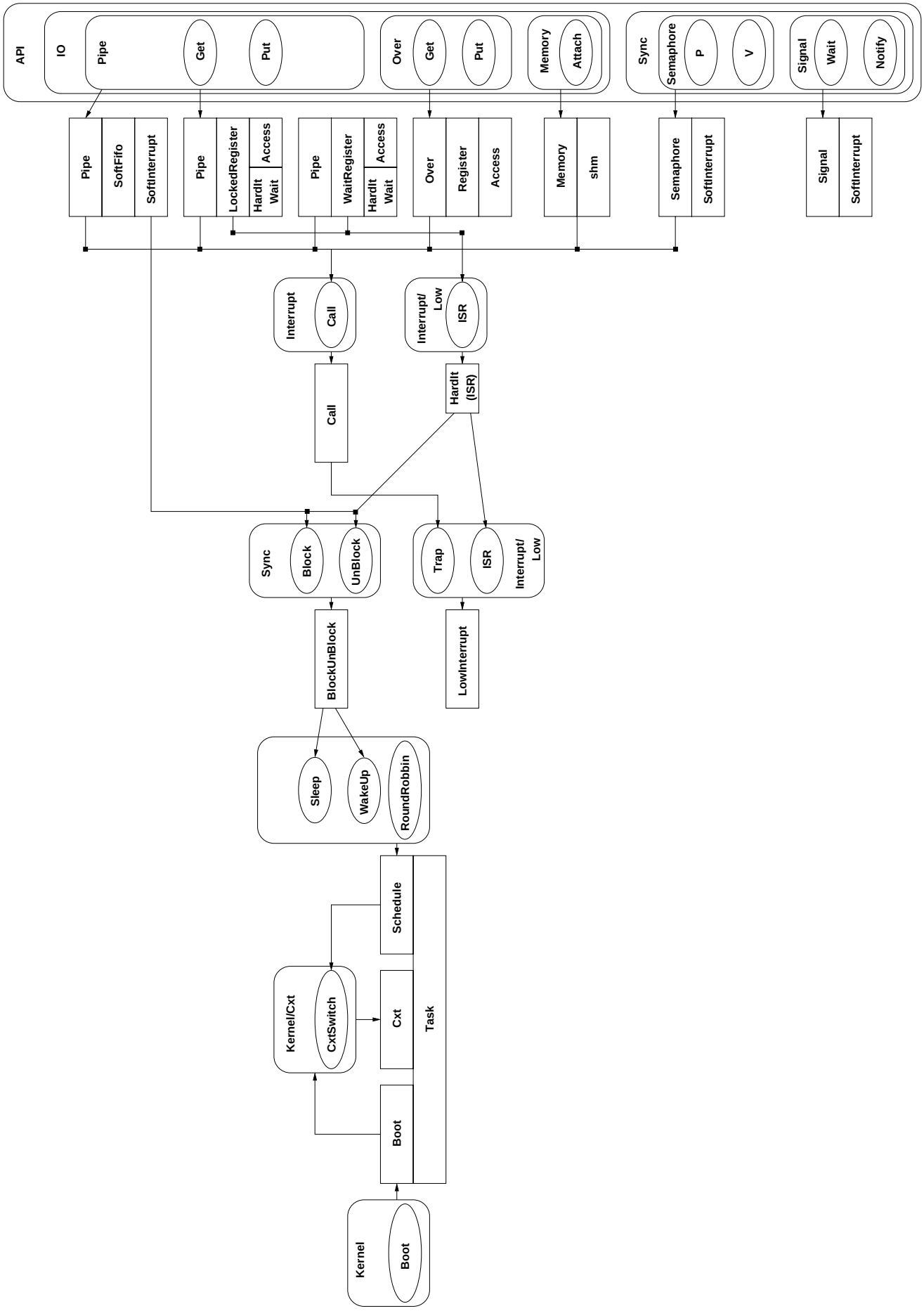

FIG. 5.8 – *Le système d'exploitation généré pour le module M2*

Extension de la bibliothèque

L'extension de la bibliothèque peut se faire suivant deux axes : l'ajout de fonctionnalités ou l'extension de sa compatibilité avec d'autres architectures.

Pour le premier axe, l'essentiel du travail consiste maintenant à ajouter des éléments d'API. Une fois que l'algorithme d'un tel élément est connu, son ajout dans la bibliothèque est très rapide : il suffit de déterminer quels sont les services de bas niveau dont il a besoin[6], puis d'encapsuler son code dans une macro qui permettra son assemblage et son adaptation lors de la génération. Les macros qui encapsulent le code des éléments d'API suivent toutes le même modèle, l'intégration du code d'un nouvel élément d'API est donc très simple. Une fois que l'utilisateur a une bonne expérience de la bibliothèque et du langage de macro, l'intégration d'un élément d'API prend de quelques minutes à quelques heures suivant la complexité de l'élément.

Pour le second axe, si le matériel à ajouter est un périphérique, il suffit d'ajouter l'élément de **Driver** correspondant. Le travail est alors identique à celui de l'ajout d'un élément d'API. Si le matériel est un processeur, il faut repérer tous les éléments dépendant du processeur, et leur ajouter les implémentations qui les rendront compatibles avec ce dernier. Ce travail peut être fastidieux si la bibliothèque est importante : et un outil permettant d'effectuer automatiquement ce type de recherche serait le bien venu.

Validité des éléments de la bibliothèque

Pour le moment, seul le test après génération permet de valider le système d'exploitation, aussi bien du point de vue des fonctionnalités que du point de vue temporel. Il serait intéressant de pouvoir être sûr avant le test que le système généré soit correcte et qu'il puisse respecter les contraintes temporelles.

Pour avoir à l'avance une estimation des propriétés temporelles du système d'exploitation, un flot d'estimation de performance basé sur les résultats obtenus avec des systèmes d'exploitation déjà générés va être étudié au cours des prochains travaux. Le but est d'ajouter dans la bibliothèque des annotations sur les propriétés temporelles de chaque élément suivant les paramètres de génération. Les possibilités d'extension du langage de macro permettront d'inclure les algorithmes d'estimation directement dans le code des macros sans avoir à modifier l'outil de génération. Ainsi, en même temps que le code du système d'exploitation sera généré, des estimations temporelles sur ce dernier seront produites.

S'assurer de la validité fonctionnelle du système d'exploitation généré est un problème complexe à résoudre. L'idéal serait de le prouver formellement. Dans un premier temps, il faudra définir une classification pour les élément et les services. Il faudra ensuite étudier la stabilité des propriétés d'un élément par composition (c'est-à-dire assemblage avec un autre élément) et adaptation.

5.3.1.3 L'outil de génération

Automatisation

Une fois la spécification et la bibliothèque définies, l'outil génère automatiquement les systèmes d'exploitation spécifiques à l'architecture et à l'application logicielle, ainsi que les fichiers permettant leur compilation.

Pour la démonstration, il manquait cependant la génération des fichiers d'adaptation du code des tâches de l'application. Ces fichiers ont donc été écrits à la main. Cela a permis de

6. Il suffit pour cela de s'inspirer des éléments d'API déjà existants.

constater que leur génération automatique est identique à la génération de n'importe quelle autre partie du système d'exploitation (la méthode est décrite dans la section 4.2.7.5).

Déroulement de la génération

La génération des deux systèmes d'exploitation pour l'application VDSL ne prend que quelques dizaines de secondes. Au cours de cette génération, de nombreux messages indiquent l'évolution des opérations, ainsi que les problèmes rencontrés. Le compte rendu est suffisamment précis : dans le cadre de la démonstration, toutes les erreurs de la spécification (aussi bien au niveau de l'architecture, qu'au niveau des paramètres) ont étés détectées au cours de la génération :

- Les erreurs au niveau de l'architecture sont détectées au moment de son l'analyse.
- Les erreurs au niveau des paramètres sont détectées à la génération du code par les algorithmes de traitement des paramètres inclus dans les macros.

5.3.1.4 Compilation du système d'exploitation généré

La compilation des systèmes d'exploitation générés conjointement avec les tâches de l'application a permis de découvrir des difficultés qui n'avaient pas été prévues initialement.

La première de cette difficulté, est que le compilateur utilisé pour le processeur ARM7 n'était pas totalement compatible avec la norme ANSI. Ce problème a été surmonté en modifiant quelques macros pour passer outre. Cela met en avant le problème du langage : il est important que le langage choisi soit vraiment portable sur tout processeur pour être utilisé. Dans le cas du langage C, la norme ANSI permet d'assurer cette portabilité.

Le deuxième difficulté est intervenue à l'édition de liens entre le système d'exploitation et les tâches. Nous rappelons que les tâches étaient décrites en C++. C'est un langage orienté objet, qui permet notamment de surcharger les méthodes (fonctions). À la compilation cette surcharge est traitée en renommant les méthodes pour que chacune méthode ait un nom unique. La difficulté est qu'il n'y a pas de convention pour ce rennomage : suivant les compilateurs ces méthodes sont renommées différemment. Dès lors, il devient difficile de faire référence à une tâche décrite en C++ à partir d'un autre langage ou d'un autre compilateur. Pour résoudre ce problème, un élément spécial de la bibliothèque ayant pour but de calculer le véritable nom de chaque tâche (c'est-à-dire après renommage) a été défini.

Remarque : la solution qui consisterait à décrire toutes les tâches et tous les élément du système d'exploitation dans le même langage est inapplicable[7] : par exemple, certaines parties comme le changement de contexte doivent être décrite en langage d'assemblage.

5.3.1.5 Validation du résultat

Nous avons vu que le système d'exploitation généré ne peut être validé que par test. Cependant, comme il est spécifique à l'application et à l'architecture, sa validation complète ne peut être effectuée que conjointement avec toute l'application.

Validation par cosimulation

Le système de cosimulation décrit dans la section 1.15 est utilisé pour valider l'ensemble de l'application. Pour ce faire les systèmes générés sont compilés avec les tâches et exécutés sur des *ISS (Instruction Set Simulator)* des processeurs ARM7. Ces simulateurs sont intégrés dans le banc de cosimulation. Cette cosimulation peut prendre beaucoup de temps car il y a de nombreux éléments à simuler.

7. Elle manque de toute façon de souplesse.

Une cosimulation plus rapide peut être obtenu en élevant le niveau d'abstraction des modèles simulés, et en exécutant nativement le code des tâches et du système d'exploitation (c'est-à-dire directement sur la station).

Le pseudo-processeur UNIX

Pour exécuter nativement le code des tâches et du système d'exploitation tout en conservant l'environnement logiciel de la station (système UNIX), le pseudoprocesseur UNIX a été défini et intégré dans la bibliothèque de génération de systèmes d'exploitation. Il s'agit en fait d'un processus classique, qui simule les interruptions du processeur par des signaux, les ports du processeur par des tubes, et les changement de contexte entre les tâches grâce aux instruction **sigsetjmp** et **siglongjmp**[8]. Ainsi, tout le code est exécuté nativement, ce qui permet d'obtenir de bonnes performances de simulation tout en utilisant les algorithmes du systèmes d'exploitation générés.

5.3.2 Résultats concernant les systèmes d'exploitation générés

5.3.2.1 Le code généré

Taille du code généré

Deux systèmes d'exploitation ont été générés pour les deux processeurs.

Pour le premier processeur (le module M1), 968 lignes de code C ont été générées, ainsi que 281 lignes de code en langage d'assemblage. Après compilation, le système d'exploitation utilise 1484 octets pour le code et 500 octets pour les données.

Pour le deuxième processeur (le module M2), 1872 lignes de code C ont été générées, ainsi que 281 lignes de code en langage d'assemblage. Après compilation, le système d'exploitation utilise 2624 octets pour le code et 1020 octets pour les données.

Comparé à des systèmes d'exploitation classiques, ou même à des systèmes d'exploitation configurables, le résultat est très bon. En effet, en moyenne la taille minimale d'un système d'exploitation embarquée est de 4k octets ; **eCos** [90] permet d'obtenir de meilleurs résultats puisque sa taille minimale est d'un peu moins de 1k octets. Cependant, nous parlons ici de taille minimale, les systèmes d'exploitation générés pour l'application fournissent déjà de nombreuses fonctionnalités.

Qualité du code généré

Le code généré peut être de qualité équivalente à celle d'un code écrit à la main. La figure 5.9 en donne un échantillon (il s'agit d'une partie du code de l'ordonnanceur). Comme le montre la figure, il est possible d'avoir un code aéré et disposant de commentaires. En effet la génération à partir de macros, permet de donner facilement n'importe quel style dans l'écriture du code et aussi de générer les commentaires.

Un avantage de l'utilisation du langage de macro complet par rapport à celle du préprocesseur du langage C, est que le code accessible par l'utilisateur n'est pas rendu confus par les directives de précompilation nécessaires à l'adaptation de code[9].

8. Ces fonctions permettent respectivement de sauvegarder et de restaurer l'état du processus.

9. Dans le cas de l'utilisation du préprocesseur, plus la flexibilité est grande, plus il y aura de directives, et plus le code sera confus.

5.3.2.2 Qualité du système d'exploitation généré

Déterminisme

Pour qu'un système d'exploitation puisse supporter des applications temps-réel, une propriété importante est qu'il soit déterministe. Pour favoriser ce déterminisme, le code des systèmes d'exploitation a été écrit de telles manières qu'il n'y ait pas de boucle dépendants des données. Dès lors, les temps d'exécutions de chaque partie des systèmes générés sont bornés de bornes connues.

Performances

Les performances d'un système d'exploitation sont en général estimées à partir du temps de changement de contexte et du temps de latence pour une interruption. D'autres temps peuvent être intéressants comme la latence d'un appel système, ou le temps de retour du système d'exploitation vers une tâche. Nous allons donner ces valeurs pour le processeur ARM7 cadencé à 25MHz.

Pour les systèmes d'exploitation générés, le changement de contexte prend 36 cycles, c'est-à-dire 1,44μs. La latence pour un interruption matérielle est de 59 cycles c'est à dire 2,36μs. Il faut ajouter à ce temps de latence celui mis par le processeur lui-même pour traiter l'interruption qui est de 4 à 28 cycles. La latence pour un appel système est de 50 cycles c'est-à-dire 2,00μs. Enfin le temps de retour du système d'exploitation à une tâche est de 26 cycles c'est-à-dire 1,04μs.

Ces résultats sont bons comparés à ceux des systèmes d'exploitation commerciaux pour des processeurs équivalents à l'ARM7 : en effet leur latence pour les interruptions (caractéristique la plus souvent donnée) ne descend jamais au dessous de 2μs quel que soit le processeur, et tourne en moyenne autour de 5μs.

5.4 Conclusion

Le flot de ciblage a été appliqué avec succès sur une application VDSL. Cette application non triviale a en fait été traitée dans le flot global développé au sein de l'équipe SLS, le flot de ciblage en étant la phase ultime pour ce qui concerne le logiciel.

Pour appliquer le flot, une bibliothèque de système d'exploitation a été développée. Cette dernière fournit déjà une grande variété de communication. Elle ne possède que peu d'éléments vraiment spécifiques au processeur, ce qui veut dire qu'elle est aisément portable vers d'autres processeurs que l'ARM7 utilisé pour la démonstration. Un simulateur de processeur effectuant une exécution native a par exemple été intégré dans la bibliothèque pour faciliter son débogage ainsi que celui de l'application.

Après utilisation, le flot s'avère rapide, et il simplifie le travail du concepteur en faisant abstraction des détails concernant le système d'exploitation. Cependant, une étape de synthèse et d'allocation préalable serait nécessaire pour s'abstraire complètement des détails de l'architecture tels que les adresses mémoire. Les systèmes d'exploitation générés sont quant à eux de très petite taille comparés aux systèmes d'exploitations embarqués commerciaux. Ils présentent aussi de bonnes performances. Un autre point positif est que les sources des systèmes d'exploitation générés sont aussi commentées et claires que si elles avaient été écrites directement à la main (c'est-à-dire sans génération, mais aussi sans les directives de configurations qui se trouvent dans le code des systèmes d'exploitation configurables). Nous pouvons remarquer à ce sujet que la qualité du système d'exploitation généré dépend fortement de la qualité de la bibliothèque.

```
/*##############################################################################*/
/*##      The scheduler : contain all the scheduling functions              ##*/
/*##############################################################################*/
 /* Services Headers */
#include "Kernel/Cxt/unixcxt.h"
#include "Kernel/Task/task.h"
#include "Kernel/IT/High/ithigh.h"

#include "schedule.h"


/*##############################################################################*/
/*##                              The tasks states                            ##*/
/*##############################################################################*/
#define READY   1
#define SLEEP   2
#define RUNNING 3

/*##############################################################################*/
/*##                              Global variables                            ##*/
/*##############################################################################*/
int cur_tid,new_tid;     /* current and new task id */


/*##############################################################################*/
/*##    Scheduling functions with :                                           ##*/
/*##                                                                          ##*/
/*##    - round-robbin                                                        ##*/
/*##############################################################################*/

/* Main scheduling function */
void __sched_schedule(void)
{
        /* Update running task id */
        int old_tid=cur_tid;
        cur_tid=new_tid;
        /* Context switching call */
        __cxt_switch(__task_tasks[old_tid].cxt,__task_tasks[cur_tid].cxt);
}


/* Round-robbin function */
void __sched_roundrobbin(void)
{
        cur_tid=__task_tasks[cur_tid].sched.next;
}
```

FIG. 5.9 – *Exemple de code généré*

Conclusion

Les systèmes embarqués spécifiques : évolution vers la complexité

Les systèmes embarqués sont les parties électroniques qui prennent place progressivement les objets usuels allant des téléphones aux automobiles.

Récemment la demande pour les systèmes embarqués et le nombre de fonctionnalités souhaitées s'est fortement accrue tandis que les délais de conception requis diminuent. Des architectures multiprocesseurs hétérogènes semblent devenir la clé pour que les systèmes embarqués puissent supporter cette complexité. En parallèle, l'intégration a fait de grands progrès : il est maintenant possible d'intégrer sur une même puce plus de 100 millions de transistors. Il est dès lors possible d'intégrer complètement un système sur une seule puce. Cependant, les concepteurs n'arrivent plus à concevoir de tels circuits dans des délais raisonnables : ils manquent de méthodologies et d'outils (voir le premier chapitre).

Objectif du projet de l'équipe SLS

L'objectif est de fournir les méthodologies et les outils qui faciliteront et accéléreront la conception des systèmes monopuces. Pour ce faire, un flot de conception descendant est proposé. Ce flot part d'une spécification de haut niveau du système à générer, et grâce à des outils automatiques de raffinement, se propose de générer le code de bas niveau correspondant. Ce flot offre aussi la possibilité de simuler le système au cours des diverses étapes de raffinement. Il met l'accent sur les architectures multiprocesseurs, ainsi que sur les communications.

Quoique encore incomplet, ce flot présenté au premier chapitre propose déjà quelques outils automatiques de raffinement, dont un outil de ciblage avec génération de systèmes d'exploitation, qui est le sujet de ce mémoire.

Le ciblage avec génération de systèmes d'exploitation

L'outil présenté au quatrième chapitre se propose de générer des systèmes d'exploitation spécifiques à une architecture et à une application logicielle. Il prend en entrée une spécification de l'architecture et de l'application, et produit en sortie le code des systèmes d'exploitation spécifique à l'application logicielle et à l'architecture en assemblant des parties de code contenues dans une bibliothèque.

La bibliothèque de système d'exploitation spécifique

Cette bibliothèque est présenté dans la section 4.1.3. Elle est le coeur de la méthodologie de génération de systèmes d'exploitation. Elle est constituée de deux parties : une partie donnant la description des divers éléments, et une partie contenant leur code.

La première partie contient de nombreux objets dont les principaux sont :

- l'élément, qui représente une partie du système d'exploitation
- le service, qui représente une fonctionnalité du système d'exploitation
- l'implémentation, qui représente une réalisation d'un élément

Un élément peut fournir et requérir des services, et possède une ou plusieurs implémentations organisées selon un arbre, les feuilles étant spécifiques aux architectures.

La deuxième partie contient le code correspondant aux implémentations sous forme de macros. Celles-ci, écrites dans un langage de macro complet, encapsulent le code final du système d'exploitation, et permettent son adaptation à l'architecture, ainsi que l'assemblage des divers éléments.

L'outil de ciblage avec génération de système d'exploitation spécifique

L'outil est présenté dans la section 4.2. Il fonctionne en plusieurs étapes :

- La première effectue l'analyse de la spécification d'entrée, en déduit les services de base requis, et les paramètres qui permettront l'adaptation du code des éléments.
- La deuxième effectue la sélection de l'ensemble des éléments nécessaires et suffisants pour l'application, grâce aux services de base, et aux informations sur l'architecture.
- La troisième étape effectue la génération de code proprement dite en distribuant les paramètres pour chaque élément et en appelant l'expanseur de macros pour chacun d'entre eux.
- La dernière étape génère les fichiers de compilation qui permettront de produire le binaire final du logiciel complet.

Application

L'outil et la bibliothèque ont été utilisés pour une application VDSL. Cette application est présentée au dernier chapitre. L'objectif était de générer deux systèmes d'exploitation pour deux processeurs ARM7, avec plusieurs protocoles de communication et de synchronisation.

Les systèmes d'exploitation ont été rapidement générés. Ils sont de très petite taille, de plus leurs performances se comparent favorablement à celles obtenues avec des méthodes manuelles à base de systèmes d'exploitation commerciaux.

Travaux futurs

L'application VDSL a permis de voir que l'outil et la méthodologie étaient réalistes. Il reste cependant des améliorations à apporter.

La principale amélioration concerne la validité du système généré : pour l'instant seuls des tests du système généré permettent de savoir s'il est fonctionnellement et temporellement correct. Il est donc important à présent d'étudier des méthodes pour avoir l'assurance que le résultat d'une génération sera correct fonctionnellement, et pour pouvoir estimer ses performances.

Une autre amélioration se situe au niveau de la spécification d'entrée : elle contient encore trop de paramètres pour pouvoir être directement écrite par le concepteur. Une étape de synthèse serait nécessaire pour la produire, à partir d'une spécification plus abstraite encore.

Perspectives

L'idée d'utiliser des architectures multiprocesseurs pour augmenter la puissance de calcul n'est pas nouvelle : elle a fait l'objet de nombreuses recherches depuis les débuts de l'informatique. Elle n'a cependant jamais connu un grand succès, principalement pour trois raisons :

- La communication entre les processeurs est souvent un goulet d'étranglement.
- La programmation de telles architectures est difficile.
- Les progrès en intégrations ont jusqu'à présent fait augmenter la puissance des processeurs exponentiellement, ce qui rend peu intéressant les efforts à fournir pour développer du logiciel pour une architecture multiprocesseur qui serait vite dépassée.

Il peut alors être légitime de se demander pourquoi l'approche multiprocesseur aurait des chances de réussir pour les systèmes embarqués spécifiques alors qu'elle a échoué auparavant. Nous pouvons remarquer que la difficulté pour programmer les architectures multiprocesseurs venait souvent du fait que les concepteurs voulaient exécuter un algorithme (souvent séquentiel) sur l'architecture parallèle. Pour les systèmes embarqués spécifiques l'approche est différente : le but n'est plus de pouvoir résoudre n'importe quel problème informatique, mais plutôt de fournir un certain nombre de fonctionnalités spécifiques faiblement corrélées les unes aux autres. Dans de telles architectures, chaque processeur est dédié à quelques tâches spécifiques, et peut donc être programmé individuellement tant qu'il ne communique pas avec les autres. La programmation multiprocesseur devient alors de la programmation monoprocesseur répétée autant de fois qu'il y a de processeurs.

Il reste cependant la problématique des communications entre les divers processeurs : elle représente à la fois un goulet d'étranglement et une difficulté de programmation du fait de l'hétérogénéité des architectures embarquées. L'aspect hétérogène des systèmes embarqués apportent une solution simple aux problèmes de coût de communication : chaque partie étant spécifique à certaines fonctions, une conception intelligente de l'architecture peut favoriser des communications locales efficaces. L'arrivée des systèmes monopuces va réduire encore davantage ces coûts. La programmabilité de ces communications multiprocesseurs hétérogènes pourra quant à elle être améliorée par des flots de conception centrés sur la communication.

Annexe A

Rive : documentation abrégée

A.1 Introduction

Rive est un programme d'expansion de macros basé sur un langage complet et extensible.

A.2 Première utilisation de Rive

Pour une première utilisation de Rive, nous allons lui faire produire le texte «Ceci est du texte.». Pour cela il suffit d'éditer un fichier, nommé **toto.gen** par exemple, pour y inscrire :

```
Ceci est du texte.
```

Dès lors, la commande «`rive toto.gen toto`» produira le fichier **toto** contenant le texte :

```
Ceci est du texte.
```

C'est-à-dire que le résultat **toto** est identique au fichier d'origine **toto.gen**, ce qui est normal car aucune macro n'était définie.

A.3 Définition d'une macro Rive

Tant qu'aucune macro n'est détectée dans le fichier source, le macro-processeur recopie directement dans le fichier de destination les caractères lus. Une macro est détectée grâce au caractère `@`.

A.3.1 Délimitation d'un macro-programme

Un macro-programme est un programme écrit dans le langage de macro. Il est délimité par les symboles `@{` et `}@`. Par exemple dans le texte suivant, un macro-programme est décrit dans les six dernières lignes :

```
Pour le moment il n'y a pas de macro,
le texte est donc recopié tel quel.
{@
    # Désormais nous sommes dans un
    # macro-programme qui va effectuer
    # une petite addition
    "1+1=" 1+1
@}
```

Le résultat est alors :
```
Pour le moment il n'y a pas de macro,
le texte est donc recopié tel quel.
1+1=2
```

A.3.2 Délimitation d'une macro-variable

Une fois qu'une variable a été définie dans un macro-programme, elle peut être utilisée dans le texte avec le caractère `@` en préfixe. Si la variable est une fonction, il est possible de lui passer des arguments. Par exemple, supposons que la macro-variable **texte** ait été définie auparavant avec la valeur «Ceci est encore du texte.», alors elle pourra être utilisée comme il suit :
```
Une macro :
@texte
```
Le résultat sera alors :
```
Une macro :
Ceci est encore du texte.
```

A.3.3 Commentaires

Les commentaires sont indiqués par le caractère # : lorsque ce caractère est rencontré dans une macro, le reste de la ligne est ignoré.

A.4 Les valeurs Rive

Les valeurs manipulées dans Rive peuvent être des constantes ou des macro-variables. Ces valeurs, faiblement typées, peuvent être interprétées différemment suivant les situations. Il y a trois types de valeurs possibles : le type chaîne de caractères, le type nombre entier relatif, et le type nombre à virgule flottante.

A.4.1 Le type chaîne de caractère

Les constantes chaînes de caractères sont représentées comme des suites de caractères consécutifs délimitées par deux guillemets `"`. Par exemple : `"Ceci est du texte."` est une constante chaîne de caractères.

Lorsque le caractère d'échappement \ est présent dans la chaîne, le caractère qui suit est interprété selon la table suivante :

Séquence	Interprétation
\n	Retour à la ligne
\t	Tabulation
\b	Bip
\<autre>	<autre>

Voici un exemple de constantes chaînes de caractères avec des séquences d'échappement :

```
{@
    "\"Le Corbeau et le Renard\" :\n"
    "\tMaître Corbeau sur un arbre perché,\n"
    "\tTenait en son bec un fromage...\n"
@}
```

Le résultat est alors :

```
"Le Corbeau et le Renard" :
        Maître Corbeau sur un arbre perché,
        Tenait en son bec un fromage...
```

A.4.2 Le type entier

Le type entier décrit des nombres entiers relatifs codés sur 64 bits.

Une constante de ce type est représentée par l'écriture de sa valeur dans la base 2, 8, 10 ou 16 :

- Lorsque la base 2 est utilisée, la valeur doit être préfixée par **0b**. Par exemple `0b1011` sera interprété comme la valeur entière 11.
- Lorsque la base 8 est utilisée, la valeur doit être préfixée par **0o**. Par exemple `0o11` sera interprété comme la valeur entière 9.
- Lorsque la base 10 est utilisée, la valeur peut ne pas avoir de préfixe, ou être préfixée par **0d**. Par exemple `0d12` et `12` seront interprétés comme la valeur entière 12.
- Lorsque la base 16 est utilisée, la valeur doit être préfixée par **0x**. Par exemple `0x23` sera interprété comme la valeur entière 35.

A.4.3 Le type flottant

Le type flottant décrit des nombres à virgule flottante codés sur 64 bits.

Une constante de ce type est représentée par l'écriture de sa valeur dans le standard du langage C.

A.5 Les expressions

Une expression Rive est une macro constituée d'opérations arithmétiques et logiques. Les opérandes sont des expressions, des constantes, des variables (voir la section A.6) ou des appels de fonctions (voir la section A.8). Pour effectuer ces opérations, les termes (macros) sont interprétés en nombre entier si aucun n'est un nombre flottant, et en nombres flottants dans le cas contraire.

Les opérations arithmétiques et logiques possibles sont données dans la table suivante :

Operation	Operateur	Exemple	(résultat)
Addition	+	3 + 4	(7)
Soustraction	−	5 − 4	(1)
Multiplication	*	5 * 6	(30)
Quotient euclidien	/	23/4	(5)
Reste Euclidien	%	23%4	(3)
Négation	−	−3	(-3)
Puissance	**	2 * *4	(16)
Logarithme	//	20//10	(1.30...)
Et bit à bit	&	3&2	(2)
Ou bit à bit	|	2|1	(3)
OuX bit à bit	^	3^1	(2)
Non bit à bit	~	~0xFFFFFFFFFFFFFFFF	(0)
Décalage droit	>>	4 >> 2	(1)
Décalage gauche	<<	1 << 2	(4)
Égalitée	==	"Toto"=="Titi"	(0)
Différence	! =	2! = 3	(1)
Supérieur	>	2 > 3	(0)
Inférieur	<	2 < 3	(1)
Supérieur ou égal	>=	4 >= 4	(1)
Inférieur ou égal	>=	4 <= 5	(1)
Et logique	&&	3&&4	(1)
Ou logique	||	3||4	(1)
OuX logique	^^	3^^4	(0)
Non logique	!	!3	(0)
Implication	=>	(1 == 2) => (0 == 3)	(1)

Les priorités entre les opérateurs sont par ordre croissant :

`(==,!=,>,<,<=,>=,=>) (<<,>>) (+,-) (*,/,%) (&,|,^,~,&&,||,!,^^) (**,//)`

Pour passer outre ces priorités il est possible d'utiliser des parenthèses, par exemple : $(1 + 2) * 3 = 9$

Remarque : contrairement aux autres opérateurs, l'égalité == effectue une comparaison entre les termes interprétés en chaînes de caractères.

A.6 Les variables

A.6.1 Définition d'une variable

La définition d'une variable consiste à associer une valeur à un identificateur. Dans Rive, les identificateurs sont des chaînes de caractères commençant par une lettre et ne contenant que des lettres, des chiffres ou des caractères _. La syntaxe de définition d'une variable est la suivante :

DEFINE <identificateur> = <macro> **END DEFINE**

Remarques :

- Le mot clé **ENDDEFINE** peut être utilisé à la place des deux mots-clé **END DEFINE**.
- Il est également possible de définir des variables tableaux, et des variables fonctions (voir les section A.7 et A.8).

Dès qu'une variable est définie, elle peut être utilisée dans une expression : elle sera alors remplacée par sa valeur. Par exemple la macro suivante sera expansée en la valeur 10 :
DEFINE cinq=5 **ENDDEFINE**

cinq+cinq

A.6.2 Portée d'une variable

Une variable n'est accessible que dans le bloc où elle a été définie. Les blocs sont implicitement délimités par les instructions bloc de Rive. Ainsi **DEFINE** et **END DEFINE** délimitent un bloc. De même toute instruction Rive se terminant par **END** délimite un bloc.

A.6.3 Définition d'une variable globale

Il est possible de définir une variable globale (visible quel que soit le bloc) grâce au mot clé **GLDEFINE**[1] qui fonctionne comme **DEFINE**.

A.6.4 Redéfinition d'une variable

Il est possible de changer la valeur d'une variable grâce au mot clé **REDEFINE** qui fonctionne comme **DEFINE**[1].

A.6.5 Annulation d'une variable

Il est possible d'annuler la définition d'une variable grâce à la construction suivante :
UNDEFINE <identificateur> **ENDDEFINE**

A.7 Les tableaux

Un tableau Rive est une liste de macros (valeur, tableau ou fonction). Il n'y a aucune restriction quant au contenu d'un tableau : des valeurs de types différents peuvent cohabiter dans un même tableau, ainsi que d'autres tableaux ou des fonctions.

A.7.1 Définition d'une variable tableau

Une variable tableau se définie avec **DEFINE**, **REDEFINE** ou **GLDEFINE** comme il suit :
DEFINE [] <identificateur> = <construction de tableau> **END DEFINE**

A.7.2 Construction de tableau

Un tableau peut être construit grâce à la liste de ses éléments, par concaténation de deux tableaux ou par extraction de parties d'un tableau.

La liste des éléments d'un tableau se déclare comme il suit :

1. Notamment la fin de définition est indiquée par **ENDDEFINE** ou **END DEFINE**.

```
DEFINE [] <identificateur> =
    [<premier élément>,<deuxième élément>,<...>]
END DEFINE
```

Voici par exemple la déclaration du tableau **tab1** contenant les valeurs 1, 2, 3 et 4 :

```
DEFINE [] tab1 = [1,2,3,4] END DEFINE
```

La concaténation de deux tableaux se déclare comme il suit :

```
DEFINE [] <identificateur> =
    <premier tableau>.<deuxième tableau>
END DEFINE
```

Voici par exemple la déclaration du tableau **tab2** contenant les valeurs 1, 2, 3 , 4, 5 et 6, grâce à une concaténation :

```
DEFINE [] tab2 = tab1.[5,6] END DEFINE
```

L'extraction de parties d'un tableau se déclare comme il suit :

```
DEFINE [] <identificateur> =
    <tableau>:(<premier indice>..<dernier indice> ou <indice>,<...>)
END DEFINE
```

Voici par exemple la déclaration du tableau **tab3** contenant les valeurs 2, 3, 4 et 6 provenant de tab2 :

```
DEFINE [] tab3 = tab2:(1..3,5) END DEFINE
```

Remarques :

- Le premier indice d'un tableau est 0.
- Il est possible de combiner toutes ces méthodes de construction de tableaux.

A.7.3 Accès aux tableaux

Pour accéder à un élément d'un tableau il suffit de donner l'identificateur du tableau suivi, entre crochets, de l'indice de l'élément. Par exemple, la macro suivante sera expansée en la valeur 12 (**tab3** est le tableau défini dans la section précédente) :

```
tab3[3]*2
```

Il est également possible de changer la valeur d'un élément d'un tableau à l'aide de **REDEFINE**. Par exemple pour remplacer le premier élément de **tab3** par la chaîne de caractères "Deux", la macro suivant pourra être utilisée :

```
REDEFINE tab3[0] = "Deux" ENDDEFINE
```

A.8 Les fonctions

Une fonction Rive est une macro qui n'est pas évaluée à sa définition, mais seulement lorsqu'elle est appelée.

A.8.1 Définition d'une fonction

Une variable fonction se définie avec **DEFINE**, **REDEFINE** ou **GLDEFINE** comme il suit :

DEFINE {<liste de paramètres>} <identificateur> = <macro> **END DEFINE**

La liste de paramètres est une liste d'identificateurs qui peuvent être utilisés dans la macro de la fonction, en plus des identificateurs globaux.

Par exemple la macro suivante définit la fonction **somme3** effectuant la somme entre trois paramètres :

DEFINE {a,b,c} somme3 = a+b+c **END DEFINE**

A.8.2 Appel d'une fonction

Pour appeler une fonction il suffit de donner l'identificateur de la fonction avec entre accolades les valeurs de ses paramètres. Par exemple la macro suivante utilise la fonction somme3 pour calculer la somme entre 3, 4 et 5 :

somme3{3,4,5}

Remarques :

- Une fonction peut être récursive, c'est-à-dire qu'elle peut s'appeler elle-même.
- Il existe de nombreuses fonctions prédéfinies dans Rive (voir la section A.11).

A.9 Les structures de contrôle

Rive permet l'évaluation conditionnelle de macros ainsi que plusieurs types de boucles.

A.9.1 Définition d'un bloc simple

Il est possible de définir un bloc sans aucune structure de contrôle comme il suit :

DO
    <macro>
END

A.9.2 Évaluation conditionnelle de macros

L'évaluation conditionnelle utilise la structure classique du «si sinon fin si» comme il suit :

IF (<condition>) **DO**
    <macro1>
ELSE
    <macro2>
END IF

La condition est elle même une macro qui sera comparée à 0 : si elle est différente alors c'est <macro1> qui sera évaluée, sinon ce sera <macro2>.

Remarques :

- La condition doit être entre parenthèses.
- La construction **ELSE** est facultative.
- Le mot clé **ENDIF** peut remplacer les mots-clé **END IF**.

A.9.3 Les boucles «tant que» et «jusqu'à»

Ces boucles s'utilisent comme il suit :

WHILE (<condition>) **DO**
    <macro>
END WHILE
UNTIL DO
    <macro>
END UNTIL (<condition>)

Remarques :

- La condition doit être entre parenthèses.
- Si la condition n'atteint jamais 0 pour **WHILE**, ou vaut toujours 0 pour **UNTIL**, alors ces boucles seront infinies.
- Comme précédemment les mots-clé **ENDWHILE** et **ENDUNTIL** peuvent être utilisés.

A.9.4 La boucle «pour»

C'est une boucle complexe : elle permet de répéter l'évaluation d'une macro en faisant varier un indice d'une valeur initiale vers une valeur finale. Cette boucle s'utilise comme il suit :

FOR <identificateur> **FROM** (<macro1>) **TO** (<macro2>) **STEP** (<macro3>) **DO**
    <macro>
END FOR

Remarques :

- macro1, macro2 et macro3 doivent être entre parenthèses.
- Le pas **STEP** est facultatif (par défaut le pas est de 1).
- Comme précédemment le mot clé **ENDFOR** peut être utilisé.
- L'identificateur est automatiquement défini par la boucle **FOR**. Sa portée est limitée à la boucle.

A.9.5 Les choix multiples

Il est possible d'offrir le choix entre plusieurs évaluations suivant la valeur d'une macro. Pour ce faire il faut utiliser la construction suivante :

CASE <macro>
    WHEN <valeur1> **DO** <macro1>
    WHEN <valeur2> **DO** <macro2>
    <...>
    WHEN OTHERS DO <macroN>
END CASE

Le choix **WHEN OTHERS** est le choix par défaut.

Remarques :

- La construction **WHEN OTHERS** est facultative (par défaut la valeur sera nulle).
- Comme précédemment le mot clé **ENDCASE** peut être utilisé.

A.10 Les références

Il est parfois utile dans une macro d'avoir plusieurs identificateurs pour une même variable. De même il peut être intéressant de passer en paramètre d'une fonction non pas une valeur, mais le moyen de traiter directement une variable.

A.10.1 L'opérateur référence

L'opérateur référence empêche l'évaluation d'une variable. Cet opérateur est représenté par la caractère '.

Par exemple pour définir un autre identificateur pour la variable **Toto** (valant la chaîne "Ceci est du texte."), la macro suivante pourra être utilisée :

```
DEFINE Toto2 = 'Toto ENDDEFINE
```

A.10.2 Les paramètres références

Pour indiquer qu'un paramètre d'une fonction doit être considéré comme une référence, il faut également utiliser l'opérateur '. Ainsi dans la fonction **ajoute**, la valeur du premier paramètre est remplacée par sa somme avec celle du deuxième paramètre :

```
DEFINE {'a,b} ajoute =
        REDEFINE a=a+b ENDDEFINE
ENDDEFINE
```

A.11 Les fonctions prédéfinies

A.11.1 Les fonctions trigonométriques

Ces fonctions comprennent les fonctions **SIN**, **COS**, **TAN**, **ASIN**, **ACOS**, **ATAN**.

Remarque : les calculs sont effectués en radian.

A.11.2 Les fonctions d'arrondi

Il est possible d'arrondir un nombre à virgule flottante vers l'entier supérieur, l'entier le plus proche ou l'entier inférieur avec les fonctions : **CEIL**, **ROUND** et **FLOOR**.

A.11.3 Les fonctions d'affichage

Lorsqu'ils sont convertis en chaînes, les nombres entiers sont représentés comme des entiers relatifs en base 10. Il est possible de les afficher en entiers naturels (non signés) en base 2, 8, 10 ou 16 avec les fonctions respectives : **BIN**, **OCT**, **DEC**, **HEX**.

A.11.4 Les fonctions de test

Ces fonctions permettent de tester l'état d'une variable. Elle retourne 1 si le test est réussi, et 0 s'il est un échec.

- **ISDEFINED** vérifie si une variable existe ou non.
- **ISTAB** vérifie si une variable est un tableau ou non.
- **ISFUNC** vérifie si une variable est une fonction ou non.

A.11.5 La fonction de sortie de bloc

La fonction **BREAK** permet de sortir d'un ou plusieurs blocs : la macro passée en argument est évaluée en entier, et sa valeur indique le nombre de blocs qui devront être quittés.

A.11.6 La fonction de mesure

La fonction **SIZEOF** retourne la taille d'une variable : si c'est un tableau, elle retourne le nombre d'éléments, sinon, elle retourne le nombre de caractères de la macro correspondante évaluée en chaîne de caractères.

A.11.7 Les fonctions d'entrée/sortie

A.11.7.1 Ouverture et fermeture d'un fichier

La fonction **OPEN** ouvre le fichier dont le nom est passé en argument. Elle retourne une valeur qui sert d'identificateur pour le fichier ouvert.

La fonction **CLOSE** ferme le fichier dont l'identificateur est passé en argument.

A.11.7.2 Lecture et écriture sur un fichier

La fonction **READ** utilise deux arguments : le premier est l'identificateur du fichier à lire, et le deuxième est le nombre de caractères à lire. Elle retourne, sous la forme d'une chaîne, les caractères lus dans le fichier.

La fonction **WRITE** utilise deux arguments : le premier est l'identificateur du fichier à lire, le deuxième est la valeur à écrire, interprétée comme une chaîne de caractères.

A.11.7.3 Position et déplacement dans un fichier

La fonction **TELL** donne la position courante dans le fichier dont l'identificateur est passé en argument.

La fonction **SEEK** prend deux arguments : le premier est l'identificateur du fichier dans lequel le déplacement doit être effectué. Le deuxième est la position à atteindre en octets.

A.11.7.4 Entrée et sortie standard

La fonction **OUTPUT** écrit sur la sortie standard la valeur passée en argument en l'interprétant en chaîne de caractères.

La fonction **INPUT** lit sur l'entrée standard une chaîne de caractères, et retourne sa valeur.

A.11.7.5 Génération d'erreur

La fonction **ERROR** met le programme d'expansion en état d'erreur. Le message d'erreur est la chaîne de caractère obtenue après réduction de la macro passée en argument de la fonction.

A.11.8 Les fonctions d'évaluation

La fonction **IDF** réduit en chaîne de caractères la macro passée en argument et l'interprète comme un identificateur de variable.

La fonction **EVAL** réduit en chaîne de caractères la macro passée en argument, et l'interprète comme une macro.

La fonction **NAME** retourne l'identificateur de la variable passée en argument.

La fonction **SYSTEM** réduit en chaîne de caractères la macro passée en argument et l'interprète comme une commande shell qu'elle exécute. La sortie de cette exécution est retournée comme résultat de la fonction.

A.11.9 Les fonctions d'extension

La fonction **INCLUDE** charge et évalue le fichier de macro dont le nom est passé en argument.

La fonction **LOAD** charge le module d'extension dont le nom est passé en argument. Elle retourne une valeur servant d'identificateur pour le module chargé. Pour pouvoir utiliser les fonctions de ce module, il faut les associer à des variables. Pour ce faire il faut utiliser la fonction **MAP** : le premier argument est l'identificateur du module contenant la fonction, le deuxième est le nom de la fonction et le troisième est le nom de la variable qui sera utilisée pour appeler la fonction. Une fois qu'une fonction d'un module est associée à une variable, cette dernière s'utilise comme une macro-fonction classique.

Glossaire

– **CAN :** Convertisseur Analogique Numérique.
– **CNA :** Convertisseur Numérique Analogique.
– **SoC :** *System on a Chip* (système sur une puce, ou système monopuce) système complexe de fonctions électroniques intégrées sur une seule puce.
– **ASIC :** *Application Specific Integrated Circuit* (CIAS, Circuits Intégrés à Applications Spécifiques) Circuits intégrés conçus pour un usage particulier selon les exigences d'un client.
– **Barres croisées :** architecture de communication matérielle où tout élément est connecté à chacun des autres.
– **codesign :** «Codéveloppement» technique consistant à développer un système hétérogène (logiciel, électronique, mécanique, etc) dans son ensemble, et non pas séparément pour chaque partie.
– **RTL :** *Register Transfer Level* (transfert de registres), modélisation du comportement d'un circuit où la barrière temporelle est le registre, et où la transition entre deux barrières temporelles représente de la logique combinatoire.
– **ISR :** *Interrupt Service Routine*, fonction appelée automatiquement par le processeur lorsqu'une interruption a eu lieu.
– **Coopératif :** Mode multitâche dans lequel une tâche active cède le contrôle des ressources du système d'exploitation à une autre application active au moment où elle n'a pas besoin de ces ressources.
– **Changement de contexte :** opération qui consiste à passer de l'exécution d'une tâche, à l'exécution d'une autre tâche par un processeur. Le changement de contexte est pris en charge par le système d'exploitation.
– **Préemptif :** Mode multitâche dans lequel les tâches actives utilisent les ressources du système d'exploitation à tour de rôle et pendant un temps déterminé.
– **Temps réel :** se dit d'un système, qui est capable de gérer des contraintes temporelles fortes, c'est-à-dire capable de respecter des délais imposés par l'environnement extérieur.
– **temporisateur :** circuit fournissant une mesure du temps. Il peut être utilisé pour générer périodiquement une interruption.
– **UNIX :** famille de systèmes d'exploitation très utilisés pour des stations de travail.
– **Threads** : (processus allégé) tâches partageant un même espace mémoire.
– **Processus** : programme qui s'exécute. Ce terme est souvent employé pour des tâches fonctionnant sur un système à mémoire virtuelle.
– **MMU :** *Memory Management Unit* (unité de gestion de mémoire), unité d'un processeur qui interprète les adresses mémoire virtuelles du logiciel pour les convertir en adresses physiques. Si une adresse virtuelle donnée ne correspond à aucune adresse physique, alors une interruption est générée, et c'est au système d'exploitation de résoudre le problème.
– **IPC :** *Inter-Process Communication* (communication interprocessus) ensemble de fonctions de communications inter-processus couramment utilisé sous UNIX. Les IPC fournissent des services de mémoire partagée, sémaphores et messagerie.

– **API :** *Application Programming Interface*, c'est l'ensemble des fonctions proposées par un logiciel pour permettre son utilisation par des programmes.
– **FIFO :** *First In First Out*, classe de protocole de communication qui assure que les premières données envoyées sont les premières données reçues.
– **ROM :** *Read Only Memory*, mémoire dans laquelle on ne peut que lire les données
– **DSP :** *Digital Signal Processor* (processeur de traitement du signal numérique), processeurs spécialisés pour les calculs de type traitement du signal. Ils disposent notamment d'une unité de multiplication/accumulation appelée *MAC*.
– **Attente active :** mécanisme de synchronisation, qui consiste à aller vérifier périodiquement l'état de la synchronisation.
– **Plug-in :** (plugiciel, ou module d'extension), logiciel d'application complémentaire qui entre automatiquement en action en présence d'un objet, et ce, sans que l'utilisateur ait à intervenir.
– **Token :** (unité lexicale), élément de langage représentant, par convention, une unité signifiante.
– **Little-endian :** (petit-boutiste) Qualifie un mode de stockage ou de transmission des nombres dans lequel l'octet le moins important apparaît en premier.
– **Big-endian :** (gros-boutiste) Qualifie un mode de stockage et de transmission des nombres dans lequel l'octet le plus important apparaît en premier.
– **DMA :** Direct Memory Access (accès direct à la mémoire), méthode de transfert dans un ordinateur où les données s'échangent entre la mémoire vive et des périphériques sans passer par le processeur.
– **Passe :** en compilation se dit d'une étape traitant complètement un fichier. Très souvent une compilation nécessite plusieurs passes pour être complètement effectuée.
– **Template :** (gabarit) Forme de référence à partir de laquelle sont créés des objets qui présentent des caractéristiques communes.
– **Langage complet :** un langage est dit complet, s'il permet de décrire n'importe quel algorithme. Un moyen classique de montrer qu'un langage est complet est de montrer qu'il est équivalent à une machine de Turing.
– **Préprocesseur :** programme ou partie de programme qui fait subir à des données un traitement préliminaire, afin de les préparer à un traitement subséquent par le processeur ou par un autre programme.
– **ISS :** *Instruction Set Simulator*, simulateur de processeurs reproduisant l'exécution de leurs instructions.

Bibliographie

[1] *The GNU Make Manual.*

[2] DEBIAN GNU/Linux web page. http://www.debian.org.

[3] *GNU implementation of the UNIX macro processor.*

[4] NucleusPLUS. available at http://www.acceleratedtechnology.com.

[5] Synthetix project. available at http://www.cse.ogi.edu/DISC/projects/synthetix/.

[6] *Manuel de référence du langage de programmation Ada.* Alsys, 1987.

[7] A. Aho, R. Sethi, and J. Ullmn. *Compilateurs. Principes, techniques et outils.* Dunod, 2000.

[8] R. Airiau, J.-M Bergé, V. Olive, and J. Rouillard. *VHDL, Langage, Modélisation, Synthèse.* Presses polytechniques et universitaires romandes, 1990.

[9] T. W. Albrecht, J. Notbauer, and S. Rohringer. HW/SW CoVerification Performance Estimation & Benchmark for a 24h Embedded RISC Core Design. *Proc. Design Automation Conf.*, pages 808–811, June 1998.

[10] P. Amblard, J.-C Fernandez, F. Lagnier, and F. Maraninchi. *Architecture logicielles et matérielles*, chapter 12. Dunod, 2000.

[11] P. Amblard, J.-C. Fernandez, F. Lagnier, F. Maraninchi, S. Pascal, and P. Waille. *Architecture logicielles et matérielles*, chapter 5, page 100. , 2000.

[12] ARM Ltd. ARM7 Data Sheet. *available at http://www.arm.com/Documentation/UserMans/PDF/ARM7vC.pdf.*

[13] A. Baghdadi, D.Lyonnard, N.E. Zergainoh, and A.A. Jerraya. An Efficient Architecture Model for Systematic Design of Application-Specific Multiprocessor SoC. In *Proc. Design Automation and Test in Europe*, pages 44–62, March 2001.

[14] A. Baghdadi, N.E. Zergainoh, W. Cesário, T. Roudier, and A.A. Jerraya. Design Space Exploration for Hardware/Software Codesign of Multiprocessor Systems. In *Proc. Eleventh IEEE International Workshop on Rapid System Prototyping*, pages 8–13, June 2000.

[15] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[16] P. Berlioux and Ph. Bizard. *Algorithme – construction, preuve et évaluation des programmes.* , 1983.

[17] G. Berry et al. The ESTEREL Language. *available at http://www-sop.inria.fr/meije/esterel/index.html.*

[18] T. Bolognesi and E. Brinksma. *Introduction to the ISO specification language LOTOS.* ISDN Systems.

[19] G. Booch. *Object-Oriented Design With Applications.* Benjamin/Cummings, 1991.

[20] R. Bosch. *CAN Specification*, 1991.

[21] S. Bourne. *Le système UNIX.* , 1985.

[22] Carnegie Mellon University. The mach project home page. *available at http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html*, 1985.

[23] A. Cataldo. Chip makers sketch plans for 3G cell phones. *EE Times, available at http://www.eetimes.com/story/OEG19990416S0026*, April 1999.

[24] W. Cesario, G. Nicolescu, L. Gauthier, D. Lyonnard, and A. Jerraya. Colif: a Multilevel Design Representation for Application-Specific Multiprocessor System-on-Chip Design. In *Proc. Twelveth IEEE International Workshop on Rapid System Prototyping*, Jun 2001.

[25] M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh, and A. Sangiovanni-Vincentelli. Synthesis of mixed software-hardware implementation from CFSM specifications. pages 209–223, October 1993.

[26] J. Cortadella and al. Task Generation and Compile-Time Scheduling for Mixed Data-control Embedded Software. In *Proc. Design Automation Conf.*, pages 489–494, June 2000.

[27] P. Coste, F. Hessel, Ph. Le Marrec, Z. Sugar, M. Romdhani, R. Suescun, N. Zergainoh, and A. A. Jerraya. Multilanguage Design of Heterogeneous Systems. *CODES Workshop on Hardware/Software Codesign*, May 1999.

[28] Coware, Inc. N2C. available at http://www.coware.com/cowareN2C.html.

[29] K. W. Derr. *Applying OMT: A Practical Step-By-Step Guide to Using the Object Modeling Technique*. SIGS Books & Multimedia.

[30] D. Desmet, D. Verkest, and H. De Man. Operating System based Software Generation for Systems-on-Chip. In *Proc. Design Automation Conf.*, pages 396–401, June 2000.

[31] C. J. DeVane. Efficient Circuit Partitioning to Extend Cycle Simulation Beyond Synchronous Circuits. *Proc. Int. Conf. on Computer Aided Design*, pages 154–161, November 1997.

[32] R. Dick, G. Lakshiminarayana, A. Raghunathan, and N. Jha. Power Analysis of Embedded Operating Systems, pp.312-315. In *Proc. Design Automation Conf.*, June 2000.

[33] R. P. Dick and N. Jha. CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems. *Proc. Int. Conf. on Computer Aided Design*, November 1998.

[34] E. W. Dijkstra. Cooperating sequential processes. *Programming Languages*, pages 43–112, 1967.

[35] A. Dorseuil and P. Pillot. *Le temps réel en mileu industriel.* Bordas, 1991.

[36] A. Dorseuil and P. Pillot. *Le temps réel en milieu industriel*, chapter 2 et 3, pages 46–74. Bordas, 1991.

[37] S. Edwards. Compiling Esterel into Sequential Code. *Proc. Design Automation Conf.*, pages 322–327, June 2000.

[38] Electronic Industries Assn. *Introduction to Edif.* , 1988.

[39] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. Hardware/software partitioning with iterative improvement heuristics. pages 71–76, November 1996.

[40] Embedded Power Corporation. RTXC. available at http://www.embeddedpower.com.

[41] Eonic. Virtuoso. available at http://www.eonic.com.

[42] R. Ernst, J. Henkel, T. Benner, W. Ye, U. Holtmann, D. Herrmann, and M. Trawn. The cosyma environment for hardware/software cosynthesis of small embedded systems. *Microprocessors and Microsystems*, May 1996.

[43] Eyring Corporation. EYRX. available at http://www.jmi.com/eyrx.com.

[44] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1994.

[45] D.D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. *SpecC Specifcation Language and Methodology*. Kluwer Academic Publishers, Mar 2000.

[46] Gallmeister. *POSIX*. O'Reilly, 1994.

[47] L. Gauthier, S. Yoo, and A.A. Jerraya. Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software. In *Proc. Design Automation and Test in Europe*, pages 679–685, March 2001.

[48] P. Graham. *ANSI Common LISP*. Prentice Hall, nov 1995.

[49] T. Grandpierre. *Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés. Partie II: Génération automatique d'exécutif distribué*. PhD thesis, Université de Paris XI, UFR de Sciences, November 2000.

[50] T. Grandpierre, C. Lavarenne, and Y.Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *7th International Workshop on Hardware/Software Co-Design*, May 1999.

[51] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language lustre. In *Proceedings of the IEEE*, volume 79, pages 1305–1320, 1991.

[52] J. Hennessy and D. Patterson. *Architecture des ordinateurs. Une approche quantitative*, chapter 8, pages 649–780. International Thomson Publishing France, 1996.

[53] N. J. Higham and D. J. Higham. *Matlab guide*. Society for Industrial & Applied Mathematics; ISBN: 0898714699, 2000.

[54] C. A. R. Hoare. Communicating sequential processes (csp). *Communications of the ACM*, 21(8), Aug 1978.

[55] J. Hou and W. Wolf. Process partitioning for distributed embedded systems. pages 70–76, 1996.

[56] Integrated systems. pSOSystem (user's manual).

[57] Intel. *8051 Reference manual*.

[58] ITU-T Recommendation Z.100. *Programming languages, SDL methodology guilines, SDL Bibliorgaphy*. 1986.

[59] J. Jeon and K. Choi. Loop pipelining in hardware-software partitioning. *Proc. Asia South Pacific Design Automation Conference*, pages 361–366, February 1998.

[60] A. A. Jerraya. Specification and validation for heterogeneous multiprocessor soc. In *Summer School on Application-Specific Multi-Processor SoC*, Jul 2001.

[61] A.A. Jerraya. La conception des circuits intégrés en pleine révolution. *Electronique*, (112), Mars 2001.

[62] Jmi Software Systems. C EXECUTIVE. available at http://www.jmi.com/cexec.html.

[63] A. Kalavade. *System Level Codesign of Mixed Hardware-Software Systems*. PhD thesis, University of California, Berkeley, CA 94720, September 1995.

[64] I. Karkowski and H. Corporaal. Design Space Exploration Algorithm For Heterogeneous Multi-processor Embedded System Design. *Proc. Design Automation Conf.*, June 1998.

[65] B. Keeth and R. J. Baker. *Dram Circuit Design : A Tutorial*. IEEE, nov 2000.

[66] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[67] K. Kim, Y. Kim, Y. Shin, and K. Choi. An Integrated Hardware-Software Cosimulation Environment with Automated Interface Generation. *Proc. of Seventh IEEE International Workshop on Rapid System Prototyping*, pages 66–71, June 1996.

[68] Sanjaya Klumar. *The Codesign of Embedded Systems : A Unified Hardware Software Representation*. Kluwer Academic Publishers; ISBN: 0792396367, nov 1995.

[69] J.-L. Krivine. *Lambda-calcul, types et modèles*. Masson, 1990.

[70] J. J. Labrosse. *MicroC/OS-II*. Publisher Group West, 1999.

[71] C. Lavarenne and Y. Sorel. Specification, preformance optimization and executive generation for real-time embedded multiprocessor applications with syndex. In *Real-Time Embedded Processing for Space Applications*, 1992.

[72] B. Lee and A. R. Hurson. Dataflow architectures and multithreading. *IEEE Computer*, pages 27–39, August 1994.

[73] E.A. Lee. Embedded software – an agenda for research. *UCB ERL Memorandum M99/63*, dec 1999.

[74] C.L. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery*, 1973.

[75] J.R. Lorch and A.J. Smith. Reducing power consumption by improving processor time management in a single user operating system. In *2nd ACM international conference on mobile computing and networking*, 1996.

[76] LynuxWorks. LynxOS. available at http://www.lynuxworks.com.

[77] D. Lyonnard, S. Yoo, A. Baghdadi, and A.A. Jerraya. Application-Specific Architecture Generation for Heterogeneous Multiprocessor System-on-Chip. In *Proc. Design Automation Conf.*, pages 518–523, June 2001.

[78] D. May et al. *Occam.* , 1982.

[79] S. Meftali, F. Gharsalli, F. Rousseau, and A.A. Jerraya. Automatic code-transformations, and architecture refinement for application-specific multiprocessor socs with shared memory. In *XIth IFIP VLSI-SOC'01*, Dec 2001.

[80] S. Meftali, F. Gharsalli, F. Rousseau, and A.A. Jerraya. An optimal memory allocation for application-specific multiprocessor system-on-chip. In *ISSS*, October 2001.

[81] Micriμm. MicroC/OS-II. available at http://www.ucos-ii.com.

[82] P. R. Moorby and D. E. Thomas. *The Verilog Hardware Description Language*. Hardcover, May 1998.

[83] J. Moscinski and Z. Ogonowski. *Advanced Control With Matlab and Simulink.* , 1995.

[84] J. Murray. *Inside Microsoft Windows Ce*. Microsoft Press;, 1998.

[85] G. Nicolescu, K. Svarstad, O. Meunier, W. Cesário, L. Gauthier, D. Lyonnard, S. Yoo, P. Coste, and A.A. Jerraya. Desiderata d'un langage de spécification pour la conception des systèmes électroniques : concepts de base et niveaux d'abstraction. *TSI*, 2001.

[86] G. Nicolescu, S. Yoo, and A.A. Jerraya. Mixed-Level Cosimulation for Fine Gradual Refinement of Communication in Soc Design. In *Proc. Design Automation and Test in Europe*, pages 754–759, March 2001.

[87] Object Management Group. Uml. *available at http://www.omg.org/technology/uml/*, 1997.

[88] John Preston and Donna M. Matherly. *Windows 3.1.* , may 1997.

[89] QNX Software Systems. QNX. available at http://www.qnx.com.

[90] Red Hat. eCos. available at http://sources.redhat.com/ecos/.

[91] W. Reisig. *Petri Nets: An introduction*. Springer-Verlag, 1985.

[92] M. Renaudin. Asynchronous circuits and systems: a promising design alternative. *Journal of microelectronic engineering*, 54:133–149, 2000.

[93] P.-C. Scholl, M.-C. Fauvet, F. Laginer, and F. Maraninchi. *Cours d'informatique : langages et prorgammation*. Masson, 1993.

[94] B. Shirazi, A. Hurson, and K. Kavi. Scheduling and load balancing in parallel and distributed system. *IEEE Computer Society Press*, 1995.

[95] David A. Solomon. *Inside Windows NT (Microsoft Programming Series)*. Microsoft Press, may 1998.

[96] R. Stallman. *The C Preprocessor*. 1991.

[97] W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[98] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.

[99] C. B. Stunkel, D. G. Shea, B. Abali, M. M. Denneau, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, M. Tsao, and P.R. Varker. Architecture and implementation of vulcan. In *8th International parallel Processing Symposium*, pages 268–274, April 1994.

[100] Sun Microsystem. ChorusOS. available at http://www.sun.com.

[101] Sun Microsystems. Java. available at http://www.jaba.sun.com.

[102] Synopsys, Inc. SystemC, Version 1.1. available at http://www.systemc.org/.

[103] Synopsys, Inc. *Synopsys Online Documentation*. Mountain View, CA, USA, 1997.

[104] H. Takada. Itron: A standard real-time kernel specification for small-scale embedded systems. *Real-Time Magazine*, 1997.

[105] A. Tanenbaum. *Systèmes d'exploitation*, chapter 3, pages 84–158. Prentice Hall, 1992.

[106] A. Tanenbaum. *Systèmes d'exploitation*. Prentice Hall, 1994.

[107] A. Tanenbaum. *Systèmes d'exploitation*, chapter 9, pages 407–660. Prentice Hall International, 1994.

[108] R. M. Thomas. *DOS 6.2 Instant Reference*. Sybex, 1993.

[109] P. W. Tuinenga. *SPICE : guide pour l'analyse et la simulation de circuits avec PSpice*. Masson, 1994.

[110] F. Vahid. Modifying Min-Cut for Hardware and Software Functional Partitioning. *CODES Workshop on Hardware/Software Codesign*, pages 43–48, March 1997.

[111] F. Vahid and D. Gajski. Slif: A specification-level intermediate format for system design. pages 116–123, March 1995.

[112] C. A. Valderrama, P. Lemarrec, and A. A. Jerraya. VCI: A VHDL-C Interface Generation Tool for Cosimulation. *Proc. IEEE International High Level Design Validation and Test Workshop*, pages 142–148, November 1997.

[113] S. Vercauteren, B. Lin, and H. De Man. A Strategy for Real-Time Kernel Support in Application-Specific HW/SW Embedded Architectures. *Proc. Design Automation Conf.*, pages 678–683, 1996.

[114] D. Verkest, W. Rosenstiel, I. Bolsens, and H. De Man. CoWare - A Design Environment for Heterogeneous Hardware/Software Systems. *Design Automation for Embedded Systems*, 1(4):357–386, 1996.

[115] W3C Recommendation. XML 1.0 specifcation, 2^{nd} edition. *available at http://www.w3c.org*, oct 2000.

[116] Wind River Systems, Inc. VxWorks 5.4. available at http://www.wrs.com/products/html/vxwks54.html.

[117] Windriver. Tornado. available at http://www.windriver.com.

[118] W. Wolf. *Computers as Components*, chapter 1. Morgan Kaufmann Publishers, 2000.

[119] W. Wolf. *Computers as Components*, chapter 2. Morgan Kaufmann Publishers, 2000.

[120] K. M. Zuberi and K. G. Shin. Emeralds: A small-memory real-time microkernel. In *SOSP99*, 1999.

Résumé

La part du logiciel est de plus en plus importante dans les circuits électroniques spécifiques. Ce logiciel, complexe, doit pouvoir être décrit en faisant abstraction du matériel : il est alors nécessaire de fournir une couche logicielle faisant l'interface entre le logiciel de haut niveau et l'architecture spécifique. Cette étape, appelée «ciblage logiciel» est une étape fastidieuse qu'il serait intéressant d'automatiser.

Ce mémoire propose de réaliser automatiquement cette étape en générant des systèmes d'exploitation spécifiques à l'architecture et à l'application logicielle. L'outil de ciblage présenté prend en entrée une spécification de l'architecture et de l'application, et produit en sortie le code des systèmes d'exploitation spécifiques pour chaque processeur en sélectionnant et assemblant des éléments contenus dans une bibliothèque. La spécification logicielle prend la forme de tâches interconnectées dont le comportement est indépendant de l'architecture : une API (pour «Application Programming Interface» en anglais) est fournie par les systèmes d'exploitation pour réaliser les opérations dépendant de l'architecture telles que les communications.

Cet outil a été utilisé pour une application VDSL. L'objectif était de générer deux systèmes d'exploitation pour deux processeurs ARM7, avec plusieurs protocoles de communication et de synchronisation. Les systèmes générés se sont avérés de très petites tailles, et leurs performances se comparent favorablement à celles des systèmes d'exploitation commerciaux.

Abstract

The software part is becoming more and more important for specific electronic systems. This complex software has to be to developed independently from the hardware: therefore it is necessary to provide a software layer interfacing high level software and specific hardware. This step, called "software targeting", is tedious. Thus, it would be interesting to automate it.

This manuscript proposes to perform automatically this step through the generation of application and architecture specific operating systems. The proposed targeting tool takes as input the architecture and application specification, and generates as output the operating systems source code for each processor through the assembly of library elements. The high level software specification is represented as a set of interconnected tasks Their behavior is independent of the architecture: an API (Application Programming Interface) is provided by the generated operating systems to encapsulate architecture dependent operations like communication ones.

The methodology described in this manuscript has been applied to a VDSL application. The goal was to generate operating systems for two ARM7 processors with many communication and synchronization protocols. The generated operating systems prove to be very small, and as fast as the best commercial operating systems.

ISBN : 2-913329-75-6 ISBNE 2913329-75-6