Modélisation polychrone et évaluation de systèmes temps réel

Abdoulaye Gamatié

N° d'ordre: 2881

THÈSE

présentée

devant l'université de Rennes 1

pour obtenir

le grade de : Docteur de l'université de Rennes 1
Mention Informatique

par

Abdoulaye El Hadji Gamatié

Équipe d'accueil : Espresso
École doctorale : MATISSE
Composante universitaire : IFSIC / IRISA

Titre de la thèse :

Modélisation polychrone et évaluation de systèmes temps réel

Soutenue le 25 Mai 2004 devant la commission d'examen

MM. :	Jean-Pierre	Elloy	Rapporteur
	Patrick	Farail	Examinateur
	Thierry	Gautier	Examinateur
	Paul	Le Guernic	Directeur de thèse
	Michel	Raynal	Président
	Yves	Sorel	Rapporteur

À la mémoire de Boubakar,
À ma mère.

Remerciements

Le travail présenté dans ce document a été réalisé à l'Institut de Recherche en Informatique et Systèmes Aléatoires (Irisa), au sein des équipes Environnement de Programmation pour Applications Temps-Réel (Ep-atr) puis Environnement de Spécification de Programmes Réactifs Synchrones (Espresso).

En premier lieu, je tiens à remercier les membres du jury qui ont accepté de juger mon travail

- Michel Raynal, Professeur à l'Université de Rennes 1, qui a accepté de présider le jury ;

- Jean-Pierre Elloy, Professeur à l'École Centrale de Nantes, et Yves Sorel, Directeur de Recherche à l'Inria, tous deux rapporteurs de ma thèse, qui ont lu et évalué mon travail ;

- Patrick Farail, Responsable du support méthode de développement logiciel embarqué à Airbus France, qui a bien voulu juger mon travail.

J'exprime également toute ma gratitude à Paul Le Guernic, pour m'avoir accueilli dans l'équipe, et m'avoir offert l'opportunité de travailler sur un sujet à la fois intéressant et plein de défis. Je suis très sensible à la confiance qu'il m'a accordée durant ces années pour mener à bien ce travail. Ses conseils m'ont été très précieux. Finalement, ce travail n'aurait sans doute pas été dans sa forme actuelle sans l'aide de Thierry Gautier, au contact de qui j'ai énormément appris. Je lui suis très reconnaissant pour sa grande disponibilité, pour son avis très critique sur ce travail, pour sa patience dans la lecture très attentive de ce document. Il n'a pas cessé de m'encourager durant ces années de thèse. Enfin, j'ai été particulièrement touché par ses qualités humaines.

Je remercie très chaleureusement tous les membres des équipes Ep-atr et Espresso pour l'ambiance agréable et détendue qu'ils ont toujours su faire régner, pour leurs conseils et leurs encouragements. Je remercie en particulier Loïc pour ses inestimables conseils pratiques, et Jean-Pierre pour sa collaboration fructueuse.

Ne pouvant remercier individuellement tous mes collègues et amis, anciens et nouveaux, qui n'ont cessé de me soutenir jusqu'au bout, je voudrais simplement leur adresser mes sincères remerciements.

Je remercie toute ma famille pour sa présence permanente à mes cotés, sans laquelle je n'aurais pas pu terminer cette thèse.

Enfin, il serait injuste de ma part de ne pas exprimer ma gratitude envers tous ceux qui m'ont permis d'arriver à ce stade de mes études : à ma famille, à mes enseignants, au système universitaire français, et à toutes les personnes qui ont cru en moi.

Table des matières

Introduction

Les systèmes réactifs temps réel

Les *systèmes réactifs* [117] [34] sont des systèmes qui interagissent de manière continue avec l'environnement physique auquel ils sont connectés. Ils reçoivent de celui-ci des flots d'informations sur leurs ports d'entrée, effectuent les traitements requis, et produisent des flots d'informations sur leurs ports de sortie. On rencontre de tels systèmes dans des applications comme le traitement du signal, la gestion des réseaux de communication ou le contrôle de processus industriels.

Il convient de distinguer les systèmes *réactifs* des systèmes *interactifs et transformationnels* qui constituent les deux autres familles de systèmes informatiques habituellement mentionnées. Dans le fonctionnement d'un système réactif, c'est l'environnement qui fixe les instants de réaction du système. Cela n'est pas le cas pour un système interactif où les interactions avec l'environnement se déroulent au rythme du système. Les systèmes d'interrogation de bases de données et les systèmes d'exploitation en sont des exemples types. Quant au système transformationnel, il requiert des données d'entrée dès son initialisation et ne fournit des données en sortie qu'après sa terminaison. C'est le cas d'un solveur d'équations ou d'un compilateur, où finalement l'unique contrainte liée au temps est qu'ils doivent se terminer en un temps fini.

Parmi les caractéristiques des systèmes réactifs [109], il convient de souligner les suivantes :

- *La concurrence.* L'évolution du système se fait de façon concurrente avec son environnement. Il est naturel de voir un tel système comme un ensemble de composants parallèles qui coopèrent pour réaliser la tâche assignée au système entier. Ainsi, le système comprend au moins deux types de processus coopérants qui assurent d'une part l'échange des données et d'autre part le traitement de celles-ci.
- *Les contraintes temporelles.* Les systèmes réactifs sont soumis à des contraintes temporelles fortes (temps de réponse borné, fréquence d'échantillonnage d'un signal, etc.), spécifiées lors de la conception. La vérification de celles-ci avant utilisation est nécessaire pour la validation du système. On parle dans ce cas de **systèmes temps réel**.
- *La sûreté.* Du fait de leurs divers rôles **critiques** (par exemple, le contrôle d'une centrale nucléaire ou bien celui du train d'atterrissage d'un avion) impliquant des enjeux importants aussi bien en vies humaines qu'en termes économiques, il est primordial d'éviter tout risque de dysfonctionnement. La sûreté de fonctionnement devient de fait une question essentielle pour ces systèmes. Cela entraîne un besoin de méthodes de développement qui intègrent des outils et techniques adéquats pour la vérification et l'analyse.
- *Le déterminisme.* C'est une propriété fortement souhaitée dans les systèmes temps réel. Elle facilite la prédiction sur les comportements d'un système. Elle permet aussi de reproduire des comportements d'un système dans l'optique, par exemple, de mettre en évidence des erreurs susceptibles de survenir lors d'une exécution de celui-ci.

– *La maintenance.* Ils sont parfois **embarqués** (dits aussi **enfouis** de façon équivalente dans le reste de ce document), posant ainsi quelques difficultés pour les modifier après la réalisation. Dans ce cas, une stratégie prédictive de maintenance, basée sur des modèles *a priori*, peut jouer un rôle important dans le choix d'une structure de système facile à modifier. Les systèmes embarqués sont surtout caractérisés par des contraintes matérielles (ressources limitées en espace mémoire et puissance de calcul).
– *La répartition.* Ce sont souvent des systèmes géographiquement **distribués** pour diverses raisons : la délocalisation des éléments d'un système (capteurs chargés de récupérer les entrées, actionneurs responsables de la production des sorties, processeurs servant de support d'exécution), le gain de performance grâce à l'utilisation de plusieurs calculateurs pour améliorer les temps de réponse des systèmes, la tolérance aux fautes à travers la duplication de certaines parties d'un système.

Au vu de toutes ces caractéristiques, la conception des systèmes réactifs temps réel requiert des méthodologies suffisamment élaborées pour répondre aux exigences d'une mise en œuvre fiable. Pour ces raisons, la prise en compte des techniques formelles dans ces méthodologies apparaît de plus en plus indispensable.

La conception des systèmes temps réel aujourd'hui

Une grande partie des réalisations concernant les systèmes temps réel repose sur les approches classiques de haut niveau. Elles sont d'une part l'*approche asynchrone*, qui définit des systèmes multi-tâche utilisant des *systèmes d'exploitation temps réel* (en anglais, *Real-Time Operating Systems* - RTOS) à l'instar de RT-POSIX et d'autre part, l'utilisation de *langages de programmation concurrente* à l'image de ADA.

Aujourd'hui, nous pouvons clairement observer certaines insuffisances de ces approches si l'on souhaite garantir à la fois la fiabilité requise par les systèmes temps réel et la réduction nécessaire du coût de conception. Une des raisons est liée notamment aux pratiques concernant le développement de ces systèmes [216]. En particulier, l'absence de méthodologies rigoureuses de programmation a longtemps entraîné la présence de *bogues*, souvent difficilement repérables dans le code. Une autre raison provient de la façon dont on procède pour la vérification des comportements du système conçu. Les techniques utilisées reposent souvent sur la simulation et le test. Dans les deux cas, il se pose d'abord la question de l'exhaustivité des situations considérées pour juger de la pertinence des résultats obtenus. De plus, ces derniers dépendent d'une plate-forme donnée ; qu'en serait-il sur d'autres plates-formes ? Le test est en soi coûteux.

Ces insuffisances liées aux approches classiques motivent la définition de nouvelles approches pour résoudre les lacunes des premières. De l'avis de plusieurs experts, tant du monde académique qu'industriel, les *approches basées sur des modèles* représentent une bonne alternative. Parmi elles, nous privilégierons ici l'*approche synchrone*, qui est fondée sur des bases mathématiques solides, permettant ainsi des raisonnements formels et facilitant par la même occasion la validation des mises en œuvre d'un système. De plus, la technologie synchrone offre les outils adéquats au développeur pour mener à bien sa mission (à travers des spécifications formelles, la génération automatique de code ou la vérification). Les *approches par modèles temporisés* proposent, au même titre que l'approche synchrone, des moyens formels pour la conception des systèmes temps réel. Cela est réalisé au travers de langages spécifiques ou bien d'extensions temporisées de concepts comme les automates.

Une autre observation résulte des thèmes d'étude ayant focalisé l'attention des experts dans le domaine des systèmes temps réel. Raj Rajkumar de Carnegie Melon University [1] note que l'étude de l'ordonnancement dans ces systèmes est l'un des sujets les plus traités. Si cela a permis d'établir plusieurs résultats intéressants, il n'en demeure pas moins que c'est au détriment des autres aspects, restés peu explorés comparativement à la problématique de l'ordonnancement. Par exemple, on peut souligner le caractère récent des travaux autour de notions comme la *qualité de service* (en anglais *Quality of Service* - QoS), utile dans le cadre du développement d'applications multimédia temps réel, ou les *intergiciels* qui offrent beaucoup de flexibilité aux architectures de système. On peut aussi mentionner l'importance de plus en plus grandissante de la modélisation à travers l'émergence de concepts tels que MDA (*Model Driven Architecture*) basé sur le *méta-modèle* UML (défini par l'OMG - *Object Management Group*), ou des technologies parmi lesquelles figurent les outils et techniques synchrones. Il y a donc nécessité de définir de nouvelles orientations sur les aspects à explorer dans les systèmes temps réel.

De manière générale, l'étude des propriétés non fonctionnelles fait l'objet de plus en plus d'attention. À ce sujet, un thème comme l'évaluation de performances suivant différentes métriques (temps de réponse, énergie consommée, etc.) doit être mis en avant.

La maîtrise d'une conception fiable des systèmes temps réel passe par la prise en compte de toute la problématique soulevée en partie par les observations ci-dessus. Il faut disposer d'approches à la fois formelles et pratiques qui permettraient d'aborder efficacement les problèmes liés aux propriétés comportementales et non fonctionnelles de ces systèmes, afin d'être capable de garantir la validité de leur mise en œuvre.

Contexte de l'étude et approche proposée

Nous présentons d'abord le contexte de notre travail, ensuite nous donnons l'idée principale de la solution que nous proposons pour répondre aux attentes considérées dans un tel contexte.

Contexte. L'étude menée dans cette thèse se place dans le cadre du projet européen IST-1999-10913 SAFEAIR [2] (Advanced Design Tools for Safety-Critical Systems), qui a débuté en janvier 2000. Ce projet a pris la suite du projet ESPRIT SACRES [3] (Safety-Critical Embedded Systems) qui avait pour mission principale de fournir aux concepteurs de systèmes critiques enfouis, une méthodologie de développement réduisant de façon significative le risque d'erreurs et le temps de conception. Les principaux résultats de ce projet ont été :

- d'une part, la définition d'une plate-forme intégrant des outils liés aux différents langages considérés. Exemple : l'outil MAGNUM STATEMATE de la société *i-Logix* [4] pour le langage STATECHARTS, et SILDEX développé et commercialisé par *TNI-Valiosys* [5] pour SIGNAL.
- d'autre part, le développement d'une approche multiformalisme de conception, où le format commun [6] des langages synchrones, DC+, a servi de pivot pour l'interopérabilité. Cette approche a permis la validation de spécifications de haut niveau à l'aide des outils de vérification formelle accessibles dans la plate-forme. Elle offre aussi la possibilité d'une

1. Communication orale : "*Advances in Large-Scale Distributed Real-time and Embedded Systems*" (IEEE *Real-Time and Embedded Technology and Applications Symposium*, Mai 2003, Washington D.C. - USA).
2. `http://www.safeair.org/.`
3. `http://www.tni.fr/sacres/.`
4. `http://www.ilogix.com/.`
5. `http://www.tni.fr/.`
6. Ce format a été défini précédemment lors du projet EUREKA-SYNCHRON (1994 - 1996).

génération de code distribué automatisée. L'approche a été mise en œuvre également dans POLYCHRONY [7], la version académique de l'environnement de programmation de SIGNAL.

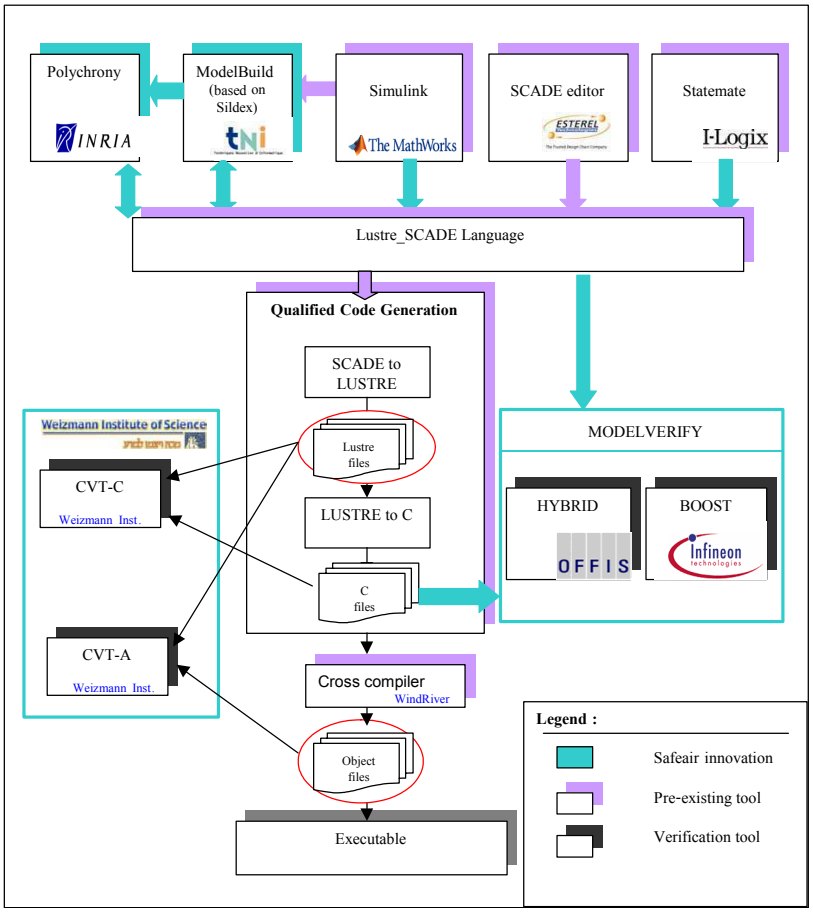


FIGURE 1 – Architecture fonctionnelle de ASDE

Le projet SAFEAIR s'adresse plus spécifiquement aux industries aéronautiques. Celles-ci sont toujours confrontées à une exigence de sécurité maximale, et elles doivent en outre répondre à la forte augmentation actuelle en fonctionnalité et en complexité. Dans ce contexte, l'avionique (ensemble des équipements d'un avion qui fonctionnent avec un matériel informatique ou électronique) représente donc un cas d'application privilégié pour les méthodes formelles en général, et la technologie synchrone en particulier. L'objectif affiché est une réduction de trente-cinq à quarante pour cent de l'effort de développement des systèmes logiciels aéronautiques, tout en augmentant leur fiabilité. Parmi les partenaires du projet, on distingue des acteurs industriels comme Airbus (France), Airbus Deutschland GmbH (Allemagne), IAI (Israël) et Snecma Control Systems (France), qui sont les utilisateurs. Les développeurs d'outils impliqués sont Telelogic Technologies Toulouse et TNI-VALIOSYS (France), OFFIS et SIEMENS (Allemagne). Enfin, les fournisseurs de technologies sont Weizmann Institute of Science (Israël), I-Logix (USA) et INRIA (France).

La démarche dans SAFEAIR a été de s'appuyer sur les résultats obtenus dans SACRES, tout en considérant les technologies du domaine de l'aéronautique, au travers d'outils tels que SIMU-

7. `http://www.irisa.fr/espresso/Polychrony/`.

LINK ou SCADE, et de bibliothèques standards de services temps réel respectant les normes de certification (DO-178B).

La première phase du projet SAFEAIR s'est achevée[8] en juillet 2002. Elle a débouché sur la réalisation d'une plate-forme de développement de systèmes avioniques, appelée ASDE (Avionics Systems Development Environment) et illustrée sur la FIG. 1. Un utilisateur de cet environnement peut construire ses modèles à partir de plusieurs composants, chaque composant ayant été conçu avec l'un des outils intégrés (comme SILDEX ou SCADE). Les communications entre ces composants peuvent être synchrones ou asynchrones. L'outil MODELBUILD (cf. FIG. 1), développé dans l'environnement, est basé sur SILDEX. Il permet de décrire et de simuler l'intégration des composants d'un modèle d'application suivant différents choix architecturaux. L'outil MODELVERIFY permet quant à lui de vérifier des propriétés du modèle global et de ses composants. L'intégration se fait par échange de fichiers SCADE.
L'environnement ASDE permet :

- une intégration des étapes de conception, depuis les outils de modélisation de niveau système jusqu'à une génération de code automatique respectant le standard[9] DO-178B spécifique au domaine aéronautique,
- une réduction significative de l'effort de validation à l'intégration grâce à l'emploi de techniques de vérification formelle,
- une approche permettant de prouver automatiquement la consistance du source et du code généré, ce qui réduit considérablement les tests unitaires.

L'équipe EP-ATR puis ESPRESSO, qui représentait l'INRIA dans SAFEAIR, a été largement impliquée dans la définition de l'architecture ASDE. Elle a fourni des résultats concernant les aspects suivants :

- définition et mise à disposition d'un ensemble de fonctionnalités de transformations correctes de programmes en vue de la vérification et de la génération de code ;
- études d'abstractions de programmes ;
- définition en SIGNAL d'une bibliothèque de composants pour une mise en œuvre temps réel ;
- étude des caractéristiques temporelles d'une application pour une mise en œuvre temps réel.

Enfin, SAFEAIR a servi de cadre de développement et d'expérimentation pour la version "domaine public" de POLYCHRONY.

Cette thèse est particulièrement concernée par les deux derniers aspects cités, étudiés par l'équipe ESPRESSO dans SAFEAIR, à savoir : *i)* la définition d'un ensemble de modèles de composants, que nous intégrons dans le cadre plus large d'une méthodologie de description d'applications temps réel en général, et dans le domaine de l'avionique en particulier ; *ii)* l'analyse des caractéristiques de ces applications en utilisant une technique basée sur des transformations de programmes SIGNAL, donnant des interprétations temporelles des applications spécifiées.

Approche proposée. Nous nous basons sur une méthodologie existante de conception d'applications distribuées en SIGNAL. Celle-ci consiste à obtenir, par des transformations de programmes respectant la sémantique du programme d'origine, un nouveau programme reflétant

8. Une seconde phase du projet, à laquelle l'INRIA ne participe pas, est en cours.
9. DO-178B : "*Software Considerations in Airborne Systems and Equipment Certification*" - `http://www.rtca.org/`.

l'architecture d'implantation considérée. Les communications entre processeurs sont représentées par les communications synchrones (instantanées) ; sur chacun des processeurs, le programme peut être mono ou multi-tâche.

Afin de définir une mise en œuvre effective sur une architecture multi-processeurs générale (contenant au moins des communications asynchrones), il faut d'une part, permettre la désynchronisation des communications et d'autre part, faire éventuellement appel aux primitives d'un système d'exploitation temps réel pour la communication et la gestion des tâches. Globalement, notre démarche est illustrée sur la FIG. 2. Elle repose sur l'utilisation du langage SIGNAL pour modéliser des applications temps réel, partant d'une description initiale (par exemple, les spécifications d'un cahier des charges). Pour cela, nous définissons une bibliothèque de modèles de composants génériques utilisables pour une mise en œuvre temps réel. Ces composants représentent des modèles de tâches, de mécanismes de communication et de synchronisation, et de services d'un système d'exploitation. Ces modèles offrent une représentation indépendante de toute plate-forme d'implantation à travers SIGNAL. Des techniques formelles peuvent alors être utilisées pour étudier les propriétés comportementales et non fonctionnelles des applications considérées.

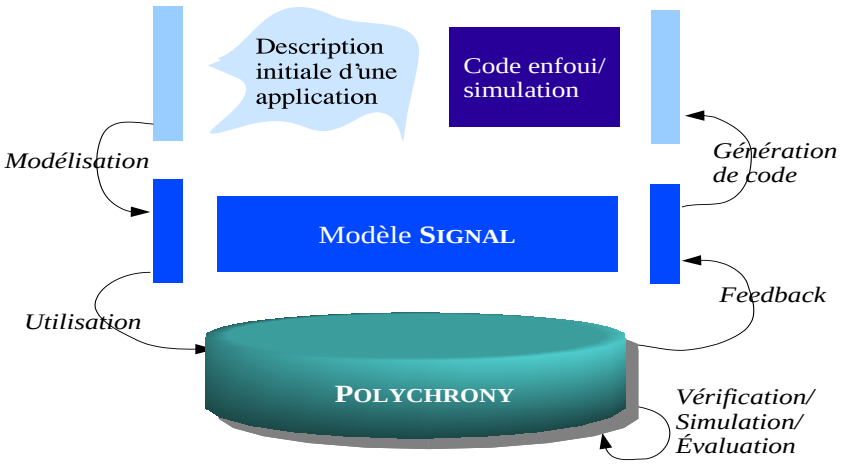


FIGURE 2 – Vue globale de notre approche.

L'environnement POLYCHRONY fournit des outils formels pour la spécification et la vérification de propriétés, la simulation et l'évaluation de performances selon une métrique donnée. Cela permet de remonter des informations utiles concernant les applications décrites (entre autres, les temps d'exécution), pour par exemple, ajuster certains paramètres au sein des modèles, et par la même occasion, modifier les spécifications initiales pour les rendre plus adéquates. Enfin, il est possible de générer automatiquement le code associé aux applications, dans différents langages (exemple : C, JAVA). Ce code peut être soit embarqué soit un code de simulation.

Organisation du document

Ce document comprend trois parties. La première présente des méthodes et outils existants, utilisés dans le développement des systèmes temps réel. La seconde traite en particulier du concept *polychrone*[10] défini autour du langage SIGNAL. Dans cette partie, les deux premiers

10. Du grec "*poly chronos*", pour signifier horloges multiples.

chapitres présentent surtout les acquis. L'apport de cette thèse se trouve en partie dans le chapitre 4, ainsi que dans le chapitre 6, où nous illustrons une approche à la modélisation par composants en SIGNAL. La partie centrale de notre contribution est exposée dans la troisième partie du document (chapitres 8 et 9), où nous montrons comment la technologie polychrone peut aider à décrire des applications dans le domaine de l'avionique, en vue de vérification et de validation.

Partie I. L'objet de cette partie est d'une part, de situer le contexte général par rapport à la conception des systèmes temps réel, et d'autre part, d'introduire des éléments qui seront utilisés dans la suite. Cette première partie débute par une introduction générale aux méthodologies de conception des systèmes temps réel. Dans le chapitre 1, nous présentons différentes approches de conception, nous insistons notamment sur certaines méthodes qui se sont affirmées en ce qui concerne la spécification et la conception des systèmes temps réel. Le chapitre 2 est consacré à la présentation d'éléments de mise en œuvre des systèmes temps réel : de la notion de *tâche* aux techniques d'*ordonnancement*, de concepts plus récents tels que les intergiciels aux langages de programmation. Le chapitre 3 donne une vue d'ensemble de la technologie synchrone sur laquelle repose l'approche proposée dans cette thèse pour la conception des systèmes temps réel : nous commençons par présenter le modèle synchrone, puis différents formalismes synchrones, et les outils et approches associés.

Partie II. La seconde partie est consacrée à la présentation des concepts et outils polychrones. On entend par polychrone, la capacité d'appréhender (descriptions et analyses) des systèmes au sein desquels il peut y avoir plusieurs *horloges* pouvant être indépendantes les unes des autres. Le langage SIGNAL, que nous présentons dans cette partie, adopte le modèle polychrone. De ce point de vue, cela lui confère un pouvoir d'expressivité plus grand par rapport aux autres langages synchrones. Ces derniers considèrent toujours l'existence d'une horloge *a priori*, la plus rapide dans le système, dont toutes les autres dépendent. Le chapitre 4 présente le langage SIGNAL utilisé comme formalisme de base pour décrire des modèles d'applications : tout d'abord, les caractéristiques du langage telles que objets manipulés, constructions, modèle sémantique et propriétés des programmes sont introduites, suivies des principes de transformation de programmes SIGNAL permettant leur analyse. C'est sur ces transformations que se base la technique d'évaluation de performances mise en œuvre dans l'environnement de développement de SIGNAL. Dans ce chapitre, nous contribuons à une vision de la modélisation de comportements "temps réel" dans le modèle sémantique polychrone. Ensuite, nous proposons un support de raisonnement qui vise l'amélioration de l'analyse statique de programmes SIGNAL. Ce support repose sur l'utilisation des intervalles pour approximer les valeurs de signaux numériques. Les chapitres 5 et 6 présentent la conception de systèmes temps réel distribués à l'aide de l'approche polychrone. Nous débutons par la description des applications distribuées temps réel telle qu'elle est mise en œuvre dans l'environnement POLYCHRONY (chapitre 5). C'est une démarche qui repose sur les résultats du projet européen SACRES. Un des objectifs de cette thèse est l'amélioration de cette approche en proposant un ensemble de modèles de mécanismes pour permettre des descriptions plus riches. Pour cela, nous exposons dans le chapitre 6 une approche à la conception par composants, sur laquelle repose en partie la définition d'une bibliothèque de modèles. Ces derniers sont utilisables dans la méthodologie introduite au chapitre 5. La démarche s'appuie sur les propriétés du modèle polychrone de SIGNAL et les transformations associées, notamment la *modularité*, l'*abstraction* et le *raffinement*. Elle est illustrée à travers la modélisation d'une file de messages bornée, gérée en FIFO (*First In First Out*).

Partie III. Dans cette dernière partie, nous considérons un cas représentatif de système temps réel, celui des systèmes avioniques. Nous montrons comment le concept polychrone permet de modéliser des applications de ce type, puis de procéder à l'évaluation de performances à des fins de validation. Dans le chapitre 7, nous revenons d'abord sur quelques aspects spécifiques aux techniques de conception et de réalisation de l'avionique. Ensuite, nous présentons le standard ARINC, plus précisément les spécifications de la série 653. Elles décrivent les entités de base qui constituent une application, ainsi que l'interface dite APEX (*APplication EXecutive*) qui relie la couche applicative au système d'exploitation. Ces spécifications nous servent de point de départ pour la définition de modèles SIGNAL d'applications temps réel embarquées dans l'avionique. Le chapitre 8 présente la réalisation de ces modèles. Il expose de façon détaillée la démarche adoptée pour définir les services de l'interface APEX en SIGNAL. Il présente également la modélisation des autres entités nécessaires pour une description complète des applications. Un cas d'étude est abordé dans le chapitre 9. Nous modélisons un exemple réaliste d'application temps réel en vue d'en évaluer les performances. Nous terminons par une présentation très brève d'une utilisation de ces modèles dans une chaîne de traduction automatique de programmes RT-JAVA en SIGNAL.

D'un point de vue scientifique, cette thèse étudie une solution pour la conception de systèmes comportant des mécanismes asynchrones (comme c'est le cas dans les systèmes temps réel) à l'aide de l'approche synchrone. Elle s'appuie particulièrement sur le modèle polychrone comme support sémantique. Des caractéristiques essentielles d'un tel modèle sont d'une part, la modularité et l'abstraction qu'il favorise dans la description de comportements d'un système, et d'autre part, la rigueur qu'il impose dans l'étude des propriétés du système en vue de la validation. Sur le plan méthodologique, ce document propose une approche de conception basée sur des raffinements à différents niveaux en partant d'une spécification polychrone donnée (représentation fine du support d'exécution d'une application dans le modèle polychrone, liaison à plus haut niveau reposant sur des services d'un exécutif temps réel). D'autre part, cette thèse met en évidence l'apport de la technologie synchrone en général pour la conception des systèmes temps réel. L'environnement POLYCHRONY, qui est utilisé ici, offre un ensemble d'outils et techniques formels permettant de décrire et d'analyser des applications temps réel. Enfin, le travail réalisé a permis de définir une bibliothèque de modèles de composants dont une grande partie repose sur les spécifications de la norme avionique ARINC 653. Ces composants sont utilisables dans POLYCHRONY pour concevoir des applications.

Première partie

Conception des systèmes temps réel

Chapitre 1

Méthodologies de conception des systèmes temps réel

La conception des systèmes temps réel est plus que jamais un véritable défi aujourd'hui, tant ceux-ci ne cessent de devenir à la fois complexes et critiques. En effet, on souhaiterait pouvoir développer des systèmes *sûrs* (garantissant des services avec le minimum de risques d'engendrer des dommages), avec des objectifs de *réutilisabilité* (i.e., réduction du coût global) et de *généralité* (par exemple, UML comme support pour formaliser une démarche de convergence). Ces trois critères résument à eux seuls les préoccupations majeures des différents acteurs, du monde industriel comme du monde académique, impliqués dans le développement des systèmes temps réel.

Plusieurs solutions ont été proposées ces dernières années. Celles-ci ont permis notamment de maîtriser un certain nombre d'aspects du problème. Par exemple, une mise en œuvre répartie en tâches est souvent plus efficace qu'une mise en œuvre séquentielle car elle permet d'effectuer plusieurs traitements à la fois. Cela aide notamment en vue du respect des contraintes liées au temps d'exécution d'un système. D'autre part, des concepts adéquats pour faciliter le travail sont disponibles aujourd'hui (des langages de programmation à l'image de ADA, des *noyaux temps réel* évoqués dans le chapitre 2 tels que *VxWorks*, etc.).

Ce chapitre est essentiellement consacré aux aspects méthodologiques des solutions qui ont été proposées. Des concepts et outils de mise en œuvre seront présentés plus en détail dans le chapitre suivant. Nous commençons par préciser ce qu'on entend par système temps réel (section 1.1.1). Ensuite, la section 1.1.2 donne une classification des différents modèles de temps, habituellement pris en compte dans la conception des systèmes temps réel. Enfin, le reste du chapitre est dédié à la présentation de méthodologies de conception : de l'approche traditionnelle dite cycle en "V" (section 1.2.1) aux approches spécifiques (section 1.2.3), en passant par la modélisation (section 1.2.2). En particulier, nous mettons l'accent sur les approches dites *orientées composants* et les approches *basées sur les méthodes formelles*. La solution qui est proposée dans cette thèse repose sur ces deux catégories d'approches.

1.1 Introduction

Nous précisons d'abord ce qu'est un système temps réel. Ensuite, nous introduisons trois façons d'appréhender le temps suivant les approches à la conception d'un tel système.

1.1.1 Qu'est-ce qu'un système temps réel ?

Il existe plusieurs définitions des systèmes temps réel dans la littérature [196] [82] [139] [55] [206]. Celle qui a été proposée par J. Stankovic est l'une des plus citées [196] :

"A real-time system is defined as a system whose correctness depends not only on the logical results of computations, but also on the time at which the results are produced."

La définition ci-dessus est fonctionnelle, celle que donne J.-P. Elloy [82] est plutôt opérationnelle :

"On qualifie de temps réel une application mettant en œuvre un système informatique dont le comportement est conditionné par l'évolution dynamique de l'état du procédé qui lui est connecté. Ce système informatique est alors chargé de suivre ou de piloter ce procédé en respectant des contraintes temporelles définies dans un cahier des charges de l'application."

Cette dernière définition clarifie mieux le sens du terme "temps réel" en dégageant notamment les rapports système/environnement. On rencontre de tels systèmes dans de nombreux domaines : les télécommunications, le contrôle de centrales nucléaires, le contrôle du trafic aérien, l'avionique, l'automobile, les hôpitaux, les systèmes multimédia, etc. Ces systèmes jouent souvent un rôle *critique* car des vies humaines et des intérêts économiques non négligeables sont mis en jeu. Le standard de certification des systèmes critiques DO-178B définit cinq niveaux de sûreté. Ces derniers décrivent les répercussions d'une panne potentielle du système : A (niveau le plus critique, une panne entraîne des conséquences catastrophiques), B (conséquences sévères), C (conséquences majeures), D (conséquences mineures), et E (conséquences sans effet). Selon le degré de tolérance admis par les systèmes temps réel vis-à-vis du non-respect des contraintes temporelles durant leur fonctionnement, ils peuvent être qualifiés de *stricts* ou de *souples*. Dans les systèmes temps réel stricts (en anglais, *hard real-time*), la violation d'une contrainte entraîne des conséquences catastrophiques. Le contrôle d'une centrale nucléaire ou le système de pilotage d'un avion sont des cas typiques. Les systèmes temps réel souples (en anglais, *soft real-time*) sont plus "tolérants". Ils n'entraînent qu'un rendement de qualité dégradée. C'est le cas par exemple d'un téléviseur, où il est acceptable d'avoir de bonnes images avec un son décalé de quelques millisecondes. On notera qu'il existe aussi des systèmes comportant à la fois des parties strictes et des parties souples. Du fait de toutes ces caractéristiques, le développement des systèmes temps réel requiert des moyens robustes permettant de garantir leur sûreté.

1.1.2 Représentations du temps dans la conception des systèmes temps réel

Selon la représentation choisie pour le temps dans la conception d'un système temps réel, l'analyse des propriétés temporelles peut être plus ou moins facilitée en vue de la validation. Les représentations habituelles du temps dans les systèmes temps réel peuvent être classées en trois types de modèles : le modèle *asynchrone*, le modèle *temporisé* et le modèle *synchrone*. Pour cerner les caractéristiques de chacun d'entre eux, nous considérons la situation schématisée sur la Fig. 1.1, donnée par Christoph M. Kirsch [135].

La Fig. 1.1 illustre les interactions entre un procédé physique $\mathcal{P}$ et un processus logiciel distribué $\mathcal{L}$. Ce dernier reçoit une entrée de $\mathcal{P}$ à l'instant physique τ_1, il effectue ensuite un traitement qui prend un certain temps, et produit une sortie à l'instant τ_2 qui est renvoyée au procédé physique. $\mathcal{L}$ évolue dans un temps qualifié de *temps logique*. Il s'agit d'un temps *discret*. Du point de vue de $\mathcal{P}$, le temps logique ne coïncide avec le temps physique qu'aux seuls instants τ_1 et τ_2. Entre ces deux instants, le temps logique peut être considéré de manière complexe comparativement au temps physique, du point de vue du processus logiciel. Sur l'exemple, après

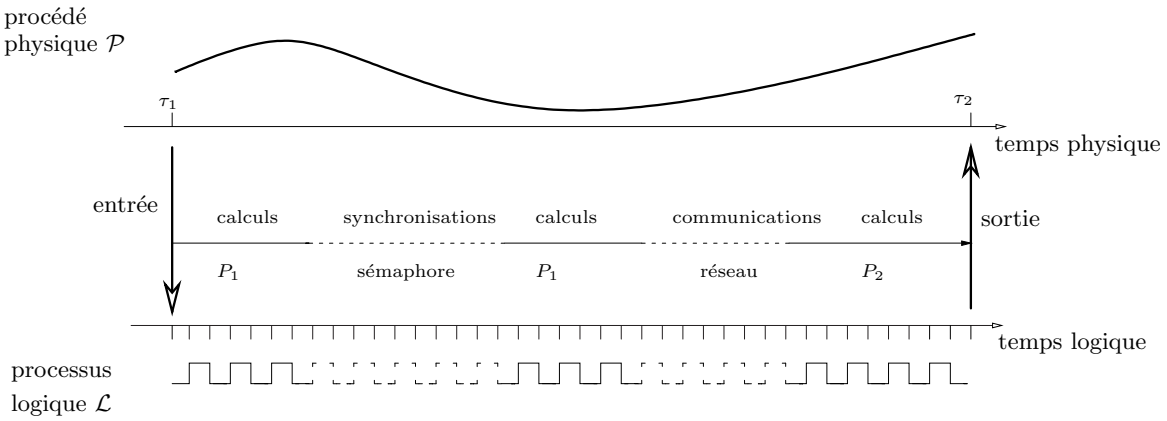


FIGURE 1.1 – Temps physique vs temps logique

s'être exécuté sur un processeur P_1, $\mathcal{L}$ demande à acquérir un sémaphore qui contrôle l'accès à une ressource particulière. Cette acquisition engendre un délai d'attente (qui peut être non borné), d'où une rupture de la continuité du temps logique. Une fois le sémaphore acquis, $\mathcal{L}$ continue son exécution sur le même processeur. Ensuite, il envoie le résultat de son traitement sur un autre processeur P_2, via un réseau de communication qui contribue aussi à la discontinuité du temps logique vu les délais résultants. Enfin, le processus finit son exécution sur P_2 et renvoie une sortie à $\mathcal{P}$.

À travers ce scénario, on observe que la relation entre le temps physique et le temps logique dépend de plusieurs facteurs, dont la performance et l'utilisation du matériel, la stratégie d'ordonnancement, les protocoles de communication et les optimisations des programmes et des compilateurs.

De façon globale, les modèles de temps mentionnés (asynchrone, temporisé et synchrone) se différencient principalement dans leur façon de caractériser le rapport attendu dans un comportement correct entre le temps physique et le temps logique d'exécution.

1.1.2.1 Modèle asynchrone

C'est le modèle "classique" utilisé par les développeurs de systèmes temps réel. Ainsi, un système est représenté par un programme, qui est constitué d'un nombre fini de *tâches* (ou processus). Celles-ci s'exécutent de façon concurrente pour réaliser la fonction affectée au système, sous le contrôle d'un dispositif particulier comme le *système d'exploitation temps réel*, appelé aussi *RTOS* (cf. chapitre 2).

Dans la situation illustrée par la FIG. 1.1, le processus logique $\mathcal{L}$ sera représenté par un ensemble de tâches dont l'exécution dépend de la politique d'ordonnancement adoptée au sein du système d'exploitation. Ce dernier s'appuie en général sur des mécanismes de très bas niveau (comme les interruptions ou les horloges matérielles). Le comportement temporel logique du système devient ainsi fortement influencé par la plate-forme d'exécution. Parmi les facteurs entrant en ligne de compte pour obtenir les temps de calcul et de communication du processus, nous pouvons citer la stratégie d'ordonnancement des tâches qui le composent, mais aussi la performance et le taux d'utilisation du support d'exécution (par exemple, le processeur). En d'autres termes, la durée d'exécution logique du processus est *a priori* inconnue ; cela induit un

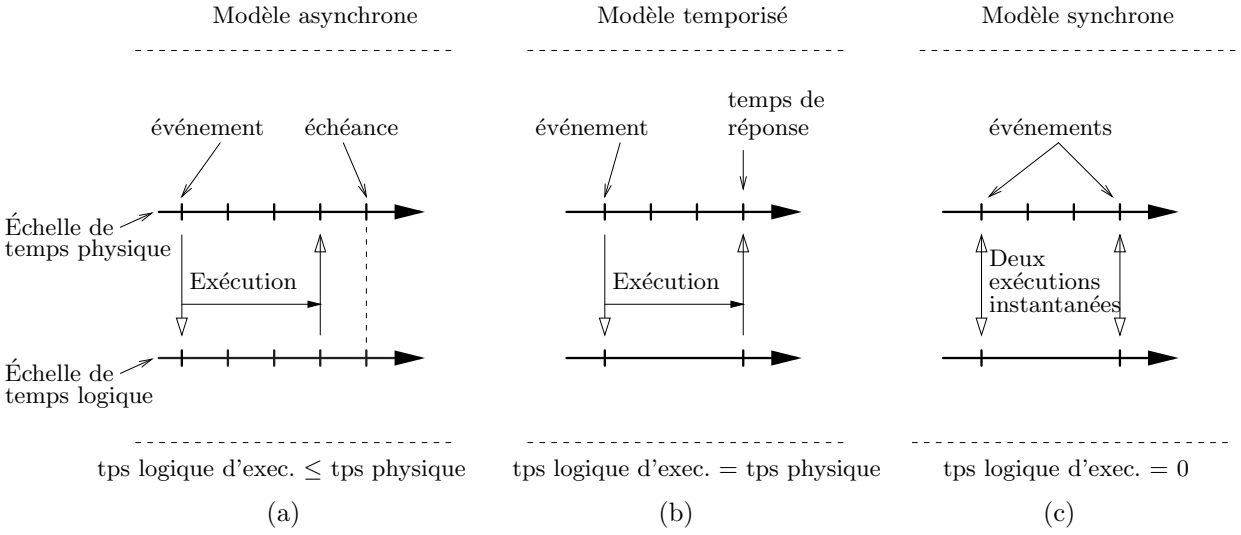


FIGURE 1.2 – Différents modèles d'exécution.

non déterminisme. C'est une caractéristique remarquable du modèle asynchrone qui le distingue des deux autres modèles. En pratique, pour contraindre le comportement temporel logique d'un système, des *échéances physiques* sont considérées. Le dépassement d'une échéance donnée peut être toléré ou non, selon le niveau critique de la tâche concernée. Sur la FIG. 1.2 (a), nous pouvons constater que l'intervalle de temps logique associé à l'exécution du processus est contenu dans celui qui a pour bornes les instants correspondants aux événements de début d'exécution et d'échéance. La programmation de systèmes temps réel à l'aide de langages comme ADA ou C repose sur ce modèle.

1.1.2.2 Modèle temporisé

Le principe de ce modèle consiste à considérer qu'aucun "dialogue" entre le procédé $\mathcal{P}$ et le processus $\mathcal{L}$ sur la FIG. 1.1 n'a lieu entre les instants τ_1 et τ_2. Les "interactions pertinentes" se passent uniquement à τ_1 et τ_2. L'intervalle $]\tau_1, \tau_2[$ est estimé à une quantité de temps logique non nulle et fixe, qui représente la durée logique des calculs et des communications effectués par le processus entre τ_1 et τ_2.

De cette façon, le programme décrivant un système est annoté avec des durées logiques spécifiées par le développeur en se basant sur des critères précis (elles dénotent une approximation des durées réelles des calculs et communications). En général, des valeurs idéales sont choisies, comme l'illustre la FIG. 1.2 (b) : les durées d'exécution logiques et physiques sont considérées comme identiques. L'avantage du modèle temporisé est que la vérification de certaines propriétés liées au comportement temporel logique du système devient possible très tôt dans la conception. Par exemple, la validité d'un ordonnancement peut être vérifiée à la compilation puisqu'on connaît toutes les informations nécessaires avant l'exécution, contrairement au modèle asynchrone. Ce modèle est très approprié pour les systèmes de contrôle embarqués qui nécessitent une prédiction de temps.

La durée d'exécution logique du programme représentant le système étant fixée *a priori* dans le modèle temporisé, deux situations particulières peuvent se présenter quant à son exécution :

i) le programme peut ne pas avoir fini de s'exécuter alors que sa durée d'exécution logique est dépassée ; *ii)* il peut aussi avoir achevé son exécution avant l'expiration du délai logique qui lui a été affecté. Dans le premier cas, une exception est habituellement générée, alors que dans le second cas, c'est la production des sorties qui se voit retardée jusqu'à expiration du délai d'exécution logique. En particulier, cette dernière situation a un grand impact sur la qualité de l'implantation du programme. Plus la durée d'exécution d'un programme est proche de la durée logique considérée, meilleur est le modèle temporisé associé, car les déphasages entre calculs et productions de sorties sont réduits. Une grande attention doit ainsi être apportée lors du choix des valeurs d'annotation des modèles.

Plusieurs formalismes de spécification de systèmes temps réel reposent sur ce modèle. Les automates temporisés en sont un exemple typique, les états sont annotés avec des durées représentant la quantité de temps pendant laquelle le système spécifié peut potentiellement rester dans ces états. En ce qui concerne les langages de programmation adoptant le modèle temporisé, nous pouvons citer *giotto* [120], utilisé dans la conception de systèmes de contrôle embarqués. Nous reviendrons plus en détail sur ce formalisme dans la section 1.2.3.4.

1.1.2.3 Modèle synchrone

L'idée du modèle synchrone est analogue à celle du modèle temporisé, dans le sens où le procédé $\mathcal{P}$ et le processus $\mathcal{L}$ ne communiquent pas entre les instants τ_1 et τ_2 sur la FIG. 1.1. La principale différence est qu'ici, les durées de calculs et de communications sont considérées comme nulles (notion d'instantanéité dans l'exécution d'un système), c'est l'*hypothèse de synchronisme*. En d'autres termes, l'intervalle $]\tau_1, \tau_2[$ est vu comme faisant entièrement partie d'un instant logique du modèle synchrone. En fait l'hypothèse n'est pas aussi "irréaliste" que cela peut paraître au premier abord. Une autre façon de voir les choses consiste à dire que le système est suffisamment rapide pour réagir à des entrées et produire les sorties correspondantes, avant la réception de prochaines entrées. Dans ce cas, la bonne question à se poser est plutôt : l'implantation matérielle du système modélisé est-elle suffisamment performante pour vérifier l'hypothèse ? À ce sujet, l'estimation des temps d'exécution au pire cas dans le modèle synchrone est essentielle pour la validation. Le modèle synchrone offre une représentation déterministe du système. Cela facilite la prédictibilité de celui-ci. Une remarque importante est qu'il est possible, tout en restant dans le modèle synchrone, de raffiner l'intervalle $]\tau_1, \tau_2[$ pour une vision détaillée des comportements : des instants intermédiaires peuvent être considérés comme significatifs du point de vue du temps logique.

Dans le modèle synchrone, ce sont les suites d'événements observés (cf. FIG. 1.2 (c)) pendant les réactions, appelées aussi *signaux*, qui déterminent complètement le référentiel temporel. Ce dernier est constitué d'*instants logiques* pendant lesquels un signal peut être *présent* (il porte alors une certaine information) ou *absent*. ESTEREL, LUSTRE ou SIGNAL sont des exemples emblématiques de langages synchrones [30]. Une présentation détaillée du modèle synchrone est donnée dans le chapitre 3.

* * *

Nous remarquons que le modèle synchrone est une abstraction du modèle temporisé. La principale différence entre ces deux modèles se situe au niveau de la production des résultats : dans le modèle synchrone, elle est instantanée du fait que les temps logiques de calculs et de communications sont abstraits ; dans le modèle temporisé, les résultats sont supposés être produits après une durée non nulle résultant de la propagation des durées des opérations intermédiaires

durant l'exécution d'un système.

À travers cette brève introduction aux différents modèles de représentation du temps dans les systèmes temps réel, nous pouvons constater que selon le choix du modèle, les aspects temporels sont appréhendés de diverses façons. Il est donc important de savoir identifier le modèle adéquat pour aborder efficacement les aspects liés au développement de tels systèmes. Par exemple, l'étude de l'ordonnançabilité [1] d'un système sera plus facile en considérant les modèles temporisés ou synchrones car leur comportement temporel logique est *a priori* connu, quantitativement ou qualitativement.

1.2 Méthodologies de conception de systèmes temps réel

1.2.1 Généralités sur l'activité de conception

Il est courant de distinguer quatre phases principales dans le développement d'un système : *spécification*, *conception*, *implantation* et *tests*. Celles-ci sont souvent organisées selon le fameux cycle en "V" (cf. FIG. 1.3) dont la branche gauche regroupe les étapes de développement, tandis que la branche droite rassemble les phases de vérification et validation. À ces quatre phases, on ajoute une cinquième qui est dédiée à la vérification formelle. De plus en plus, les techniques formelles deviennent incontournables pour la validation d'une mise en œuvre.

D'autre part, cette vision du développement des systèmes est aujourd'hui en train d'être remise en cause avec l'apparition de nouvelles approches telles que l'*ingénierie dirigée par les modèles* (plus connue sous l'appellation anglo-saxonne de *Model Driven Architecture* - MDA), basée sur UML. Par *modèle*, nous sous-entendons une représentation d'une entité, ne prenant en compte que les caractéristiques pertinentes identifiées, en vue d'une étude précise (en d'autres termes, il s'agit d'une abstraction de l'entité). Un *méta-modèle* [2] est quant à lui, un ensemble de concepts et leurs relations à utiliser pour définir des modèles. Ainsi, l'approche MDA envisage le développement plutôt à travers des transformations systématiques de modèles, indépendamment de toute plate-forme. Dans le même temps, l'utilisation des intergiciels (ou *middleware*) devient de plus en plus répandue dans le développement de systèmes à grande échelle du fait de la flexibilité et du confort qu'ils apportent. En effet, l'intergiciel offre aux applications le moyen d'accéder aux ressources gérées par le système d'exploitation via une *abstraction* uniforme, favorisant ainsi l'interopérabilité, la sécurité, etc.

Avant d'aborder ces aspects récents liés au développement des systèmes (cf. section 1.2.3.3 en ce qui concerne UML et MDA, et chapitre 2 pour les intergiciels), nous rappelons d'abord ce en quoi consistent les différentes phases du cycle en "V".

- la *spécification* : elle commence en général par l'établissement d'un cahier des charges qui permet à la fois l'expression et l'analyse des besoins (exemple : services offerts, contraintes). Il est important que le cahier des charges soit compréhensible par tous les acteurs impliqués dans le développement du système. Ainsi, la spécification peut s'appuyer sur des langages informels comme le langage naturel, semi-formels à l'image d'UML, ou bien formels, c'est le cas du formalisme B.
  Un exemple de méthode de spécification et d'analyse très populaire pour les systèmes temps réel est SA-RT (Structured Analysis with Real-Time) [214] [118]. C'est l'extension

1. Par *ordonnançabilité* d'un système temps réel, nous entendons le fait que l'exécution de ce dernier respectera les contraintes temporelles qui lui sont imposées.
2. L'on pourrait faire le parallèle suivant : programmes / langage de programmation *vs* modèles / méta-modèle.

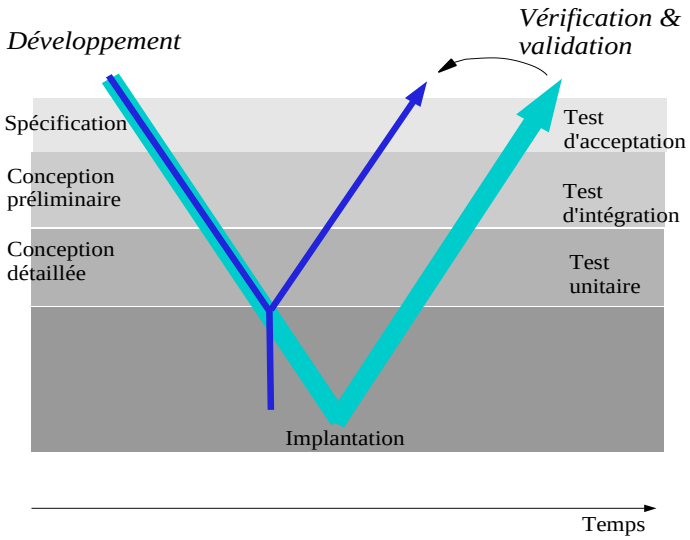


FIGURE 1.3 – Passage du cycle en "V" au cycle en "Y".

temps réel de l'approche SA (Structured Analysis), proposée par de Marco [75]. SA-RT repose sur un langage de spécification à base d'automates qui permet de décrire les aspects fonctionnel, comportemental et traitement des données. Un système est ainsi représenté de façon hiérarchique par un ensemble de diagrammes. Ces derniers décrivent d'une part, les transformations des données, exprimées à travers des "processus de traitement de données" et d'autre part, le séquencement, qui reflète la dynamique du système, en utilisant des "processus de contrôle" spécifiés à l'aide d'automates d'états. Un avantage de la méthode SA-RT est qu'elle facilite l'analyse fonctionnelle d'un système, d'où son succès dans l'industrie. En revanche, elle reste dans le cadre des approches informelles, et cela ne facilite pas la validation.

- la *conception* : celle-ci définit une vue exécutive du système, et aboutit souvent à un ensemble de modules coopératifs qui se partagent des ressources critiques. Sur la FIG. 1.3, cette étape est constituée de deux sous-étapes : la conception *préliminaire* qui permet de décrire l'architecture globale du système (éléments logiciels et matériels, ainsi que leurs interconnexions), et la conception *détaillée* qui donne une vue plus précise de chacun des éléments dégagés lors de la phase préliminaire. Cela facilite leur transformation en programmes exécutables. Dans la suite de ce chapitre, nous donnons quelques méthodes importantes, utilisées lors de la phase de conception.

- l'*implantation* : elle définit la mise en œuvre du programme représentant le système. Elle nécessite donc l'utilisation de langages de programmation, par exemple les langages C ou ADA pour les systèmes temps réel. Par ailleurs, ces systèmes peuvent requérir les services d'un système d'exploitation ou d'un noyau temps réel pour contrôler les *tâches* qui les constituent. À ce propos, l'intergiciel joue le rôle de l'interface via laquelle la couche applicative (spécifique à un domaine) peut accéder de façon sûre et transparente aux fonctionnalités de la couche système. Toutes ces notions seront abordées en détail dans le chapitre suivant.

- les *tests* : ils ont pour but de vérifier le fonctionnement correct du système ainsi développé.

Pour cela, il faut déceler les éventuelles erreurs présentes dans le programme issu de la phase précédente et corriger celles-ci. Les *tests unitaires* vérifient que chaque élément du système global satisfait sa spécification. Les *tests d'intégration* quant à eux doivent essayer de mettre en évidence des erreurs dues à des interactions imprévues entre des éléments du système après l'intégration de ces derniers. D'autre part, les besoins fonctionnels et non fonctionnels exprimés en début de projet sont validés pendant ces tests. Enfin, les *tests d'acceptation*, qui constituent l'ultime phase, permettent de détecter des omissions ou des erreurs dans la définition de besoins : c'est la *validation*.
Pour un système temps réel, les tests comprennent d'une part la vérification de propriétés fonctionnelles (savoir si le comportement attendu est celui qui est observé), et d'autre part des analyses non fonctionnelles, typiquement, l'évaluation de performances selon une métrique donnée (exemple : temps de réponse). Des techniques souvent employées sont l'utilisation de "batteries de tests" ou bien la simulation. Celles-ci ne permettent pas une analyse exhaustive. On est en général obligé de recourir en plus à des techniques un peu plus sophistiquées comme la génération automatique de tests (qui consiste à dériver une suite de tests, à partir des spécifications d'un système ou bien d'un ensemble d'*objectifs de tests* ; les tests sont ensuite utilisés pour une implantation donnée du système afin de déterminer la conformité de celui-ci vis-à-vis des spécifications).

- la *vérification formelle* : elle présente l'avantage de reposer sur des techniques et outils mathématiques qui permettent d'étudier les propriétés de systèmes complexes avec la garantie de raisonnements corrects. Parmi les méthodes existantes pour la vérification formelle, nous distinguons trois grandes catégories : le *model checking* (qui consiste en étude d'accessibilité de certains états d'un système représenté sous forme d'automate à états finis), l'*interprétation abstraite* (basée sur une théorie d'approximation de la sémantique de systèmes dynamiques discrets tels que l'exécution d'un programme) et la preuve interactive (reposant sur des outils qualifiés d'"assistants de preuve"). Alors que la première catégorie de méthodes est adaptée seulement dans le cas de systèmes admettant un modèle fini, les deux autres catégories permettent d'appréhender les propriétés d'un système sans disposer nécessairement d'une représentation finie de celui-ci. La technologie synchrone offre plusieurs outils de vérification basés sur ces méthodes.

Nous remarquerons qu'il est possible (et d'ailleurs souhaitable) que la vérification et la validation interviennent assez tôt dans le développement d'un système. Cela conduit à une optimisation du cycle classique en un cycle en "Y", comme le montre la Fig. 1.3, où le trait vertical représente la génération automatique de code certifié, combinée avec la réutilisation de composants. Le projet SafeAir s'appuie notamment sur une telle démarche. Nous pouvons ainsi noter les gains en temps et en effort qui en résultent.

1.2.2 Importance de la modélisation

Aujourd'hui, il est largement admis que la *modélisation* joue un rôle essentiel dans la conception des systèmes. Elle permet d'une part, de caractériser les propriétés d'un système et d'autre part, de donner une représentation formelle qui décrit une vue opérationnelle de celui-ci, comprenant son organisation globale, ainsi que le comportement des éléments dont il est constitué. L'utilisation du modèle d'un système facilite la prise de décisions très tôt dans sa conception : pour un choix donné d'architecture, on peut simuler et étudier les comportements induits, on peut alors décider de s'en contenter, ou bien de revoir l'organisation du système ; et tout cela se fait relativement vite.

Pour les systèmes temps réel, il est important que les modèles puissent permettre la prise en compte des aspects temporels. Cela dépend souvent du langage utilisé pour décrire les modèles. Par ailleurs, un langage ayant une sémantique formelle solide favorise la validation. Les avantages de la modélisation pourraient se résumer aux points suivants [191] :

- *coût raisonnable* (en général, la réalisation du modèle d'un système est moins coûteuse que sa mise en œuvre effective) ;
- *flexibilité* des modèles (la modification d'un modèle se fait plus facilement) ;
- *richesse d'expressivité* (il est possible de décrire les comportements déterministes d'un système et les comportements non déterministes de son environnement) ;
- *résolution des problèmes liés au facteur échelle* (on dispose de mécanismes adéquats comme la composition ou l'abstraction) ;
- *techniques et outils formels* (lorsqu'il s'agit d'une modélisation formelle) ;
- *prédictibilité* des caractéristiques du système modélisé (exemple : temps d'exécution).

Parmi les outils largement répandus pour la modélisation des systèmes, nous pouvons citer le méta-modèle UML, sur lequel nous reviendrons dans la section 1.2.3.3. Il offre des concepts de modélisation à la fois intéressants et faciles à utiliser. Un exemple de méthodologie de modélisation de systèmes embarqués temps réel basée sur ce langage est ACCORD/UML [99]. Elle permet la prise en compte des contraintes de temps de réponse et de modularité au niveau architecture du système. Les langages synchrones servent aussi à la modélisation, c'est notamment le cas du langage SIGNAL dans cette thèse.

1.2.3 Méthodologies de conception

La conception d'un système requiert habituellement une ou plusieurs *méthodes*, reconnues pour être efficaces pour le type de système en question. Ici, on entend par méthode un ensemble de principes cohérents qui dictent la façon dont on doit s'y prendre pour concevoir le système. Au vu de l'actuelle complexité croissante des systèmes, il est nécessaire de disposer d'un certain nombre de méthodes coopérantes pour adresser de manière appropriée les différentes phases de conception mentionnées ci-dessus. Les *méthodologies* fournissent alors les stratégies d'application des méthodes ainsi retenues.

Parmi les méthodologies existantes qui s'adressent à la conception de systèmes, nous ne citerons que les plus populaires [194]. Nous présentons d'abord l'approche dite en *cascade* (en anglais, *waterfall*), définie par W. Royce en 1970. L'intérêt de ce modèle de conception est ici simplement historique. En effet, c'est le premier modèle qui fut proposé pour répondre aux besoins des concepteurs de logiciels quelconques, suivant une méthodologie bien identifiée. En ce sens, il permet de mesurer combien les pratiques dans ce domaine ont évolué. Nous introduirons ensuite de façon brève deux autres approches, utilisées parfois dans la conception de systèmes temps réel. Il s'agit de la *programmation exploratoire* et du *prototypage*. Enfin, nous nous attarderons particulièrement sur deux familles d'approches : les approches *orientées composants* et les approches *basées sur les méthodes formelles*. La démarche méthodologique que nous proposons s'inspire de ces deux approches.

1.2.3.1 Approche en cascade

L'approche en *cascade* consiste en un schéma de conception simple, constitué des phases du cycle en "V", auxquelles vient s'ajouter une phase de maintenance (cf. FIG. 1.4). Celle-ci est la plus longue dans la mesure où elle concerne l'évolution future du système. Elle comprend par exemple la correction d'erreurs non détectées au cours des phases précédentes (comme les

tests), l'adaptation des fonctionnalités du système pour de nouveaux besoins de l'utilisateur, etc.

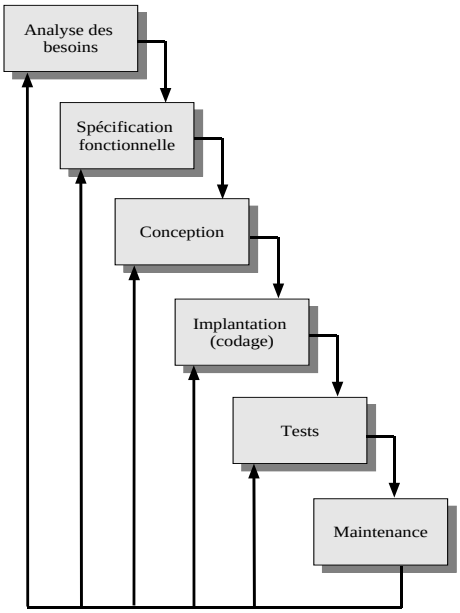


FIGURE 1.4 – Les itérations dans le modèle de la cascade.

Dans cette approche, l'acceptation d'une phase est requise impérativement avant le passage à la phase suivante (exemple : passage entre l'étape de conception et celle d'implantation). Cela veut dire *a priori* qu'il faut être sûr que tous les problèmes liés à la phase courante ont été résolus. Il est recommandé de cerner de manière précise l'ensemble des besoins avant de se lancer dans le processus. Malheureusement, l'expérience montre qu'il est difficile (pour le client) de décrire exactement tous les besoins au tout début de la conception. En fait, dans la pratique, il arrive fréquemment que les phases se chevauchent, entraînant ainsi des itérations pour faire des mises à jour aux niveaux précédents. Par exemple, si une erreur de conception n'est constatée que durant la phase d'implantation, il faut alors revenir aux phases précédentes pour la corriger. Il en résulte donc un risque élevé d'itérations fréquentes. La solution souvent adoptée pour résoudre le problème consiste à écourter des phases (telles que la spécification), pour passer aux suivantes. Cela constitue justement un inconvénient majeur de l'approche : les difficultés susceptibles d'être résolues durant l'étape courante sont repoussées aux étapes d'après (donc, à des échéances ultérieures), voire ignorées. Au final, le système obtenu n'est pas tout à fait conforme aux attentes de départ. Un autre inconvénient de l'approche en cascade est l'impossibilité de disposer d'une version provisoire opérationnelle, avant la fin complète du processus global. Ce qui entraîne un délai de livraison assez long du système à développer.

Malgré toutes ces lacunes, l'approche en cascade reste encore utilisée (notamment pour concevoir des systèmes de petite taille) du fait de sa simplicité dans le déroulement des phases de conception. Un modèle de conception proche du schéma en cascade est le modèle en *spirale*, introduit par B. Boehm en 1988 [44].

1.2.3.2 Approches exploratoires et prototypage

La *programmation exploratoire* et le *prototypage* sont deux autres démarches méthodologiques qui ont été proposées pour combler les insuffisances de l'approche en cascade. Il est important

de les mentionner car elles sont parfois utilisées dans la conception des systèmes temps réel. Cependant, nous n'insisterons pas sur elles car elles ne nous serviront pas par la suite.

L'approche exploratoire consiste à développer un système opérationnel aussi rapidement que possible, puis à le modifier jusqu'à ce qu'il fonctionne conformément aux attentes du client. Ainsi, il s'agit plutôt d'obtenir un système adéquat, mais pas forcément correct. La validation se ramène ainsi à montrer simplement l'adéquation du système. Cette approche s'adresse davantage au domaine de l'*Intelligence Artificielle*, où il arrive de rencontrer des systèmes dont on ne peut spécifier tous les besoins *a priori*.

Quant à l'approche par prototypage, elle procède de façon similaire à la première, par contre, l'objectif est la détermination des besoins du système. Ces besoins vont pouvoir être pris en compte dans la conception finale du système.

Certains modèles de conception de systèmes temps réel combinent ces deux approches. C'est le cas du modèle "3V" proposé par Mosnier et Bortolazzi [165]. Il comporte trois cycles en "V" (d'où son nom) : *i)* le premier cycle consiste à définir de façon rapide et non détaillée, puis à simuler les fonctionnalités globales du système, ce qu'on pourrait considérer comme une étape exploratoire ; *ii)* le second cycle du modèle correspond à la mise en œuvre d'un prototype du système, suivant le cycle en "V" habituel [3] ; *iii)* enfin, le dernier cycle correspond à la réalisation du système lui-même selon le cycle en "V" également. La méthodologie SETTA (Systems Engineering for Time Triggered Architectures) [186] repose sur ce modèle, elle a été proposée pour le développement de systèmes temps réel de type *time-triggered* [139].

* * *

Les approches méthodologiques introduites ci-dessus sont utilisées dans le développement des systèmes de façon générale. Pour les systèmes temps réel en particulier, elles ne sont pas suffisantes pour prendre en compte toutes les exigences de mise en œuvre : entre autres, la garantie de sûreté de fonctionnement qui est un aspect critique, et la capacité à répondre aux lois du marché (fournir le produit dans des délais raisonnables, avec un coût acceptable). Les approches que nous présentons ci-après ont été définies dans le but de résoudre ces lacunes.

1.2.3.3 Approches orientées composants

Ce sont des approches qui ont émergé en réalité dans la seconde moitié des années quatre-vingt-dix. Plus précisément, par rapport [4] au cycle en V, elles ont été définies dans l'optique de :

- réduire la quantité d'effort nécessaire dans le développement et la maintenance d'un système,
- réduire aussi le coût financier de la réalisation du système,
- favoriser une augmentation de la productivité.

La notion centrale dans ces approches est le *composant*. Une définition de ce dernier qui nous paraît pertinente a été donnée par Szipersky [199] :

> *"A component is a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently and is subject to third-party composition."*

3. C'est-à-dire, spécification, conception, implantation, vérification et validation.
4. Une comparaison des approches orientées composants et l'approche en cascade peut être trouvée dans [4].

Ainsi, l'idée de l'approche orientée composants consiste à structurer un système en composants. Ces composants sont le plus souvent déjà disponibles, il s'agit en fait d'une ré-utilisation. Cela veut dire qu'il faut disposer d'une bibliothèque suffisamment fournie pour accéder aux composants dont on a besoin. Chaque composant doit être caractérisé par une interface qui résume d'une certaine manière ses propriétés [5] (signatures de ses opérations, modèles d'interactions avec l'environnement...). Pour les systèmes temps réel, il est souhaitable que les propriétés non fonctionnelles, comme le temps d'exécution, apparaissent. Cela permet des analyses précoces des performances du système considéré. D'autre part, les sources des composants peuvent être dans un langage comme C, C++, ou bien en binaire. La bibliothèque peut être privée ou publique. Dans les deux cas, pour faciliter toute ré-utilisation, il est important que les composants soient décrits de façon suffisamment précise, facilement utilisables ou adaptables. Dans un tel contexte, la conception d'un système revient en grande partie à identifier parmi les composants requis ceux qui sont les plus adéquats, ainsi que leurs liens. La phase d'implantation devient fortement simplifiée. Cela conduit à un cycle en "Y" comme présenté sur la FIG. 1.3.

Dans ce qui suit, nous présentons deux supports fondamentaux aux méthodes de l'approche orientée composants. Ceux-ci contribuent de façon significative au développement de cette approche. Il s'agit des *langages de description d'architecture* et du langage UML. Pour chacun, nous mentionnons quelques unes des méthodes associées, importantes pour le domaine du temps réel.

Langages de description d'architecture. Une grande partie des réflexions menées sur les architectures logicielles reste consacrée à l'étude de leurs structures [6]. Ces réflexions auront largement contribué au gain d'effort dans le développement et la maintenance des systèmes actuels. L'étude de l'architecture du système est une étape qui doit intervenir très tôt dans son développement.

Les langages de description d'architecture (ou ADL - *Architecture Description Languages*) [64] [162] permettent des représentations haut niveau de systèmes à grande échelle (des aspects tels que les algorithmes, les structures de données ou les circuits ne sont pas décrits). Les notions de *composant* et de *connecteur* sont fondamentales car elles jouent un rôle essentiel dans la structuration d'un système. D'autre part, ces langages sont généralement accompagnés d'outils, entre autres pour la spécification, l'intégration, la simulation et l'analyse de certains aspects (exemple : propriétés des composants du point de vue de la composition). Les ADL se distinguent par le *style* adopté pour décrire les architectures.

On compte parmi les plus connus le langage WRIGHT [12], développé à l'université Carnegie Mellon (USA) sous la conduite de Garlan. Ce langage a été proposé pour remédier à l'insuffisance des approches existantes sur les aspects formels. Le langage ACME [96] est quant à lui issu de la collaboration de plusieurs équipes de recherche sur les architectures logicielles, toujours sous la conduite de Garlan. Il joue le rôle de format d'échange pour la technologie dédiée à la description et l'analyse d'architectures. Sa structure générique et extensible permet de profiter des avantages des autres langages et outils. On peut aussi mentionner MODECHART [127], défini à l'université du Texas (Austin, USA) par une équipe dirigée par Al Mok. Ce langage sert à

5. On parle aussi de *contrat*.

6. Historiquement, on retiendra que Edsger Dijkstra [78] fit remarquer en 1968 qu'une structuration bien réfléchie d'un système facilite l'identification de bon nombre de ses caractéristiques. Cette remarque étant largement prise en compte aujourd'hui, la conception d'un système ne consiste plus à passer directement à sa programmation comme c'était le cas auparavant, il faut d'abord étudier son architecture (souvent suivant une méthodologie donnée). Plus tard, David Parnas [173] contribuera aussi au développement de cette philosophie de conception.

spécifier des systèmes temps réel. Sa sémantique RTL (real-time logic) permet de valider les spécifications. Il existe d'autres langages de description d'architecture dédiés à la conception des systèmes temps réel, comme *AIL_Transport* dans l'automobile, qui permet la description d'architectures électroniques embarquées [192]. Le langage MetaH [210] est sans doute le plus populaire aujourd'hui. Il est utilisé dans la conception de systèmes temps réel embarqués, le plus souvent dans le domaine de l'avionique. C'est un langage qui est développé au Honeywell Technology Center (USA) sous la direction de Steve Vestal[7]. Dans le chapitre 8, nous analyserons le lien entre la démarche adoptée dans MetaH pour concevoir les applications avioniques, et l'approche que nous proposons, qui est basée sur le langage Signal. Pour cela, nous présentons les principales caractéristiques de cet Adl.

- **Une introduction au langage** MetaH**.** Ce langage [210] [123] permet de développer des systèmes embarqués suivant une approche *bottom-up* : une description MetaH (cf. Fig. 1.5) d'un système consiste en l'assemblage d'"éléments logiciels" (des morceaux de code source d'origines diverses, écrits dans un langage quelconque) avec des "éléments matériels" (composants représentant des processeurs, des mémoires ou des bus) ; cet assemblage définit ainsi l'architecture complète du système. Le langage permet au développeur de spécifier explicitement le type de matériel requis par chaque élément logiciel pour s'exécuter. L'outil associé se charge alors de réaliser des affectations en fonction des ressources matérielles disponibles.
  Parmi les entités de base utilisées au niveau logiciel, nous pouvons mentionner les suivantes : *process* (comme P et Q sur la Fig. 1.5), *thread*, *package*, *monitor*, *port*, et *event*. La composition hiérarchique de ces objets est réalisée au moyen de mécanismes particuliers : *connexion*, *macro* et *mode*. Ce dernier désigne un groupe d'entités (processus/threads) qui s'exécutent de manière exclusive. Les modes peuvent être vus comme les différentes configurations possibles du système décrit. Les macros résultent d'un découpage hiérarchique des spécifications. Elles jouent un rôle principalement dans la structuration du système.
  Au niveau matériel, les principales primitives sont : *processor* (dans l'exemple de la Fig. 1.5, le processeur considéré est de type DSP), *device*, *memory*, et *channel*/*bus*. Tous ces objets sont reliés au sein d'une description globale de système par des *connexions*.
  L'architecture de l'application décrite sur la Fig. 1.5 comprend deux macros ($M1$ et $M2$) pour la partie logicielle, et un processeur de type DSP. La macro $M2$ contient un seul mode ($m21$) qui regroupe deux *process* (P et Q), alors que $M1$ est composée de deux modes ($m11$ et $m12$, la flèche entre deux modes indique un changement de mode, qui consiste en la désactivation du mode quitté et en l'activation du mode pointé par la flèche).
  L'environnement associé au langage MetaH offre un ensemble d'outils facilitant la réalisation, notamment le partitionnement logiciel/matériel d'une application, l'étude de l'ordonnançabilité, des analyses de sûreté et de fiabilité. Par ailleurs, Il permet aussi de générer du code embarqué, distribué, prenant en compte les aspects liés à la tolérance aux fautes.
  Par rapport aux approches automatisées (ou semi automatisées) existantes, dédiées à la conception des systèmes embarqués temps réel et utilisant les systèmes d'exploitation temps réel, celle de MetaH se distingue surtout par les aspects énumérés ci-après.
  – L'intégration des composants du système peut être réalisée de façon automatique

7. Les travaux autour de ce langage sont sponsorisés principalement par deux organismes : DARPA (Defense Advanced Research Projects Agency) et US Army.

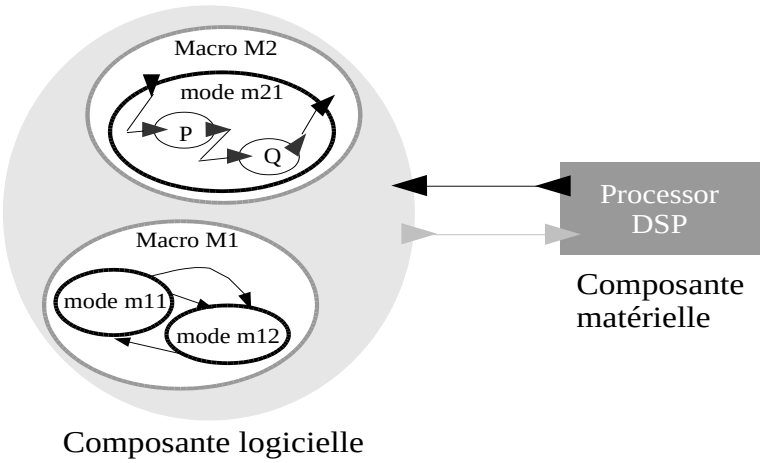


FIGURE 1.5 – Description graphique METAH d'une application.

via les spécifications METAH. L'effort de développement est ainsi réduit, de même qu'une bonne partie des éventuelles erreurs qui peuvent en résulter. Enfin, des analyses statiques permettent d'éliminer du code embarqué inutile.

– Il est possible d'analyser des spécifications partiellement décrites en METAH. Un mécanisme de co-génération produit en même temps du code et des modèles analytiques sous forme d'automates hybrides, à partir des spécifications. Il garantit aussi la cohérence des résultats d'analyses effectuées sur un modèle vis-à-vis du comportement final de l'implantation décrit par le code.
– Les concepts du langage (notamment les composants matériels et logiciels) sont définis de façon à ce qu'ils soient le plus indépendants du contexte d'utilisation, cela favorise leur réutilisabité. Enfin, METAH facilite une reconfiguration rapide de l'architecture du système (par exemple, en cas de changement de composant matériel, de besoins fonctionnels), sans modifier les modules sources utilisés.

Cependant, nous pouvons relever quelques restrictions de la conception à l'aide de METAH. La mise en œuvre d'un système dépend fortement de la politique d'ordonnancement des processus considérée dans METAH : cet ordonnancement se base d'une part, sur des informations relatives au chemin d'exécution des processus, et d'autre part, sur les annotations temps réel qui caractérisent les modules sources. C'est une technique particulière à METAH. Cela entraîne peu de flexibilité puisque que toute plate-forme candidate à un déploiement du système doit supporter ce paradigme. Une autre restriction concerne le mécanisme de communication entre processus. En effet, les échanges de messages ne peuvent avoir lieu qu'à des moments bien précis : en début d'exécution ou bien en fin. Un tel choix est surtout motivé par le fait qu'il facilite l'analyse des comportements d'une application.

UML (Unified Modeling Language). UML est un standard de l'*Object Management Group* (OMG) [170] dont la définition propose des concepts "objets" sous forme de notation, et leurs relations en vue d'exprimer des modèles. Ce méta-modèle permet de spécifier, construire, visualiser et documenter un système. Le but est de faciliter la représentation et la compréhension de solutions aux problèmes de conception. Le choix d'une notation unifiée pour l'activité de conception fournit surtout une base commune pour la compréhension, à tous les acteurs prenant part à un même projet.

La modélisation à l'aide d'Uml offre principalement quatre points de vue concernant [128] :

- les *aspects statiques du système* ("le qui") : diagrammes de classes reflétant divers aspects : description des objets et de leurs relations, modularité, contrats, relations, généricité, héritage et structuration en paquetages,
- la *perception du système par l'utilisateur* ("le quoi") : cas d'utilisation,
- les *aspects dynamiques du système* ("le quand") : diagrammes de séquences (scénarios), diagrammes de collaboration (entre objets), diagrammes d'états et transitions et diagrammes d'activités,
- la *vision implantation* ("le où") : diagrammes de composants et de déploiement.

Comme nous pouvons le constater, les descriptions Uml reposent sur divers types de diagrammes. Pour une présentation détaillée de ceux-ci, nous renvoyons le lecteur à [170]. Ces diagrammes peuvent être annotés de contraintes afin de lever les éventuelles ambiguïtés qui peuvent y apparaître. Le langage utilisé pour spécifier ces contraintes est *Object Constraint Language* (Ocl), développé par IBM.

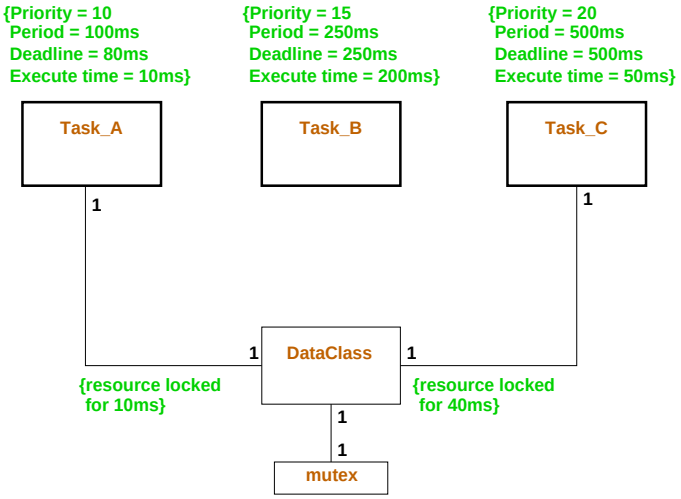


Figure 1.6 – Diagramme de classes représentant un modèle Uml.

Le diagramme de classes dessiné sur la Fig. 1.6 montre quelques-uns des éléments de notation utilisés dans Uml [79]. Il s'agit d'un modèle élémentaire qui a été défini sous l'environnement de conception Rhapsody. Il représente un système composé de trois tâches `Task_A`, `Task_B` et `Task_C`, qui s'exécutent de façon concurrente. Elles sont modélisées par les classes (représentées par des rectangles) en gras (pour dénoter des entités *actives*). Chaque tâche est caractérisée par des annotations temps réel qui indiquent sa *priorité*, sa *période d'activation*, son *échéance* et son *temps d'exécution*. Par exemple pour `Task_A`, ces paramètres valent respectivement 10, 100 millisecondes, 60 millisecondes et 10 millisecondes. Les autres classes sont décrites par des rectangles simples. Ici, l'unique ressource partagée par les tâches est dénotée par `DataClass`. Par ailleurs, l'accès à cette ressource se fait de façon exclusive pour les tâches concernées. Cela est réalisé à l'aide d'un sémaphore. Ce protocole d'accès est représenté sur le diagramme par la classe `mutex`, d'où l'*association* entre `DataClass` et cette dernière. On remarquera que le lien entre les deux classes porte des informations de *cardinalités* (ou *arités*) pour indiquer le nombre

d'instances de chaque classe pouvant être en relation. Sur le schéma, toutes les arités sont égales à "1". Par exemple, on affectera un et un seul sémaphore à chaque ressource et inversement (dans la situation considérée, seules `Task_A` et `Task_C` utilisent `DataClass`).

Si le méta-modèle UML offre une grande richesse d'expressivité du point de vue syntaxique, il ne dispose pas en revanche de sémantique formelle clairement définie. Cela conduit à des problèmes de gestion de la cohérence entre les nombreuses vues qui peuvent constituer une spécification. Certains travaux ayant porté sur la recherche d'une solution au problème se sont concentrés sur une seule vue, par exemple [213] se concentre sur les diagrammes d'états (STATECHARTS). D'autres ont plutôt essayé d'unifier des concepts normalement éparpillés entre plusieurs vues, dans le but d'assurer une meilleure cohérence [144]. Un autre inconvénient d'UML réside dans le fait que son utilisation est fortement dépendante des outils : par exemple, des projets UML réalisés avec des outils différents ne peuvent coopérer.

L'évolution du langage UML aujourd'hui est due en majeure partie à son succès dans le monde industriel. Il faut ainsi adapter le langage aux besoins de différents domaines. En particulier, on peut observer les efforts fournis afin de le rendre adéquat à la conception des systèmes temps réel.

Plusieurs extensions du langage sont proposées pour l'adapter à la description des systèmes temps réel [79] [188]. Ces extensions montrent comment UML peut faciliter la conception et le développement des systèmes temps réel au travers d'exemples concrets. Pour cela, certains aspects relatifs au temps réel ont été incorporés dans le modèle, c'est le cas du *profil* UML dit SPTS (*Schedulability, Performance, and Time Specification*), sur lequel nous reviendrons par la suite. Un profil UML décrit une spécialisation du méta-modèle à un domaine particulier (par exemple, un profil standard UML pour les intergiciels est CORBA [169], introduit au chapitre 2). Des langages plus spécifiques se basent sur une notation UML. C'est le cas notamment d'ADL comme *AIL_Transport* déjà mentionné, ou bien *cotre* [38] [1] défini dans le cadre du projet RNTL [8] du même nom. Le langage *cotre* sert de support à une méthodologie de modélisation et de validation d'architectures de logiciels temps réel avioniques : de la conception à l'implantation cible. Il est défini formellement au travers d'une méta-modélisation UML.

Le profil SPTS a été défini par l'OMG [9] [188] dans le but de dégager les paradigmes adéquats d'utilisation d'UML pour le traitement d'aspects concernant la modélisation du temps, de l'ordonnançabilité, et de la performance dans les systèmes temps réel. Cela offre la possibilité de décrire des modèles qui permettent des prédictions quantitatives concernant des aspects critiques dans les systèmes temps réel tels que l'ordonnancement ou la performance, tout en profitant des autres avantages d'UML. Plus précisement, les aspects abordés dans ce profil sont :

- la modélisation de *ressources* bornées logiques et physiques, auxquelles des informations quantitatives peuvent être associées ;
- la modélisation de caractéristiques temporelles liées à la *qualité du service* (QoS). On distingue deux principales approches pour aborder cet aspect : la première se base sur l'ordonnancement dans un système (par exemple, en utilisant le nombre d'échéances ratées comme mesure de QoS), tandis que la seconde propose de gérer la qualité de service à un niveau d'abstraction plus élevé dans un système (par exemble, au travers d'un intergiciel).

8. `http://www.telecom.gouv.fr/rntl/AAP2001/Fiches_Resume/COTRE.htm`.

9. Document détaillé accessible à l'adresse `http://www.omg.org/cgi-bin/doc?ptc/2003-03-02`. Les compagnies qui ont participé à l'élaboration des résultats sont : ARTiSAN Software Tools, Inc. - I-Logix Inc. - Rational Software Corp - Telelogic AB - TimeSys Corporation - Tri-Pacific Software Inc.

La première approche est plutôt adaptée aux systèmes "fermés" (c'est-à-dire, des systèmes pour lesquels tout est décidé au moment initial de développement - aucun changement n'est envisagé après). À l'inverse, la seconde approche convient surtout aux systèmes "ouverts" (i.e., ceux dans lesquels de nouvelles applications peuvent être prises en compte, et dont les anciennes peuvent être remplacées) ;
- la définition de *modèles* flexibles *de temps* et des mécanismes associés (exemple : horloge physique, *timer*) ;
- la définition d'un *modèle de concurrence* comprenant entre autres les *tâches* et leur mécanisme de gestion ;
- l'incorporation de méthodes d'*analyse d'ordonnancements* telles que *Rate Monotonic Analysis* (RMA) ;
- l'incorporation de techniques d'*analyse de performances.*

Les méthodologies de conception de systèmes temps réel basées sur UML favorisent, d'une part, une démarche qui repose sur des modèles, et d'autre part, elles permettent une génération de code (C, C++, Java) qui prend en compte les propriétés structurelles et comportementales spécifiées dans les diagrammes. Ces deux aspects constituent notamment des caractéristiques-clés du concept technologique *MDA* (*Model Driven Architecture* ou ingénierie dirigée par les modèles) [193], défini par l'OMG. Les idées de base de ce concept reviennent à la définition de modèles indépendants des plates-formes, ainsi que des transformations associées pour en faire des modèles spécifiques à une plate-forme donnée. Les modèles sont décrits à l'aide d'UML. On pourrait remarquer ici que ces idées ne sont pas totalement neuves. Il s'agit en fait de la mise en œuvre de principes bien connus qui sont l'abstraction et le raffinement, souvent manipulés en informatique théorique. Ainsi, l'apport majeur du concept MDA réside plutôt dans la facilité d'utilisation de ces principes, offerte en pratique. Cela devient possible grâce aux environnements de conception définis pour supporter le concept. Nous mentionnons à cet égard les environnements suivants : ROSE-RT [10] dont l'approche MDA associée est *UML-RT* ; RHAPSODY qui est développé par *i-Logix* (États-Unis), et dans lequel a été définie l'approche *ROPES* [11] ; et enfin, l'environnement associé à l'approche *ARTiSAN* [12]. Par ailleurs, nous soulignerons que la *transformation de modèle* est de plus en plus au cœur de la problématique MDA [43]. D'après [13] J.-M. Jézéquel, "*un processus de développement peut être décrit comme un ensemble partiellement ordonné de transformations de modèles, qui prend comme entrées des modèles, et qui produit des modèles comme sorties*". La transformation est donc une opération centrale dans l'approche MDA car elle conditionne plusieurs aspects liés à la construction des modèles (exemples : automatisation, raffinement).

La popularité de la technologie autour d'UML dans l'industrie d'aujourd'hui est due en grande partie aux concepts pratiques de modélisation qu'elle offre. Les différents acteurs impliqués dans la conception d'un système s'y retrouvent assez aisément. Cela facilite entre autres la communication entre différentes équipes de développement, qui est un aspect primordial. Par ailleurs, de nombreux outils sont disponibles pour aider les utilisateurs. Cependant, cette technologie souffre toujours du manque d'une sémantique formelle bien définie pour le langage, même si beaucoup d'efforts ne cessent d'être accomplis en ce sens. C'est d'autant plus nécessaire que cela faciliterait la validation.

10. `http://www.rational.com/products/rosert`.
11. Rapid Object-oriented Process for Embedded Systems - `http://www.ilogix.com`.
12. `http://www.artisansw.com`.
13. *Model-driven engineering : Basic principles and challenges.* - Exposé invité à *Formal Methods for Components and Objects (FMCO'03)*, Leiden, Netherlands, Novembre 2003.

Quel rapprochement entre les ADL et UML ? Cette présentation des deux supports pour des approches orientées composants, à savoir les ADL d'une part, et UML de l'autre nous conduit à faire les quelques observations suivantes. D'abord, nous pouvons remarquer que l'espace des besoins qu'on souhaiterait couvrir avec UML est plus vaste que celui concerné par les ADL. Par conséquent, le modèle conceptuel de UML a un pouvoir expressif qui dépasse largement celui des ADL. Pour autant, pouvons-nous en conclure que UML est en quelque sorte un "super-ensemble" pour les ADL ? À notre connaissance, l'étude qui a tenté d'apporter le plus d'éléments de réponse a été menée par N. Medvidovic et al. [161]. Cette étude a clairement exposé deux *stratégies* permettant d'utiliser UML pour modéliser les architectures logicielles, à la façon des ADL :

- **stratégie 1** : utiliser UML tel qu'il est défini (dans sa façon standard) pour modéliser les concepts architecturaux traditionnellement considérés avec un ADL quelconque (exemples : composants, connecteurs) ;
- **stratégie 2** : adapter UML, en lui rajoutant des aspects propres aux ADL, pour en faire sous une autre forme un ADL.

Parmi les conclusions relatives à l'examen des deux stratégies, il en ressort qu'UML pourrait tout à fait jouer le rôle d'un ADL. Cependant, cela a un prix très coûteux. La diversité des ADL s'explique par le fait qu'ils sont souvent dédiés à des domaines bien précis. De ce fait, le style qu'ils adoptent est choisi en conséquence. C'est la source d'un des problèmes centraux que [161] souligne : UML ne permet pas de prendre en compte certains styles, étant donné qu'il adopte un style principalement orienté objet.

Une autre étude que nous pouvons mentionner concerne la définition du langage AADL (Avionics Architecture Description Language) [86] [1] qui est dédié à la description des systèmes avioniques. Il est développé par la *Society of Automotive Engineers* (SAE). C'est un bon exemple, qui combine d'une part l'ADL METAH, et d'autre part UML.

1.2.3.4 Approches basées sur les méthodes formelles

Les approches formelles ont par le passé connu certaines réticences de la part des développeurs, quant à leur utilisation dans l'activité de conception. I. Sommerville en donne quelques raisons dans [194]. Entre autres, il a été reproché à ces approches la quantité d'effort non négligeable, et pas forcément justifiée aux yeux des développeurs, qu'elles requièrent (par exemple pour rédiger des spécifications formelles). Aujourd'hui, le constat est plutôt le changement des mentalités en faveur de ces approches. Ainsi, les méthodes formelles apparaissent de plus en plus nécessaires à l'activité de conception. Elles contribuent de façon significative à l'amélioration de la fiabilité des systèmes conçus. En particulier, elles apparaissent indispensables pour ce qui concerne la conception des systèmes temps réel [182].

Les approches formelles reposent d'abord sur l'utilisation de formalismes de spécification fondés sur des concepts mathématiques. Grâce à ces derniers, des raisonnements corrects peuvent être établis. De fait, plusieurs approches formelles sont orientées vérification, c'est le cas de celles qui sont basées sur les automates temporisés. Les autres approches se veulent plus générales, elles permettent au moins de spécifier, vérifier et générer du code. L'approche synchrone fait partie de ces dernières. Elle offre des formalismes à l'aide desquels un système peut être décrit selon différents niveaux d'abstraction. Ces descriptions peuvent éventuellement reposer sur des mécanismes de raffinement, comme cela est le cas pour le langage SIGNAL : partant d'une description initiale, on procède à des transformations sur celle-ci, à l'aide de règles bien précises,

jusqu'à l'obtention d'une description finale qui reflète la mise en œuvre souhaitée. D'autre part, ces formalismes facilitent la vérification de la consistance des spécifications vis-à-vis des propriétés à démontrer.

Un autre avantage pratique des méthodes formelles réside dans le caractère systématique de bon nombre des raisonnements et traitements effectués lors de la conception (exemples : vérification, codage). Cela peut être facilement automatisé dans des outils, garantissant la correction des transformations effectuées. Ainsi, la plupart des approches formelles fournissent des outils pour accompagner le développeur dans sa tâche. Il en résulte ainsi un gain de temps et une diminution des risques d'erreurs, habituellement liés aux transformations manuelles.

Ces dernières années, de nombreuses méthodes de développement de systèmes temps réel, basées sur l'utilisation de concepts et outils formels, ont été définies. Certaines d'entre elles sont spécifiques à des domaines particuliers (exemple : télécommunications, automobile), tandis que d'autres se présentent comme plus générales.

Nous présentons trois familles d'approches formelles qui concernent toutes la conception des systèmes temps réel : la première famille est basée sur les *algèbres de processus* ; les deux autres reposent quant à elles sur les notions d'états et de transitions. Nous distinguons d'une part, celles qui sont basées sur des *évolutions explicites* d'états, et d'autre part, celles qui sont basées sur des *évolutions implicites* d'états.

Approches basées sur les algèbres de processus. Des langages tels que CSP (*Communicating Sequential Processes*) de Hoare [122], ou CCS (*Calculus of Communicating Systems* de Milner [164] furent proposés dans le but d'étudier les systèmes concurrents communicants. Ils sont à l'origine de l'approche *algébrico-axiomatique* de la théorie de la concurrence, connue sous le nom d'algèbre de processus (en anglais, *process algebra*). Celle-ci vise notamment à aider à une meilleure compréhension du parallélisme dans les systèmes, et à fournir un cadre théorique pour en cerner les mécanismes de base.

	$a, b, c...$	(actions)
	$X, Y, Z...$	(variables)
$e ::= \tau \mid a \mid \bar{a}$		(événements)
$p ::= \mathsf{nil} \mid e.p \mid p\backslash A \mid p + q \mid p \parallel q \mid \mathsf{rec}\, X.p$		(processus)

FIGURE 1.7 – Syntaxe de CCS

Les algèbres de processus consistent en des calculs définis dans un formalisme algébrique (i.e. constitué d'un domaine et d'un ensemble d'opérations). Les comportements de programmes concurrents peuvent ainsi être spécifiés à l'aide de constructions syntaxiques, ayant une sémantique formelle bien établie et souvent exprimée de façon opérationnelle.

Pour illustrer cette famille de formalismes, nous donnons un rapide aperçu de CCS (syntaxe et sémantique opérationnelle) ainsi qu'une de ses extensions temporisées. CCS est caractérisé par des communications instantanées non bloquantes. Celles-ci mettent en jeu deux acteurs au plus (l'émetteur et le récepteur).

Les objets de base sont décrits sur la FIG. 1.7, ce sont :

- les *actions* atomiques ;
- les *variables* ;
- les *événements*, parmi lesquels on distingue l'événement *silencieux* τ, et les co-événements/actions notés avec une "barre" (exemple : $\bar{a}$). On dit que a et $\bar{a}$ sont complémentaires (avec $a \neq \tau$), et par convention $\bar{\bar{a}} = a$;
- les *processus*. Parmi ceux-ci, *nil* représente le processus incapable de toute action. Les processus peuvent être combinés soit à l'aide d'un opérateur de *choix non-déterministe* (dénoté par $+$), soit avec un opérateur de *composition parallèle* (dénoté par $\|$). Il existe aussi un opérateur externe de *préfixage* noté ".", pour exprimer qu'on effectue une action spécifiée en préfixe (exemple : $e.p$ est le processus qui effectue l'action e, puis agit comme p). Les définitions récursives quant à elles sont obtenues à l'aide du mot-clé rec (dans l'expression $\mathsf{rec}\, X.p$ sur la FIG. 1.7, X est une variable). Enfin, l'opérateur "\" dénote la restriction au sein d'un processus.

$$e.p \xrightarrow{e} p \quad (r_1) \qquad \frac{p \xrightarrow{e} p'}{(p+q) \xrightarrow{e} p'} \quad (r_2) \qquad \frac{q \xrightarrow{e} q'}{(p+q) \xrightarrow{e} q'} \quad (r_3)$$

$$\frac{p \xrightarrow{e} p'}{(p\backslash A) \xrightarrow{e} (p'\backslash A)} \; {\scriptstyle e \notin A} \quad (r_4) \qquad \frac{p[(\mathsf{rec}\, X.p)/X] \xrightarrow{e} p'}{(\mathsf{rec}\, X.p) \xrightarrow{e} p'} \quad (r_5)$$

$$\frac{p \xrightarrow{e} p'}{(p \parallel q) \xrightarrow{e} (p' \parallel q)} \quad (r_6) \qquad \frac{q \xrightarrow{e} q'}{(p \parallel q) \xrightarrow{e} (p \parallel q')} \quad (r_7) \qquad \frac{p \xrightarrow{e} p' \quad q \xrightarrow{\bar{e}} q'}{(p \parallel q) \xrightarrow{\tau} (p' \parallel q')} \quad (r_8)$$

FIGURE 1.8 – Sémantique opérationnelle de CCS

La sémantique opérationnelle de CCS est rappelée sur la FIG. 1.8, dans une syntaxe à la Plotkin. Il s'agit d'un ensemble de règles d'inférence qui décrivent une fonction de l'ensemble des processus vers un système de transition d'états. Il en résulte un graphe dans lequel les sommets représentent les états d'un processus, et les arcs représentent les événements produits par le processus à chaque changement d'état. On note $p \xrightarrow{e} q$ la transition de l'état p à l'état q produisant l'événement e. Dans les règles, la récriture est exprimée sous la forme $\dfrac{\{prem\}}{\{concl\}}$, où *prem* et *concl* représentent respectivement les prémisses et conclusions. Pour donner une idée sur la façon dont les règles sont interprétées, nous expliquons les règles (r_1), (r_5) et (r_6). La règle (r_1) exprime une transition où le processus $e.p$ (état de départ) effectue une action atomique e (spécifiée en préfixe), puis se retrouve dans l'état représenté par p (état d'arrivée, celui-ci contient les actions restant à effectuer). Dans la règle (r_5), on définit le comportement du processus $\mathsf{rec}\, X.p$ comme étant celui de p, dans lequel toutes les occurrences de la variable X sont remplacées par le processus $\mathsf{rec}\, X.p$. Enfin, la règle (r_6) spécifie le fait que la composition parallèle de deux processus p et q, où le premier effectue une action e et devient p', alors que le second n'agit pas, se comporte comme le processus qui agit (c'est-à-dire p), devenant ainsi la composition de p' et q.

Il existe plusieurs extensions temporisées de CCS. On trouvera dans [209] une large présentation de celles-ci, et de façon générale, les autres algèbres de processus temporisées. L'extension

introduite ici a été proposée par Chen [61] (elle est appelée *Chen's Timed CCS*[14]). Dans cette algèbre, le domaine considéré pour le temps n'est pas fixé (il peut être $\mathbb{R}, \mathbb{Q}, \mathbb{N}$...). L'auteur étudie notamment certains aspects concernant la décidabilité et la complétude de l'algèbre en considérant des instanciations avec des domaines de temps discrets ainsi que des domaines de temps denses. Par ailleurs, il donne une sémantique opérationnelle des constructions à l'aide de règles à la Plotkin, et une sémantique dénotationnelle en utilisant des structures particulières, appelées *timed synchronization trees*.

$$p \quad ::= \quad \mathsf{nil} \mid e(t)^{bsup}_{binf}.p \mid p\backslash A \mid p + q \mid p \parallel q \mid \mathsf{rec}\, X.p \qquad \text{(processus)}$$

FIGURE 1.9 – Syntaxe de Timed CCS selon Chen [61].

L'action représente la principale construction pour introduire le temps dans l'algèbre. La FIG. 1.9 décrit la forme d'un processus selon l'algèbre de Chen (les autres objets de base ne changent pas). Dans le processus $e(t)^{bsup}_{binf}.p$, t est une variable temporelle, qui peut apparaître comme variable libre dans p ; les expressions de temps $binf$ et $bsup$ décrivent respectivement les bornes inférieures et supérieures (incluses) de l'intervalle dans le lequel varie t. Ce processus n'agit pas pendant une durée δ, telle que $0 \leq binf \leq \delta \leq bsup \leq \infty$. Il exécute ensuite l'action e et devient le processus dans lequel toutes les occurrences de t sont substituées par δ. Une algèbre proche de celle de Chen est celle qui a été définie par Daniels [74], où le temps est décrit à l'aide d'intervalles sur $\mathbb{R}$.

Comme nous venons de le voir pour CCS, les autres algèbres de processus ont aussi leurs extensions temporisées pour spécifier les comportements de systèmes temps réel. Parmi les toutes premières algèbres de processus temporisées, nous mentionnerons *ATP* (*Algebra of Timed Processes*) définie par Nicollin et Sifakis [168]. Dans *ATP*, le temps est discret. Cette algèbre est très proche de *ACP* (*Algebra of Communicating Processes*), une autre algèbre dont la version temporisée a été donnée par Baeten et Bergstra [20]. Enfin, une extension temporisée de CSP a été proposée par Schneider [187]. En dehors de la forme considérée pour le temps dans ces formalismes (c'est-à-dire, soit dense soit discrète), nous retiendrons qu'ils se distinguent surtout par rapport aux caractéristiques des algèbres sous-jacentes non temporisées.

Des propositions plus récentes visent à fournir des modèles avec une expressivité qui prendrait en compte les aspects critiques des systèmes temps réel, tels que la gestion de ressources, l'ordonnancement. C'est notamment le cas de l'algèbre *ACSR* définie par Brémond-Grégoire et Lee [51]. Celle-ci permet de spécifier les priorités, de décrire des mécanismes de partage de ressources, de préemption et d'exception. L'outil associé s'appelle VERSA [63]. Il offre la possibilité de spécifier et vérifier des systèmes temps réel dans le formalisme de *ACSR*. Dans [84], les auteurs étudient un système de commande d'une gouverne d'avion à l'aide d'une algèbre proche de *ACSR*, appelée *TPAP*. Enfin, nous mentionnerons le langage de spécification RT-LOTOS [71] qui est fondé sur les algèbres de processus. C'est une extension temporisée du langage LOTOS dont la sémantique repose sur CCS et CSP.

*
* *

Après les approches basées sur les algèbres de processus, nous présentons deux autres familles d'approches formelles. Celles-ci ont en commun le fait d'avoir des automates comme modèles sous-jacents.

14. D'autres extensions prennent une appellation analogue, par exemple *Wang's Timed CCS* [209].

Bref rappel sur les automates à états finis. Les automates représentent un outil à la fois simple et pratique pour décrire les comportements d'un système. On associe ainsi une signification aux états et transitions de l'automate. Par exemple, chaque état dénotera une phase de traitement tandis que les transitions constitueront les phases de communication. Un *automate à état fini* (en anglais, *Finite State Machine* - FSM) peut être défini comme suit.

Définition 1 *Un automate à états finis Aut est un quintuplet* $< X, \mathcal{Q}, q_0, \mathcal{A}, \delta >$ *où :*
- X *représente un alphabet d'entrée fini,*
- $\mathcal{Q}$ *est un ensemble fini d'états,*
- $q_0 \in \mathcal{Q}$ *est l'état initial,*
- $\mathcal{A} \subset \mathcal{Q}$ *contient les états d'acceptation (dits aussi états terminaux ou finaux),*
- $\delta \subset \mathcal{Q} \times X \times \mathcal{Q}$ *est une relation de transition.*

□

L'automate de la FIG. 1.10 permet de reconnaître les séquences de la forme $a^i bbc^j$, $i \geq 0, j \geq 0$ (exemples : *bb*, *abb*, *bbc*, *aaaaaaaabbc*, *aabbcccc*). Il est formellement défini par le quintuplet suivant :
$< \{a, b, c\}, \{q_0, q_1, q_2\}, q_0, \{q_2\}, \{(q_0, a, q_0), (q_0, b, q_1), (q_1, b, q_2), (q_2, c, q_2)\} >$

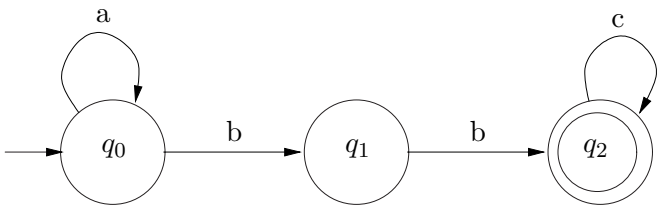


FIGURE 1.10 – Automate reconnaissant les séquences de la forme $a^i bbc^j$, $i, j \geq 0$.

Sur cet automate, l'état initial q_0 est repéré par une flèche ne provenant d'aucun état. L'état d'acceptation q_2 est quant à lui, mis en évidence par deux cercles imbriqués. Une séquence est reconnue comme étant de la forme $a^i bbc^j$, $i, j \geq 0$ lorsque cet état est atteint.

Les approches décrites ci-après sont classifiées selon la façon dont apparaît le modèle d'automate dans les supports associés. Les approches basées sur des *évolutions explicites d'états* utilisent les automates pour décrire les comportements d'un système. Cela permet d'observer exactement les différents états par lesquels le système ainsi représenté transite. Parmi les formalismes servant de support à ces méthodes nous pouvons citer SDL, les automates temporisés, les automates hybrides. Dans les autres approches, les notions d'états et de transitions n'apparaissent pas explicitement. Nous les qualifions d'approches basées sur des évolutions implicites d'états. C'est le cas notamment des approches fondées sur des formalismes tels que les langages synchrones ou les réseaux de Petri.

Approches basées sur des évolutions explicites d'états. Les approches présentées ici utilisent comme support des extensions d'automates à états finis.

- **Approches fondées sur SDL (Specification and Description Language).** Le langage SDL [83] est un langage formel, orienté objet et graphique comme UML, utilisé

pour spécifier et concevoir des systèmes réactifs temps réel. C'est un langage qui est en constante évolution et standardisé par l'ITU (*International Telecommunication Union*) [125]. Une description SDL consiste en un ensemble d'éléments structurels, à savoir les *systèmes*, les *blocs* et les *processus*. Un système peut se décomposer en blocs, qui à leur tour peuvent être répartis en processus. Ces éléments communiquent entre eux, ou bien avec l'environnement. Ces communications sont réalisées d'une part, via des *signaux asynchrones* et d'autre part, à travers des *appels synchrones* (en mode *remote procedure call* - RPC). Le comportement de chaque processus est représenté par une *machine à états finis étendue* (en anglais, *extended FSM*). Il s'agit d'une FSM qui intègre la notion de variable. En plus des événements d'entrée/sortie, on peut associer aux transitions de la FSM des prédicats et des actions sur les variables. Sur la FIG. 1.11, nous avons

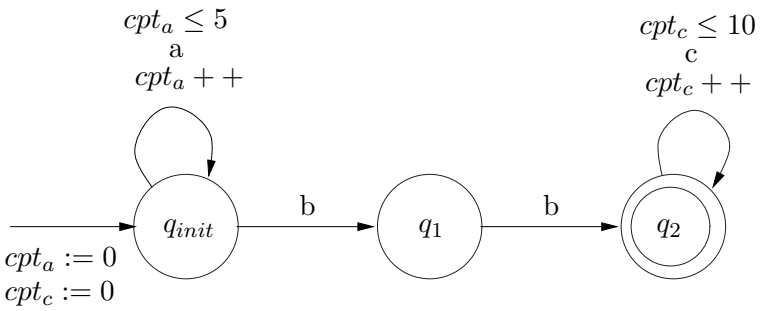


FIGURE 1.11 – Automate étendu reconnaissant les séquences de la forme a^ibbc^j, $i \leq 5, j \leq 10$.

ajouté les variables cpt_a et cpt_c à l'automate de la FIG. 1.10. Elles servent à compter respectivement le nombre de caractères '`a`' et '`c`' lus, depuis le début de la reconnaissance d'une séquence. Initialement, ces variables sont mises à zéro. Elles sont incrémentées par la suite sur certaines transitions. Les prédicats associés aux transitions de lecture des '`a`' et '`c`' limitent le nombre de caractères dans les séquences reconnaissables. Ces dernières sont désormais de la forme a^ibbc^j, $i \leq 5, j \leq 10$ (exemples : la séquence *aaaaaaaabbc* n'est pas reconnaissable, alors que *bb*, *abb*, *bbc*, *aabbcccc* le sont).
Dans le langage SDL, une donnée est décrite par un type de données abstrait. Celui-ci comprend :

- un ensemble de valeurs,
- un ensemble d'opérations applicables à ces valeurs,
- un ensemble de règles algébriques (équations) qui définissent le comportement des opérations lorsqu'elles sont effectuées.

On retrouve parmi les types prédéfinis des types classiques tels que *integer, real, boolean* ou *character*. On note aussi des types spécifiques comme *time* (temps absolu - date) et *duration* (temps relatif - différence de deux dates) liés au temps, ou *PId* pour identifier les instances de processus. Par ailleurs, d'autres types de données peuvent être définis à l'aide de constructions telle que *array* et *struct*. Cette notion de type de données abstrait donne à SDL une orientation objet. Ainsi, le mécanisme d'héritage est présent dans le langage. Cela favorise la réutilisation des constructions (des éléments structurels tels que les blocs, les processus, pouvant être définis en tant que types).

Le langage SDL est très utilisé dans le domaine des télécommunications. Toutefois, on le retrouve dans d'autres domaines comme l'aéronautique, l'espace ou l'automobile. Parmi les approches basées sur ce langage, on distingue [11], où les auteurs combinent UML et

SDL pour développer des systèmes embarqués temps réel. Les approches [69] et [142] utilisent aussi SDL. Par ailleurs, nous pouvons mentionner la plate-forme TAU, développée par la société *Telelogic*[15] (États-Unis / Suède). Elle comprend deux outils basés sur SDL :

- SDT - Un outil de conception de système à l'aide de SDL. Il permet aussi de modéliser un système en utilisant la méthode OMT, qui est orientée objet. Enfin, il offre la possibilité de générer du code.
- ITEX - Il est utilisé pour la génération de cas de test à partir de SDT.

- **Approches fondées sur les automates temporisés.** Il s'agit d'une variante d'automate étendu, où les variables ajoutées dans l'automate sont discrètes, et appelées *horloges* [15]. Les automates temporisés ont été introduits pour servir de modèle formel pour spécifier les systèmes temps réel. Les horloges décrivent l'évolution du temps. Elles ont toutes la même fréquence. Chaque transition d'un état à un autre peut être étiquetée par des contraintes sur les horloges (condition de franchissement) d'une part, et des affectations de valeurs sur les horloges (principalement des remises à zéro) d'autre part. Les contraintes sur les horloges sont définies inductivement de la manière suivante :

$$contrainte ::= x \leq c \mid c \leq x \mid x < c \mid c < x \mid contrainte_1 \wedge contrainte_2$$

où x est une variable d'horloge et c une constante rationnelle.

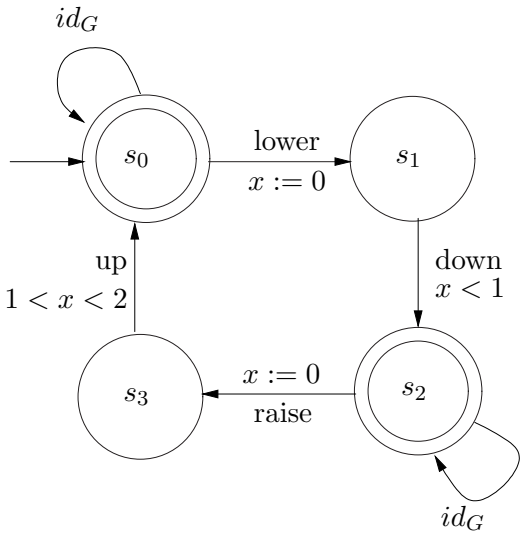


FIGURE 1.12 – Spécification du comportement d'une barrière dans un passage à niveau.

L'automate temporisé dessiné sur la FIG. 1.12 représente le comportement attendu d'une barrière, lors de la traversée d'un passage à niveau [15]. Dans cette description, l'alphabet fini d'actions (c'est-à-dire, l'ensemble X dans la définition 1) est représenté par l'ensemble suivant : $\{id_G, up, down, lower, raise\}$. L'unique état initial est s_0. Ce dernier est également état terminal, de même que l'état s_2. La barrière est ouverte dans l'état s_0, et fermée dans l'état s_2. Elle communique avec un dispositif de contrôle à travers les signaux *lower* (requête de fermeture) et *raise* (requête d'ouverture). Les événements

15. `http://www.telelogic.com`.

up et *down* dénotent respectivement l'ouverture et la fermeture de la barrière. Enfin, la barrière peut effectuer des transitions sans effet (repérées par l'action id_G) dans les états s_0 et s_2. Les états s_1 et s_3 sont des états intermédiaires entre la réception d'une requête et la réalisation effective de l'action associée à celle-ci : dans l'état s_1 (resp. l'état s_3), la barrière est en train d'être fermée (resp. ouverte) ; la sortie de cet état indique que la barrière a été fermée (resp. ouverte), dénoté par l'événement *down* (resp. l'événement *up*).
Dans l'automate de la FIG. 1.12, il n'y a qu'une horloge, représentée par la variable x. La contrainte associée à la transition *down* exprime le fait que la fermeture de la barrière doit être effective en une unité de temps au plus, à la suite d'une requête de fermeture. De façon similaire, la contrainte décrite sur la transition *up* spécifie que l'ouverture de la barrière doit survenir entre une et deux unités de temps, après une requête d'ouverture. On remarquera que l'horloge x est remise à zéro avant chaque décompte.

À coté des automates temporisés classiques (au sens d'Alur et Dill), nous pouvons en mentionner d'autres comme les extensions temporisées des *Input/Output automata* (ou *I/O automata*) de N. Lynch [155], appelées *timed I/O automata* [133]. Les I/O automates sont des automates au sein desquels les transitions sont étiquetées par des *actions*. Trois types d'actions sont distingués dans ces automates : les actions en *entrée*, les actions en *sortie* et les actions *internes*, dénotées respectivement par $in(Aut)$, $out(Aut)$ et $internal(Aut)$ pour un I/O automate Aut. Une propriété fondamentale est que $out(Aut)$ et $internal(Aut)$ sont sous le contrôle de Aut (c'est lui qui les génère). Les actions $out(Aut)$ sont transmises à l'environnement de façon instantanée. Quant aux actions $in(Aut)$, elles sont produites par l'environnement, et envoyées instantanément aussi à Aut. Les I/O automates sont appropriés pour décrire des systèmes à événements discrets.

Les outils autour des automates temporisés est très développée. Des outils comme UPPAAL [26] ou KRONOS [219] en font partie. Il existe plusieurs approches orientées vérification, basées sur ces outils. Parmi elles, nous citons TAXYS [65] car c'est celle qui se rapproche le plus de l'approche présentée dans cette thèse. Elle a été proposée dans le cadre d'une collaboration entre France Télécom et le laboratoire de recherche Verimag (Grenoble, France). Cette approche permet de modéliser, programmer et vérifier des systèmes embarqués temps réel. À partir de spécifications exprimées dans le langage synchrone ESTEREL [36] et annotées de contraintes temporelles (temps d'exécution, échéance), sont générés des modèles sous forme d'automates temporisés. La vérification des modèles est faite avec l'outil KRONOS [219], et le compilateur SAXO-RT [215] (développé par France Télécom) les compile en C.

- **Approches fondées sur les automates hybrides.** C'est une autre famille d'automates [14] dédiée à la spécification de systèmes réactifs comportant des composantes à variations continues en plus de composantes à variations discrètes. Typiquement, ce sont des systèmes composés d'un environnement physique sous contrôle d'un processus logiciel. La principale différence entre automates hybrides et automates temporisés est que le modèle des automates hybrides intègre, en plus des horloges, d'autres variables pour décrire des évolutions continues. Ces évolutions se traduisent par des équations différentielles. Les automates hybrides offrent donc un pouvoir d'expressivité plus grand que les automates temporisés.

L'automate hybride illustré sur la FIG. 1.13 représente le fonctionnement d'un thermo-

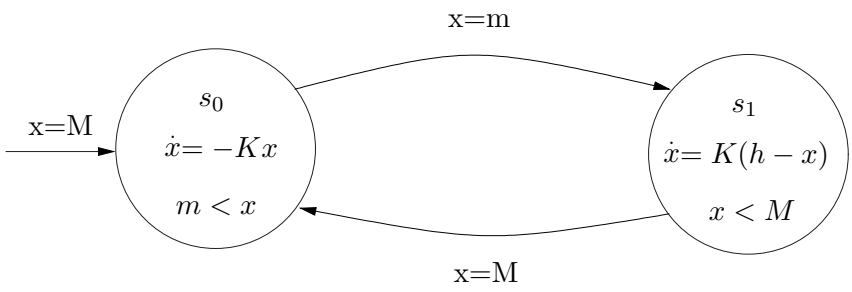


FIGURE 1.13 – Automate hybride représentant le fonctionnement d'un thermostat.

stat [14]. L'idée consiste à maintenir la température d'une enceinte, représentée par la variable continue x, entre un seuil minimal m et un seuil maximal M. Elle diminue suivant la fonction exponentielle $x(t) = \theta e^{-Kt}$, où t dénote le temps, θ est la température initiale et K est une constante qui dépend des caractéristiques de l'enceinte. Lorsque le thermostat est allumé, la température varie selon la fonction $x(t) = \theta e^{-Kt} + h(1-e^{-Kt})$, avec h une constante qui dépend de la puissance du thermostat. Dans l'état s_0 de l'automate, le thermostat est éteint et la diminution de la température est décrite par l'équation différentielle $\dot{x}= -Kx$. Dans s_1, le thermostat est allumé et la montée de la température est décrite par l'équation $\dot{x}= K(h-x)$. Les invariants associés respectivement à s_0 et s_1 sont $m < x$ et $x < M$ et les transitions entre les deux états sont gardées par des conditions sur x.

On retiendra qu'il existe aussi des *I/O automates hybrides*, toujours développés par N. Lynch et son équipe [154]. En ce qui concerne la technologie autour des automates hybrides, on peut mentionner l'outil de *model checking* HYTECH [119], développé par T. Henzinger et son équipe à l'université de Californie, à Berkeley.

Nous noterons que certains supports d'approches basées sur des évolutions explicites d'états sont combinés avec une *logique temporelle* pour étudier les comportements d'un système. Par exemple, l'outil KRONOS basé sur les automates temporisés utilise l'extension temporisée de la logique temporelle CTL, appelée *TCTL* (*Timed Computation Tree Logic*) [13].

Approches basées sur des évolutions implicites d'états. Pour illustrer cette famille de méthodes, nous distinguons tout d'abord les méthodes basées sur les réseaux de Petri. Ensuite, nous introduisons les démarches basées sur l'utilisation de langages spécifiques, l'approche GIOTTO d'une part, et les langages synchrones d'autre part.

- **Approches fondées sur les réseaux de Petri.** Les réseaux de Petri (ou RdP) figurent parmi les premiers concepts définis pour décrire des systèmes avec des comportements réactifs et concurrents. En particulier, ils offrent, au même titre que les GRAFCET [68], un support pour la spécification et la conception de systèmes temps réel automatisés.

  Définition 2 *Un réseau de Petri $\mathcal{R}$ est un quadruplet $< P, T, A, M >$ où :*
  – *P dénote un ensemble fini de places (qui contiennent des jetons),*
  – *T est un ensemble fini de transitions (qui indiquent comment les jetons passent d'une place à l'autre),*
  – *$A \in (P \times T) \cup (T \times P)$ représente l'ensemble des arcs,*
  – *$M_{init} : P \longrightarrow \mathbb{N}_+$ est un marquage initial dans $\mathcal{R}$ (c'est-à-dire, les jetons contenus dans les places initialement).*

□

Les RdP ont une sémantique formelle bien définie qui évite toute ambiguïté dans les spécifications. Dans un RdP, les places peuvent contenir des *jetons* (ou *tokens*) qui doivent franchir des transitions pour changer de place. Les arcs associent aux différentes transitions des places en amont et en aval de la transition.

FIGURE 1.14 – Évolution d'un *réseau de Petri de base*.

La FIG. 1.14 illustre la règle d'évolution d'un *RdP de base*[16]. Le *tirage* d'une transition (t) requiert au moins un jeton dans toutes les places placées en amont (p_1 et p_2). Une fois la transition effectuée, les jetons sont retirés des places en amont, et ajoutés à toutes les places en aval (p_3 et p_4). Implicitement, chaque marquage d'un RdP peut être vu comme un état de l'automate sous-jacent ; et les transitions ont lieu lors des changements de places par les jetons.

Les descriptions à l'aide de RdP fournissent ainsi des représentations graphiques qui facilitent la compréhension des spécifications. Les modèles résultants peuvent être exécutés, et le système décrit devient analysable. Cependant, un gros inconvénient des RdP réside dans le fait que les modèles ont vite tendance à croître rapidement dès que le système considéré est complexe.

Dans [33], les auteurs donnent un aperçu de différentes variantes de RdP. En particulier, nous mentionnons les *extensions temporisées* dont celles proposées par Ramchandani [178] et Sifakis [190] figurent parmi les toutes premières. Les exécutions dans un RdP de base (i.e. les franchissements des transitions) se font de façon instantanée, donc sans aucune consommation de temps. Cela est suffisant pour permettre l'étude de propriétés qualitatives d'un système (par exemple, synchronisation ou causalité), mais pas les propriétés quantitatives de celui-ci (typiquement, les performances du système). Pour prendre en compte l'aspect quantitatif dans les RdP, des informations de temps sont associées aux transitions, aux places, aux arcs, aux jetons, ou à une combinaison de ces éléments du RdP. Dans [178], Ramchandani associe une "durée de tirage" à chaque transition (i.e., la quantité de temps que met le système pour tirer la transition). Sifakis quant à lui, attribue aux places des "durées de détention" de jetons [190] (i.e., un jeton doit rester dans une place pendant cette quantité de temps, avant de pouvoir provoquer le tirage d'une transition). Parmi les études récentes sur les RdP temporisés, nous pouvons citer les travaux de Parosh et Nylén [2], où ils définissent un algorithme ainsi qu'un prototype pour vérifier des propriétés de sûreté. Les RdP qu'ils utilisent sont caractérisés par des jetons possédant chacun une horloge représentant son "âge". Dans [60], Cassez et Roux proposent eux une traduction de RdP à transitions temporisées en automates temporisés. Un avantage immédiat de cette traduction est l'accès à la technologie des automates temporisés.

16. Il existe plusieurs types de RdP.

- **Approche Giotto.** L'approche GIOTTO [120] a été définie par Thomas A. Henzinger et son équipe, à l'université de Californie à Berkeley. Elle offre un modèle de programmation abstrait pour les systèmes de contrôle embarqués, à contraintes temps réel strictes.

  Cette approche est basée sur un langage du même nom, *giotto* [121]. Le modèle *giotto* d'une application est assez proche d'une description METAH. Il consiste en un ensemble de *tâches*, essentiellement périodiques, regroupées au sein d'un certain nombre de *modes*. Les tâches s'exécutent de façon concurrente au sein de chaque mode. Les communications sont effectuées via des *ports* (variables typées) et des *drivers*. Ces derniers jouent le rôle de routeurs et relient les ports des tâches et des modes entre eux.

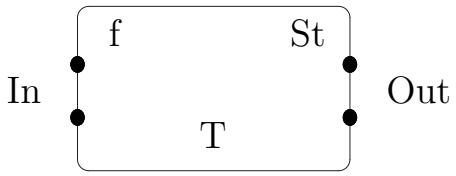


FIGURE 1.15 – Tâche *giotto*.

  Sur la FIG. 1.15, *In* (resp. *Out*) dénote l'ensemble des ports d'entrée (resp. de sortie). Une tâche T est caractérisée par un *état*, représenté par *St*, qui est vu comme un ensemble de ports privés, dont les valeurs sont inaccessibles en dehors de la tâche. Par ailleurs, chaque tâche possède une fonction *f*, qui calcule la valeur des ports de sortie et de l'état suivant de la tâche, à partir de la valeur des ports d'entrée et l'état courant de la tâche. Cette fonction est mise en œuvre par un programme séquentiel dans un langage de programmation quelconque. Aucun point de synchronisation n'apparaît dans *f*. Toutes les synchronisations sont faites à l'extérieur des tâches.

  À tout instant, ce sont les modes actifs au sein de l'application qui indiquent les tâches susceptibles de s'exécuter. Celles-ci sont activées de façon régulière dans le temps. C'est pour cela que la sémantique de *giotto* est qualifiée de *time-triggered* [139] (c'est-à-dire "à déclenchement temporel"). L'avantage majeur d'une telle sémantique est la prédictibilité des comportements temporels du système. Ainsi, la mise en œuvre des fonctions associées aux tâches est accompagnée d'annotations concernant leurs temps d'exécution. L'approche GIOTTO se base sur le modèle temporisé pour la conception des systèmes temps réel. Ce choix est en partie motivé par le souci de s'affranchir des contraintes liées aux plates-formes d'implantation. Le compilateur associé au langage utilise des annotations (exemples : temps d'exécution au pire cas, directives d'ordonnancement des tâches de la part de l'utilisateur) comme informations permettant de vérifier les propriétés d'un système (par exemple, l'ordonnançabilité).

- **Approche synchrone.** C'est l'approche qui est adoptée dans cette thèse. On aura l'occasion d'y revenir plus en détail dans le chapitre 3. On mentionnera simplement que les langages synchrones [30] [109] permettent de spécifier et concevoir des systèmes temps réel. Ils ont une sémantique formelle qui est essentielle à la validation. Les langages synchrones sont souvent classés en deux familles : les langages *impératifs* et les langages orientés *flot de données*, déclaratifs. Dans la première famille, on peut citer ESTEREL [36] et les variantes synchrones de STATECHARTS [115] qui permet la définition d'une hiérarchie d'automates dans une représentation graphique pouvant exprimer la concur-

rence et la communication, comme ARGOS [157] ou SYNCCHARTS [17]. Les langages LUSTRE [111], LUCID SYNCHRONE [59], et SIGNAL [32] sont orientés flot de données.

1.3 Résumé

Les systèmes temps réel sont des systèmes dont la correction dépend aussi bien de leurs propriétés fonctionnelles que de leurs contraintes temporelles. Deux familles de systèmes temps réel sont distinguées : les systèmes *stricts* où la violation d'une contrainte entraîne des conséquences catastrophiques (en vies humaines ou en termes économique), et les systèmes *souples* qui sont plus "tolérants" vis-à-vis d'un dysfonctionnement donné. La prise en compte du temps dans leur conception est réalisée à travers trois modèles : *i)* le *modèle asynchrone* dans lequel le système est décomposé en tâches dont l'exécution doit respecter des échéances et éventuellement des périodes, sous le contrôle d'un système d'exploitation ou d'un noyau temps réel (utilisables également dans des approches basées sur les deux autres modèles) ; *ii)* le *modèle temporisé* où le temps d'exécution est fixé *a priori*, facilitant ainsi l'analyse des comportements temporels du système ; *iii)* et enfin, le *modèle synchrone* dans lequel le temps d'exécution est totalement abstrait, et seules la simultanéité et les précédences des événements observables du système sont importantes. Cette abstraction du temps dans le modèle synchrone offre un niveau de description suffisant pour aborder des propriétés comportementales pertinentes d'un système temps réel, tout en s'affranchissant des contraintes liées à une plate-forme de mise en œuvre particulière. Par ailleurs, nous pouvons observer que dans les approches "asynchrones", spécification / conception et implantation sont intimement liées alors qu'elles sont découplées dans les approches "synchrones". Notre étude se base sur le modèle synchrone.

À travers le panorama, non exhaustif, présenté dans ce chapitre, nous avons essayé de donner une idée de la diversité des méthodologies de conception des systèmes temps réel. En particulier, nous avons insisté sur les *approches orientées composants* ainsi que les *approches formelles*. La conception orientée composants présente l'avantage de faciliter la construction de systèmes, grâce notamment à la modularité et la réutilisation. L'utilisation des méthodes formelles dans le développement d'un système offre des bases mathématiques pour des raisonnements solides. Cela est indispensable à l'analyse et à la validation de systèmes critiques. Toutes ces approches reposent sur des modèles de haut niveau qui permettent de décrire les systèmes.

Si nous avons pu observer une grande diversité des approches de haut niveau existantes pour le développement des systèmes temps réel, les concepts de mise en œuvre demeurent quant à eux, communs à toutes ces approches. Dans le chapitre qui suit, nous rappelons en quoi consistent ces concepts, dont nous nous servons aussi par la suite.

Chapitre 2

Éléments de mise en œuvre

Nous introduisons ici des éléments fondamentaux pour la mise en œuvre des systèmes temps réel. L'utilisation des concepts ainsi décrits se place donc au niveau de la phase dite d"'implantation" dans le cycle en V, présenté au chapitre 1, en section 1.2.1.

Le langage d'assemblage a longtemps été utilisé pour programmer des systèmes temps réel (c'est encore le cas aujourd'hui dans l'usage de micro-contrôleurs pour applications en grande série). Ensuite, des langages un peu plus éloignés de la machine, comme C, ont été préférés au langage d'assemblage, de trop bas niveau. La mise en œuvre d'applications temps réel complexes, à l'aide de ces langages, inclut souvent l'utilisation de services d'un exécutif temps réel pour la gestion des tâches. Ainsi, une grande partie des applications industrielles repose sur une telle solution. Enfin, des langages plus évolués intégrant des concepts de tâche, de communication et de synchronisation ont vu le jour. C'est le cas de langages comme Ada ou plus récemment Real-Time Java. Les langages synchrones, que nous considérons dans cette thèse, s'inscrivent aussi dans une modélisation de haut niveau.

Ce chapitre est organisé de la façon suivante :

- Il fixe d'abord un cadre terminologique pour les notions qui seront utilisées par la suite. Ainsi, la section 2.1 introduit les notions de *tâches* et de *ressources*. Elle en présente notamment les caractéristiques. Les *communications* sont abordées à travers le modèle d'architecture appelé *Time-Triggered Architectures* (tta).
- Il présente ensuite l'ordonnancement des tâches, à savoir, les algorithmes qui constituent la base des études actuelles sur l'ordonnancement, ainsi que les techniques d'analyse de l'ordonnancement dans un système (section 2.2).
- Les *intergiciels*, qui virtualisent les aspects systèmes au travers d'interfaces de plus haut niveau sont l'objet de la section 2.3.
- Les langages de programmation d'applications temps réel sont également abordés à travers deux exemples que sont Ada et Real-Time Java (section 2.4).

2.1 Notions de tâches et de ressources

2.1.1 Tâches

Une tâche peut être définie comme un ensemble d'actions atomiques (le code), ordonnées dans le temps et effectuées lors de l'exécution d'un programme. Elle est séquentielle et dispose d'un *contexte*, formé par l'ensemble des informations que les actions associées peuvent consulter ou modifier.

Tâches périodiques et apériodiques. Toute exécution d'une tâche nécessite au préalable son *activation* (c'est-à-dire, l'occurrence d'un événement qui dénote une demande de mise en route). Les tâches peuvent être classifiées selon la prédictibilité de leurs dates d'activation.

Dans les systèmes temps réel, certaines tâches sont exécutées de façon répétitive. Par exemple dans un avion, on peut vouloir ajuster la vitesse et l'altitude chaque 100 millisecondes. Ces informations provenant de capteurs seront utilisées par des tâches *périodiques* qui contrôlent les ailerons, le gouvernail ou la propulsion, afin de maintenir la stabilité de l'appareil. La périodicité de ces tâches est connue du concepteur, et les instants d'activation de ces tâches peuvent être fixés à l'avance. Ainsi, une tâche périodique est caractérisée par une période d'activation notée T, qui représente le nombre d'unités de temps séparant deux activations consécutives de celle-ci.

À l'inverse, certaines tâches sont *apériodiques*. Elles sont activées occasionnellement. Par exemple, lorsque le pilote d'un avion veut changer de direction pendant un vol, les tâches associées à cette action se retrouvent activées. Par leur nature, les tâches apériodiques ne peuvent être prédictibles dans le temps. De plus, ce sont souvent les plus urgentes (par conséquent, il faut disposer de suffisamment de ressources en réserve pour les exécuter aux moments opportuns). Les tâches apériodiques dont l'intervalle de temps entre deux activations successives est borné sont appelées tâches *sporadiques*. On qualifie de temps minimum d'inter-arrivée, la quantité de temps inférieure à toute durée écoulée entre deux activations consécutives d'une tâche sporadique.

Attributs / paramètres d'une tâche. Les tâches sont par ailleurs caractérisées par un certain nombre d'attributs servant d'informations pendant l'exécution. Parmi ces attributs, on retrouve souvent la *priorité*, la *périodicité* et l'*échéance* (en anglais, *deadline*).

La priorité permet d'élire une tâche parmi un ensemble de tâches susceptibles de s'exécuter. La priorité d'une tâche peut être *statique* ou *dynamique*. Dans le premier cas, la priorité affectée à la tâche est fixe durant toute l'exécution du programme tandis que dans le second, elle peut varier. Ces variations dépendent de divers facteurs tels que l'accès aux ressources allouées aux tâches. La périodicité d'une tâche précise la nature de son activation (exemple : périodique, apériodique). L'échéance indique la date à ne pas dépasser lors de l'exécution d'une tâche. Dans les systèmes temps réel stricts, le non respect d'une échéance conduit à des conséquences catastrophiques. Cela n'est pas le cas dans les systèmes temps réel souples puisque le manquement de l'échéance n'entraînera qu'un rendement de qualité inférieure. L'échéance d'une tâche périodique ou apériodique peut être soit *absolue* (elle est représentée par une date), soit *relative* à la date d'activation de celle-ci (exprimée par une durée égale à la différence entre sa date d'activation et celle de son échéance absolue).

États d'une tâche. Au sein d'un système, une tâche peut être dans divers *états*. Nous distinguons les suivants :

- dans l'état *dormant* (en anglais, *idle*), la tâche est en mémoire mais elle n'est pas considérée par les opérations d'ordonnancement ;
- dans l'état *prêt* (en anglais, *ready*), la tâche est en attente de ressource CPU pour s'exécuter ;
- dans l'état *en exécution* (en anglais, *running*), la tâche est en train de s'exécuter ;
- dans l'état *en attente* (en anglais, *waiting*), la tâche est en attente d'un signal ou d'une ressource afin de poursuivre son exécution.

2.1.2 Ressources

C'est l'ensemble des dispositifs matériels dont les tâches ont besoin pour accomplir les actions qui leur sont associées. Parmi ces dispositifs, on peut mentionner les mémoires, les registres et les bus. Les *processeurs* sont des ressources spéciales, évidemment indispensables à l'exécution de n'importe quelle tâche dans le système.

La gestion des ressources dans les système temps réel est l'un des aspects les plus critiques dans leur conception. En effet, l'accès à une ressource donnée est très souvent restreint à une tâche au plus, c'est le principe d'*exclusion mutuelle*. D'autre part, pour éviter qu'une tâche monopolise indéfiniment une ressource partagée (ce qui pourrait ainsi entraîner un non respect des échéances par d'autres tâches en attente), un mécanisme de *préemption* est proposé. Grâce à ce dernier, l'utilisation d'une ressource par une tâche peut être suspendue temporairement à tout moment pour reprendre ultérieurement. Enfin, deux phénomènes caractéristiques engendrés par le partage de ressources critiques au sein d'un système sont l'*inversion de priorité* et l'*interblocage*. Ils sont abordés dans la section 2.2.2.3.

2.1.3 Communications

La mise en œuvre d'un système temps réel à grande échelle consiste souvent en un partitionnement de celui-ci en sous-systèmes, qui doivent interagir en communicant des informations (données) pour accomplir la mission affectée au système global. Ce découpage doit impérativement satisfaire des contraintes telles que le respect des propriétés temporelles (le système doit être capable d'acquérir et de traiter des événements dans les bons délais), la modularité (elle facilite l'adaptabilité du système, d'où une certaine souplesse) ou la maintenabilité (l'effort nécessaire à la localisation d'erreurs de fonctionnement ainsi que leur correction doit être minimisé). Dans cette section, nous présentons essentiellement le modèle d'architecture TTA (*Time-Triggered Architecture*) [139], dû à H. Kopetz et ses collègues du projet MARS (*Maintainable Real-Time System*) au *Technische Universität* de Vienne (Autriche).

Le modèle TTA. Ce modèle d'architecture permet de concevoir des systèmes distribués temps réel à grande échelle suivant une démarche constructive. Les sous-systèmes (ou nœuds) formant un système présentent la particularité d'être quasi autonomes. Leur intégration dans le système selon le modèle TTA préserve leurs propriétés initiales : on parle d'architectures *composables*. Par exemple, on rencontre des systèmes basés sur ce modèle dans le domaine de l'automobile (comme les systèmes de contrôle du freinage ou de la direction assistée). Les architectures TTA comportent deux principaux traits caractéristiques : d'une part la *notion commune de temps* d'elles imposent à tous les sous-systèmes, et d'autre part, la prise en compte d'*interfaces* via lesquelles les nœuds interagissent.

Dans un système distribué temps réel, chaque site possède en général une horloge physique locale qui, en pratique diffère de celle du voisin (il est donc indispensable de disposer d'un mécanisme global permettant de coordonner les activités des différents sites afin de garantir le fonctionnement souhaité). Ainsi, dans une architecture TTA, tous les sous-systèmes sont supposés avoir accès à un temps global basé sur une échelle de temps physique. L'émission de signaux de contrôle (pour activer par exemple les tâches au sein des sous-systèmes) est effectuée sur la base de ce temps global. De plus, ce dernier doit être tolérant aux pannes puisque toute défaillance de celui-ci est susceptible d'avoir de graves répercussions sur le fonctionnement du système entier (en effet, tous les sous-systèmes en dépendent).

Les communications entre un sous-système et le reste du système sont effectuées via des interfaces. À un haut niveau d'abstraction, ces dernières sont constituées de *firewalls* temporels empêchant les sous-systèmes d'échanger des signaux de contrôle avec leurs voisinage. Chaque *firewall* comporte un certain nombre d'informations connues des sous-systèmes concernés par l'interface qui le contient : les noms et structures de données qui transitent dans ce *firewall* ; les instants de l'horloge globale auxquels les sous-systèmes doivent accéder à celui-ci ; et la durée de vie des données dans le *firewall*. Le système de communication se charge de connecter les interfaces entre elles. De plus, il assure le transport de données d'une interface à une autre dans des délais bornés déterministes, connus *a priori*. Les instants auxquels ces données sont récupérées au niveau d'une interface émettrice et délivrées aux interfaces réceptrices sont spécifiés dans des tables. Ces dernières sont remplies avant l'exécution et elles sont connues de toutes les entités communiquantes. La conséquence immédiate de cette mise en œuvre des communications est que toute interface peut être caractérisée (i.e., ses propriétés temporelles ainsi que le format des données échangées via celle-ci peuvent être spécifiés) sans aucune contrainte de connaissance du type d'activité au niveau des sous-systèmes. Ce qui favorise l'hétérogénéité du système global.

Le modèle d'architecture LTTA (*Loosely Time-Triggered Architecture*) est une variante de TTA qui n'impose pas une synchronisation strictes aux horloges locales des nœuds. Dans ce modèle : *i)* les accès au bus de communication se font de façon "quasi-périodique" [1] et non bloquante ; *ii)* les lectures et écritures sont effectuées indépendamment à chaque extrémité du bus en phase avec les horloges locales associées ; *iii)* le bus se comporte comme une mémoire partagée, c'est-à-dire, il conserve les valeurs qui sont périodiquement rafraîchies suivant une horloge locale. Un tel fonctionnement peut donc facilement induire des problèmes de communications, notamment à cause des risques de pertes de messages. Dans [31], Benveniste et al. proposent un protocole robuste de communication qui permet d'éviter les pertes de messages entre des entités d'une architecture de type LTTA. Ils illustrent également une utilisation de ce protocole dans le cadre d'un déploiement de programmes synchrones sur des architectures distribuées.

*
* *

Nous observons que le modèle TTA est très intéressant pour la conception des systèmes distribués temps réel à cause de ses caractéristiques comme la composabilité, la flexibilité offerte son système de communication vis-à-vis des activités associées aux sous-systèmes, l'existence d'une horloge globale tolérante aux pannes qui rythme le fonctionnement du système entier. Cela favorise la prédictibilité de ce dernier et permet de garantir plus facilement les propriétés attendues. Cependant, d'autres modèles tels que LTTA montrent que la synchronisation stricte (présente dans TTA) n'est pas indispensable pour assurer la robustesse des communications dans un système distribué temps réel.

2.2 Ordonnancement des tâches

Le problème de l'ordonnancement dans un système consiste en la définition d'un *ordonnanceur*, c'est-à-dire le module chargé d'affecter les ressources disponibles aux tâches qui composent le système, de façon à ce qu'elles s'exécutent tout en respectant les contraintes temporelles qui leur sont imposées. De nombreux travaux ont été accomplis sur la question de l'ordonnancement des tâches dans les systèmes temps réels [153] [198]. À ce sujet, la *validité* (aucune tâche ne

1. On entend par *quasi-périodique* le fait que les différentes horloges suivant lesquelles les écritures, lectures et mises à jour de valeurs effectuées dans le bus ne sont pas synchronisées.

manquera son échéance) et l'*optimalité* (l'ordonnancement proposé est le meilleur parmi les ordonnancements valides) sont deux critères importants pour décider de la qualité d'un algorithme d'ordonnancement (en pratique, on se contente souvent d'ordonnancements valides).

2.2.1 Caractéristiques générales

Dans cette section, nous précisons certaines caractéristiques [2] permettant de classifier les algorithmes d'ordonnancement. Les définitions données sont fortement inspirées de celles que donne Buttazzo [57].

Statique vs dynamique. Un algorithme d'ordonnancement est dit *statique* lorsque la prise de décisions concernant l'allocation de ressources aux tâches est basée sur des paramètres fixes, en particulier, les priorités qui leur sont affectées avant l'exécution du système.

Nous qualifions un algorithme de *dynamique* lorsque l'attribution des ressources aux tâches repose sur des paramètres variables lors de l'évolution du système.

Hors ligne vs en ligne. Nous parlons d'algorithme *hors ligne*, lorsque les décisions d'ordonnancement sont prises avant l'exécution du système (typiquement durant la phase de compilation). La mise en œuvre d'un tel algorithme nécessite la connaissance de certaines informations : les paramètres de toutes les tâches, une estimation des temps d'exécution au pire cas de celles-ci, les contraintes de *précédence* ou d'exclusion mutuelle. L'ensemble de ces informations est exploité au sein d'une structure de données figée, généralement une table (*table-driven* [139]), qui est consultée régulièment pour choisir la tâche qui doit s'exécuter. En général, ce choix ne requiert pas beaucoup de temps. Les systèmes temps réel stricts ont longtemps utilisé ce type d'algorithme.

Les algorithmes d'ordonnancement hors ligne ont plusieurs avantages [137] :

- les ordonnancements obtenus sont souvent efficaces ;
- ils peuvent réduire significativement les surcharges dues aux changements de contexte ;
- la mise en œuvre de l'exclusion mutuelle pour les accès aux ressources ne nécessite pas de recours aux *sémaphores* ;
- ils favorisent la prédictibilité (déterminisme) du système, ce qui favorise les détections d'erreurs au plus tôt.

Cependant, les algorithmes d'ordonnancement hors ligne sont souvent insuffisants [137] :

- dans un système ordonnancé suivant ces algorithmes, il est souvent nécessaire de revoir l'ordonnancement entier du système après la moindre modification des caractéristiques d'une tâche quelconque ;
- ils sont plutôt adaptés à des systèmes avec un petit nombre de tâches ;
- ils ne conviennent pas aux tâches sporadiques ou apériodiques ;
- enfin, du fait de certains facteurs liés à l'exécution effective des fragments de code (comme les interruptions), il peut en résulter un sur-dimensionnement dans les estimations *a priori* (par exemple, dans le calcul des temps d'exécution au pire cas).

Dans un algorithme *en ligne*, la prise des décisions concernant l'allocation des ressources aux tâches est effectuée à l'exécution du système. Il s'agit d'un algorithme flexible qui s'adapte

2. Dans la littérature, on trouve diverses définitions de ces mêmes notions, menant quelques fois à des confusions. À ce sujet, L. Cucu présente, dans sa thèse [73], une vision synthétique des définitions courantes.

au scénario d'évolution des tâches. Pour cela, l'état courant du système global doit être connu à tout moment (les tâches activables, les ressources affectables, etc.). Cependant, le choix des tâches qui doivent s'exécuter, souvent guidé par les priorités (*priority-driven* [153]), requiert une quantité de temps qui peut être non négligeable.

*
* *

Nous pouvons noter qu'un algorithme d'ordonnancement hors ligne peut être statique mais pas dynamique. Ce n'est pas le cas d'un algorithme en ligne qui peut, lui, être dynamique ou statique.

Préemptif vs non préemptif. Un algorithme d'ordonnancement est qualifié de *préemptif*, si une tâche en cours d'exécution peut se voir retirer le processeur au profit d'une autre tâche prête à s'exécuter et de priorité plus forte. Les algorithmes utilisés pour les systèmes temps réel admettent souvent la préemption. Cette dernière doit cependant être appréhendée avec précautions, notamment par rapport à l'estimation de son coût, pour assurer qu'un système ordonnançable respectera bien un fonctionnement temps réel correct.

Dans un algorithme *non préemptif*, une tâche détenant le processeur s'exécute jusqu'à ce qu'elle termine complètement son exécution. Ce genre d'algorithme est surtout intéressant pour des systèmes où le temps d'exécution des tâches est court comparativement au temps consommé par les changements de contexte. Par la suite, nous ne nous intéresserons qu'aux algorithmes préemptifs.

2.2.2 Algorithmes d'ordonnancement

2.2.2.1 Algorithmes statiques

Ordonnancement statique cyclique. L'algorithme d'ordonnancement *statique cyclique* (en anglais, *static cyclic scheduling*) est une technique ancienne utilisée depuis les années soixante. Celle-ci consiste à stocker un scénario d'exécution pré-établi des tâches dans une structure de données, appelée *échéancier*. L'évolution du système va ainsi reposer sur ce scénario. L'avantage d'un tel ordonnancement est qu'il est très facile à mettre en œuvre. En particulier, il est adéquat pour des systèmes où les tâches sont strictement périodiques et où les priorités sont égales aux échéances. Par contre, une limitation majeure est liée aux contraintes spatiales à respecter pour implanter l'échéancier. En particulier, cela peut poser quelques difficultés dans le cas des systèmes embarqués, au sein desquels la taille de la mémoire disponible est souvent restreinte. Une solution à ce problème consiste alors à mettre les tâches en séquence.

Ordonnancement par contraintes de précédence. Cette technique consiste à spécifier de façon statique des contraintes de *précédence* selon lesquelles les tâches doivent s'exécuter. Ces contraintes peuvent être exprimées à travers un *graphe de précédence* [87] comme sur la Fig. 2.1 (chaque T_i représente une tâche, et les m_j correspondent aux messages échangés). Ce type d'ordonnancement est utile par exemple pour décrire un système comportant un mécanisme *pipeline*, où, après réception des données d'entrée, il y a des communications successives entre les tâches regroupées en chaîne, jusqu'à l'émission des données de sortie par la dernière tâche.

Enfin, une technique reposant sur la notion de contraintes de précédence a été proposée par [217]. Elle comprend deux étapes : d'abord, les auteurs établissent un ordonnancement statique

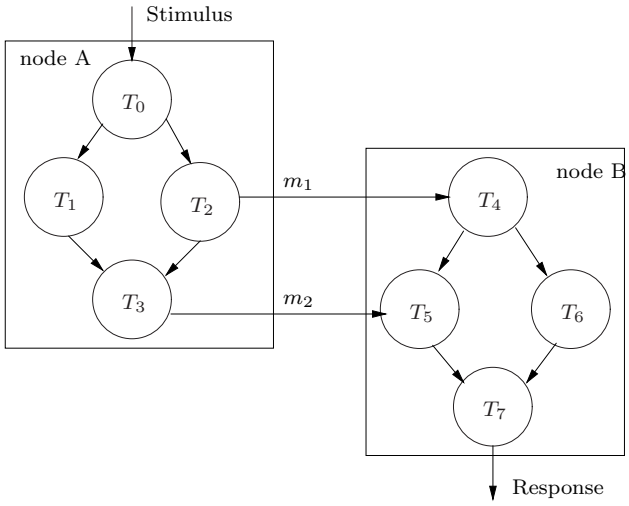


FIGURE 2.1 – Graphe de précédence associé à un ensemble de tâches distribuées.

préliminaire, où la politique est telle que c'est la tâche ayant l'échéance la plus proche qui est prioritaire (cette politique [3] appelée E.D.F., est présentée dans la section 2.2.2.2) ; ensuite, ils modifient l'ordonnancement en considérant des heuristiques pour essayer de satisfaire des contraintes de précédence.

Rate Monotonic Scheduling (R.M.S.) proposé par Liu et Layland [153]. Les auteurs considèrent les hypothèses générales suivantes :

- Aucune tâche ne comporte de section critique non préemptible, et le coût engendré par une préemption est négligeable ;
- Seules les demandes en ressources liées aux calculs sont significatives (la mémoire, les entrées/sorties et tous les autres besoins en ressources sont négligés) ;
- Toutes les tâches sont indépendantes, et il n'y a aucune contrainte de précédence entre elles ;
- Toutes les tâches sont périodiques.

C'est un algorithme hors ligne qui repose sur des priorités statiques. Plus la période d'une tâche est courte, plus elle a une priorité forte. Par exemple, sur la trace de la FIG. 2.2, la tâche $T1$ est la plus prioritaire car sa période, $[E2, E3]$, est inférieure à $[EA, EB]$ qui est celle de la tâche $T2$.

Le modèle de tâche souvent utilisé dans les analyses d'ordonnancement considère l'échéance relative de chaque tâche comme étant identique à la période de celle-ci. L'algorithme fournit alors des résultats très intéressants du point de vue de l'ordonnançabilité et de l'utilisation du processeur. Considérons un ensemble de n tâches périodiques, où chaque tâche identifiée par l'indice i a les caractéristiques suivantes :

- P_i dénote sa période,
- E_i représente son échéance,

3. L'Algorithme défini par E.D.F. consiste à "classer les tâches par échéances croissantes". Généralement, il est considéré comme étant dynamique, pour des tâches dont on ne connaît pas *a priori* les dates d'activation et les échéances. Cependant, il peut être également utilisé statiquement lorsque ces informations sont connues, et c'est le cas dans [217].

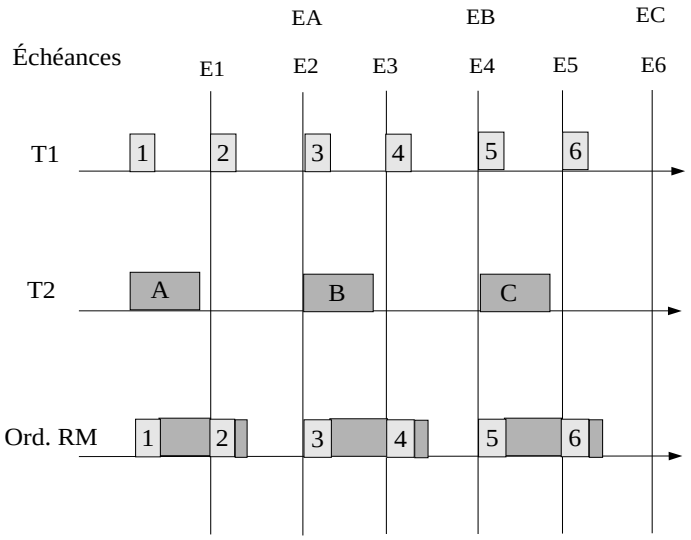


FIGURE 2.2 – Une exécution monoprocesseur suivant la politique R.M.S.

– C_i indique son temps d'exécution au pire cas.

L'ordonnancement des n tâches suivant l'algorithme R.M.S est réalisable si l'utilisation U donnée par la formule $U = \sum_{i=1}^{n} C_i/P_i$ satisfait la propriété suivante [153] :

$$U \leq n(\sqrt[n]{2} - 1)$$

Cette propriété est une condition suffisante, mais pas nécessaire. La condition nécessaire et suffisante a été donnée un peu plus tard par Joseph et Pandhya dans [131]. Elle repose sur le calcul du *temps de réponse au pire cas* R_i d'une tâche T_i. Celui-ci est défini comme étant la somme :

– du temps écoulé entre l'activation et le début de l'exécution de T_i,
– du temps d'exécution effectif de T_i,
– du temps d'attente de T_i au cas où elle est préemptée.

Il est obtenu à l'aide du calcul suivant :

$$R_i = C_i + \sum_{\forall k \in hp(i)} \lceil \frac{R_i}{P_k} \rceil C_k$$

où $hp(i)$ est l'ensemble des tâches de plus forte priorité que la tâche T_i. L'équation est ensuite résolue en utilisant une *relation de récurrence*, sous la forme $R_{n+1} = relation\_rec(R_n)$ ($relation\_rec$ est définie par le terme droit de l'équation ci-dessus). La condition exacte d'ordonnançabilité est alors donnée par :

$$R_i \leq E_i, \forall i$$

L'hypothèse simplificatrice d'égalité entre la période et l'échéance, souvent considérée dans les modèles de tâches (dans [153] notamment), n'est plus nécessaire.

Deadline Monotonic (D.M.) proposé par Leung et Whitehead [151]. Il s'agit d'un algorithme hors ligne préemptif à priorité statique. On suppose que l'échéance relative d'une tâche T_i est au plus égale à sa période ($E_i \leq P_i$). La priorité la plus forte est affectée à la tâche ayant la plus

courte échéance. L'algorithme D.M peut ainsi être vu comme étant équivalent à R.M.S lorsque les échéances et les périodes des tâches coïncident. La FIG. 2.3 illustre une exécution de deux tâches T_1 et T_2, suivant cet algorithme.

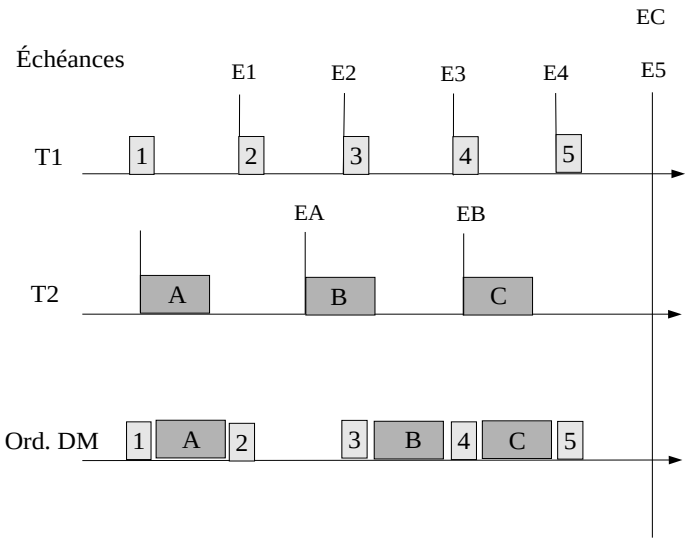


FIGURE 2.3 – Exécution monoprocesseur suivant la politique D.M.

Les auteurs proposent une condition suffisante qui garantit l'ordonnançabilité d'un ensemble de tâches [151]. Elle est donnée par la formule suivante :

$$\sum_{i \in [1..n]} \frac{C_i}{E_i} \leq n(\sqrt[n]{2} - 1)$$

2.2.2.2 Algorithmes dynamiques

Round Robin (ou algorithme du *tourniquet*). Dans cette politique, le système d'exploitation gère les tâches de façon circulaire (en *fifo - First In First Out*). Chacune d'entre elles se voit attribuer les ressources pour s'exécuter. Lorsque la tâche active termine, le système d'exploitation alloue le processeur à la tâche suivante dans la file. Cette politique est très adaptée aux systèmes à temps partagé, où on définit pour les tâches une durée maximale qui correspond au *quantum* de temps d'utilisation du processeur. L'algorithme du tourniquet est aussi utilisé lorsque plusieurs tâches possèdent la même priorité.

Earliest Deadline First (E.D.F.) proposé aussi par Liu et Layland [153]. Les hypothèses considérées ici sont quasi identiques à celles de R.M.S sauf que les tâches ne sont pas nécessairement périodiques. Les priorités sont dynamiques. Les tâches sont classées selon leurs échéances. Lors de chaque ordonnancement, la tâche élue est celle dont l'échéance absolue est la plus proche. La particularité de cette politique est son optimalité par rapport à la minimisation du *retard maximum* (l'intervalle de temps entre la fin d'exécution d'une tâche et son échéance). Il est illustré sur la FIG. 2.4, où nous considérons l'exécution de deux tâches T_1 et T_2.

La condition nécessaire et suffisante d'ordonnançabilité d'un ensemble de n tâches périodiques

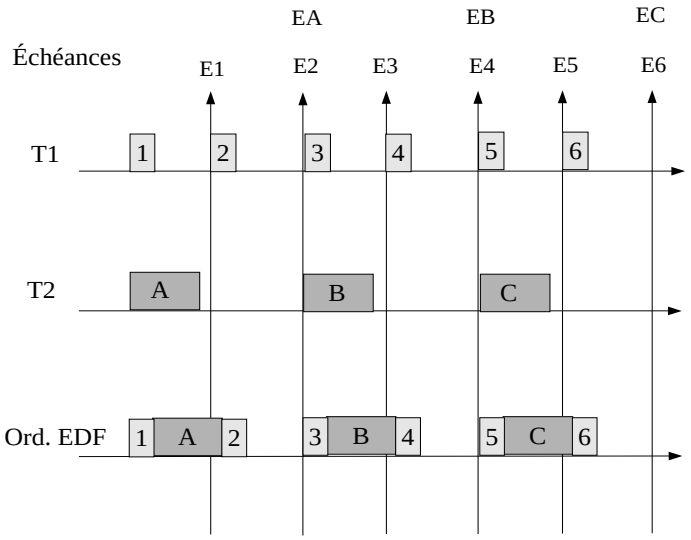


FIGURE 2.4 – Exécution monoprocesseur suivant la politique E.D.F.

suivant l'algorithme E.D.F est donnée par la condition suivante sur l'utilisation :

$$U \leq 1$$

2.2.2.3 Synchronisation des tâches pour le partage des ressources

Les priorités dynamiques sont utiles pour résoudre les problèmes succeptibles de se produire lors des partages de ressources en exclusion mutuelle. Parmi ces problèmes, nous mentionnons les *inversions de priorités* et les *interblocages*.

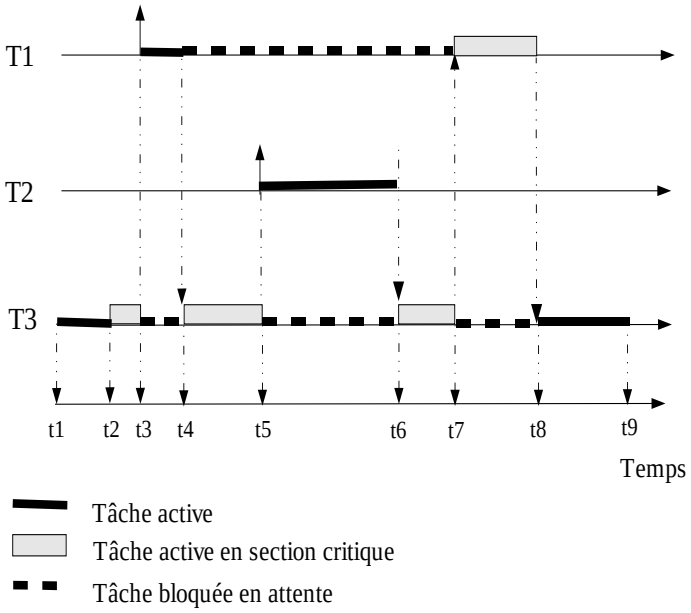


FIGURE 2.5 – Problème de l'inversion de priorité.

L'inversion de priorité survient lorsqu'une tâche tente d'accéder à une ressource critique et n'y parvient pas tout de suite, à cause d'une autre tâche de priorité inférieure. Nous avons représenté une telle situation sur la FIG. 2.5. Elle concerne trois tâches $T1$, $T2$ et $T3$, où $T1$ est plus prioritaire que $T2$ qui est elle-même plus prioritaire que $T3$. La tâche $T1$ (en tant que client) sollicite $T3$ (qui joue le rôle de serveur) pour réaliser une opération sur une ressource particulière ($T2$ n'est pas concernée). Le scénario illustré montre que $T1$, la tâche la plus prioritaire, est retardée durant l'intervalle de temps $[t5,\ t6]$ par une tâche de priorité inférieure, en l'occurrence $T2$. L'activation de cette dernière a causé la préemption de $T3$ alors qu'elle n'avait pas fini son exécution. Ce qui oblige $T1$ à attendre plus longtemps.

L'*héritage de priorité* (en anglais, *priority inheritence*) est la technique souvent utilisée pour résoudre ce problème : la tâche détenant la ressource hérite de la priorité de la tâche requérant la ressource et de plus forte priorité. C'est ce qui est représenté sur la FIG. 2.6. La tâche $T3$ hérite de la priorité de $T1$ à l'instant $t4$. Elle ne peut donc plus être préemptée par $T2$ à l'instant $t5$, contrairement au scénario de la FIG. 2.5. Enfin, elle reprendra sa priorité initiale en sortie de section critique, c'est-à-dire à l'instant $t6$. La tâche $T1$ peut alors poursuivre son exécution sans être retardée par $T2$. Grâce à ce mécanisme d'héritage assez simple, l'inversion de priorité peut être évitée. Cependant, d'après V. Yodaiken (FSMLabs - *Finite State Machine Labs*), cette technique est loin d'être la solution adéquate. Il en donne plusieurs raisons dans [218].

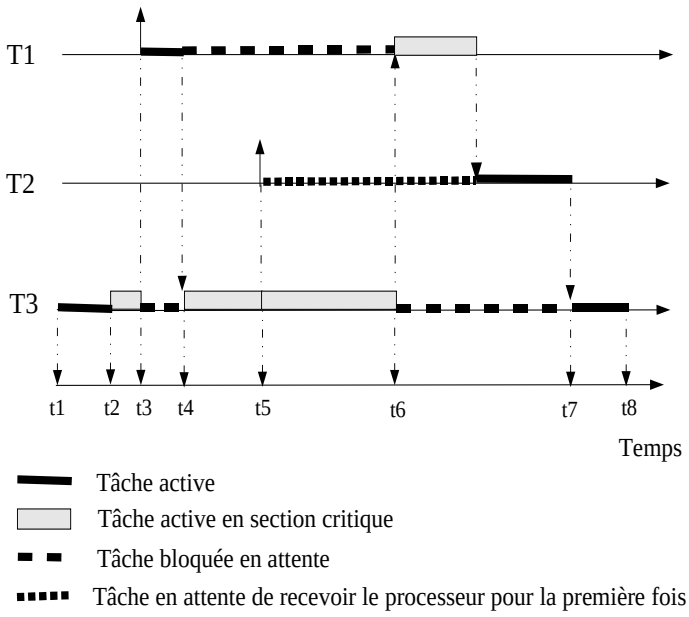


FIGURE 2.6 – Héritage de priorité.

L'héritage de priorité ne prévient pas l'interblocage entre tâches (un ensemble de tâches est en interblocage si chaque tâche attend un événement que seule une autre tâche de l'ensemble peut engendrer). Le protocole de la *priorité plafonnée* (en anglais, *priority ceiling*) permet de traiter de façon efficace les situations d'interblocage entre tâches. Considérons une tâche T_i et une ressource r, le principe de cette technique peut se résumer de la manière suivante :

- on associe à chaque ressource r une "priorité maximale" parmi les priorités des tâches qui peuvent utiliser r ;
- une tâche T_i qui détient une ressource r hérite de la priorité maximale des tâches plus

prioritaires qu'elle qui sont bloquées sur r, jusqu'à ce que T_i libère r ;
– par contre, la tâche T_i ne pourra acquérir la ressource r que si la priorité associée à r est strictement supérieure à celles de toutes les ressources déjà possédées par les autres tâches du système.

2.2.3 Analyse et mise en œuvre

2.2.3.1 Méthodes d'analyse

On distingue principalement deux méthodes concernant l'étude de l'ordonnançabilité d'un système. Dans la méthode *analytique*, le système est abstrait via ses paramètres. En outre, on considère un certain nombre d'hypothèses : l'ordonnanceur est déterministe, les temps d'exécution au pire cas des tâches sont connus *a priori*, etc. Les tâches du système sont spécifiées au travers de leurs caractéristiques (exemples : périodicité, échéance). Il s'agit ensuite de vérifier que ces paramètres satisfont ou non des conditions semblables à celles que nous avons illustrées pour les algorithmes dynamiques dans la section 2.2.2.2. Des analyses d'ordonnançabilité basées sur cette méthode sont présentées dans [153], [151], [136] ou [189]. Dans [24] et [56], d'autres analyses tenant compte de l'impact des caches mémoires sur l'ordonnancement des tâches dans les systèmes temps réel sont proposées. Les principaux inconvénients de cette méthode proviennent du nombre d'hypothèses et d'approximations pris en compte. En pratique, le modèle analysé est souvent éloigné du système réel, et cela peut entraîner des résultats pessimistes.

La seconde méthode repose plutôt sur la construction d'un modèle "explicite", moins abstrait que dans la première méthode. Ce modèle décrit de façon détaillée l'ensemble des comportements possibles du système à l'aide de formalismes tels que les automates temporisés ou hybrides, ou bien les réseaux de Petri. Ensuite, l'ordonnançabilité du système peut être étudiée à l'aide d'algorithmes définis sur ces modèles. Un exemple d'étude utilisant les automates temporisés est celui de TAXYS [65]. Les auteurs montrent comment l'outil KRONOS [219], qui sert à faire du *model checking*, peut aider à vérifier l'ordonnançabilité dans une application temps réel embarquée. Un autre exemple concerne le langage METAH, dont l'environnement de conception propose des automates hybrides comme support pour analyser l'ordonnançabilité du système décrit. Dans cette seconde méthode, une difficulté majeure concerne la construction de modèles adéquats.

2.2.3.2 Mise en œuvre

L'ordonnancement est mis en œuvre le plus souvent à travers une tâche spécifique appelée *serveur des tâches*. C'est une tâche à laquelle on donne généralement une priorité très forte. Lors de son exécution, elle provoque l'activation des autres tâches. Le serveur des tâches est pris en charge par le *système d'exploitation* (exemples : *LynxOS*, *OSE*, *RTLinux*) ou le *noyau* (ou *exécutif*) temps réel (exemple : *VxWorks*). Le système d'exploitation fournit un support logiciel complet permettant de gérer les ressources d'un calculateur. Celles-ci comprennent : *i)* des unités physiques telles que le processeur ou la mémoire ; *ii)* des modules logiciels formés par les programmes systèmes, qui permettent par exemple de transformer des *programmes utilisateurs sources* en *programmes objets*. Le noyau représente le sous-ensemble du système d'exploitation qui gère le processeur et les ressources nécessaires à l'exécution des programmes objets. Pour des applications embarquées, le système d'exploitation se ramène souvent à un noyau temps réel (par exemple, on peut noter des *micro-noyaux* comme *Mach*[4], ou des *nano-noyaux* comme

4. `http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html`.

Adeos[5], qui sont des noyaux d'une taille très réduite, pouvant aller jusqu'à seulement 1.8 Ko).

Dans les systèmes à grande échelle et hétérogènes, on est obligé de suivre une méthodologie bien précise pour mettre en œuvre l'ordonnancement des différentes tâches. Une approche consiste notamment à scinder le système en plusieurs sous-systèmes, pour lesquels on peut choisir la politique d'ordonnancement la mieux appropriée. Ensuite, il faut définir les modalités de regroupement de l'ensemble.

L'*ordonnancement à niveaux multiples* [197] est une technique qui permet de hiérarchiser la description de l'ordonnancement dans un système en distinguant différents niveaux de granularité. C'est le cas dans les systèmes avioniques dont la mise en œuvre repose sur l'approche *modulaire intégrée* [7] (cf. chapitre 7).

2.3 Intergiciels

L'*intergiciel* (plus connu sous l'appellation anglo-saxonne *middleware*), comme le montre la Fig. 2.7, a pour rôle d'assurer l'interfaçage entre la partie applicative d'un système structuré par couches et le système d'exploitation. Cela permet de résoudre les problèmes d'interopérabilité et d'intégration des applications. Une telle organisation est notamment intéressante pour des systèmes où résident des applications réparties concurrentes. Ces dernières peuvent être *a priori* de natures différentes, mais grâce à la transparence offerte par le *middleware*, elles accèdent de façon uniforme aux informations stockées dans le système, qui ne sont pas forcément compatibles au niveau du format.

L'importance grandissante des méthodes orientées composants a été soulignée dans le chapitre précédent. Dans ces méthodes où on est potentiellement amené à assembler des entités hétérogènes, il est important de disposer de moyens garantissant l'interopérabilité. Parmi les technologies prévues à cet effet, nous citons CORBA (*Common Object Request Broker Architecture*) [169] défini par l'OMG, .NET [163] de Microsoft, ou Jini [130] défini par la communauté "Java". Parmi elles, seule CORBA offre une extension temps réel.

CORBA. C'est un standard de l'OMG qui spécifie une infrastructure de communication pour la couche applicative, au sein de l'organisation d'un système. Il est le fruit d'un concensus entre industriels à propos de la gestion de systèmes distribués hétérogènes, conçus suivant l'approche orientée objet. CORBA propose un modèle de coopération de type client/serveur entre des applications réparties. Des fonctionnalités de chaque application peuvent être exportées sous forme d'objets CORBA. Les interactions entre applications sont réalisées à l'aide d'appels de procédure distante. Les applications utilisant un objet représentent les clients, et l'application en attente de requête des clients est le serveur (la partie du serveur qui met en œuvre l'objet est appelée *servant*). Par ailleurs, les applications coopérantes peuvent être programmées dans des langages différents, indépendamment des plates-formes sur lesquelles elles s'exécutent. Cette interopérabilité est réalisée à travers des spécifications d'interfaces bien précises au niveau application, le formalisme utilisé est appelé *langage de définition d'interface* (ou IDL - *Interface Definition Language*). La syntaxe adoptée dans un IDL est une syntaxe indépendante de toute technologie qui permet d'exprimer l'encapsulation d'objets. On peut faire une analogie entre les déclarations dans un IDL et la spécification de classes abstraites en C++ : elles contiennent des attributs et des signatures d'opérations. D'autre part, l'héritage est permis pour accroître la

5. `http://www.nongnu.org/adeos`.

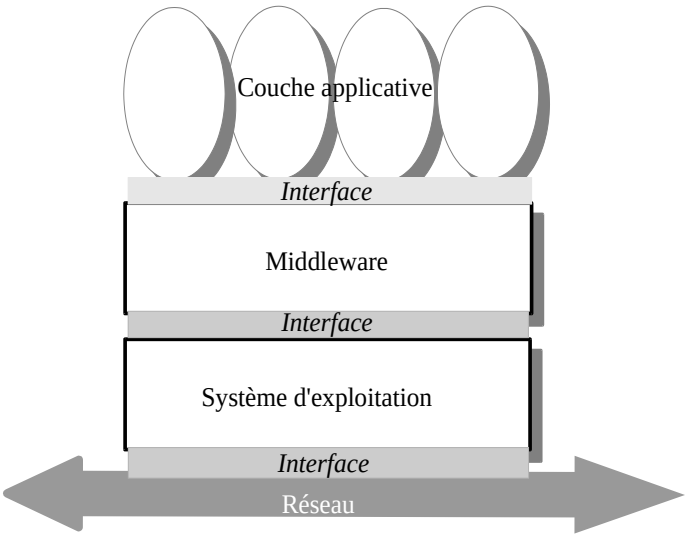


FIGURE 2.7 – Couche *middleware* dans l'organisation d'un système.

réutilisation. La compilation de spécifications IDL produit des fichiers d'interfaces ainsi que les squelettes de programme associés. En plus de l'IDL, il y a besoin d'un mécanisme d'échange de données entre entités de la couche applicative. C'est le rôle de l'*ORB* (*Object Request Broker*) et il le fait de manière transparente.

Une extension temps réel de CORBA, connue sous le nom de REAL-TIME CORBA [171], a été proposée dans le but de favoriser d'abord la prédictibilité *bout-en-bout*[6] des *activités* d'un système afin d'être en mesure de vérifier *a priori* que celui-ci satisfait bien les besoins. Par ailleurs, REAL-TIME CORBA permet une gestion plus adéquate des ressources dans un système temps réel (par exemple, le processeur, la mémoire ou le réseau). REAL-TIME CORBA définit des interfaces et des mécanismes qui facilitent la combinaison de l'ORB et des applications. Ces dernières accèdent aux ressources via les interfaces. Les activités qui composent ces applications sont quant à elles gérées à l'aide des mécanismes de l'ORB (devenu REAL-TIME ORB). Celui-ci fait appel au système d'exploitation temps-réel sous-jacent pour ordonnancer les tâches représentant les activités et assurer l'exclusion mutuelle lors des accès aux ressources. La définition d'une tâche dans REAL-TIME CORBA est compatible avec la norme POSIX. Le type des priorités affectées à ces tâches est *a priori* indépendant de toute plate-forme. Les durées de possibles inversions de priorité et des latences d'opérations lors de l'exécution du système sont bornées. Cela est réalisé à l'aide de mécanismes tels que l'héritage de priorité ou l'invocation d'opération avec *timeout*. Un système décrit selon REAL-TIME CORBA comporte principalement quatre types de composants : les applications, les entités chargées du transport des données échangées, le REAL-TIME ORB et les mécanismes d'ordonnancement du système d'exploitation.

6. On parle aussi de prédictibilité *horizontale* ; c'est-à-dire, être capable d'étudier *a priori* les interactions entre clients et serveurs lors de l'exécution du système. La prédictibilité *verticale* concerne les interactions entre la couche applicative et l'interface réseau.

2.4 Langages de programmation d'applications temps réel

Nous présentons deux langages de programmation d'applications temps réel : Ada qui est un langage qui connaît un usage répandu dans le milieu industriel depuis plusieurs années maintenant, et Real-Time Java présenté comme le futur langage adapté pour les applications temps réel critiques. Ces deux langages partagent le fait d'être de sérieux concurrents à C et C++, dans des domaines où ces derniers sont habituellement utilisés : les systèmes temps réel et les systèmes embarqués.

2.4.1 Ada

Son histoire remonte aux années 1970. À la base, sa "création" fut motivée par le souhait du département de la défense américain [7] d'avoir un langage de programmation unique en lieu et place des nombreux autres langages auxquels avaient recours alors ses développeurs, pour les systèmes embarqués. Ainsi, la première ébauche proposée en 1979 fut normalisée finalement en 1983, sous l'appellation Ada 83. La version révisée [8] la plus récente du langage date de 1995, elle est connue sous le nom de Ada 95 (par la suite, nous parlerons juste de Ada ; nous préciserons Ada 83 si besoin).

Le langage Ada met avant tout l'accent sur la diminution du coût des logiciels. Il privilégie beaucoup la maintenabilité. Nous pouvons mentionner en quelques points des caractéristiques fortes du langage :

- La modularité : il intègre des concepts de la programmation orientée objet comme l'héritage (réalisé à l'aide du type spécial désigné avec le mot-clé `tagged` dont l'opération de dérivation associée est `new`). Cela permet une structuration efficace du code, qui facilite sa lecture. Il offre des notions telles que les paquetages, la compilation séparée, la généricité et les exceptions.
- Un typage fort : Ada impose des contrôles stricts qui permettent de diagnostiquer des erreurs très tôt dans le cycle de conception.
- La portabilité : il fournit des garanties de développement d'applications portables sur différentes architectures de machines (par exemple, les précisions des calculs numériques dans les programmes doivent être les mêmes, ou au moins équivalentes, sur des machines différentes).
- Des concepts de la programmation temps réel : Ada permet des descriptions multi-tâche temps réel au travers d'un concept d'*annexe*. Il s'agit de sous-ensembles de notions distinguées par le langage, qui correspondent à différents domaines d'utilisation. Ces notions ne modifient ni la syntaxe ni les règles du langage telles que spécifiées dans la norme principale. Par exemple, les annexes requises pour la programmation d'applications distribuées temps réel sont celles qui portent sur : la *programmation système* (décrivant entre autres des paquetages et des *pragmas* pour la gestion directe des interruptions), les *systèmes temps réel* (spécifiant la gestion de tâches, la prise en compte de priorités dynamiques de tâches, des politiques d'ordonnancement des tâches et de gestion de files d'attente, la gestion du temps physique [9], etc.), les *systèmes distribués* (définissant un programme comme un ensemble de partitions qui s'exécutent sur différents processeurs, ainsi que les échanges de messages entre les partitions).

7. US DoD (*Department of Defense*), déjà à la base du langage COBOL dans les années 1960.
8. `http://www.adahome.com`.
9. Ce temps est donné par l'horloge interne de la machine, il dépend donc de la plate-forme d'implantation.

Au vu des caractéristiques ci-dessus, nous pouvons constater les nombreux avantages du langage Ada. Cependant, il comporte aussi quelques inconvénients (notamment pour la programmation d'applications temps réel critiques) comme le *non déterminisme* induit par l'instruction `delay` du langage. Par exemple, une tâche est suspendue en exécutant le code suivant :

```
delay 10.0;
```

L'on est tenté de penser que la tâche ne sera suspendue que pendant 10 unités de temps. En réalité, elle est suspendue pendant **au moins** 10 unités de temps. Après l'expiration de la durée spécifiée, il n'est pas garanti qu'un processeur sera disponible immédiatement pour exécuter la tâche [22]. En clair, la sémantique de cette instruction précise uniquement qu'il ne se passera rien avant la fin du délai. Au-delà, rien n'est dit, et donc tout est possible.

La présence d'ambiguités comme celle que nous venons de souligner dans la sémantique du langage Ada a conduit à la définition du langage Sparc, qui se restreint à un sous-ensemble de Ada ayant une sémantique formelle définie [23]. Le modèle de concurrence de Sparc consiste en une tâche unique (c'est-à-dire, le programme principal) qui accède aux objets globaux (i.e., ses propres variables). Une autre particularité de Sparc est la non prise en compte du mécanisme d'exception. Le modèle multi-tâche et le mécanisme d'exception présents dans Ada sont jugés trop complexes et problématiques pour en donner des définitions formelles dans Sparc. Enfin, des annotations (qualifiées de "commentaires formels") sont introduites dans le code sous forme de *pré/post conditions*, afin de permettre des analyses statiques très tôt dans le développement d'une application.

2.4.2 Real-Time Java

De nombreux efforts ont été fournis ces dernières années dans la "communauté Java" pour introduire des concepts de la programmation temps réel dans le langage, donnant ainsi naissance à Real-Time Java. Parmi les groupes de travail les plus actifs, nous distinguons le *J-Consortium*[10] et le *Real Time Expert Group* de *Sun*[11]. Ces groupes ont soumis notamment des propositions visant la normalisation du langage.

Du fait de ses caractéristiques intéressantes, le langage Java se présente en bon candidat pour une extension temps réel.

- La programmation à l'aide de Java comporte tous les avantages d'un langage orienté objet (par exemple, modularité et réutilisation, plusieurs bibliothèques de fonctions sont accessibles dans l'environnement de programmation). D'autre part, les restrictions qu'elle impose en font un langage plus sûr, comparé à d'autres langages de la même famille comme C++.
- Les applications sont portables grâce aux concepts de machine virtuelle (ou *Java Virtual Machine* - JVM). Le code source est compilé sous forme de *bytecode* intermédiaire, indépendamment de toute plate-forme. Ce code pourra alors être interprété dans un environnement d'exécution Java particulier.

Cependant, on peut noter quelques uns de ses inconvénients, qui suscitent encore quelques doutes quant à son utilisation pour programmer des applications temps réel.

10. `http://www.j-consortium.org`.
11. `http://www.rtj.org/team.html`.

– L'exécution d'un code source JAVA nécessite sa compilation en *bytecode*, qui ensuite, peut être interprété par la machine virtuelle. Cela requiert davantage de ressources systèmes que l'exécution d'un langage machine, d'où une baisse possible en rapidité et en performances.
– Le mécanisme de ramasse-miettes (ou *garbage collector*) utilisé par les environnements d'exécution JAVA afin de faciliter la gestion de la mémoire, est mis en œuvre à travers une tâche. Celle-ci est déclenchée quand il le faut, en préemptant la tâche active pour une durée non connue *a priori*. Cela complique la prédiction sur l'ordonnancement des tâches, et donc la garantie des contraintes temps réel.
– Plus généralement, on peut remarquer l'absence de "paradigmes temps réel" dans le langage (exemples : tâches cycliques, horloge temps réel).

Les spécifications [45] [126] établies par les groupes de travail ont pour objectif de proposer des solutions aux inconvénients du langage JAVA mentionnés ci-dessus. Elles proposent une évolution du langage en apportant quelques modifications à certains de ses mécanismes tels que la gestion de la mémoire, des exceptions ou des événements asynchrones, pour faciliter la prédictibilité. D'autre part, des concepts temps réel sont également introduits sous forme de classes bien identifiées : tâches périodiques, horloges, etc.

À coté des spécifications qui viennent d'être évoquées, nous pouvons aussi mentionner l'existence de profils définis pour REAL-TIME JAVA. Ceux-ci se concentrent principalement sur des sous-ensembles des spécifications du langage, selon les besoins du domaine d'application considéré. Par exemple, le profil *Ravenscar High Integrity Profile* [141], qui est compatible avec la spécification J2ME [12] de Sun, est dédié aux systèmes temps réel stricts.

Enfin, d'autres travaux comme ceux effectués dans le projet RNTL (Réseau National des Technologies du Logiciel) Expresso [13] vont dans le même sens. Ce projet a pour objectif de réaliser un environnement de développement temps réel basé sur une spécification du langage REAL-TIME JAVA.

*
* *

Ada vs Real-Time Java. Ces deux langages ont davantage de points communs que de différences. Nous remarquerons que chacun d'entre eux a été défini avec le souci de sûreté dans la programmation : ils sont fortement typés et imposent des contrôles stricts de programmes. Ils adoptent tous les deux une approche orientée objet. Ils offrent des concepts de la programmation temps réel (exemples : gestion multi-tâche, manipulation du temps). Pour une étude comparative exhaustive de ADA et REAL-TIME JAVA, le lecteur pourra se référer à [52] et [53]. Les auteurs examinent principalement un rapprochement des deux langages du point de vue des aspects orientés objet et temps réel. Le domaine des systèmes temps réel est très large, et chacun des deux langages peut être plus ou moins adapté selon les besoins. Dans des systèmes temps réels souples tels que le multimédia, REAL-TIME JAVA peut jouer un rôle important. Cela est justifié par le fait que le langage JAVA est déjà bien installé dans la programmation d'applications WWW (*World Wide Web*). Cependant, dans les systèmes temps réel stricts, où la sûreté et la fiabilité sont des critères indispensables, le langage ADA reste un bon candidat du fait qu'il soit très encré dans l'industrie.

12. `http://java.sun.com/j2me/docs/index.html`.
13. `http://www.telecom.gouv.fr/rntl/FichesA/Expresso.htm`.

Nous pouvons mentionner d'autres langages de programmation d'applications temps réel tels que PEARL, LTR (Langage Temps Réel) ou CHILL (*Ccitt HIgh Level Language*). Le premier est notamment très populaire dans l'industrie allemande. Le second qui date des années soixante dix, est un prédécesseur du langage ADA, et il est surtout connu en France (c'est un standard du Ministère français de la Défense). Enfin, le dernier, développé également dans les années soixante dix, est standardisé par le CCITT (Comité Consultatif International de Télégraphie et Téléphonie). La version de 1996, CHILL96, est proche du langage ADA. CHILL est surtout utilisé dans le domaine des télécommunications.

2.5 Résumé

Ce chapitre a été consacré à la présentation de quelques éléments de mise en œuvre pour les systèmes temps réel. Nous avons exposé les principaux algorithmes d'ordonnancement (comme les ordonnancements statiques, *Rate Monotonic Scheduling* ou *Earliest Deadline First*), ainsi que la problématique du partage de ressources. Pendant de nombreuses années, une partie significative des travaux sur les systèmes temps réel a porté sur ces deux aspects. Les langages de programmation de tels systèmes intègrent ces notions. C'est le cas notamment du langage ADA qui est très utilisé dans le développement d'applications temps réel strictes (exemple : applications embarquées dans les avions). Plus récemment, on a assisté à l'apparition du langage REAL-TIME JAVA pour la programmation d'applications temps réel. D'autre part, des concepts tels que les *intergiciels* sont proposés pour améliorer l'interopérabilité dans les plates-formes d'implantation. Leur popularité est parfaitement illustrée par la technologie CORBA à travers son extension temps réel, appelée REAL-TIME CORBA.

Après avoir donné une vision générale des approches et concepts de base pour la mise en œuvre des systèmes temps réel, nous allons nous concentrer sur le cas spécifique des outils de conception basés sur le modèle synchrone.

Chapitre 3

Introduction à la technologie synchrone

3.1 Présentation

Nous avons introduit au début de ce document (chapitre 1), trois façons de représenter le temps au sein d'un système temps réel. Nous nous concentrons ici sur l'un des modèles, à savoir le modèle synchrone. Il a été introduit pour faciliter la conception et l'analyse des système réactifs temps réel.

3.1.1 Motivation de l'approche

Considérons les propos suivants [109] :

- le train s'arrêtera *dans au plus dix secondes*
- le train s'arrêtera *après cent mètres*
- le train s'arrêtera *après quarante tours de roues*

D'un point de vue conceptuel, ces propos sont de même nature car ils font tous référence au temps que mettra le train à s'arrêter. Cependant, ils l'expriment de façons différentes ; les unités de mesure sont des "secondes", des "mètres" et des "tours de roues". Cet aspect du temps est qualifié de *multiforme*.

Les propos ci-dessus peuvent s'exprimer de façon unique, en utilisant une notion de contrainte de *précédence d'événements*. Nous dirons par exemple :

- l'événement *arrêt du train* doit précéder la dixième (resp. centième) occurrence suivante de l'événement *seconde* (resp. *mètre*).

Dans cette nouvelle façon d'énoncer les faits, les informations pertinentes sont les occurrences des événements d'une part, et les précédences de l'autre. Elles suffisent pour exprimer l'écoulement du temps sous toutes ses formes. C'est la base du *modèle synchrone*.

Dans ce modèle, la notion de *temps physique* est remplacée par une notion d'*ordre*. Seules la *simultanéité* et la *précédence* des événements observables du système considéré sont prises en compte. La *référence temporelle* ou *horloge* est déterminée par la séquence des événements, donnant lieu à des instants dits *logiques*, qui représentent les *réactions* du système. À chacun de ces instants, un nombre quelconque (positif ou nul) d'événements peut apparaître, lesquels sont considérés comme simultanés. La durée physique des calculs et des communications qui ont

lieu pendant les occurrences des événements d'un instant logique est ainsi abstraite (on parle d'*abstraction synchrone*). Toutefois, ces contraintes temporelles physiques doivent être vérifiées lors de l'implantation du système, ou même, si possible, lors d'une modélisation de cette implantation.

La FIG. 3.1 montre un système et une trace d'exécution possible. Chaque réaction du système est caractérisée par : d'abord, l'acquisition d'entrées provenant de son environnement (par exemple, le procédé physique contrôlé) puis, le traitement de ces données et enfin, la production du résultat. Ici, le système a deux entrées $E1$ et $E2$, et une sortie S. Les instants logiques sont $t1$, $t2$, $t3$ et $t4$. Ils correspondent aux différentes *réactions* du système (ici, il y en a quatre). Ils se succèdent dans l'échelle de temps discrète ainsi dérivée. Les instants τi désignent les points dans le temps physique auxquels on observe les événements impliqués dans les réactions du système.

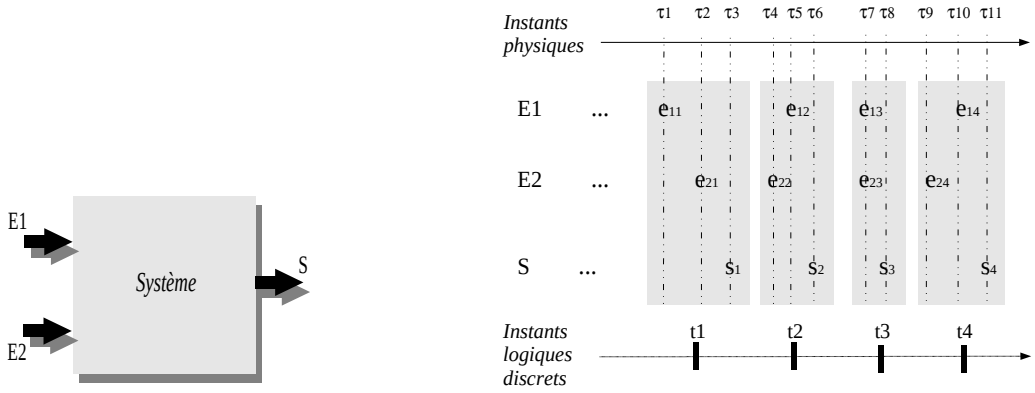


FIGURE 3.1 – Une trace d'exécution synchrone d'un système.

3.1.2 Des modèles synchrones

Nous énumérons dans cette section différents modèles sémantiques synchrones en nous inspirant de la synthèse réalisée par M. Nebut dans [166]. Pour plus de détails, nous renvoyons donc le lecteur à ce document.

Un système réactif comprend au moins deux types de composants pour assurer d'une part, les échanges des données avec l'environnement et d'autre part, le traitement des données acquises. La modélisation synchrone d'un tel système doit donc prendre en compte son échelle de temps et celle de l'environnement, les composants du système pouvant eux-même avoir leurs propres horloges : on parle du temps *multi-horloges* synchrone. Les modèles sémantiques synchrones se distinguent essentiellement par leur façon de traiter le temps multi-horloges [147], et plus particulièrement l'*absence*.

Le modèle dit *endochrone* [1] se caractérise par le fait que le statut des entrées du système (absence/présence) est déterminé par le système lui-même. L'environnement de ce dernier se voit donc obligé de respecter [2] les contraintes spécifiées sur le statut des entrées par le système. Le langage LUSTRE repose sur le modèle endochrone : tout programme possède une horloge

1. Au sens étymologique, traduit par "*temps de l'intérieur*".

2. Pour assurer un fonctionnement correct d'un système endochrone "plongé" dans un environnement ne respectant pas les contraintes qu'il impose, on peut utiliser une interface de communication qui permet de prendre en compte ces contraintes.

principale (ou horloge *maîtresse*) qui décrit la présence des signaux. Le modèle endochrone peut être rapproché du mode d'exécution *demand-driven*.

Dans le modèle *exochrone*[3] ou *réactif*, c'est plutôt l'environnement qui fournit le statut des entrées lues par le système. Ce dernier *réagit* donc à toute configuration de son environnement. Le langage ESTEREL adopte ce dernier modèle. On qualifie aussi ce langage de *réactif* pour exprimer la même chose [147] (un programme ESTEREL réagit toujours en présence de ses entrées, quelle que soit leur configuration). Le modèle exochrone peut être associé au mode d'exécution *event-driven*.

Le modèle sémantique *polychrone* regroupe les modèles endochrones et exochrones. Il offre ainsi un plus grand pouvoir d'expressivité. On peut décrire divers types de comportements : spécifications partielles ou exécutables, non déterminisme. Cela permet de concevoir un système de façon incrémentale sans pour autant être contraint par la nécessité d'une description complète et exécutable. Bien entendu, il peut en résulter des descriptions complexes dont l'analyse n'est pas facile, d'où la nécessité de disposer de méthodologies bien appropriées pour faciliter la conception. Le langage SIGNAL est fondé sur ce modèle.

3.2 Langages et outils synchrones

Les langages synchrones [109] [30] sont souvent classés en deux familles : les langages adoptant un style *impératif* et les langages orientés *flot de données*. Dans la première famille, on peut citer ESTEREL [36] qui est textuel ; ou STATECHARTS [115] qui est fondé sur la définition d'une hiérarchie d'automates avec des représentations graphiques permettant d'exprimer la concurrence et la communication. Les langages LUSTRE [111], LUCID SYNCHRONE [59], et SIGNAL [32] sont orientés flot de données. Dans cette section, nous donnons un apercu des différents langages synchrones. Le lecteur pourra se référer à [166] pour une comparaison exhaustive de ces langages.

3.2.1 L'approche impérative

3.2.1.1 Statecharts et quelques extensions

Le langage STATECHARTS [115] a été concu au *Weizmann Institute of Science*[4] (Israël). Il est sans aucun doute le formalisme graphique le plus populaire pour décrire les systèmes réactifs. Il s'agit d'une extension des machines à états, qui prend en compte les aspects suivants :

- *communication* : les transitions entre états sont similaires à celles d'une machine de Mealy[5]. Elles sont de la forme événement/action, où l'occurrence de l'événement et le déclenchement de l'action associée sont simultanés.
- *concurrence* : on peut exprimer des comportements parallèles sans pour autant avoir une explosion d'états (en partie grâce à la hiérarchie d'états).
- *hiérarchie* : un état peut encapsuler d'autres états (appelés aussi sous-états). On distingue deux types d'états au sein de l'automate. Un *état-et* contient des sous-états concurrents qui peuvent évoluer simultanément et communiquer par un mécanisme de diffusion d'événements internes dans tout l'automate. Dans un *état-ou*, les sous-états évoluent de

3. Au sens étymologique, traduit par "*temps de l'extérieur*".

4. `http://www.weizmann.ac.il`.

5. Une machine de Mealy est une machine à états finis qui produit des sorties sur ses transitions. C'est en quelque sorte le pendant de la machine de Moore, où les sorties sont produites dans les états.

manière exclusive. Le choix de l'état lors de l'entrée dans un état-ou peut se faire de deux manières : soit il existe un sous-état pointé statiquement par un *connecteur défaut* (transition sans origine) ; soit c'est l'état qui était actif lorsque l'état-ou a été quitté, indiqué par un *connecteur historique*.
La hiérarchie permet notamment de modéliser des comportements préemptifs (lorsqu'on quitte un état, on quitte également ses sous-états). D'autre part, cette hiérarchie facilite la représentation de systèmes complexes.

La FIG. 3.2 donne un exemple de spécification sous forme de STATECHARTS. L'automate ainsi représenté contient deux états au niveau le plus externe : `Idle` est l'état initial (repéré par un connecteur défaut) et `Running`. Les transitions entre ces deux états sont déclenchées par les occurrences des événements `e` et `f`. Aucune action n'est associée à ces derniers.
L'état `Running` est raffiné en un état-et, où les sous-états concurrents sont `Sub_Running_Up` et `Sub_Running_Down`. Ceux-ci sont à leur tour raffinés en états-ou. Par exemple, dans `Sub_Running_Up`, S_1 et S_2 vont s'exécuter de manière exclusive. Nous remarquerons que la transition entre S_2 et S_1 est étiquetée par un événement `a` qui peut déclencher une action, qui correspond ici à l'émission de l'événement `b`. Enfin, `H` est un connecteur historique.

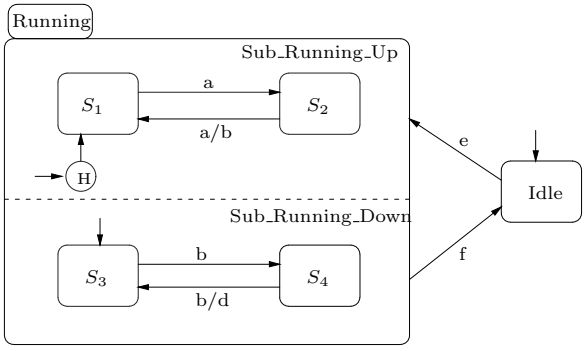


FIGURE 3.2 – Une description à l'aide de STATECHARTS.

La configuration courante d'un automate STATECHARTS est définie par l'ensemble des états actifs. Le passage d'une configuration à une autre se fait à chaque *step*, qui représente l'unité de réaction. La prise en compte des actions provoquées par des événements (externes ou internes) dépend de la sémantique considérée [211].

Dans la variante STATEMATE [116], ces actions ne deviennent effectives qu'au *step* suivant (d'où un décalage temporel d'un *step* dans l'émission des sorties). STATEMATE définit deux modèles de temps pour les réactions : *i)* dans le modèle appelé **step**, le système doit acquérir des entrées à chaque *step* ; *ii)* tandis que dans le modèle **superstep**, le système enchaîne les actions par diffusion d'événements internes tant qu'il y a des transitions tirables, sans acquisition d'entrées. STATEMATE est mis en œuvre dans l'outil MAGNUM de la société *i-Logix*.

Le langage ARGOS [157] développé à l'IMAG (Grenoble) est une variante purement synchrone de STATECHARTS. Il dispose d'un environnement de modélisation et de validation, appelé ARGONAUTE. ARGOS promeut plutôt une utilisation conjointe avec d'autres langages synchrones comme LUSTRE ou ESTEREL (présentés ci-après). Le formalisme SYNCCHARTS [17] repose sur STATECHARTS et s'inspire d'ARGOS. Il a été introduit comme une version graphique de ESTEREL. Défini à la base au laboratoire I3S à Sophia-Antipolis, il est développé actuellement par ESTEREL TECHNOLOGIES. La sémantique de SYNCCHARTS est entièrement synchrone.

Enfin, nous notons l'existence d'une version de STATECHARTS pour le méta-modèle UML et le langage MATLAB [6] (adapté pour les problèmes scientifiques, et permettant de visualiser graphiquement des données sous forme 2D ou 3D). L'intégration d'un tel formalisme à UML apporte à ce dernier une base formelle pour exprimer les propriétés dynamiques des systèmes modélisés. Les propriétés statiques sont spécifiées à l'aide du langage OCL.

3.2.1.2 Esterel

ESTEREL [49] [36] est un langage qui a été développé au Centre de Mathématiques Appliquées [7] (CMA) de Sophia Antipolis, à travers la collaboration de deux organismes : l'École Nationale Supérieure des Mines de Paris et l'INRIA. Il repose d'une part sur des structures de contrôle impératives classiques telles que la séquence, l'itération, et d'autre part sur des instructions qualifiées de réactives dont la sémantique est basée sur la notion d'instant [8]. L'intérêt d'un langage synchrone impératif comme ESTEREL est surtout de faciliter des descriptions modulaires de systèmes réactifs où le contrôle joue un rôle primordial. Pour cela, il offre les constructions de haut niveau nécessaires.

Les objets de base sont le *signal* et le *module*. Le signal est caractérisé par son statut à tout instant : absent ou présent. Un signal peut porter une valeur ou non, dans le deuxième cas, on parle de signal *pur*. Il existe un signal pur prédéfini dans le langage, `tick`, qui correspond à l'horloge la plus rapide qui régit les activations. L'émission d'un signal peut être de deux natures : soit elle est environnementale, dans ce cas le signal est une entrée du programme plongé dans l'environnement émetteur, soit le signal est émis par un programme, en tant que sortie, avec l'instruction `emit`. Le module est une construction qui permet de structurer un programme ESTEREL. Syntaxiquement, il est constitué d'une interface (signaux d'entrée/sortie du module) et d'un corps (composé d'instructions impératives et réactives, spécifiant le comportement du module). Le corps est exécuté instantanément à chaque activation du module.

Parmi les caractéristiques importantes du langage ESTEREL, nous pouvons mentionner les suivantes :

- *communications* et *synchronisations* : elles se font à travers la diffusion instantanée des signaux entre les entités (modules).
- mécanismes de *préemption* et d'*attente* : la préemption est réalisée via des instructions spéciales. Par exemple, dans `suspend P when s`, `P` n'est exécuté qu'au sein des instants où le signal `s` est absent. L'instruction `pause` permet de suspendre l'exécution courante jusqu'au prochain instant d'activation. L'instruction `await s` provoque une suspension de l'exécution jusqu'à l'occurrence du signal `s`. On remarquera que ces deux dernières instructions ne sont pas instantanées (car elles ne terminent pas forcément dans le même instant logique où leur exécution débute). Enfin, il existe aussi des mécanismes d'exception dans le langage.
- deux types de *compositions* : deux instructions `I1` et `I2` peuvent être composées de façon *séquentielle* `I1;I2`, dans ce cas, `I2` est exécutée seulement après la terminaison de `I1`. Les deux instructions peuvent aussi être composées en utilisant un opérateur de composition parallèle `I1 || I2`. L'instruction résultant de la composition parallèle de `I1` et `I2` s'arrête lorsque toutes les instructions composées ont terminé.

6. `http://www.mathworks.com`.
7. `http://www-sop.inria.fr/cma`.
8. Les instants logiques auxquels le système décrit est supposé réagir.

```
                                                    loop
                                                      pause;
                                                      pause;
  loop                                                emit e1;
    await e1 ;                                      end loop
    call code_E1();                                ||
  end loop                                          loop
||                                                    pause;
  loop                                                pause;
    await e2 ;                                        pause;
    call code_E2();                                   emit e2;
  end loop                                          end loop
```

FIGURE 3.3 – Code ESTEREL représentant une application (à gauche) et son environnement (à droite).

Le programme ESTEREL décrit sur la FIG. 3.3 représente une application (partie gauche) et son environnement (partie droite). Les deux sous-programmes communiquent par échange de signaux : l'environnement émet deux événements `e1` et `e2`, récupérés par l'application. Cela est exprimé à travers la composition parallèle de deux boucles (`loop ... end loop`) contenant les émissions d'événements (`emit e1` et `emit e2`). L'application attend donc l'émission des événements (`await e1` et `await e2`). Sur réception de ces derniers et en fonction de l'événement, un traitement spécifique (`code_E1()` pour `e1`) est effectué via un mécanisme d'appel (`call`). On remarquera l'opérateur de composition séquentielle entre la réception et l'appel au sein des boucles, dans l'application.

Un aspect intéressant des langages synchrones est leur capacité à analyser les propriétés des programmes. En particulier, l'analyse de la consistance de l'ensemble des contraintes sur les horloges (instants de présence) des signaux permet de vérifier des propriétés sur les comportements du programme (exemple : absence de blocage). Cette analyse fait généralement partie de la compilation. Par exemple, un programme source ESTEREL peut être compilé en automates [101], en circuits [35], ou bien dans un format intermédiaire sous la forme d'un graphe de flots de contrôle [176].

Des versions académiques de plates-formes de développement à l'aide du langage ESTEREL, comprenant compilateur et outils de vérification, sont proposées par le CMA, France Télécom et l'université de Columbia (New-York). La version industrielle, appelée ESTEREL STUDIO est commercialisée par la société ESTEREL TECHNOLOGIES. On rencontre des utilisations concrètes de ESTEREL dans des domaines tels que l'avionique ou l'automobile, pour la spécification d'applications embarquées. Il est aussi utilisé pour la conception conjointe ou *co-design* (cf. POLIS ci-dessous).

3.2.2 L'approche déclarative

Les langages orientés flot de données trouvent leur origine dans des études qui remontent aux années 70. Parmi ces études, on peut citer les travaux de Kahn sur la sémantique des réseaux de processus communiquant par files d'attentes [132]. Il utilise les flots de données pour

représenter le comportement. Nous soulignons également l'apport de Dennis et son équipe au M.I.T. [9], qui ont beaucoup contribué au développement des langages orientés flot de données [76] [77]. D'autres travaux allant dans le même sens ont conduit à la définition du formalisme LUCID. Celui-ci est le premier langage fondé sur les flots de données ; il a été défini par Ashcroft et Wadge [212]. Le modèle de programmation proposé par LUCID est assez proche des mathématiques, facilitant ainsi le raisonnement sur les comportements des programmes. Cependant, le langage n'encourage pas une approche opérationnelle dans la programmation, comme c'est le cas dans les langages impératifs classiques. D'où quelques difficultés, notamment en ce qui concerne la définition d'un schéma de génération de code séquentiel efficace.

Toutes ces études (entre autres) ont largement servi d'inspiration pour le développement des langages synchrones que nous présentons dans cette section, à savoir LUSTRE, LUCID SYNCHRONE et SIGNAL. Seuls les deux premiers langages sont abordés ici. SIGNAL est présenté en détail dans le chapitre suivant, consacré à la technonologie polychrone.

3.2.2.1 Lustre

Le langage LUSTRE [111] est développé au laboratoire Verimag [10] (Grenoble). Il est adapté pour spécifier des systèmes réactifs qui manipulent essentiellement des flots de données sous forme d'équations.

Dans LUSTRE, les objets de base sont le *signal* et le *nœud* (*node*). Un signal dénoté par la variable `x` représente le flot de données infini $x_1, x_2, ..., x_k, ...$ où x_i est la i^{eme} occurrence de `x`. Ainsi, chaque signal est caractérisé par :

- l'ensemble des instants auxquels il apparaît, appelé son *horloge*. Cette dernière est encodée par un flot booléen dans lequel les occurrences valant *vrai* dénotent la présence du signal. Sur la trace de la FIG. 3.4, `b` représente l'horloge du signal `y`. On parle de signaux *synchrones* lorsque ceux-ci ont la même horloge.
- la suite des valeurs portées lorsqu'il est présent. Le type de ces valeurs peut être simple (sur la FIG. 3.4, `x` et `b` sont respectivement de type entier et booléen) ou composite (par exemple tableau).

Un nœud représente l'unité de programmation, et définit une fonction (le langage LUSTRE est fonctionnel). Il comprend d'une part une interface formée par les signaux d'entrée/sortie, et d'autre part un corps constitué d'un ensemble d'équations, avec éventuellement des variables locales.

`b` :	*faux*	*vrai*	*faux*	*vrai*	*faux*	*faux*	*vrai*	*vrai*	...
`x` :	1	2	3	4	5	6	7	8	...
`y = x when b` :		2		4			7	8	...
`z = current y` :	*nil*	2	2	4	4	4	7	8	...

FIGURE 3.4 – Un exemple de trace d'opérateurs LUSTRE.

Parmi les opérateurs, on distingue des extensions des opérateurs classiques de l'arithmétique, de la logique, et conditionnels (`if... then...else...`). L'opérateur `pre` permet de mémoriser la valeur précédente d'un signal. La valeur initiale d'un signal est définie à l'aide de l'opérateur

9. `http://web.mit.edu`.
10. `http://www.verimag.imag.fr/SYNCRONE`.

`->` : l'expression à gauche de la flèche représente la valeur d'initialisation de l'expression de droite (cf. l'exemple sur la FIG. 3.5). Tous ces opérateurs imposent à la fois aux arguments et au résultat d'être synchrones.
L'opérateur `when` permet d'extraire une sous-suite d'un signal. Seuls les arguments doivent être synchrones (par exemple sur la FIG. 3.4, `b` et `x` ont la même horloge). Enfin, l'opérateur `current` permet de mémoriser les valeurs d'un signal à l'horloge la plus rapide du nœud (cf. FIG. 3.4 où le signal `z` est défini avec cet opérateur, initialement il vaut *nil*).
À chaque nœud est associée une horloge de base (horloge la plus rapide), représentée par un flot booléen qui porte la valeur *vrai* à toutes ses occurrences. Toutes les constantes du nœud sont à cette horloge. Les signaux d'entrée sont soit à l'horloge de base soit à une horloge moins fréquente spécifiée par un signal booléen (i.e. incluse dans l'horloge de base). De cette façon, à chaque invocation d'un nœud, il y a au moins un signal qui est disponible.
Le programme LUSTRE de la FIG. 3.5 décrit un compteur réinitialisable. Une invocation du nœud correspondant est : `val = COUNTER(0, true)`.

```
node COUNTER(init : int; reset : bool)
returns (n : int)
let
     n = init -> if reset then init
                           else pre(n) + 1
tel
```

FIGURE 3.5 – Code LUSTRE représentant un compteur réinitialisable.

La compilation d'un programme LUSTRE peut être obtenue dans un format dit *Object Code* (OC), dont la structure de contrôle est donnée sous forme d'automates étendus [114]. Elle peut aussi produire du code dans les formats communs aux langages synchrones comme *Declarative Code* (DC) [85].

Une plate-forme académique de LUSTRE est fournie par le laboratoire Verimag. Elle comporte un compilateur, ainsi que d'autres outils de vérification (par exemple, LESAR [112]) et de test. La version industrielle du langage est mise en œuvre dans l'outil SCADE (Safety Critical Application Development Environment), commercialisé actuellement par ESTEREL-TECHNOLOGIES. À travers SCADE, le langage LUSTRE est utilisé dans des domaines critiques tels que l'avionique. Par exemple, dans le projet SAFEAIR, il représente le format d'interopérabilité. Les *automates de modes* [158] sont un formalisme synchrone basé sur LUSTRE. Ils ont été introduits pour modéliser les modes d'exécution d'un système. C'est une notion qui est présente dans beaucoup de systèmes temps réel, par exemple les systèmes embarqués dans les avions. Un automate de modes peut être vu comme une sorte de réseau, où les nœuds représentent des programmes LUSTRE. Les états courants de l'automate déclenchent l'exécution des nœuds associés.

3.2.2.2 Lucid synchrone

LUCID SYNCHRONE [59] est un langage développé au Laboratoire d'Informatique de Paris 6. C'est un langage fonctionnel d'ordre supérieur qui manipule des séquences infinies de valeurs. Il adopte des caractéristiques de LUSTRE (en s'inspirant de certains opérateurs, par exemple, l'opérateur `->` de LUSTRE est dénoté ici par `fby` - *followed by*), et des langages fonctionnels

(la syntaxe est un sous-ensemble de celle de OCAML - *Objective CAML*). Une spécificité de LUCID SYNCHRONE est son *calcul d'horloges* qui est effectué à l'aide de calculs de types. Par ailleurs, il existe un encodage du langage dans l'outil d'aide à la preuve COQ, cela dans le but d'aborder la question de la certification des programmes synchrones [47].

3.2.3 Autres approches

En plus des langages synchrones cités ci-dessus, il existe d'autres formalismes qui sont des extensions synchrones de langages ou concepts existants.

Reactive-C est un langage dédié à la programmation réactive [48]. Il est basé sur le langage C. Dans ce langage, les comportements d'un programme sont exprimés en termes de réactions en réponse à des stimuli. Il en résulte un temps reposant sur ces stimuli, et un instant est une paire (stimulus/réaction). Un autre langage proche de Reactive-C est ECL (Esterel-C Language), défini au laboratoire Cadence de Berkeley (États-Unis) [143].

À l'image des langages synchrones, le formalisme des GRAFCET [68], utilisé dans de domaine de l'automatique, décrit des réactions instantanées. On peut faire un parallèle entre l'évolution d'un modèle GRAFCET et les modes d'exécution dans STATEMATE : dans l'interprétation dite *sans recherche de stabilité* (SRS), un pas d'évolution du modèle GRAFCET correspond à l'acquisition des entrées, au calcul d'un pas simple d'évolution et à l'émission des sorties. Dans l'interprétation *avec recherche de stabilité* (ARS), un pas d'évolution correspond à l'acquisition des entrées, au calcul d'un pas itéré d'évolution et à l'émission des sorties. Un pas itéré correspond à une suite de pas simples d'évolution SRS jusqu'à l'obtention d'une situation qui ne peut évoluer sans acquisition de nouvelles entrées (qualifiée de situation stable). Le mode *superstep* de STATEMATE partage avec l'interprétation ARS le fait qu'ils présentent chacun des risques de non terminaison (pouvant conduire à un blocage vis-à-vis de l'environnement). La combinaison du formalisme synchrone et celui des GRAFCET permet de profiter d'une part, des avantages de la technologie synchrone qui a des bases formelles solides, et d'autre part, de l'aspect pratique des descriptions à l'aide de GRAFCET. C'est le but de l'extension synchrone de GRAFCET, appelée S-GRAFCET [89] dont la plate-forme associée est développée à l'Université de Nice [11]. L'étude réalisée par F. Jiménez dans [129] contribue à une conception sûre des automatismes industriels.

Le formalisme ESTELLE SYNCHRONE est une extension récente du standard ISO (*International Organization for Standardization*) ESTELLE [12], pour la description de systèmes réactifs distribués [179]. ESTELLE permet de décrire formellement des systèmes distribués. De telles descriptions comprennent des modules et des canaux par lesquels les modules échangent des messages. L'extension synchrone ajoute simplement un nouveau type de module appelé *systemsynchrony* pour spécifier des comportements réactifs. La structure et la sémantique de tels modules empruntent celles de STATECHARTS.

Enfin, les *objets synchrones* font partie également des extensions qu'on peut trouver dans la littérature [134] [46] [50]. L'idée de base consiste à voir les objets comme des entités réactives indépendantes, communiquant par des événements. La durée d'un instant est celle que prend l'ensemble des réactions des objets à la suite d'une activation.

11. `http://www.i3s.unice.fr/gaffe/tools.html`.
12. `http://www.estelle.org`.

3.3 La technologie synchrone dans le co-design

De nombreux travaux ont mis en évidence l'apport des langages et environnements de développement synchrones, pour faciliter la conception de cibles d'exécution logicielles et/ou matérielles. À ce sujet, on pourra se reporter entre autres, aux travaux de Kountouris [140] sur le *co-design* et le langage SIGNAL, de Le Lann [148] ou Rocheteau [180] sur la conception de circuits, respectivement à l'aide de SIGNAL et LUSTRE. Le langage ESTEREL est aussi utilisé pour le co-design (cf. environnement POLIS, présenté en section 3.3.2). Ci-après, nous présentons quelques-uns des environnements de conception, connus tant dans le monde académique qu'industriel.

3.3.1 Ptolemy

PTOLEMY [13] est un environnement défini à l'Université de Berkeley (États-Unis), pour la conception conjointe. Il propose un cadre dans lequel sont possibles la spécification, la simulation et l'implantation de systèmes embarqués [149]. Les spécifications consistent en des blocs hétérogènes interconnectés qui décrivent les fonctionnalités d'un système. Pour cela, le concepteur dispose d'un certain nombre de modèles d'exécution qu'il peut choisir pour un bloc donné. Par exemple, l'évolution des blocs peut être donnée sous forme d'automates, elle peut aussi être exprimée en termes d'événements discrets ou de graphes flot de données synchrones. Le modèle réactif synchrone est lui aussi présent au sein de l'environnement [81].

3.3.2 Polis

POLIS [14] est un autre environnement de conception conjointe, également défini et réalisé à l'Université de Berkeley. Il permet de décrire, simuler et synthétiser des systèmes embarqués dans un cadre uniforme (où la représentation adoptée pour le logiciel et le matériel est la même) [21].

Ainsi, un système est décrit sous forme de réseaux où les nœuds sont des *Codesign Finite State Machines* (CFSM) représentant les différents composants du système. Une CFSM est un modèle proche de la machine à états finis classique, dans lequel les transitions sont effectuées à la suite d'événements, *a priori* indépendants. Les CFSM ont un comportement synchrone, par contre, elles communiquent de manière asynchrone à travers le réseau. Il en résulte donc une représentation de type *Globally Asynchronous, Locally Synchronous* (GALS) du système. Le langage ESTEREL est utilisé pour spécifier les composantes synchrones, tandis que les communications se font à travers des tampons. Par ailleurs, le langage ECL [143] du laboratoire Cadence est aussi utilisé dans POLIS.

3.3.3 SynDEx

L'environnement de conception SYNDEX [15] (*Synchronous Distributed Executive*) est développé à l'INRIA Rocquencourt. Il propose une méthode qualifiée d'"Adéquation Algorithme Architecture (A^3)", pour le prototypage rapide et l'optimisation de la mise en œuvre d'applications distribuées temps réel embarquées [195] [104] [106]. SYNDEX utilise une représentation interne proche du *graphe hiérarchisé aux dépendances conditionnées* (GHDC) du langage SIGNAL (cf.

13. `http://ptolemy.eecs.berkeley.edu`.
14. `http://www-cad.eecs.berkeley.edu/Respep/Research/hsc`.
15. `http://www-rocq.inria.fr/syndex`.

chapitre suivant).

Pour concevoir un système selon l'approche SynDEx, on commence d'abord par décrire deux vues du système : *logicielle* et *matérielle*. Ensuite, c'est l'outil qui se charge de répartir la représentation logicielle sur l'architecture matérielle considérée de façon à pouvoir exploiter au mieux le parallélisme disponible sur l'architecture cible multi-processeur, en minimisant le temps d'exécution de l'application. Un exemple d'application concrète de la méthodologie SynDEx est illustré dans [138]. Il s'agit de la conception et la mise en œuvre du prototype d'un véhicule électrique semi-autonome (appelé CyCab). Nous pouvons mentionner les fonctionnalités suivantes de SynDEx :

- spécification et vérification d'un algorithme d'application à l'aide d'un graphe flot de données conditionné ou d'un formalisme synchrone tel que Signal ;
- spécification d'une architecture matérielle multi-processeur sous forme d'un graphe ;
- heuristique pour la répartition et l'ordonnancement de l'algorithme d'application sur l'architecture multi-processeur, avec optimisation du temps de réponse ;
- visualisation des performances temps réel pour le dimensionnement de l'architecture ;
- génération automatique des exécutifs distribués temps réel (ce qui réduit ainsi les efforts de développement que nécessitent leur codage et leur mise au point).

Nous reviendrons sur SynDEx dans le chapitre 5, où nous présentons la méthodologie de conception d'applications distribuées temps réel dans la plate-forme Polychrony.

3.4 Résumé

Nous avons présenté la technologie synchrone à travers les nombreux formalismes et outils basés sur celle-ci. Dans le modèle de conception associé, un système temps réel évolue via une suite de réactions. Au sein d'une réaction, des décisions peuvent être prises en se basant sur la notion d'*absence*. Chaque programme est caractérisé par un état global et les communications sont effectuées à travers un mécanisme de diffusion instantanée. Ainsi, l'évolution d'un système temps réel est vue dans une échelle de temps logique. Le temps physique est abordé dans les phases ultérieures de la conception. Le concepteur peut donc se concentrer d'abord sur les propriétés du système indépendamment de toute plate-forme de mise en œuvre.

Trois modèles sémantiques synchrones sont distingués suivant le mode de communication proposé entre le système et son environnement. Dans le modèle *endochrone*, le statut des entrées reçues de l'environnement par le système est déterminé par ce dernier. Dans le modèle *exochrone*, c'est plutôt l'environnement qui détermine le statut des entrées fournies au système. Le modèle *polychrone* se veut plus général. Il regroupe les deux autres modèles. Il existe divers langages basés sur ces modèles : par exemple, le langage Lustre est endochrone tandis qu'Esterel est exochrone ; Signal quant à lui, adopte le modèle polychrone. Les langages synchrones ont la particularité d'avoir leurs sémantiques clairement définies en termes mathématiques. Cela favorise l'utilisation des méthodes formelles dans la conception, pour spécifier et vérifier des propriétés. Ils disposent généralement d'environnements de développement (incluant par exemple des interfaces graphiques pour la programmation, des outils de preuve et des générateurs automatiques de code). Toutes ces caractéristiques des langages synchrones en font des formalismes très intéressants pour la description et l'analyse des systèmes temps réel. Nous avons donc choisi Signal pour notre étude. Il est présenté dans le chapitre suivant.

Deuxième partie

Concepts et outils polychrones

Chapitre 4

Langage SIGNAL et polychronie

Une caractéristique fondamentale qui distingue SIGNAL des autres langages synchrones est son modèle sémantique *polychrone* [166] [147]. La *polychronie* de ce langage permet typiquement de décrire des systèmes au sein desquels les composants fonctionnent à des horloges différentes. Ainsi, la description d'un système ne requiert pas *a priori* la définition d'une horloge globale comme c'est le cas dans les descriptions endochrones, par exemple, celles du langage LUSTRE. Cette horloge peut être construite, lorsque cela est possible, à l'aide de transformations définies dans le modèle polychrone. En outre, la polychronie permet de concevoir des composants supplémentaires sans avoir à modifier ceux qui existent déjà (ce qui est également possible en LUSTRE). Le modèle polychrone est aussi pratique pour spécifier de façon naturelle des fonctionnements non déterministes (par exemple, pour décrire les comportements d'un environnement interagissant avec une application embarquée temps réel). Il offre donc un grand pouvoir d'expressivité. Il fournit des transformations formelles permettant de raffiner des spécifications polychrones (données en termes de relations) en des descriptions exécutables (exprimées à l'aide de fonctions ou d'automates). Plus généralement, le modèle synchrone permet de telles transformations. Ainsi grâce aux raffinements, on peut à partir de modèles polychrones d'un système, dériver des modèles endochrones ou exochrones associés.

Après une introduction générale aux objets de base du langage SIGNAL dans la section 4.1, nous présentons quelques propriétés formelles de son modèle sémantique dans la section 4.2. Pour les besoins de cette thèse, nous adoptons la sémantique dénotationnelle fondée sur les *tags* [147] (section 4.2). Elle est bien adaptée pour caractériser les comportements temps réel. Dans cette sémantique, les *tags* représentent des instants appartenant à un ensemble dense partiellement ordonné. Il existe une autre sémantique dénotationnelle de SIGNAL basée sur des suites infinies appelées *traces* [145]. Elle se distingue principalement de la sémantique basée sur les *tags* par les aspects suivants : le temps logique est représenté par un ensemble totalement ordonné (l'ensemble des entiers naturels $\mathbb{N}$) ; et l'absence d'événements est spécifiée explicitement. La sémantique présentée dans [32] donne quant à elle une vision opérationnelle de SIGNAL. Dans la section 4.2.2, nous abordons la représentation de comportements temps réel à l'aide du modèle polychrone. Ensuite, dans les sections 4.3 et 4.4, nous introduisons respectivement deux types d'abstractions de programmes SIGNAL : l'*abstraction syntaxique* et l'*abstraction sur valeurs*. Elles consistent en des transformations systématiques d'un programme permettant d'obtenir un nouveau programme qui reflète un aspect particulier du programme initial, ou qui sert d'*observateur* de ce programme (pour évaluer par exemple le comportement temps réel de ce celui-ci).

4.1 Présentation du langage

SIGNAL [32, 146, 98] est un langage conçu pour la spécification et la conception d'applications temps réel qui se place dans la famille des langages synchrones. C'est un langage orienté flot de données adoptant un style déclaratif.

Les objets manipulés sont appelés des *signaux*. Ce sont des suites infinies de valeurs typées indexées par un temps logique. Pour un signal `x`, la suite $(x_t)_{t\in\mathbb{N}}$ représente les valeurs qui lui sont associées dans le temps (l'indice t dénote un instant logique). Parmi les types admis de valeurs, on retrouve ceux des langages de programmation classiques comme les entiers, réels ou booléens. Il existe également des types spécifiques comme le type *événement pur*, qu'on note `event` dans la syntaxe SIGNAL. À chaque instant logique t, un signal `x` peut être soit *présent*, soit *absent*. Dans le premier cas, il porte une certaine information v (autrement dit, $x_t = v$), et dans le second il est noté dans la sémantique du langage [146] au moyen du symbole $\perp$ (i.e. $x_t = \perp$). L'ensemble des instants où un signal est présent est appelé son *horloge*. Dans la syntaxe du langage, l'horloge d'un signal `x` est notée[1] `^x`. Toutefois, il faut observer que la notion de présence d'un signal est relative. Ainsi, on parlera de la présence d'un signal par rapport à la présence/absence d'autres signaux. En particulier, des signaux qui ont la même horloge sont dits *synchrones*.

Un *processus* en SIGNAL est un système d'équations qui exprime les relations fonctionnelles et temporelles entres les signaux impliqués. Les processus constituent les autres objets de base du langage. Ils peuvent être combinés par composition pour obtenir d'autres processus. Un programme SIGNAL est un processus.

4.1.1 Les opérateurs du langage

SIGNAL repose sur six constructions de base qui forment le *langage-noyau*. Dans ces constructions de base, on distingue les opérateurs sur signaux, eux-mêmes classés en deux catégories : les opérateurs dits *monochrones* (*fonctions* ou *relations*, et opérateur de *décalage*) qui imposent aux signaux mis en jeu d'être synchrones ; et les opérateurs *polychrones* (opérateurs de *sous-échantillonnage* et de *mélange déterministe*) qui n'imposent pas *a priori* aux signaux impliqués d'avoir la même horloge. Les processus définis à l'aide de ces opérateurs sont appelés *processus élémentaires*. Les autres constructions de base portent sur les processus : ce sont la *composition parallèle* et le *confinement*. L'ensemble de ces opérateurs offre suffisamment d'expressivité pour en dériver d'autres, utiles pour la structuration et le confort dans les spécifications. Parmi ces extensions, nous mentionnons ci-après des opérateurs qui permettent de manipuler explicitement le contrôle.

4.1.1.1 Opérateurs de base sur les signaux

Nous présentons les constructions de base, portant sur les signaux.

- *Opérateurs monochrones*

  Fonctions[2] **instantanées.** Ce sont des extensions canoniques aux suites, des fonctions élémentaires classiques (exemples : opérations arithmétiques ou logiques). Ainsi, le

1. On notera aussi $\widehat{x}$.
2. Plus généralement, il s'agit de relations.

processus `y := f(x1,...,xn)` où les `xi` et `y` sont des signaux et `f` une fonction n-aire extension aux suites d'une fonction f, est défini par :

$$(\forall t \in \mathbb{N})\ y_t = \begin{cases} \perp & \text{si } x1_t = ... = xn_t = \perp \\ f(x1_t, ..., xn_t) & \text{sinon} \end{cases}$$

Tous les signaux `xi` et `y` ont la même horloge. Pour le processus `x := a + b`, où `a`, `b`, et `x` sont des signaux entiers, une trace possible est la suivante :

`a` :	$\perp$	5	$\perp$	$\perp$	4	2	0	$\perp$	...
`b` :	$\perp$	3	$\perp$	$\perp$	1	9	8	$\perp$	...
`x` :	$\perp$	8	$\perp$	$\perp$	5	11	8	$\perp$	...

Décalage ou retard. Cet opérateur permet de décaler une suite vers des indices inférieurs. On peut de cette façon accéder aux valeurs portées "dans le passé" par un signal. Ainsi, dans l'équation [3] `y := x $ 1 init c`, où `x` et `y` sont des signaux et `c` une constante d'initialisation (du même type que `y`), on a la relation :

- $(\forall t \in \mathbb{N})\quad x_t = \perp \quad \Leftrightarrow \quad y_t = \perp$
- $(\exists t_i \in \mathbb{N})\quad x_{t_i} \neq \perp \quad \Rightarrow \quad y_{t_0} = c,\ \ (\forall t_i > 0)\ y_{t_{i+1}} = x_{t_i}$

où $t_0 = inf\{t \mid x_t \neq \perp\}$ (le premier instant logique où `x` est présent) et $t_{i+1} = inf\{t \mid t > t_i \wedge x_t \neq \perp\}$ (le suivant immédiat d'un instant logique au sein de l'ensemble des instants où `x` est présent). Ici aussi les signaux `x` et `y` sont synchrones. Soit le processus `pre_x := x $ 1 init 3.14`, où `x` et `pre_x` sont des signaux réels. Une trace possible est la suivante (on introduit ici une horloge plus rapide que celle de `x` et `y`, dénotée par `h`) :

`h` :	*tick*	*tick*	*tick*	*tick*	*tick*	*tick*	*tick*	*tick*	*tick*	*tick*	...
`x` :	$\perp$	1.7	2.5	$\perp$	$\perp$	6.5	$\perp$	2.4	1.3	5.7	...
`pre_x` :	$\perp$	3.14	1.7	$\perp$	$\perp$	2.5	$\perp$	6.5	2.4	1.3	...

L'opérateur de retard existe aussi sous une *forme généralisée*. Cette version de l'opérateur permet d'accéder à la valeur qu'avait un signal k instants auparavant. Elle est exprimée à l'aide de l'équation `y := x $ k init tab_init`, où `tab_init` est un vecteur borné contenant les constantes d'initialisation, et `k` est un signal entier portant une valeur au moins égale à 0 et inférieure à la longueur de `tab_init`.

- *Opérateurs polychrones*

Sous-échantillonnage ou filtrage. Il permet d'extraire des éléments d'une suite, sous une certaine condition. Dans l'équation `y := x when b`, où `x` et `y` sont deux signaux du même type, et `b` est un signal booléen, `y` vaut `x` lorsque `b` est présent et porte la valeur *vrai* ; sinon `y` est absent (i.e. il vaut $\perp$). L'horloge de `y` est définie par l'intersection de celle de `x` et de l'ensemble des instants où `b` porte la valeur *vrai*.
Dans la trace suivante, `x` et `y` sont de type entier :

`x` :	...	$\perp$	5	$\perp$	4	8	7	3	$\perp$	...
`b` :	...	$\perp$	*vrai*	*faux*	$\perp$	*faux*	*vrai*	$\perp$	*vrai*	...
`y` :	...	$\perp$	5	$\perp$	$\perp$	$\perp$	7	$\perp$	$\perp$	...

3. On écrit aussi `y := x $ init c`.

L'expression `b when b` peut être abrégée en `when b`.

Mélange déterministe. Cet opérateur permet d'obtenir l'entrelacement fonctionnel de deux suites. Dans le processus `y:= u default v`, le signal `y` prend la valeur portée par le signal `u` lorsque celui-ci est présent, autrement `y` vaut `v` (en particulier, il est absent lorsque `u` et `v` sont absents). L'horloge de `y` est définie par l'union des horloges de `u` et `v`.
Une trace associée à ce processus est donnée ci-après (tous les signaux sont de type entier).

u :	...	⊥	5	⊥	4	8	⊥	⊥	3	⊥	...
v :	...	⊥	51	17	⊥	32	⊥	20	13	⊥	...
y :	...	⊥	5	17	4	8	⊥	20	3	⊥	...

4.1.1.2 Opérateurs de base sur processus

Composition parallèle. La composition de deux processus `P` et `Q`, notée `P | Q`, est l'union des systèmes d'équations définis par ces processus et donc leur sémantique est la conjonction de leurs comportements mutuels. Les deux processus communiquent via leurs signaux communs ; les signaux d'entrée de `P` peuvent être des signaux de sortie de `Q` et vice-versa. Aucun signal ne peut être défini au même instant dans les deux processus (SIGNAL respecte le principe d'assignation unique). On montre facilement les propriétés algébriques suivantes :

- la *commutativité* : `P | Q = Q | P`
- l'*associativité* : `P | (Q | R) = (P | Q) | R`
- l'*idempotence* : `P | P = P`

Confinement ou restriction. C'est l'opérateur qui permet de restreindre la portée d'un signal à un processus. L'expression[4] `P where x` rend le signal `x` inaccessible à l'extérieur du processus `P`. On peut alors montrer l'existence d'*éléments absorbants* (les processus vides sur un ensemble non vide de variables).

4.1.1.3 Opérateurs dérivés portant sur le contrôle pur

Nous indiquons ici les opérateurs qui sont utilisés par la suite pour exprimer des propriétés liées au contrôle.

- *Extraction d'horloge* : dans l'équation `h := ^x`, le signal de type événement pur `h` représente l'horloge de `x`. Cette équation est définie comme `h := (x = x)`.
- *Synchronisation explicite* : l'équation `x1 ^= x2` spécifie que les signaux `x1`, `x2` sont synchrones (l'opérateur est trivialement étendu à plusieurs signaux sous la forme `x1 ^= ... ^= xn`). Cette équation se définit comme `(| h' := (^x1 = ^x2) |) where h'`.
- *Intersections, unions et différences d'horloges* : l'équation `h' := x1 ^* x2` définit le signal `h'` comme étant l'intersection des horloges des signaux `x1` et `x2`. La définition de cette équation est `h' := ^x1 when (^x2)`. L'union et la différence ensembliste s'expriment respectivement au moyen des opérateurs ^+ et ^−.

```
process MODELE =
    { paramètres }
    ( ? entrées;
      ! sorties; )
  spec (| propriétés de l'interface |)
    (| spécification du comportement interne du processus |)
  where
    déclarations locales % signaux et sous-modèles %;
  end;
```

FIGURE 4.1 – Modèle de processus SIGNAL.

4.1.2 Modèle de processus SIGNAL

Les processus élémentaires définis sur les signaux peuvent être combinés par composition et confinement pour définir d'autres processus plus complexes sous la forme de *modèles de processus*. Un tel modèle est représenté sur la FIG. 4.1. Il est caractérisé par :

- une *interface* qui comporte un ensemble de paramètres statiques (exemple : constantes d'initialisation), un ensemble de signaux d'entrée repéré par le symbole `?`, un ensemble de signaux de sortie introduit à l'aide du symbole `!`, et des propriétés (essentiellement des relations de synchronisation et de dépendance) entre les signaux d'entrée/sortie, qui sont décrites dans la partie `spec`.
- un *corps* qui décrit d'une part le comportement interne du modèle, et d'autre part les déclarations locales. Ces dernières peuvent contenir des signaux, des définitions de sous-modèles, ou bien des interfaces de sous-modèles externes (compilés séparément ou bien matériels). Le symbole `%` sert à délimiter les commentaires.

```
process DECOMPTEUR =
     ( ? integer x;
       ! integer y;
     )
  (| y := x default (zy - 1)
   | zy := y $ 1 init 0
   | x ^= when (zy <= 0)
   |)
 where
    integer zy;
 end;
```

x	:	2	⊥	⊥	3	⊥	⊥	⊥	5	⊥	⊥	⊥	⊥	...
zy	:	0	2	1	0	3	2	1	0	5	4	3	2	...
y	:	2	1	0	3	2	1	0	5	4	3	2	1	...

FIGURE 4.2 – Modèle de processus SIGNAL avec *sur-échantillonnage*, et sa trace.

Exemple 1 (sur-échantillonnage) *Le processus* SIGNAL `DECOMPTEUR`*, donné sur la* FIG. *4.2,*

4. On écrit aussi `P/x` pour exprimer le confinement.

est un exemple classique qui spécifie un décompteur. Le fonctionnement de celui-ci est le suivant : il décrémente la valeur du signal d'entrée `x`*, et recommence le même traitement avec la prochaine entrée. Durant les calculs, il renvoie les valeurs intermédiaires de* `x` *sur son unique sortie* `y`*.*

Comme on peut le constater sur la trace de la FIG. *4.2, l'horloge la plus fréquente de ce processus n'est pas celle du signal d'entrée* `x`*. Celle-ci est synthétisée par le processus lui-même. On parle de* ***sur-échantillonnage****. Ce dernier consiste à ajouter de façon inductive un nombre arbitraire d'instants entre deux cycles d'horloge. Ce qui permet de spécifier des contraintes entre les entrées et sorties d'un système, de telle sorte qu'aucune entrée ne peut se présenter tant que les contraintes considérées n'ont pas été satisfaites par les calculs (intermédiaires) d'une sortie donnée. Ce mécanisme de sur-échantillonnage est une des caractéristiques spécifiques du modèle polychrone.*

4.2 Quelques propriétés formelles

Nous donnons une présentation formelle des objets de base du langage, ainsi que certaines notions utiles par la suite. Pour cela, nous adoptons le modèle sémantique polychrone proposé dans [147]. Il est basé sur le "méta modèle" *tagged signal* défini par Lee et Sangiovanni-Vicentelli [150], dans le but de pouvoir étudier différents modèles d'exécution dans un cadre uniforme.

4.2.1 Modèle sémantique polychrone

Soient :
- $\mathcal{X}$ un ensemble dénombrable de variables ;
- $\mathbb{B} = \{ff,tt\}$ l'ensemble des booléens où *ff* et *tt* dénotent respectivement *faux* et *vrai* ;
- $\mathcal{V}$ un ensemble représentant le domaine des opérandes dans une expression, contenant au moins $\mathbb{B}$;
- $\mathbb{T}$ un ensemble dense muni d'une relation d'ordre partiel notée $\leq$, dont les éléments sont appelés *tags*, tel que : toute paire $\{t_1, t_2\}$ avec $(t_1, t_2) \in \mathbb{T}^2$ admet une borne inférieure $inf\{t_1, t_2\}$ dans $\mathbb{T}$.

4.2.1.1 Définitions de base

Nous introduisons dans cette section les notions de base qui permettent de caractériser le modèle polychrone. Nous commençons par la notion de *points d'observation* :

Définition 3 (points d'observation) *On appelle ensemble de points d'observation (ou points d'observation en abrégé) tout ensemble de tags $\mathcal{T}$ qui satisfait les propriétés suivantes :*

1. *$\mathcal{T} \subset \mathbb{T}$,*
2. *$\mathcal{T}$ est dénombrable,*
3. *toute paire de tags de $\mathcal{T}$ possède un minorant dans $\mathcal{T}$.*

□

Les ensembles $\mathcal{T}$ et $\mathbb{T}$ vont servir de référentiels temporels suivant le type d'échelle considérée :

– $\mathcal{T}$ fournit une échelle de temps discret correspondant aux instants logiques auxquels sont "observés" les événements, ou l'absence d'événement d'un système qui s'exécute ;
– $\mathbb{T}$ permet de plonger les comportements du système, décrits suivant l'échelle de temps logique $\mathcal{T}$, dans une échelle de "temps réel" pour refléter une mise en œuvre de ce système sur une plate-forme donnée.

On appelle *chaîne* tout sous-ensemble C de $\mathbb{T}$ totalement ordonné possédant un minorant. L'ensemble des chaînes est représenté par le symbole $\mathcal{C}$. Pour un ensemble de points d'observation $\mathcal{T}$, on note $\mathcal{C}_\mathcal{T}$ l'ensemble des chaînes de $\mathcal{T}$. Les notations $min(C)$ et $pred_C(t)$ représentent respectivement le minimum et le prédécesseur immédiat du *tag* t dans une chaîne C. Nous présentons ci-après les notions de base (illustrées sur la Fig. 4.3). Les définitions sont toutes relatives à un ensemble de points d'observation donné. Cela ne sera pas mis explicitement en indice afin d'alléger l'écriture.

Définition 4 (événement) *Un événement e sur un ensemble de points d'observation $\mathcal{T}$ (un événement e, en abrégé) est un couple $(t, v) \in \mathcal{T} \times \mathcal{V}$.* □

L'ensemble des événements sur $\mathcal{T}$ est noté $\mathcal{E}_\mathcal{T}$. La notion définie ci-après, appelée *signal*, constitue un ensemble d'événements.

Définition 5 (signal) *Un signal sur un ensemble de points d'observation $\mathcal{T}$ (un signal, en abrégé) est une fonction partielle $s \in \mathcal{C}_\mathcal{T} \rightharpoonup \mathcal{V}$, qui aux points d'observation d'une chaîne dans $\mathcal{C}_\mathcal{T}$ associe des valeurs.* □

Le domaine de définition d'un signal s (l'ensemble des t de $\mathcal{T}$ tels que $s(t)$ est défini) est noté $tags(s)$. L'ensemble des signaux sur $\mathcal{T}$ est noté $\mathcal{S}_\mathcal{T}$.

Définition 6 (comportement) *Pour un ensemble de points d'observation $\mathcal{T}$, un comportement b sur $X \subseteq \mathcal{X}$ est une fonction $b \in X \longrightarrow \mathcal{S}_\mathcal{T}$ qui associe à chaque variable $x \in X$ un signal s sur $\mathcal{T}$.* □

On note $\mathcal{B}_{\mathcal{T},X}$ l'ensemble des comportements de domaine $X \subseteq \mathcal{X}$ sur un ensemble de points d'observation $\mathcal{T}$, et $\mathcal{B}_\mathcal{T}$ l'ensemble des comportements définis sur l'union de tous les ensembles de variables sur $\mathcal{T}$. On note $vars(b)$ le domaine d'un comportement b. L'ensemble des *tags* associé à b est représenté par $tags(b) = \bigcup_{x \in vars(b)} tags(b(x))$. Pour un ensemble de comportements B, on note $tags(B) = \bigcup_{b \in B} tags(b)$.

Pour un comportement b, défini sur Y et un ensemble $X \subset Y$, on note $b_{|X}$ sa projection sur X, c'est-à-dire, $vars(b_{|X}) = X$ et $\forall x \in X, b_{|X}(x) = b(x)$. La projection de b sur le complémentaire de X dans Y est dénotée par $b_{/X}$. Pour tout signal s, l'expression $s|_{\leq t}$ dénote le préfixe de s jusqu'au *tag* t, c'est-à-dire : $s|_{\leq t} = \{(t', v) \in s \mid t' \leq t\}$. Le préfixe d'un comportement est obtenu en considérant le préfixe de chacun de ses signaux.

Dans le modèle polychrone, les comportements d'un système sont spécifiés sur un ensemble discret tel que chaque instant dénote l'occurrence d'événements du système. Une telle description prend pleinement en compte la simultanéité et la précédence d'événements. C'est une vision qui facilite l'étude des aspects comportementaux d'un système temps réel (par exemple, l'ordonnancement). La validation des spécifications polychrones ainsi définies est réalisée à travers

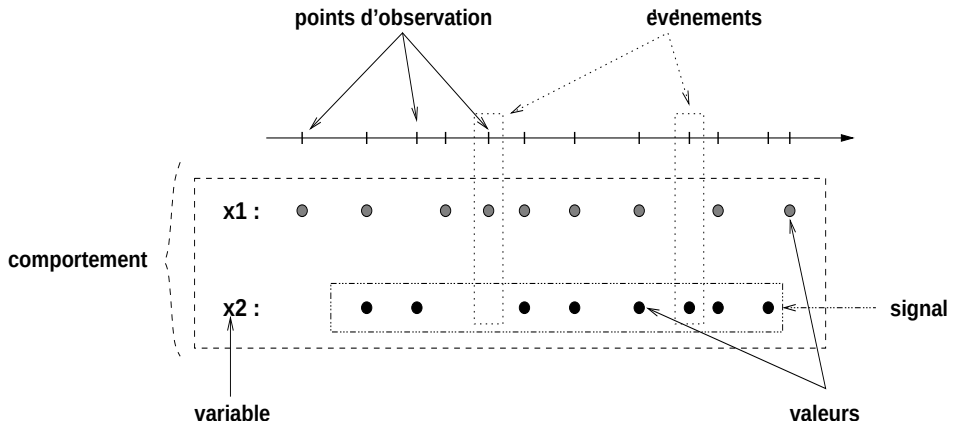


FIGURE 4.3 – Événements, signaux, et comportements.

un déploiement sur une échelle de temps continu (représentant le temps physique). À chaque événement du modèle polychrone, on peut alors associer l'instant physique auquel il apparaît lors d'une exécution effective sur une plate-forme donnée. Des aspects quantitatifs non traités dans le modèle polychrone abstrait peuvent alors l'être (par exemple, vérification de contraintes sur les durées des calculs).

Pour un modèle quelconque de système temps réel, le passage d'une échelle de temps discret à une échelle de temps continu doit tenir compte des performances de la plate-forme d'implantation. Selon la vitesse des processeurs disponibles pour réaliser les calculs et communications, un événement e observé à l'instant logique t peut être produit à des instants physiques différents durant des exécutions effectives. Pour une suite d'événements (i.e. un signal), cela induit des délais variables entre ses événements successifs. Par contre, ces derniers sont toujours produits dans le même ordre.

Le plongement d'un ensemble de comportements définis sur une échelle de temps initiale dans une autre échelle est facilement modélisable dans le modèle polychrone. L'intuition consiste à voir un signal comme un *élastique* qui porte des marques ordonnées. Lorsqu'il est *étiré*, ses marques demeurent dans le même ordre et on peut en ajouter d'autres entre deux marques étirées. Quand on le relâche, toutes les marques redeviennent proches les unes des autres. De plus, elles restent dans le même ordre. Au sein d'un comportement, si chaque signal est étiré de façon identique, l'ordre partiel entre les marques reste inchangé. Le mécanisme d'*étirement* de comportements (en anglais, *stretching*) est défini ci-après.

Définition 7 (étirement d'un comportement) *Pour un ensemble de points d'observation $\mathcal{T}$, et b_1, b_2 deux comportements de $\mathcal{B}_\mathcal{T}$, b_1 est moins étiré que b_2 (ou b_2 est un étirement de b_1), noté $b_1 \leq_{\mathcal{B}_\mathcal{T}} b_2$, ssi $vars(b_1) = vars(b_2)$ et il existe une bijection $f : tags(b_1) \to tags(b_2)$ suivant laquelle b_1 et b_2 sont isomorphes :*

$$\begin{array}{ll} \forall x \in vars(b_1) & f(tags(b_1(x))) = tags(b_2(x)), \\ \forall x \in vars(b_1)\ \forall t \in tags(b_1(x)) & b_1(x)(t) = b_2(x)(f(t)), \\ \forall t_1, t_2 \in tags(b_1) & t_1 \leq t_2 \Leftrightarrow f(t_1) \leq f(t_2), \end{array}$$

et telle que

$$\forall C \in \mathcal{C}_\mathcal{T}, \forall t \in C \quad t \leq f(t).$$

□

Sur la FIG. 4.4, nous avons illustré deux fonctions définissant deux étirements possibles (f' et f'') d'un comportement. Comme nous le verrons par la suite, la notion d'étirement joue un rôle très important dans la description de comportements temps réel. L'étirement étant une relation d'ordre partiel sur l'ensemble des comportements, on définit l'*équivalence* de comportements *modulo étirement*.

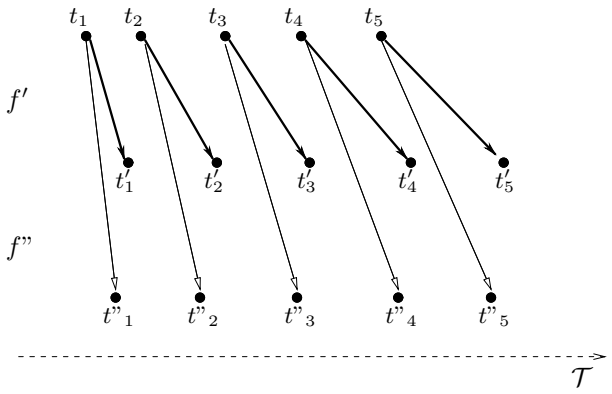


FIGURE 4.4 – Exemples d'étirements de comportement.

Définition 8 (équivalence modulo étirement) *Pour un ensemble de points d'observation $\mathcal{T}$, deux comportements b_1 et b_2 sont dits équivalents modulo étirement, noté $b_1 \lesseqgtr b_2$, ssi il existe un comportement b_3 moins étiré que chacun d'entre eux, c'est-à-dire :*

$$b_1 \lesseqgtr b_2 \text{ ssi } \exists b_3 \quad b_3 \leq_{\mathcal{B}_\mathcal{T}} b_1 \quad et \quad b_3 \leq_{\mathcal{B}_\mathcal{T}} b_2$$

□

La classe d'équivalence d'un comportement b selon étirement forme un *semi-treillis* admettant un comportement minimal. On appelle comportements *stricts* les comportements *minimaux* pour l'étirement sur $\mathcal{T}$. Étant donné un comportement b, l'ensemble de tous les comportements qui lui sont équivalents par étirement sur $\mathcal{T}$ définit sa *clôture par étirement* sur $\mathcal{T}$ notée b^*.

Définition 9 (clôture d'un ensemble de comportements) *On appelle clôture d'un ensemble p de comportements sur un ensemble de points d'observation $\mathcal{T}$, l'ensemble noté p^* union des clôtures b^* des comportements b de p :*

$$p^* = \bigcup_{b \in p} b^*$$

□

On définit de cette façon les *processus*.

Définition 10 (processus) *Pour un ensemble de points d'observation $\mathcal{T}$, on appelle processus tout ensemble de comportements $p \in \mathcal{P}(\mathcal{B}_\mathcal{T})$ clos par étirement, i.e., $p^* = p$.* □

On notera $vars(p)$ l'ensemble des variables des comportements que contient le processus p. On dira que p est défini sur $vars(p)$. Tout processus non vide p comporte un sous-ensemble $p_{\downarrow} \subseteq p$ de comportements stricts (pour tout $b_1 \in p$, il existe un unique $b_2 \in p_{\downarrow}$ tel que $b_2 \leqslant b_1$). Dans le reste de ce chapitre, on distinguera la représentation *syntaxique* d'un processus SIGNAL notée P, de sa représentation *sémantique* associée, notée $[\![\mathtt{P}]\!]$ (i.e., l'ensemble clos par étirement qui contient tous les comportements admis par P). Nous définissons ci-après les opérations sur les processus.

Définition 11 (composition) *Pour un ensemble de points d'observation $\mathcal{T}$ sur lequel sont définis deux processus p_1 et p_2, la composition synchrone $p = p_1 \mid p_2$ est un processus (ensemble de comportements clos par étirement sur $\mathcal{T}$), tel que :*

$$\begin{aligned} vars(p) &= vars(p_1) \cup vars(p_2), \\ p &= (\{b \mid b_{|vars(p_1)} \in p_1,\ b_{|vars(p_2)} \in p_2\})^* \end{aligned}$$

□

Nous avons mentionné quelques propriétés algébriques importantes de l'opérateur de composition dans la section 4.1.1.2. Une autre propriété de la composition est la *monotonie* [147].

Propriété 1 (monotonie) *Pour tous processus p_1, p_2, p_3, on a*

$$p_1 \subseteq p_2 \Rightarrow (p_1 \mid p_3) \subseteq (p_2 \mid p_3)$$

Définition 12 (restriction) *La restriction, notée p/x, d'un processus p défini sur X à un processus défini sur $X \backslash \{x\}$, est définie par :*

$$p/x = (\{b_2 \mid \exists b_1 \in p \ \wedge \ b_2 = b_{1/\{x\}}\})^*$$

□

Dans la définition ci-dessus, la clôture est nécessaire comme nous pouvons le constater dans la situation suivante : si dans tout comportement de p défini sur $\{x, y\}$, les occurrences de x et y alternent, alors dans tout comportement $b_{1/\{x\}}$ il existe, entre tout couple d'événements de tags t_1 et t_2, au moins un tag t'_1 auquel n'est associé aucun événement (seul x était présent à t'_1).

On note par le même symbole "|" la composition dans la syntaxe et dans la sémantique, et on a pour tous processus SIGNAL P et Q :

$$[\![\mathtt{P \ | \ Q}]\!] = [\![\mathtt{P}]\!] \mid [\![\mathtt{Q}]\!]$$

De même pour la restriction, on a :

$$[\![\mathtt{P/x}]\!] = [\![\mathtt{P}]\!]/\mathtt{x}$$

À présent, nous introduisons deux notions importantes permettant d'établir l'équivalence de comportements polychrones, y compris après leur désynchronisation. Pour cela, nous précisons d'abord la notion de *synchronisation* entre signaux dans le modèle polychrone. Étant donné un signal identifié par la variable x, son horloge, notée $\hat{x}$, est un signal qui porte la valeur *vrai* si et seulement si le signal est présent (les instants de présence étant définis par l'ensemble des *tags* associés au signal) :

$tags$	:	t_1	t_2	t_3	t_4	...
x	:	$v1$	$v2$	$v3$	$v4$	...
$\hat{x}$	:	tt	tt	tt	tt	...

Les relations de synchronisation sont exprimées à l'aide d'équations entre horloges de signaux. Par exemple, pour exprimer le fait que deux signaux identifiés par x et y sont synchrones, on écrit : $\hat{x} = \hat{y}$.

4.2.1.2 Comparaison de comportements désynchronisés

L'étirement définit une relation d'équivalence qui préserve à la fois la simultanéité et l'ordre des événements au sein d'un comportement. En d'autres termes, deux comportements dont l'un représente un étirement de l'autre possèdent les mêmes relations de synchronisation. Le modèle polychrone définit une notion de *relâchement* de comportement (en anglais, *relaxation*), moins "contraignante" que l'étirement dans le sens où, au sein d'un comportement issu du relâchement d'un autre comportement, les relations de synchronisation ne sont pas nécessairement préservées. En revanche, l'ordre des événements pour chaque signal reste inchangé. Le relâchement permet ainsi de comparer des comportements suivant l'ordre des points d'observation. Cela est utile notamment pour caractériser la désynchronisation de descriptions synchrones. En effet, le déploiement de ces dernières sur un modèle d'architecture distribuée requiert souvent un relâchement de certaines contraintes sur les horloges des signaux.

Définition 13 (relâchement d'un comportement) *Pour un ensemble de points d'observation $\mathcal{T}$, un comportement b_2 est un relâchement du comportement b_1, noté $b_1 \sqsubseteq b_2$, ssi*

$$vars(b_1) = vars(b_2) \quad et \quad \forall x \in vars(b_1),\ b_{1|\{x\}} \leq_{\mathcal{B}_\mathcal{T}} b_{2|\{x\}}$$

□

Le relâchement est aussi une relation d'ordre partiel sur les comportements. Il permet de définir l'*équivalence des flots* de valeurs de signaux appartenant à différents comportements. Intuitivement, deux comportements sont *flot-équivalents* (en anglais, *flow-equivalent*) si et seulement si ils ont le même domaine et les valeurs de chacun de leurs signaux sont dans le même ordre.

Définition 14 (équivalence de flots) *Pour un ensemble de points d'observation $\mathcal{T}$, deux comportements b_1 et b_2 sont dits flot-équivalents, noté $b_1 \approx b_2$, ssi il existe un comportement b_3 tel que*

$$b_3 \sqsubseteq b_1 \quad et \quad b_3 \sqsubseteq b_2$$

□

La classe d'équivalence d'un comportement b selon l'équivalence de flots forme un semi-treillis qui admet un comportement strict, noté $b_\approx$ (i.e. le représentant de la classe). Dans le modèle polychrone, l'équivalence de flots est un critère "minimal" de correction pour les raffinements d'une description purement synchrone d'un système sur une architecture asynchrone. En effet, les comportements initiaux de la description et ceux résultant de transformations de celle-ci en vue d'une exécution asynchrone, doivent avoir les occurrences de leurs signaux qui apparaissent dans le même ordre.

4.2.2 Modélisation polychrone de comportements temps réel

Nous abordons ici un certain nombre d'aspects liés à la modélisation de systèmes temps réel dans le cadre polychrone. Une observation importante est que les notions introduites dans le modèle facilitent le raisonnement en ce qui concerne les choix de conception : des questions relatives aussi bien à la spécification qu'à la mise en œuvre peuvent être abordées dans le même modèle.

Mise à l'échelle d'exécution des modèles. Supposons le plongement d'une échelle de temps dans une autre, où l'intervalle entre les instants est plus grand que dans l'échelle initiale. Grâce au mécanisme d'étirement, les comportements définis sur l'échelle initiale peuvent être ramenés à la nouvelle échelle.

Un cas typique où cette propriété du modèle polychrone trouve son utilité est la conception orientée composants dont l'un des avantages majeurs est la réutilisation. Dans un tel contexte, en général, lors de la définition des composants d'un système temps réel (où chacun de ces composants s'exécute à la fois sur un seul processeur), le développeur n'a pas vraiment à se préoccuper de la fréquence à laquelle le code produit sera exécuté ultérieurement. Cela dépend des plates-formes d'exécution considérées et du contexte applicatif de leur utilisation. Les fréquences d'exécution observées sont donc potentiellement variées. Grâce à l'étirement, une même spécification polychrone d'un composant donné peut être déployée sur différents supports d'exécution temps réel, n'ayant pas forcément les mêmes fréquences de calcul.

Un modèle polychrone abstrait d'un système peut être déployé sur différentes plates-formes, n'ayant pas *a priori* les mêmes fréquences d'exécution. Cela favorise la réutilisation.

*
* *

Afin d'illustrer nos propos en ce qui concerne les autres aspects abordés dans la suite, nous utiliserons des graphes orientés comme support de représentation des comportements d'un processus. Considérons l'exemple simple d'un processus `P` consistant en une addition de deux entiers : `P` $\equiv$ `z := x + y`. Dans une mise en œuvre effective de `P`, à chaque instant où il est effectué, le calcul de la valeur de `z` requiert au préalable les valeurs de `x` et `y`. Il y a donc des contraintes de précédence entre le signal `z` et les signaux `x` et `y`. De telles contraintes peuvent aussi être spécifiées explicitement dans un programme SIGNAL, entre des signaux qui ne sont pas liés *a priori* par des dépendances fonctionnelles (ce n'est pas le cas dans `P`, où des dépendances fonctionnelles existent entre `x`, `y` et `z`). Nous représentons les contraintes de notre exemple par $x \rightarrow z$ et $y \rightarrow z$ (on parle aussi de dépendances de valeurs entre signaux [39]). Ces dépendances sont valides à une certaine horloge (ici, il s'agit de l'horloge commune des signaux impliqués dans `P` puisque l'addition est un opérateur monochrone).

En considérant de telles relations de dépendance entre signaux d'un processus SIGNAL quelconque p, on munit tout comportement b de p d'un ordre partiel représenté par $\preccurlyeq_b$ qui est la fermeture transitive de la plus petite relation satisfaisant : $\forall x_1, x_2 \in vars(b)$, $t_1 \in tags(x_1)$, $t_2 \in tags(x_2)$,

$$\begin{array}{lcl} t_1 \leq t_2 & \Rightarrow & (t_1, x_1) \preccurlyeq_b (t_2, x_1) \\ x_1 \rightarrow x_2, t_1 \in tags(x2) & \Rightarrow & (t_1, x_1) \preccurlyeq_b (t_1, x_2) \end{array}$$

Pour le processus `P` $\equiv$ `z := x + y`, on a pour tout comportement $b \in [\![\texttt{P}]\!]$ et pour tout $t \in tags(b)$, $(t, x) \preccurlyeq_b (t, z)$ et $(t, y) \preccurlyeq_b (t, z)$. C'est ce qui est exprimé à travers les sous-graphes

de la FIG. 4.5. Intuitivement, chaque comportement b peut être caractérisé à l'aide de tels sous-graphes (chacun d'entre eux correspond à une réaction instantanée du processus). Chaque t_i représente un instant logique du modèle polychrone (pour l'instant, nous ne faisons pas apparaître l'échelle de temps physique. Elle sera mise en évidence sur la FIG. 4.6).

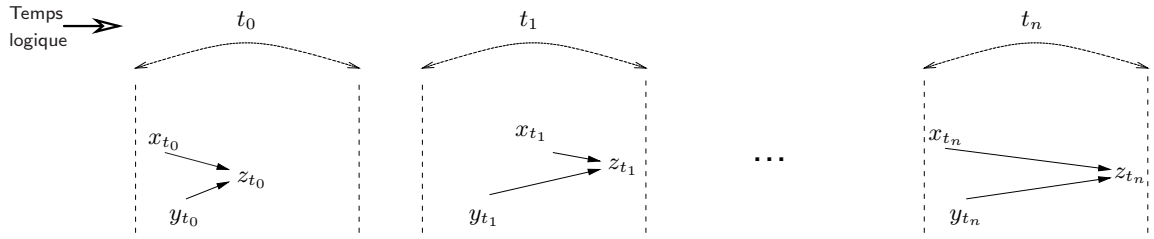


FIGURE 4.5 – Sous-graphes associés à $b \in [\![\texttt{z := x + y}]\!]$ respectant l'ordre $\preccurlyeq_b$.

Ces sous-graphes peuvent donner lieu à diverses interprétations de mise en œuvre : les ordres d'arrivée des paramètres d'entrée `x` et `y` varient durant l'évolution du système ; la production de la sortie `z` est plus longue à certains instants logiques. Dans ce qui suit, nous nous baserons sur des notions définies à la section 4.2.1 pour exposer intuitivement une façon d'aborder le "temps réel" dans le modèle polychrone.

Contraintes temps réel. Le déploiement des sous-graphes de la FIG. 4.5 sur une échelle de temps continu modélise une exécution effective. Les instants physiques auxquels les événements apparaissent y sont dénotés par les τ_k sur la FIG. 4.6. Du point de vue du modèle polychrone, le sous-graphe[5] de l'instant logique t_0 sur la FIG. 4.6 peut être "étiré" autant qu'on veut, pourvu que l'ordre partiel $\preccurlyeq_b$ soit respecté. Par exemple, on pourrait imaginer un étirement par rapport à l'échelle de temps continu qui fournit la valeur de z_{t_0} entre les instants physiques τ_4 et τ_5. L'introduction d'une échelle de temps physique peut être vue d'une certaine manière comme un pas vers la prise en compte de contraintes temps réel (i.e., spécification des bornes de temps de réaction). Typiquement, on peut imposer que la réaction à l'instant t_0 se produise strictement entre les instants τ_0 et τ_4. Dans ce cas, le sous-graphe considéré de b ne peut être étiré qu'entre ces deux instants physiques. Nous observons ainsi que le fait de considérer des contraintes temps réel sur le modèle polychrone d'un système se traduit de manière intuitive, par la définition d'un *intervalle de clôture*[6] pour chaque sous-graphe caractérisant un comportement. Ce genre de contraintes est généralement imposé par l'environnement du système. Par exemple, cela peut être à travers une clause de la forme : "émettre des sorties tant d'unités de temps *au plus* après avoir reçu des entrées". Le système se voit ainsi imposer un délai critique à ne pas excéder lors de son fonctionnement. Ce qui se traduit simplement dans le modèle polychrone par le choix d'une longueur maximale d'étirement correspondant à ce délai, pour tout sous-graphe d'un comportement du processus SIGNAL spécifiant le système.

On doit naturellement se poser la question de la vérification des contraintes induites par le choix d'un intervalle de clôture pour un comportement (en d'autres termes, est-ce que tous les sous-graphes caractérisant ce dernier satisfont les contraintes de l'intervalle ?). Cela s'inscrit dans le cadre de la validation du modèle d'un système. Plusieurs approches sont envisageables pour y répondre. Celle que nous considérons dans cette thèse, proposée initialement par A.

5. Par abus de langage, nous nous autorisons à utiliser le terme "étirement" pour ces graphes.

6. Tout étirement "licite" des sous-graphes d'un comportement admissible ne doit pas excéder cet intervalle défini sur l'échelle de temps continu.

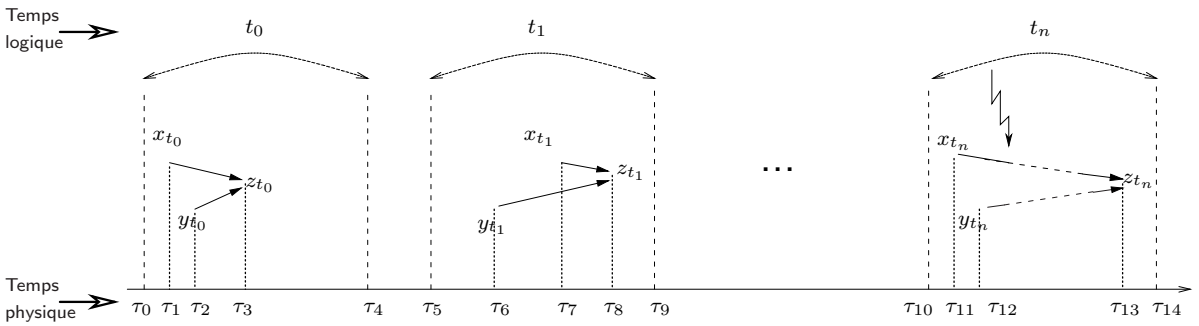


FIGURE 4.6 – Sous-graphes associés à $b \in [\![\texttt{z := x + y}]\!]$ sur une échelle de temps physique.

Kountouris [140], est entièrement basée sur le modèle polychrone. Une présentation détaillée en est donnée dans la section 4.4.3. La technique utilisée consiste à dériver à partir d'un programme SIGNAL un autre programme (servant d'observateur), qui reflète des propriétés *non fonctionnelles* du programme de départ. En l'occurrence, ce programme observateur détermine des durées de calculs effectués dans le programme initial. Pour cela, il prend en compte des paramètres caractérisant une plate-forme (par exemple, des durées d'exécution d'opérations élémentaires comme la somme de deux entiers codés sur 32 bits) et des dates d'acquisition d'entrées. À partir de ces informations, il calcule des dates de production des sorties. La durée globale des calculs peut alors être déduite. En utilisant une telle technique, nous pouvons identifier et écarter les comportements qui ne respectent pas la contrainte imposée par l'intervalle de clôture. D'autres approches de validation reposent sur l'utilisation de techniques analytiques à l'image de celles qu'offre l'*algèbre Max-Plus* [9]. Enfin, des techniques basées sur les modèles temporisés peuvent aussi être mentionnées. Par exemple, dans TAXYS [65], les automates temporisés servent de support pour l'analyse temporelle.

Les contraintes imposées par l'environnement permettent de déterminer l'intervalle de clôture pour les sous-graphes caractérisant un comportement. Il existe un autre type de contraintes, induites cette fois-ci par le support d'exécution. Il s'agit de durées minimales nécessaires à la réalisation des opérations effectuées par un système. En général, on dispose d'une estimation des temps d'exécution des opérations élémentaires (comme les opérations arithmétiques ou logiques) sur une plate-forme. C'est un moyen d'avoir *a priori* une idée du temps minimal requis pour une exécution globale. La prise en compte de ces contraintes dans un modèle du système peut entraîner le rejet de certains comportements pour la mise en œuvre considérée. Ce sont les comportements au sein desquels des sous-graphes ne sont pas suffisamment étirés pour que la longueur de l'étirement soit supérieure aux durées minimales imposées par la plate-forme considérée (pour que ceux-ci satisfassent les contraintes, ils doivent être étirés davantage).

La FIG. 4.7 illustre les deux types de contraintes temps réel que nous venons de distinguer. Ici, g_1, g_2, g_3, g_4 et g_5 symbolisent des étirements de sous-graphes caractérisant un même comportement b (i.e., ils expriment des propriétés fonctionnelles identiques ; ils se différencient uniquement à travers l'étirement). On remarque que g_1, g_2, g_3 et g_4 respectent l'intervalle de clôture défini par l'environnement tandis que ce n'est pas le cas de g_5. En revanche, par rapport aux contraintes imposées par la plate-forme d'implantation, g_1 exhibe une exécution plus rapide que ne le permet la plate-forme considérée. Dans ce contexte particulier, la prise en compte des contraintes temps réel entraînera le rejet du comportement b car il ne respecte pas toujours celles-ci. Ainsi, le modèle polychrone concret d'un système temps réel (c'est-à-dire,

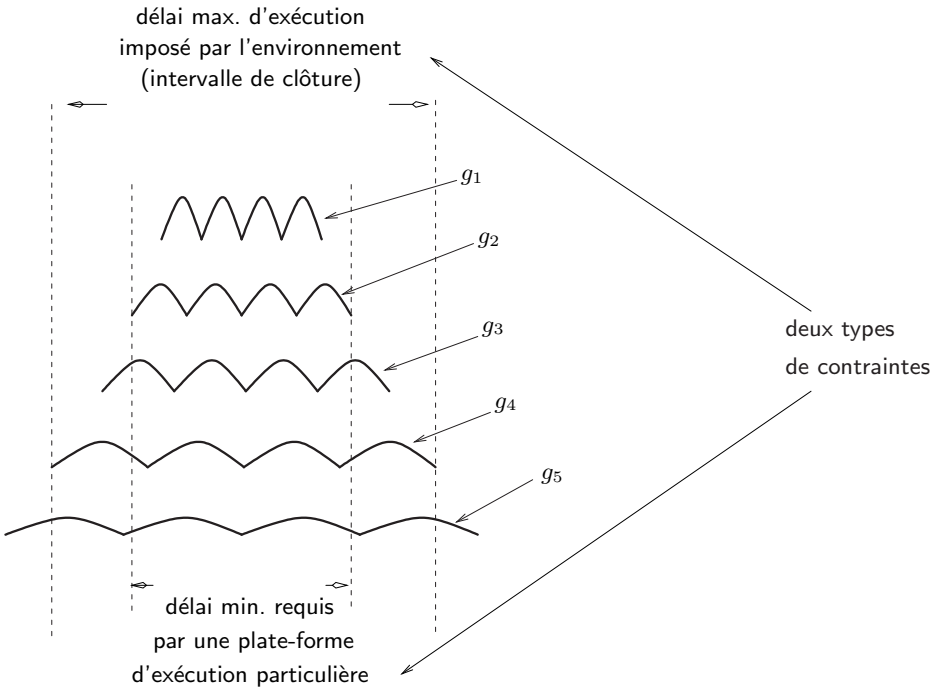


FIGURE 4.7 – Deux types de contraintes sur le délai d'exécution.

où les contraintes du support d'exécution et de l'environnement sont prises en compte) résulte d'une restriction du nombre de comportements du modèle abstrait associé.

Interruptions dans une exécution temps réel. Lors d'exécutions d'un système temps réel, des interruptions sont susceptibles de se produire. Par exemple, des calculs peuvent être interrompus afin de réaliser des communications ou des synchronisations (ce qui a pour conséquence de rallonger la durée des calculs). Du point de vue du modèle polychrone abstrait, ces interruptions ne "brisent" pas les instants logiques. En effet, les calculs étant entièrement reflétés par les sous-graphes caractérisant chaque comportement du modèle, toute éventuelle interruption est mise en évidence par un étirement de ces sous-graphes. Une longue interruption se traduit donc par un long étirement des sous-graphes. Par exemple sur la FIG. 4.6, le sous-graphe de l'instant logique t_n peut être interprété comme le résultat induit par des interruptions par rapport au sous-graphe de l'instant t_0 (la production de l'événement z_{t_n} dans le graphe de l'instant t_n consomme un délai plus long que dans celui de l'instant t_0, après réception des événements x_{t_n} et y_{t_n}).

Nous avons vu que tout processus SIGNAL est clos par étirement. Autrement dit, il contient tous les comportements possibles du système spécifié (y compris donc ceux qui peuvent résulter de la survenue d'interruptions lors de ses exécutions). Ces comportements traduisent les mêmes propriétés fonctionnelles et ne se différencient qu'à travers l'étirement. Par conséquent, le choix d'un représentant quelconque de cet ensemble est suffisant pour étudier les aspects fonctionnels du système modélisé par le processus. Cette caractéristique du modèle polychrone permet de se concentrer sur les aspects purement comportementaux du fonctionnement d'un système temps réel, tout en restant capable d'aborder les aspects non fonctionnels.

Une autre façon de modéliser les interruptions consiste à considérer des signaux d'entrée supplémentaires dont la présence dénote l'occurrence d'événements à la base de celles-ci. Dans

ce cas, la production des sorties sera différée ou non en fonction du statut de ces nouveaux signaux. Dans cette seconde façon de représenter les interruptions, on est amené à briser les instants logiques. Pour cela, on procède au raffinement de l'échelle temporelle.

Raffinement temporel. Le mécanisme de sur-échantillonnage du langage SIGNAL, introduit précédemment (cf. section 4.1.2), permet le raffinement temporel de descriptions polychrones : à partir d'une échelle de temps logique donnée, on peut définir une échelle plus fine, grâce à laquelle on peut observer de possibles décalages temporels d'événements produits par le système durant son évolution. Ce raffinement temporel ne modifie pas l'ordre partiel $\preccurlyeq_b$ dans les sous-graphes associés à tout comportement b du processus. Seule l'échelle de temps change. Puisqu'il y a potentiellement plus de points d'observation sur une échelle de temps sur-échantillonné, les sous-graphes peuvent être étirés de façon à préciser certains décalages temporels non visibles entre des événements sur l'échelle initiale. Par exemple, la production de la valeur de `z` dans le processus `P` $\equiv$ `z := x + y` nécessite un certain délai après réception des valeurs de `x` et `y`. Nous pouvons donc imaginer un schéma d'exécution *2-step*, composé de *deux instants logiques* : le premier dénote l'acquisition des entrées (i.e., valeurs de `x` et `y`) et le second, la production des sorties (i.e., valeur de `z`). Cela revient tout simplement à décomposer en deux, par sur-échantillonnage, chaque instant logique t_k de la FIG. 4.5. De façon similaire, on peut continuer à raffiner la description jusqu'à la mise en évidence des décalages [7] entre les instants d'acquisition d'une part et ceux de production d'autre part. On irait alors vers un schéma d'exécution *n-step* (où n dénote le nombre d'instants logiques résultant du sur-échantillonnage de l'échelle de temps initiale). Plus généralement, le raffinement temporel d'un processus est défini comme suit :

Définition 15 (raffinement temporel) *Soit p un processus défini sur un ensemble de points d'observation $\mathcal{T}_1$, le raffinement temporel de p sur un ensemble de points d'observation $\mathcal{T}_2$ tel que $\mathcal{T}_1 \subset \mathcal{T}_2$ est défini par la clôture par étirement de p sur $\mathcal{T}_2$.* □

Nous observons ainsi que des comportements temps réel peuvent être décrits de façon précise dans le modèle polychrone. Le plongement de l'ensemble des points d'observation dans une échelle de temps physique se fait alors très facilement.

* * *

Le modèle polychrone offre à travers ses caractéristiques un cadre formel intéressant pour décrire et étudier différents aspects liés à la conception de systèmes temps réel. Dans les chapitres suivants, nous nous attacherons à montrer la pertinence de ce modèle à travers l'utilisation du langage SIGNAL, qui en est le formalisme associé.

4.3 Abstraction syntaxique

Un programme SIGNAL `P` décrit à la fois les synchronisations entre ses signaux ainsi que le calcul de leurs valeurs. L'isolation de ces aspects permet de traiter plus facilement des propriétés spécifiques du programme. Celui-ci peut donc être *abstrait* au travers d'un autre programme `P'` qui reflète uniquement les aspects concernés par les propriétés qu'on souhaite vérifier. Le programme `P'` est obtenu à l'aide de transformations syntaxiques. Celles-ci permettent souvent des décompositions en paires du programme initial `P`, distinguant par exemple :

7. En général, les entrées ne sont pas forcément reçues en même temps (idem pour la production des sorties).

– la partie *combinatoire*[8], induite par les processus élémentaires statiques (fonctions instantanées, sous-échantillonnage et mélange déterministe) et la partie *mémorisation*, définie par la seule construction primitive dynamique du langage (décalage) ;
– la partie *contrôle*, constituée par les booléens et les horloges, et la partie *calcul*.

Une présentation détaillée de ces différentes décompositions se trouve dans [166]. Nous introduisons ici le principe de base de ces transformations.

4.3.1 Principe

Il résulte immédiatement de la définition de la composition que pour tous processus P, P'

$$([\![\mathtt{P}]\!] \mid [\![\mathtt{P'}]\!])_{|vars([\![\mathtt{P}]\!])} \subseteq [\![\mathtt{P}]\!]$$

On peut montrer plus précisément que

$$[\![\mathtt{P}]\!] = ([\![\mathtt{P}]\!] \mid [\![\mathtt{P'}]\!])_{|vars([\![\mathtt{P}]\!])} \text{ ssi } [\![\mathtt{P}]\!]_{|vars([\![\mathtt{P'}]\!])} \subseteq [\![\mathtt{P'}]\!]_{|vars([\![\mathtt{P}]\!])}$$

Autrement dit, tout comportement de $[\![\mathtt{P}]\!]$ est compatible avec au moins un comportement de $[\![\mathtt{P'}]\!]$.
On a donc, lorsque $vars([\![\mathtt{P}]\!]) = vars([\![\mathtt{P'}]\!])$

$$[\![\mathtt{P}]\!] \mid [\![\mathtt{P'}]\!] = [\![\mathtt{P}]\!] \text{ ssi } [\![\mathtt{P}]\!] \subseteq [\![\mathtt{P'}]\!]$$

Si P' dénote une propriété Q, elle sera donc satisfaite par tous les comportements de P ssi

$$[\![\mathtt{P}]\!] \mid [\![\mathtt{Q}]\!] = [\![\mathtt{P}]\!]$$

L'*abstraction syntaxique par décomposition* d'un programme SIGNAL P consiste à dériver syntaxiquement, à partir des équations qui constituent celui-ci, un autre programme $\alpha(\mathtt{P})$, où α dénote cette abstraction. $\alpha(\mathtt{P})$ est ensuite utilisable pour analyser P.

Définition 16 (abstraction syntaxique par décomposition) *Étant donné un processus* P*, le processus* $\alpha(\mathtt{P})$ *tel que* $vars([\![\alpha(\mathtt{P})]\!]) \subseteq vars([\![\mathtt{P}]\!])$ *résulte d'une abstraction syntaxique par décomposition* α *de* P *ssi :*

$$[\![\mathtt{P}]\!] \mid [\![\alpha(\mathtt{P})]\!] = [\![\mathtt{P}]\!]$$

ou de façon équivalente ssi :

$$[\![\mathtt{P}]\!] \subseteq [\![\alpha(\mathtt{P})]\!]$$

□

Puisque l'abstraction α d'un programme P repose sur une décomposition, elle induit une relation de *complémentarité* entre $\alpha(\mathtt{P})$ et une partie de P. Si $\rho(\mathtt{P})$ dénote une telle partie (ρ est mis pour "résidu"), à une abstraction α d'un programme SIGNAL P, on peut associer une[9] décomposition syntaxique (α, ρ) définie par un ensemble de règles de récriture :

$$\mathtt{P} \longrightarrow \alpha(\mathtt{P}) \mid \rho(\mathtt{P})$$

8. Appelée aussi partie *statique*.
9. En fait, parmi une infinité.

satisfaisant :

$$[\![\mathtt{P}]\!] = [\![\alpha(\mathtt{P}) \mid \rho(\mathtt{P})]\!]$$

On montre que $\alpha(\mathtt{P})$ est bien une abstraction de P ; en effet,

$$\begin{array}{rcll}
[\![\mathtt{P}]\!] \mid [\![\alpha(\mathtt{P})]\!] & = & [\![\alpha(\mathtt{P}) \mid \rho(\mathtt{P})]\!] \mid [\![\alpha(\mathtt{P})]\!] & \text{(par définition)} \\
& = & [\![\alpha(\mathtt{P})]\!] \mid [\![\rho(\mathtt{P})]\!] \mid [\![\alpha(\mathtt{P})]\!] & \text{(par définition de } \mid) \\
& = & [\![\alpha(\mathtt{P})]\!] \mid [\![\alpha(\mathtt{P})]\!] \mid [\![\rho(\mathtt{P})]\!] & \text{(par commutativité de } \mid) \\
& = & [\![\alpha(\mathtt{P})]\!] \mid [\![\rho(\mathtt{P})]\!] & \text{(par idempotence de } \mid) \\
& = & [\![\alpha(\mathtt{P}) \mid \rho(\mathtt{P})]\!] & \text{(par définition de } \mid) \\
& = & [\![\mathtt{P}]\!] &
\end{array}$$

L'abstraction α résultant de cette décomposition doit de plus être un endomorphisme idempotent : lorsqu'on abstrait par α un programme P', issu déjà d'une première application de la même abstraction à un programme P, le résultat P'' ne doit pas induire de nouveau relâchement de la sémantique de P'.

Définition 17 (décomposition syntaxique) *On appelle décomposition syntaxique tout couple de transformations (α, ρ) satisfaisant pour tout processus élémentaire* P *:*

$$[\![\alpha(\mathtt{P}) \mid \rho(\mathtt{P})]\!] = [\![\mathtt{P}]\!]$$

et tel que α est idempotente :

$$[\![\alpha(\alpha(\mathtt{P}))]\!] = [\![\alpha(\mathtt{P})]\!]$$

La décomposition syntaxique (α, ρ) est alors définie inductivement pour la composition et le confinement :

$$\begin{array}{rcl}
\alpha(\mathtt{P}_1 \mid \mathtt{P}_2) = \alpha(\mathtt{P}_1) \mid \alpha(\mathtt{P}_2) & \textit{et} & \rho(\mathtt{P}_1 \mid \mathtt{P}_2) = \rho(\mathtt{P}_1) \mid \rho(\mathtt{P}_2) \\
\alpha(\mathtt{P/x}) = (\alpha(\mathtt{P}))\mathtt{/x} & \textit{et} & \rho(\mathtt{P/x}) = (\rho(\mathtt{P}))\mathtt{/x}
\end{array}$$

□

Exemples d'utilisations :

- Pour étudier la consistance des contraintes de synchronisation de P, on abstrait le programme en ne gardant que ses synchronisations. Celles-ci correspondent à la plus grande relation qui lie les horloges de P (cf. FIG. 4.10).
- Pour étudier la dynamique du système d'équations formant P, on abstrait celui-ci par ses variables d'état. Elles sont définies par les opérateurs dynamiques du langage (cf. FIG. 4.8). Par conséquent, on ne s'intéresse spécifiquement qu'à ces opérateurs.

Dans [166], l'abstraction α a été reformulée dans le cadre de l'*interprétation abstraite* [72] qui est une théorie générale permettant d'approximer la sémantique des systèmes dynamiques discrets. L'interprétation abstraite exprime l'analyse statique comme une correspondance formelle entre la sémantique concrète d'un programme et une sémantique abstraite guidée par la propriété à prouver. Une propriété importante de α qui a été démontrée dans [166] est sa monotonie.

Propriété 2 *Pour tous programmes* $\mathtt{P}_1$ *et* $\mathtt{P}_2$*, on a*

$$[\![\mathtt{P}_1]\!] \subseteq [\![\mathtt{P}_2]\!] \Rightarrow [\![\alpha(\mathtt{P}_1)]\!] \subseteq [\![\alpha(\mathtt{P}_2)]\!]$$

4.3.2 Abstractions usuelles

Nous présentons dans cette section deux exemples d'application d'abstractions syntaxiques à un programme SIGNAL pour en extraire des informations. Ces exemples concernent plus précisément l'extraction de la partie statique (i.e., la partie qui ne comporte aucune expression définie à l'aide de constructions dynamiques comme le *retard*), et l'extraction de la partie définissant le contrôle d'un programme.

Abstraction statique (ou combinatoire). Une définition de cette abstraction est obtenue à travers le partitionnement des constructions du noyau de SIGNAL en deux classes : d'une part, les primitives portant sur des données purement combinatoires (fonctions instantanées, filtrage et mélange déterministe) et d'autre part, l'unique primitive sur signaux qui induit une *mémorisation* (retard). En utilisant ce partitionnement, un programme `P` est décomposé syntaxiquement en une partie statique $\alpha_{stat}(\texttt{P})$ et une partie dynamique $\rho_{stat}(\texttt{P})$.

La FIG. 4.8 indique les parties $\alpha_{stat}(\texttt{P})$ et $\rho_{stat}(\texttt{P})$ pour chaque construction de base sur signaux. On remarquera que pour le retard, il existe aussi une partie statique qui exprime le fait que les signaux impliqués sont synchrones. On vérifie facilement que α_{stat} préserve bien la sémantique des programmes et elle est sémantiquement idempotente. Pour tout processus élémentaire `P`, on montre que

$$[\![\alpha_{stat}(\texttt{P})]\!] \mid [\![\texttt{P}]\!] = [\![\texttt{P}]\!]$$

En effet, pour les fonctions instantanées, le filtrage et le mélange déterministe, on le vérifie par idempotence de la composition. En ce qui concerne l'opérateur de retard, la propriété reste vraie puisque l'équation `y ^= x` n'ajoute aucune contrainte supplémentaire à celle qui correspond à `P` (c'est-à-dire, `y := x $ 1 init c`).

P	$\alpha_{stat}(\texttt{P})$	$\rho_{stat}(\texttt{P})$
`y := f(x1,...,xn)`	`y := f(x1,...,xn)`	$\emptyset$
`y := x $ 1 init c`	`y ^= x`	`y := x $ 1 init c`
`y := x when b`	`y := x when b`	$\emptyset$
`y := u default v`	`y := u default v`	$\emptyset$

FIGURE 4.8 – Parties statiques et dynamiques des processus primitifs sur signaux.

Abstraction par le contrôle. Le contrôle d'un programme SIGNAL est formé d'une part, de sa partie purement booléenne, et d'autre part, des synchronisations. Pour l'extraire, nous considérons donc deux abstractions intermédiaires permettant d'obtenir les deux parties qui le composent.

– *Abstraction purement booléenne.* Étant donné un programme `P`, cette abstraction consiste à isoler autant que possible le composant $\alpha_{bool}(\texttt{P})$ défini uniquement sur des variables de type booléen. Des raisonnements purement booléens peuvent ainsi être appliqués à ce composant. La FIG. 4.9 illustre une décomposition syntaxique de `P` qui permet de définir $\alpha_{bool}(\texttt{P})$ ainsi que sa partie complémentaire $\rho_{bool}(\texttt{P})$. Dans le tableau, `expr` [10], `non_bool_expr` [11] et `bool_expr` [12] représentent des expressions élémentaires, c'est-à-dire,

10. Toute expression de valeur non booléenne.
11. Typiquement, toute expression définie avec un opérateur de comparaison arithmétique comme `<`, `<=`, `>` ou `>=`.
12. Typiquement, toute expression définie avec un opérateur logique tel que `and`, `or`, `not`.

définies à l'aide d'opérateurs élémentaires, où les opérandes sont des variables ou des constantes.

P	$\alpha_{bool}(\mathtt{P})$	$\rho_{bool}(\mathtt{P})$
`x := expr` et $Dom(x) \neq \mathbb{B}$	$\emptyset$	P
`x := non_bool_expr` et $Dom(x) = \mathbb{B}$	$\emptyset$	P
`x := bool_expr` et $Dom(x) = \mathbb{B}$	P	$\emptyset$

FIGURE 4.9 – Séparation des parties booléennes et non booléennes d'un processus.

Les variables booléennes qui ne sont pas définies dans la partie $\alpha_{bool}(\mathtt{P})$ sont considérées comme des entrées du programme abstrait. La transformation α_{bool} vérifie bien les propriétés d'une abstraction syntaxique :

$$[\![\alpha_{bool}(\mathtt{P})]\!] \mid [\![\mathtt{P}]\!] = [\![\mathtt{P}]\!]$$

– *Abstraction par les synchronisations.* Pour définir cette abstraction, on se base sur le fait qu'une équation SIGNAL spécifie une relation portant à la fois sur les valeurs des variables impliquées et sur les horloges des variables (représentant les synchronisations). L'abstraction α_{sync}, illustrée sur la FIG. 4.10, consiste alors à ne garder que les informations d'horloges d'un programme P. Le programme $\alpha_{sync}(\mathtt{P})$ résultant est en relation de complémentarité avec P lui-même (i.e. $\rho_{sync}(\mathtt{P}) = \mathtt{P}$). D'autre part, puisqu'il n'induit aucune contrainte supplémentaire sur P, on montre trivialement que :

$$[\![\alpha_{sync}(\mathtt{P})]\!] \mid [\![\mathtt{P}]\!] = [\![\mathtt{P}]\!]$$

P	$\alpha_{sync}(\mathtt{P})$
`y := f(x1,...,xn)`	`y ^= x1 ^= ... ^= xn`
`y := x $ 1 init c`	`y ^= x`
`y := x when b`	`y ^= x when b`
`y := u default v`	`y ^= u ^+ v`

FIGURE 4.10 – Extraction des synchronisations.

On observe que α_{sync} représente une abstraction de l'abstraction statique α_{stat} déjà mentionnée, c'est-à-dire :

$$[\![\alpha_{stat}(\mathtt{P})]\!] \subseteq [\![\alpha_{sync}(\mathtt{P})]\!]$$

Au moyen des abstractions α_{bool} et α_{sync}, le contrôle d'un programme SIGNAL P est déduit en composant les sous-programmes résultant de l'application de celles-ci à P. On peut donc définir l'abstraction par le contrôle comme suit :

$$\alpha_{cont}(\mathtt{P}) = \alpha_{bool}(\mathtt{P}) \mid \alpha_{sync}(\rho_{bool}(\mathtt{P}))$$

Celle-ci est en relation de complémentarité avec la partie *calculs* (représentée par $\rho_{cont}(\mathtt{P})$ qui peut être défini comme $\rho_{bool}(\mathtt{P})$. En effet, de $\mathtt{P} = \alpha_{bool}(\mathtt{P}) \mid \rho_{bool}(\mathtt{P})$, on décompose $\rho_{bool}(\mathtt{P})$ suivant les synchronisations ; on obtient alors $\mathtt{P} = \alpha_{bool}(\mathtt{P}) \mid (\alpha_{sync}(\rho_{bool}(\mathtt{P})) \mid \rho_{bool}(\mathtt{P}))$, d'où on déduit l'égalité suivante : $\mathtt{P} = \alpha_{cont}(\mathtt{P}) \mid \rho_{bool}(\mathtt{P})))$.

Le programme $\alpha_{cont}(\mathtt{P})$ convient pour la vérification de propriétés sur P à l'aide de l'outil de *model-checking* SIGALI [160], disponible dans POLYCHRONY. Cet outil utilise un encodage du programme fourni en entrée dans le corps $\mathbb{Z}/3\mathbb{Z}$, représenté par l'ensemble $\{-1, 0, 1\}$: les valeurs -1, 1 et 0 dénotent respectivement la "présence avec la valeur *faux*", la "présence avec

la valeur *vrai*" et l' "absence". Cet encodage prend en compte complètement la valeur des signaux booléens, mais pas celle des signaux numériques. Nous reviendrons plus en détail sur SIGALI dans le chapitre 6. L'analyse statique effectuée par le compilateur SIGNAL, appelée *calcul d'horloges* [39] [16] [167], se base sur la partie *contrôle statique* de P. Cette partie est équivalente au programme $\alpha_{stat}(\alpha_{cont}(\mathtt{P}))$.

Concrètement, étant donné un programme P à analyser par le compilateur, ses parties contrôle et traitements de données sont d'abord séparées [13] [39] [16]. Ensuite, une *analyse structurelle* est effectuée sur la partie contrôle statique du programme : cela consiste en un ensemble de transformations sémantiquement préservantes au système d'équations d'horloges. Il en résulte alors une *forme* dite *résolue* de P. La représentation interne de celle-ci, adoptée par le compilateur, est appelée *graphe hiérarchisé aux dépendances conditionnées* (ou GHDC). Ce graphe sert de base à toutes les analyses proposées à la compilation. Chacun de ses nœuds contient un ensemble de signaux présents à une même horloge. Cela permet par exemple d'étudier la consistance des flots de données spécifiés dans un processus (i.e., la compatibilité des contraintes imposées sur les horloges des signaux). On peut aussi vérifier des propriétés comme l'absence totale de réaction de la part du système décrit (en d'autres termes, les contraintes dans le système d'équations d'horloges imposent à toutes les horloges d'être nulles). Par ailleurs, la forme résolue facilite l'établissement d'un ordre suivant lequel le programme pourrait être exécuté. Le calcul correspondant s'appuie en partie sur l'ordre partiel des horloges, qui correspond à l'inclusion des ensembles d'instants (une horloge pouvant être plus fréquente qu'une autre).

4.4 Abstraction sur valeurs

Dans cette section, nous présentons de nouvelles transformations de programmes SIGNAL. Celles-ci permettent notamment d'aborder des propriétés quantitatives d'un programme (à savoir, une approximation des valeurs de signaux numériques à l'aide d'intervalles afin d'étendre le calcul d'horloges − section 4.4.2 − et une estimation des coûts d'exécution pour la validation temporelle − section 4.4.3). Elles reposent sur la notion d'*observateur* que nous introduisons dans la section 4.4.1.

4.4.1 Principe pour la définition d'un observateur

Un observateur Q d'un programme P consiste en une abstraction de celui-ci dans laquelle on spécifie des propriétés de P qu'on souhaite vérifier. C'est un programme qui n'ajoute aucune contrainte sur les comportements de P.

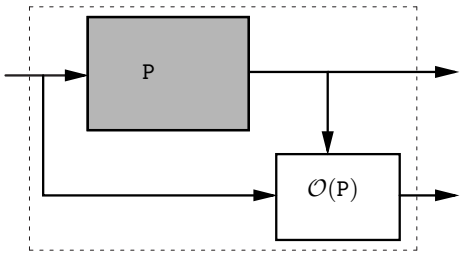


FIGURE 4.11 – Un programme P et son observateur $\mathcal{O}(\mathtt{P})$.

13. Cela peut être réalisé à l'aide des transformations syntaxiques qui viennent d'être présentées.

Comme le montre la FIG. 4.11, l'observateur ($\mathcal{O}(\mathtt{P})$) reçoit du programme "observé" ($\mathtt{P}$) les signaux requis pour une analyse donnée, et indique si les propriétés considérées sont satisfaites (par exemple, à travers des signaux booléens en sortie).

Définition 18 (observateur) *Un programme* $\mathtt{Q}$ *est un observateur d'un programme* $\mathtt{P}$*, noté* $\mathtt{Q} = \mathcal{O}(\mathtt{P})$ *ssi*

$$(\mathtt{Q} \mid \mathtt{P})_{|vars([\![\mathtt{P}]\!])} = \mathtt{P} \quad et \quad vars([\![\mathtt{P}]\!]) \cap vars([\![\mathtt{Q}]\!]) \neq \emptyset$$

□

L'utilisation d'observateurs pour la vérification est très pratique du fait que ceux-ci peuvent être décrits dans le même formalisme que le programme observé. Il n'est donc pas nécessaire de combiner plusieurs formalismes contrairement à ce qu'on constate dans certaines techniques d'analyse (comme par exemple celles qui associent la logique temporelle aux automates pour faire du *model-checking*). La vérification de propriétés de sûreté en LUSTRE est notamment basée sur une utilisation d'observateurs synchrones qui expriment les propriétés à vérifier [113]. Ceux-ci calculent une sortie booléenne qui demeure vraie tant que la propriété considérée est satisfaite.

Le même principe est applicable pour analyser des modèles décrits en SIGNAL. En particulier, il permet d'appréhender certains aspects quantitatifs tels que des propriétés liées aux valeurs de signaux lors des calculs, ou l'estimation du coût d'une opération pour valider des comportements temps réel. L'observateur d'un programme SIGNAL peut être obtenu à travers des transformations formellement justifiées par le modèle sémantique du langage (de façon similaire à ce qui a été présenté à la section 4.3.1). Dans les sections 4.4.2 et 4.4.3, nous illustrons deux schémas de construction d'observateurs pour aborder des propriétés numériques et évaluer des temps d'exécution d'un programme $\mathtt{P}$. Dans chacun de ces schémas, l'observateur $\mathcal{O}(\mathtt{P})$ est une *interprétation* de $\mathtt{P}$, considérée sur un nouveau domaine de calcul pour certaines variables de $\mathtt{P}$. Par exemple, dans la transformation décrite en section 4.4.2, les signaux numériques sont considérés sur des ensembles finis d'intervalles et non sur des ensembles de valeurs simples comme $\mathbb{N}$ ou $\mathbb{R}$ (i.e., à tout instant, la valeur d'un signal est représentée par un intervalle formé par les valeurs susceptibles d'être prises par ce signal). L'interprétation introduite dans la section 4.4.3 effectue quant à elle des calculs sur des types de données qui dépendent d'une métrique (par exemple, un type "date" pour déterminer les instants de disponibilité de signaux pendant les calculs). Lorsqu'on compose $\mathtt{P}$ avec chacune de ces deux interprétations, son comportement original n'est en aucun cas perturbé. C'est une propriété primordiale que celles-ci doivent satisfaire.

4.4.2 Extension du calcul d'horloges

L'abstraction utilisée dans l'outil SIGALI ne permet de prendre en compte que les valeurs de signaux booléens. Dans les équations portant sur des signaux numériques, seules les relations de synchronisation sont encodées. Quelques travaux ont été menés dans le but de rendre plus "robustes" les techniques d'analyse dans POLYCHRONY [42] [166]. L'étude que nous avons abordée dans [90] s'inscrit dans le cadre de l'extension du calcul d'horloges. Nous avons étudié une abstraction de programmes SIGNAL dans laquelle les domaines de valeurs numériques sont représentés par des ensembles finis d'intervalles. Cette étude sert surtout de base de réflexion pour aller vers des solutions adéquates.

L'idée consiste dans un premier temps à abstraire syntaxiquement un programme P afin d'en extraire la partie statique des calculs numériques, c'est-à-dire le programme $\alpha_{stat}(\rho_{cont}(\mathtt{P}))$. Ce dernier est ensuite considéré sur un nouveau domaine où les calculs ne portent plus sur de "simples" valeurs de signaux, mais sur des intervalles qui approximent l'ensemble des valeurs que peut prendre chaque signal. Par exemple, pour le processus $\mathtt{P} \equiv$ `z := x + y`, on obtient l'interprétation suivante :

$$\mathcal{O}(\mathtt{P}) \equiv \mathtt{I_z} := (\mathtt{I_x} + \mathtt{I_y})\ \mathtt{when\ (\hat{}z)} \quad (\text{"SIGNAL avec intervalle"})$$

où $\mathtt{I_v}$ désigne un "*signal-intervalle*" dont chaque occurrence indique des valeurs associées à l'expression v. Si on compose P et $\mathcal{O}(\mathtt{P})$, avec x, y, $\mathtt{I_x}$ et $\mathtt{I_y}$ comme signaux d'entrée, on récupère en sortie la valeur exacte de z (calculée à partir de celles de x et y) ainsi que l'intervalle $\mathtt{I_z}$ (qui contient logiquement la valeur calculée de z). Ces deux sorties sont produites à la même horloge ^z. L'arithmétique des intervalles est une théorie développée notamment pour les calculs numériques [10]. Nous nous en sommes servis pour définir des opérations sur un nouveau type d'expressions, appelées *i-formules*, destinées à représenter une interprétation à l'aide d'intervalles.

Une i-formule est une expression qui combine des notions de la logique propositionnelle et des calculs sur intervalles. Elle est de la forme suivante :

$$((x_1, I_1) \wedge exp\_bool_1) \vee ... \vee ((x_m, I_m) \wedge exp\_bool_m)$$

où chaque couple (x_k, I_k) dénote l'appartenance d'une variable numérique x_k à l'intervalle I_k ; le terme exp_bool_k représente une formule de la logique des propositions. Typiquement, de telles formules peuvent encoder des horloges. Par exemple, l'interprétation $\mathtt{I_z} := (\mathtt{I_x} + \mathtt{I_y})$ `when (^z)` est traduite par la i-formule suivante :

$$z \in I_{x+y} \ \wedge \ horl_z$$

où $horl_z$ est une formule propositionnelle encodant l'horloge de z (qui est aussi l'horloge commune).

Les i-formules sont mises en œuvre par des structures de données appelées IBDD (*Interval Binary Decision Diagrams*), inspirées des BDD. La FIG. 4.12 illustre les IBDD associés aux i-formules suivantes :

$$\begin{aligned} I_1 &\equiv (x_1, [0, 10[) \wedge B) \vee ((x_2,]5, 13]) \wedge B') \\ I_2 &\equiv (x, [0, 10[) \wedge B) \vee ((x,]20, 30]) \wedge B') \end{aligned}$$

Les valeurs *false* et *true* sont représentées respectivement par les constantes 0 et 1. Chaque étiquette d'un arc indique la valeur de l'expression booléenne correspondant au sommet d'origine. Les sommets supérieurs d'un IBDD dénotent des appartenances de variables numériques à des intervalles, tandis que les sommets inférieurs débouchent sur des formules propositionnelles (les feuilles simples et celles qui sont représentées par un triangle sont des BDD). L'opérateur "+" exprime ici la disjonction de B et B'.

L'interprétation du calcul numérique basée sur les i-formules peut améliorer le raisonnement effectué sur les programmes SIGNAL lors du calcul d'horloges. Elle apporte des informations supplémentaires qui ne peuvent être déduites par le compilateur dans sa version actuelle. L'abstraction par le contrôle statique ne suffit pas pour déduire certaines informations par rapport

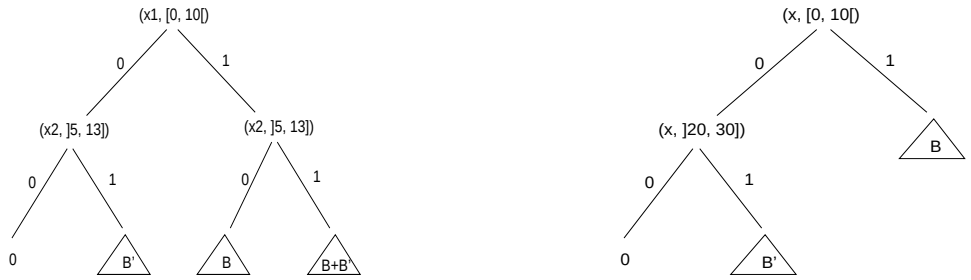


FIGURE 4.12 – IBDD représentant I_1 (à gauche) et I_2 (à droite).

aux valeurs de signaux numériques. Typiquement, avec une représentation sous forme de i-formules, on peut détecter l'exclusion de deux horloges définies par des expressions numériques dont les intervalles sont disjoints (il suffit d'évaluer leur conjonction). Ce qui n'est pas possible actuellement avec l'abstraction prise en compte dans le calcul d'horloges, d'où l'intérêt d'une extension.

4.4.3 Évaluation de comportements temps réel

Cette section présente une technique générale permettant de définir une *interprétation non fonctionnelle* de programmes SIGNAL. Par, "non fonctionnelle", nous sous-entendons une interprétation qui permet d'observer la façon dont les comportements attendus d'un programme sont obtenus (par exemple, cela peut être du point de vue consommation de temps ou d'énergie). La technique, qui repose sur l'application d'un *morphisme de programmes*, a été proposée par Apostolos Kountouris pendant sa thèse [140], puis développée dans le cadre du projet SAFEAIR, pour l'évaluation de performances. L'idée générale consiste à dériver à partir d'un programme SIGNAL initial, un autre programme (l'observateur associé), qui se comporte de la même façon que le programme original du point de vue du contrôle ; par contre, les aspects purement fonctionnels diffèrent. Le nouveau programme va ainsi refléter des aspects *non fonctionnels* de celui de départ, selon une métrique de performance donnée (comme le temps d'exécution). Les sections suivantes présentent la mise en œuvre de la technique dans le contexte du langage SIGNAL.

4.4.3.1 Approche générale

Grâce au schéma d'abstraction syntaxique présenté à la section 4.3, un programme SIGNAL P peut être décomposé en ses parties contrôle $\alpha_{cont}(\mathtt{P})$ et traitements de données $\rho_{cont}(\mathtt{P})$ (cf. FIG. 4.13) :

$$[\![\mathtt{P}]\!] = [\![\alpha_{cont}(\mathtt{P}) \mid \rho_{cont}(\mathtt{P})]\!]$$

L'ensemble des entrées de P est dénoté par $E = E_b \cup E_D$. Il est composé en partie de signaux booléens et événements (E_b), qui sont utilisés par la partie contrôle ($\alpha_{cont}(\mathtt{P})$) pour calculer l'ensemble des horloges d'activation (H) de la partie traitements de données ($\rho_{cont}(\mathtt{P})$). Les autres entrées (E_D) sont directement utilisées par $\rho_{cont}(\mathtt{P})$. Parmi les signaux calculés par ce dernier, il y a des booléens représentant des informations de contrôle (B), qui sont envoyés à $\alpha_{cont}(\mathtt{P})$. Les autres signaux produits par $\rho_{cont}(\mathtt{P})$ (i.e. S_D) constituent avec les événements de sortie de $\alpha_{cont}(\mathtt{P})$ (i.e. H_S), les sorties du processus P ; en d'autres termes, $S = S_D \cup H_S$. Cette décomposition du programme P permet de distinguer les parties nécessaires à la construction d'un observateur $\mathcal{O}(\mathtt{P})$. En l'occurrence, la partie contrôle de P sera utilisée telle quelle dans $\mathcal{O}(\mathtt{P})$. Quant à sa partie traitements de données, elle servira à définir la composante de l'observateur qui reflète les calculs basés sur la métrique choisie pour l'évaluation (exemples : temps

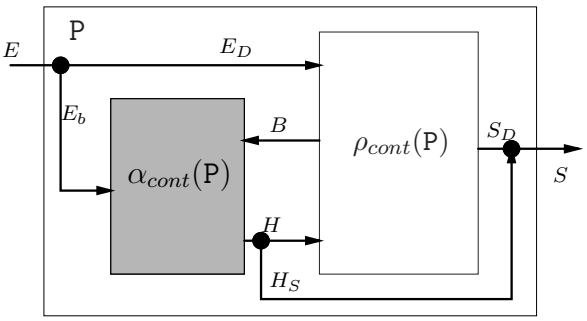


FIGURE 4.13 – Décomposition d'un processus P, en ses parties contrôle et traitements de données.

de réponse, énergie).

Par la suite, nous adoptons la notation suivante : pour P désignant un programme, le programme désignant son interprétation *non* fonctionnelle suivant une métrique m est noté $\mathcal{N}_m(\mathtt{P})$ (i.e., un observateur de P suivant m).

Définition 19 *Étant donné un processus* SIGNAL P*, son interprétation non fonctionnelle suivant une métrique m est un autre processus* SIGNAL $\mathcal{N}_m(\mathtt{P})$ *ayant une partie contrôle identique, et une partie traitements de données qui reflète les informations relatives à m :*

$$\alpha_{cont}(\mathcal{N}_m(\mathtt{P})) = \alpha_{cont}(\mathtt{P}) \quad et \quad \rho_{cont}(\mathcal{N}_m(\mathtt{P})) = \mathcal{N}_m(\rho_{cont}(\mathtt{P}))$$

□

Le schéma de dérivation de l'interprétation non fonctionnelle d'un processus P est donné sur la FIG. 4.14. Les nouvelles entrées du processus dérivé, représentées par $\mathcal{N}_m(E)$, dénotent des données requises pour calculer les informations relatives à la métrique m. Elles sont respectivement synchrones aux entrées E. Il en est de même pour les sorties $\mathcal{N}_m(S)$. Enfin, toutes les informations de contrôle (c'est-à-dire, B et E_b) deviennent des entrées du processus $\mathcal{N}_m(\mathtt{P})$.

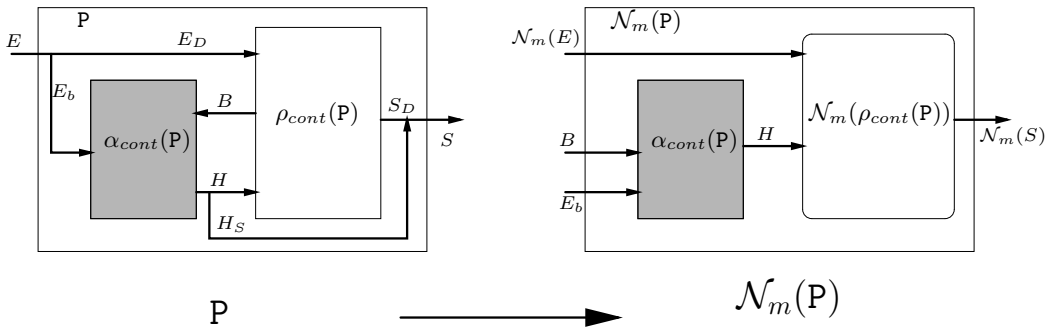


FIGURE 4.14 – Dérivation de l'interprétation non fonctionnelle d'un processus P.

À présent que nous disposons d'un modèle générique d'interprétation non fonctionnelle d'un programme, nous pouvons appréhender un certain nombre de questions intéressantes dans le cadre du choix d'une implantation à adopter. Par exemple, nous pouvons évaluer l'impact de ce choix par rapport à l'énergie consommée par l'exécution du programme, au temps d'exécution, ou un autre aspect.

4.4.3.2 Application : évaluation temporelle

La métrique considérée pour le modèle présenté dans cette section est le temps d'exécution. Ainsi, la notation $\mathcal{N}_t(\mathtt{P})$ exprime le modèle temporel d'un processus P, où t désigne une métrique "temps".

Caractéristiques de l'interprétation temporelle. Étant donné un programme SIGNAL P, on associe à chacun des signaux apparaissant dans celui-ci une "image temporelle". Par convention, cette dernière est dénotée à l'aide de l'identifiant du signal associé, précédé du préfixe "*date_*" :

$$\forall x \in \mathtt{P} \; \exists \; date\_x \in \mathcal{N}_t(\mathtt{P}) \quad (\mathcal{N}_t(x) = date\_x)$$

Le signal *date_x* représente la date de disponibilité des valeurs de x dans le programme P. Les deux signaux sont synchrones :

$$x \; \hat{}= \; date\_x$$

Par ailleurs, on attribue à chaque processus élémentaire de P, une certaine quantité de temps $\delta_{\mathtt{P}}$. Celle-ci est du même type que les dates de disponibilité des valeurs de signaux. Elle symbolise le délai nécessaire pour que l'opération correspondante soit effectuée. Nous rappelons que la structure des processus SIGNAL est récursive. Chaque processus est composé de sous-processus. Ces derniers sont eux-mêmes constitués d'autres sous-processus. Les constructions du noyau du langage représentent les processus atomiques dans ce schéma récursif. Ce sont aussi les nœuds du GHDC associé au processus. La mesure de $\delta_{\mathtt{P}}$ dépend de plusieurs facteurs parmi lesquels nous pouvons citer l'implantation matérielle du programme. Par exemple, une addition de grands entiers, codés sur 32 bits, peut se faire en un cycle sur un processeur disposant d'un additionneur 32 bits (c'est le cas du Pentium IV d'Intel). Par contre, il faut plus d'un cycle sur un processeur ne disposant que d'un additionneur 16 bits (c'est notamment le cas du "8086"). La quantité $\delta_{\mathtt{P}}$ dépend aussi du type de données considéré dans les calculs. Par exemple, une addition sur des entiers ne consomme pas la même quantité de temps qu'une addition sur des flottants. Le modèle temporel des processus doit tenir compte de tous ces facteurs.

Pour chaque processus du noyau, un modèle générique a été défini. Le lecteur trouvera ces modèles dans [140]. Nous illustrons ci-après à travers l'exemple d'une simple addition, l'obtention de l'interprétation temporelle des fonctions instantanées.

Exemple 2 *Soit le processus suivant :* `x := y + z`*, il s'agit de déterminer son interprétation temporelle. Sur la* FIG. *4.15, le nœud à gauche montre la façon dont l'addition de deux entiers est vue dans le GHDC. Cette vision est opérationnelle. Pour calculer la valeur du signal* `x`*, on a besoin de celles de* `y` *et* `z` *(sur le schéma, la valeur de chaque signal est représentée par l'identifiant correspondant). D'autre part, ce calcul requiert aussi les horloges nécessaires à l'activation du nœud. L'entrée h_x est l'horloge commune à* `x`*,* `y` *et* `z`*. L'interprétation temporelle du processus est donnée par le nœud de droite, sur la même figure. Toutes les données d'entrée ont été remplacées par leurs dates de disponibilité (ajout du préfixe "date_" à chaque identifiant). Les signaux date_begin$_i$ et date_done$_i$ ont été ajoutés pour indiquer la fin du traitement d'un nœud afin de passer aux nœuds suivants. Ainsi, la présence de date_begin$_i$ signifie que tous les nœuds précédant le nœud courant ont été traités. Lorsque ce nœud a également été traité (ici, quand la valeur de x a été calculée), date_done$_i$ devient alors présent. Les nœuds suivants dans le graphe peuvent à leur tour être traités. L'addition simple des valeurs de* `y` *et* `z` *est remplacée par le calcul de la date de* `x`*, noté δ_+. Comme nous l'avons déjà évoqué, ce calcul dépend d'un certain nombre de critères liés notamment à l'architecture d'implantation, au type des données, etc.*

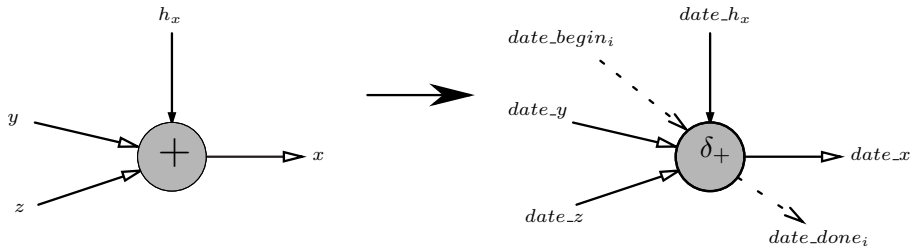


FIGURE 4.15 – Nœud du graphe correspondant à l'addition `x := y + z` (gauche) et son modèle temporel (droite).

Le calcul d'une date sera donc spécifié à l'aide d'un processus paramétré comme sur la FIG. *4.16. Les types* `type_y` *et* `type_z` *sont génériques dans le processus* `CALCUL_DATE_ADD` *(par exemple, (*`type_y`*,* `type_z`*) peut être (*`integer`*,* `integer`*), (*`real`*,* `real`*) ou (*`integer`*,* `real`*)). On retrouve les mêmes entrées et sorties que sur la* FIG. *4.15. Dans le corps du processus, la seconde équation définit la date de* `done_i`*. Nous allons plutôt nous concentrer sur la première qui exprime le calcul de la date de* `x`*.*

```
process CALCUL_DATE_ADD
     { type type_y, type_z; }
     ( ? date_type date_y, date_z, date_h_x, date_begin_i;
       ! date_type date_x, date_done_i )
     (| date_x := MAX2( MAX3( date_y, date_z, date_h_x), date_begin_i when ^date_x)
                  + DELAI_ADD{type_y, type_z}()
      | date_done_i := (date_x default date_begin_i) cell ^date_done_i
      |)
```

FIGURE 4.16 – Processus SIGNAL exprimant le modèle temporel de `x := y + z`.

Intuitivement, celle-ci est égale à la date de disponibilité maximale des entrées, à laquelle est ajouté le délai de l'opération d'addition. Ce délai est donné par la fonction `DELAI_ADD`*. Celle-ci est paramétrée par les types des opérandes (*`type_y` *et* `type_z`*) : c'est là que sont pris en compte les paramètres liés à l'architecture d'implantation. En fait, nous disposons d'une base de données dans laquelle nous avons collecté des informations concernant diverses plates-formes. Elles indiquent par exemple le coût sur un processeur ϕ, d'une addition de deux entiers, de deux flottants, d'un entier et d'un flottant, idem pour les autres opérations "élémentaires" du langage. Ainsi, connaissant a priori quelques caractéristiques d'une architecture matérielle, nous pouvons arriver à estimer le temps de calcul nécessaire pour une opération donnée. Il est donc indispensable de disposer d'une base de données suffisamment fournie.*

Obtention des résultats pour des analyses. La FIG. 4.17 montre comment est réalisée la *co-simulation* d'un programme avec son interprétation temporelle, en vue d'évaluer le temps d'exécution.

Un *générateur de dates* fournit les dates de disponibilité ($date\_e_i$) pour chaque entrée (e_i) du programme issue du *générateur d'entrées*. À chaque activation de l'ensemble, des sorties (s_j) ainsi que leurs dates de présence correspondantes ($date\_s_j$) sont produites. En plus des dates

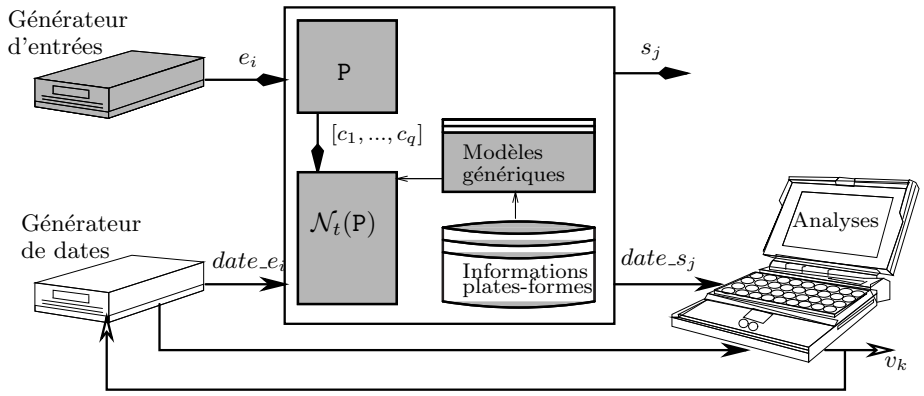


FIGURE 4.17 – Co-simulation d'un programme SIGNAL et de son interprétation temporelle.

des entrées, le calcul des dates des sorties requiert la *configuration courante du contrôle*. Celle-ci est représentée par une valuation du vecteur de conditions $[c_1, ..., c_q]$, qui correspond aux signaux booléens intermédiaires B, calculés par la partie traitements de données du processus P (cf. FIG. 4.14). Enfin, une bibliothèque SIGNAL qui contient les modèles génériques pour chaque processus élémentaire, accompagnée d'une base de données regroupant des informations sur les temps d'exécution d'opérations élémentaires (exemple : addition/multiplication/division d'entiers) sur des plates-formes données, sont utilisées. On peut vérifier un certain nombre de propriétés temporelles par rapport aux différents comportements du programme. Par exemple, la *latence optimale* du système spécifié (c'est-à-dire, le temps que consomme la production des données de sortie une fois les entrées acquises) peut être obtenue en mettant simplement les dates en entrée à zéro. On peut aussi calculer le temps d'exécution d'un système après un nombre quelconque d'activations, en ré-injectant dans le générateur de dates les informations sur les dates de sortie pour accumulation.

4.5 Résumé

SIGNAL est un langage synchrone orienté flot de données adapté à la spécification et la mise en œuvre d'applications temps réel. Nous avons présenté ses fondements mathématiques grâce auxquels on caractérise des comportements *polychrones*, c'est-à-dire des comportements de systèmes au sein desquels existent plusieurs horloges différentes. On considère deux référentiels temporels : une échelle discrète, dénotée par $\mathcal{T}$, qui définit les instants logiques auxquels on observe la présence et l'absence d'événements d'un système temps réel ; et une échelle dense, représentée par $\mathbb{T}$, qui sert pour le temps physique et dans laquelle des comportements décrits sur $\mathcal{T}$ sont plongés pour modéliser une mise en œuvre donnée. Des notions telles que le mécanisme d'étirement, définies dans le modèle sémantique, permettent de représenter le passage de $\mathcal{T}$ à $\mathbb{T}$. Par ailleurs, ce modèle sémantique offre la possibilité de transformer un programme dans le but d'en extraire les parties pertinentes pour une analyse donnée. Les techniques de vérification développées dans POLYCHRONY, l'environnement de conception associé à SIGNAL, utilisent ces transformations. Les principaux outils qui mettent en œuvre ces techniques sont le compilateur et le *model-checker* SIGALI. Le premier permet de vérifier les propriétés fonctionnelles statiques d'un système (par exemple, une absence de réaction, la présence de contraintes d'horloges non satisfaites), tandis que le second permet de traiter en plus des propriétés fonctionnelles dynamiques (par exemple, l'accessibilité d'un état du système). Un exemple illustrant une analyse à l'aide de SIGALI est présenté dans le chapitre 6. Les propriétés non fonctionnelles peuvent être étudiées quant à elles grâce à des interprétations de programmes SIGNAL qui reposent

sur l'application d'un *morphisme de programmes*. Cette technique offre un moyen d'évaluer les performances d'une application temps réel décrite en SIGNAL. Dans le chapitre 9, le schéma proposé pour l'évaluation des temps d'exécution de l'application étudiée utilise cette technique. Toutes ces caractéristiques du langage SIGNAL sont fournies sous forme de fonctionnalités dans l'environnement POLYCHRONY.

Les transformations de programmes SIGNAL présentées dans ce chapitre facilitent l'analyse des comportements spécifiés. Elles se prêtent également à la définition de schémas en vue du déploiement des programmes sur des architectures matérielles. Cela correspond au passage d'une description de l'échelle de temps logique $\mathcal{T}$ à l'échelle de temps physique $\mathbb{T}$ (la technique d'évaluation de comportements temps réel introduite en section 4.4.3 permet de valider un tel passage). Le chapitre suivant décrit une démarche de conception d'applications distribuées temps réel à l'aide de SIGNAL. Cette démarche est basée sur des transformations s'appuyant sur les propriétés du *graphe hiérarchisé aux dépendances conditionnées* (représentation interne d'un programme au sein du compilateur).

Chapitre 5

Méthodologie de conception d'applications distribuées temps réel à l'aide de SIGNAL

La mise en place de systèmes à grande échelle nécessite parfois l'interconnexion d'autres systèmes qui peuvent être hétérogènes à travers un réseau (par exemple, c'est le cas avec le réseau *Internet* qui relie des systèmes situés sur des sites géographiques différents). On peut citer en exemple les supercalculateurs, les systèmes industriels ou les stations de travail. Les raisons d'une telle répartition sont diverses : gain de performance engendré par l'utilisation de plusieurs unités de calculs pour un meilleur temps de réponse, délocalisation géographique des éléments du système, réplication des systèmes pour la tolérance aux fautes.

Dans la conception des applications temps réel distribuées, un certain nombre d'aspects critiques doivent être pris en compte. Par exemple, en ce qui concerne les échanges entre tâches situées sur des sites (calculateurs) différents, on ne peut pas négliger le temps des communications. On doit s'assurer que les mécanismes de transfert sont suffisamment robustes (i.e., il ne doit pas y avoir de perte de messages). Pour ces raisons, la répartition d'une application doit suivre une méthodologie bien définie qui offre les concepts nécessaires pour, d'une part, exprimer le parallélisme présent dans ces applications et d'autre part, permettre la validation de la description qui en résulte.

Plusieurs approches ont été proposées pour apporter des solutions aux problèmes que pose la conception des applications distribuées. Nous avons déjà évoqué des modèles d'architectures distribuées temps réel tels que TTA ou LTTA et l'existence de certaines approches qui sont basées sur des langages tels que ADA [174], ou qui définissent des langages spécifiques comme c'est le cas dans GIOTTO [120] ou METAH [210] (cf. chapitres 1 et 2). Enfin, d'autres utilisent des mécanismes de distribution automatique de programmes. Par exemple, dans SYNDEX [195] [106], l'outil se base sur certaines heuristiques pour répartir une application de façon à exploiter au mieux les ressources de calcul disponibles. Dans toutes ces démarches, une préoccupation majeure vise à faciliter le travail du concepteur en le déchargeant des problèmes liés au parallélisme, laissés à des outils adéquats tels que les compilateurs. Un autre point crucial concerne la validation des applications. Celle-ci détermine la correction d'une mise en œuvre vis-à-vis des exigences. À ce jour, on peut trouver de nombreux outils et de nombreuses techniques permettant de simuler une application afin d'en évaluer les performances, de prouver des propriétés de sûreté de fonctionnement, d'équité, etc. Plus particulièrement en ce qui concerne les applications distribuées temps réel, la vérification des propriétés non fonctionnelles comme les contraintes

temporelles demeure une tâche très délicate.

Dans ce chapitre, nous présentons une méthodologie de conception d'applications distribuées dans l'environnement POLYCHRONY. Étant donné un programme SIGNAL initial P spécifiant une application, on procède par transformations successives respectant la sémantique de P, jusqu'à obtenir un nouveau programme P' reflétant l'architecture d'implantation voulue. Les transformations considérées sont formellement justifiées grâce à certains résultats théoriques établis dans le modèle polychrone. La section 5.1 donne une vision générale de la méthodologie. Elle en présente notamment les fondements théoriques (section 5.1.1), ainsi que la mise en œuvre (section 5.1.2). Dans la section 5.2, nous évoquons les améliorations qui sont proposées. Il s'agit principalement de la définition d'un ensemble de composants pour modéliser des mécanismes asynchrones. Nous introduisons dans les sections 5.2.2.1 et 5.2.2.2 les approches sur lesquelles nous nous basons pour décrire ces mécanismes (leur modélisation à proprement parler est abordée dans les chapitres 6 et 8). Enfin, nous présentons quelques approches adoptant des démarches similaires.

5.1 Présentation de la méthodologie

Les caractéristiques de l'asynchronisme présent dans un système distribué temps réel apparaissent *a priori* comme un obstacle à la description d'un tel système dans le modèle synchrone : des sous-systèmes faiblement couplés en général, l'absence d'horloge globale, des communications de type "*send / receive*", etc. En fait, ce genre de système repose en général sur une architecture *globalement asynchrone localement synchrone* (GALS). Par conséquent, sa conception implique à la fois des techniques asynchrones et synchrones. La méthodologie de conception présentée ici s'adresse précisément aux systèmes adoptant une architecture de type GALS. Elle permet la production, par des transformations locales à chaque "composant synchrone" du système, à la fois du code pour ces composants et des protocoles de communication à mettre en œuvre.

Parmi les études qui ont été menées à propos de la distribution de programmes synchrones, nous citons les travaux autour de la mise en communication de programmes SIGNAL compilés séparément, menés par B. Chéron [62]. Ceux d'O. Maffeïs [156] avaient pour but de montrer comment abstraire les programmes SIGNAL, et donner l'ordonnancement et les partitionnements qualitatifs des graphes associés. A. Girault, [100] quant à lui, s'est intéressé à la répartition d'automates synchrones dans le cadre du langage LUSTRE ; ses travaux s'adaptent aussi à ESTEREL. Les travaux de P. Aubry [18] vont dans le même sens en explorant la répartition manuelle et semi-automatique de programmes flot de données synchrones. L'étude réalisée par T. Grandpierre [103] montre, avec les réseaux de Petri, comment d'une spécification synchrone flot de données (venant de LUSTRE ou SIGNAL), on peut en dériver une implantation distribuée minimisant un temps de réponse qui respecte la spécification. Le langage ESTEREL a été utilisé par O. Hainque [108] pour la spécification et la programmation d'applications réparties temps réel de grande taille. Par ailleurs, d'autres résultats ont été établis dans le cadre du projet européen SACRES [184] concernant la génération de code distribué à partir de programmes synchrones [27] [97]. Il en a résulté une méthodologie de conception d'applications distribuées utilisant SIGNAL. Celle-ci a été améliorée pendant le projet européen SAFEAIR [102] [185] qui a pris la suite de SACRES.

L'avantage des langages synchrones par rapport aux langages asynchrones tels que ADA pour le développement d'une application distribuée temps réel réside d'une part, dans le déterminisme

global de l'application qui peut être facilement garanti et d'autre part, dans l'existence de techniques de preuves indispensables pour des applications critiques. Cela explique le recours à des formalismes synchrones dans les études mentionnées ci-dessus. La démarche de conception présentée ici est entièrement fondée sur le langage SIGNAL. Elle tient compte d'un certain nombre de critères de base en ce qui concerne les programmes distribués. Ces critères assurent la préservation de la sémantique originale d'une application après les transformations engendrées par sa répartition (par exemple, relâchement de certaines contraintes de synchronisation). Ils sont présentés dans la section suivante.

5.1.1 Fondements théoriques

La nature polychrone de SIGNAL permet de modéliser un système au sein duquel chaque composant peut avoir son horloge propre. Cela en fait un langage adapté pour décrire un système selon une architecture de type GALS (celle-ci étant caractérisée par des composants faiblement couplés). Ses parties asynchrones, formées essentiellement par les mécanismes et protocoles de communication, seront alors représentées par des modèles synchrones correspondants. Il se pose ensuite la question de l'équivalence des comportements du modèle polychrone ainsi obtenu, et de la mise en œuvre asynchrone du système. En d'autres termes, il s'agit de déterminer les conditions sous lesquelles on peut passer d'un comportement synchrone à un comportement asynchrone et inversement, de façon équivalente comme l'illustre la FIG. 5.1. Sur la trace synchrone dessinée, on remarque que les contraintes de synchronisation entre les différents événements sont mises en évidence à travers la présence du symbole d'absence ("⊥"). Cela n'est pas le cas sur la trace asynchrone associée, où ces informations sont perdues. Seul l'ordre des valeurs affectées aux variables est préservé. Afin de pouvoir décider de l'équivalence de ces deux traces pour un même système, deux notions fondamentales ont été définies : l'*endochronie* et l'*endo-isochronie*[1].

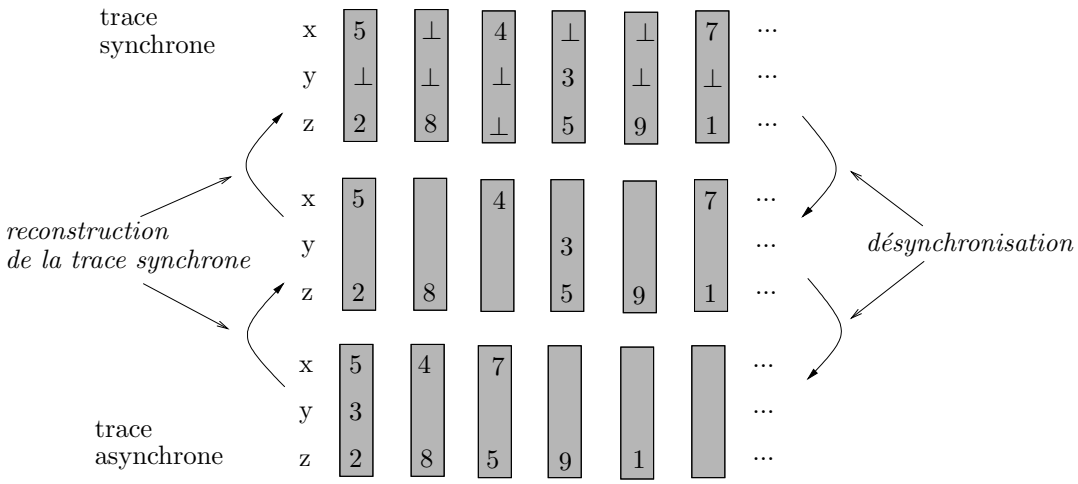


FIGURE 5.1 – Désynchronisation d'un comportement synchrone et resynchronisation du comportement asynchrone résultant.

Endochronie. La FIG. 5.2 montre les hiérarchies d'horloges [39] associées à deux processus SIGNAL. Celle de gauche, qui consiste en un seul arbre d'horloges, caractérise un processus en-

1. On parle aussi d'"*isochronie*" [29], qui est en effet une notion similaire, mais plus générale (nous y reviendrons dans le paragraphe consacré à l'endo-isochronie).

dochrone [2]. Les horloges sont organisées selon une hiérarchie basée sur l'inclusion d'ensembles d'instants. Par exemple, h_2 et h_3 sont incluses dans h_1. L'horloge la plus rapide du processus se trouve à la racine. Lorsqu'un processus n'est pas endochrone, il ne peut être représenté par un seul arbre d'horloges. Sur la FIG. 5.2, c'est le cas pour le processus de droite. On peut le rendre endochrone en construisant une horloge plus rapide que ses trois racines. Celle-ci deviendra alors la racine unique.

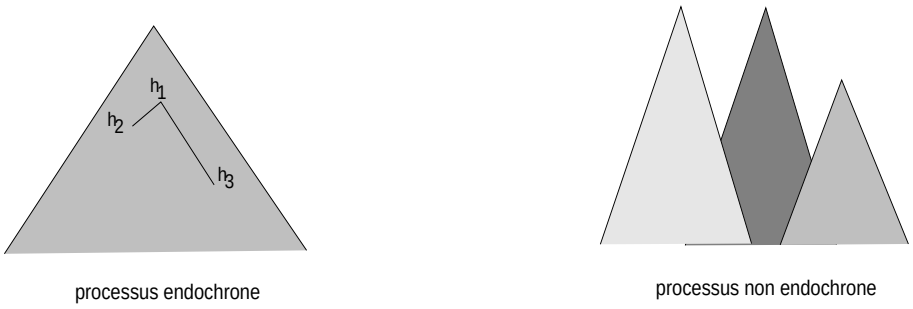


FIGURE 5.2 – Hiérarchies d'horloges dans un processus SIGNAL.

Grâce à la hiérarchie définie dans l'arbre d'horloges, un processus endochrone sait, à partir d'un flot de valeurs significatives (i.e. des stimulations externes, donc asynchrones), reconstruire exactement le flot de ses entrées ainsi que celui de ses sorties. En d'autres termes, pour faire le lien avec la définition que nous avons donnée au chapitre 3 (section 3.1.2), le système se base sur l'arbre d'horloges associé au processus qui le modélise pour décider quand les entrées sont prises en compte.

Définition 20 (endochronie) *Un processus p est endochrone sur un sous-ensemble (éventuellement vide) $E \subseteq vars(p)$ ssi*

$$\forall b, b' \in p,\ (b_{|E})_{\approx} = (b'_{|E})_{\approx} \Rightarrow b \lesseqgtr b'$$

Le processus p est dit endochrone s'il est endochrone sur l'ensemble de ses signaux d'entrée. □

Dans la définition ci-dessus, le membre gauche de l'implication spécifie que les comportements b et b' du processus p ont la même classe d'équivalence de flots, sur les signaux associés aux variables de l'ensemble E. En d'autres termes, par rapport à ces signaux, l'ordre des valeurs observées dans b est identique à celui des valeurs observées dans b'. Seules les synchronisations peuvent varier. Dire que p est endochrone revient simplement à considérer que b et b' sont alors équivalents modulo étirement. En effet, si l'on dispose de flots identiques de valeurs, les instants de consommation et production de valeurs étant *a priori* entièrement déterminés dans un processus endochrone, alors les comportements produits par le processus ont forcément les mêmes synchronisations. Celles-ci sont induites par l'arbre d'horloges associé à p. Au final, b et b' étant caractérisés par des flots de valeurs et des synchronisations identiques, ils sont équivalents modulo étirement (d'où le membre droit de l'implication).

L'importance du déterminisme a déjà été soulignée en ce qui concerne les systèmes temps réel. Il facilite l'étude des comportements d'un système puisque ceux-ci sont prévisibles. Dans

2. Une hiérarchie formée de plusieurs arbres d'horloges caractérise un processus qualifié d'exochrone (cf. chapitre 3 - section 3.1.2) : les réactions du processus dépendent de l'environnement, qui décide quels arbres d'horloges sont activés au sein de chaque réaction.

le modèle polychrone, le déterminisme d'un processus est caractérisé par le fait que ses comportements dépendent entièrement, et à tout instant d'exécution, du statut et des valeurs des entrées du processus.

Définition 21 (déterminisme) *Un processus p est déterministe sur un sous-ensemble quelconque $E \subseteq vars(p)$ ssi, pour tous $b_1, b_2 \in (p)_{\leqslant}$, pour tout $t \in tags(b_1) \cap tags(b_2)$:*

$$(b_{1|E})|_{\leq t} = (b_{2|E})|_{\leq t} \Rightarrow (b_1)|_{\leq t} = (b_2)|_{\leq t}$$

Le processus p est dit déterministe s'il est déterministe sur l'ensemble de ses signaux d'entrée. □

Il en résulte la propriété suivante [147] :

Propriété 1 *Tout processus endochrone est aussi déterministe.* □

L'endochronie est un critère adéquat pour étudier l'équivalence entre les observations internes (synchrones) et externes (asynchrones) d'un système.

Endo-isochronie. L'endo-isochronie définit les conditions suivant lesquelles la composition synchrone est équivalente à la composition asynchrone (c'est-à-dire impliquant des communications de type "*send / receive*"). Ainsi, les communications asynchrones entre processus endo-isochrones peuvent être entièrement remplacées par des modèles synchrones de protocoles de communication.

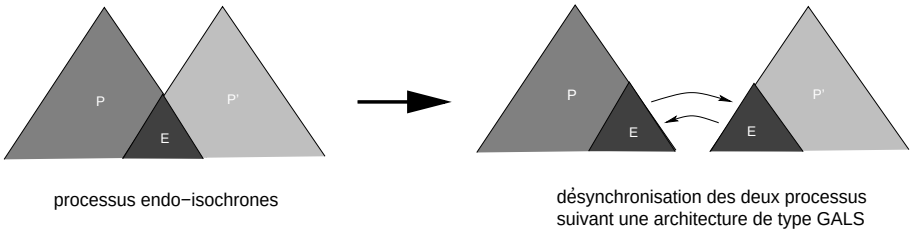


FIGURE 5.3 – Processus endo-isochrones et désynchronisation sur une architecture de type GALS.

Les processus P et P' composés sur la FIG. 5.3 (à gauche) sont tous les deux endochrones. La restriction de leurs comportements à l'ensemble des variables communes E est aussi endochrone (on dit aussi qu'ils communiquent de façon endo-isochrone). Ces processus peuvent être déployés sur une architecture GALS tout en préservant les comportements définis par leur composition synchrone. Pour cela, chacun d'entre eux est muni d'une interface endo-isochrone (représentée par une duplication de E sur la FIG. 5.3). Ces interfaces échangent des informations dont elles ont besoin l'une de l'autre (typiquement, un signal défini dans l'un des processus et utilisé dans l'autre). Les instants auxquels ces informations doivent être acquises sont facilement déduits du fait que E est endochrone. Avant de définir l'endo-isochronie, on précise d'abord ce qu'on entend par composition asynchrone.

Définition 22 (composition asynchrone) *Étant donné un comportement b appartenant à un processus p tel que $X = vars(p)$, et un comportement b' d'un autre processus p' tel que*

$X' = vars(p')$, la composition parallèle $p \parallel p'$ admet les comportements b'' relâchés sur les signaux communs à b et b' (i.e. $E = X \cap X'$)

$$p \parallel p' = \{b'' \mid \exists (b, b') \in p \times p', b_{|X \setminus X'} \lesssim b''_{|X \setminus X'} \wedge b_{|E} \sqsubseteq b''_{|E} \wedge b''_{|X' \setminus X} \lesssim b'_{|X' \setminus X} \wedge b'_{|E} \sqsubseteq b''_{|E}\}$$

□

D'autre part, on définit l'*invariance de flots* qui permet d'assurer que le raffinement d'une spécification synchrone $p \mid p'$ en une mise en œuvre asynchrone $p \parallel p'$, préserve les flots de valeurs des signaux pour tout comportement.

Définition 23 (invariance de flots) *La composition de deux processus p et p' est flot-invariante ssi, pour tout comportement $b \in p \mid p'$, pour tout comportement $b' \in p \parallel p'$, $(b_{|E})_{\approx} = (b'_{|E})_{\approx}$ implique $b \approx b'$ avec E représentant les signaux d'entrée de $p \mid p'$.* □

Nous définissons à présent l'*endo-isochronie* qui représente l'autre notion fondamentale avec l'endochronie, nécessaire pour concevoir des applications distribuées suivant une approche compositionnelle, à partir de spécifications polychrones des différents composants.

Définition 24 (endo-isochronie) *Deux processus endochrones p et p' sont endo-isochrones*[3] *ssi $(p_{|E}) \mid (p'_{|E})$ est endochrone (avec $E = vars(p) \cap vars(p')$).* □

L'endo-isochronie est une version plus restrictive de l'*isochronie* [29]. En effet, elle impose à toute paire de processus que la restriction à leur "partie communication" soit endochrone. C'est un critère suffisant mais pas nécessaire pour que la paire soit isochrone. La définition de l'endo-isochronie s'inscrit donc dans une démarche plutôt constructive.

La composition de deux processus endo-isochrones est flot-invariante [147].

* * *

De façon globale, la conception d'un système suivant une architecture globalement asynchrone localement synchrone, utilisant le modèle polychrone, consiste à modéliser le système en un ensemble de composants endochrones, qui communiquent de façon endo-isochrone. L'endochronie et l'endo-isochronie sont deux critères vérifiables statiquement. La section suivante présente la mise en œuvre de l'approche dans POLYCHRONY.

5.1.2 Mise en œuvre dans POLYCHRONY

On considère une application que l'on voudrait répartir sur une architecture composée de plusieurs processeurs. La méthodologie de distribution, illustrée sur la FIG. 5.4, a été développée dans POLYCHRONY, l'environnement de conception associé au langage SIGNAL (à savoir, un environnement graphique pour la spécification, des outils tels que le compilateur ou le *model-checker* SIGALI pour l'analyse des descriptions et la génération automatique de code). Elle comprend les étapes ci-après.

3. Par abus de langage, on parlera aussi de communication endo-isochrone entre p et p'.

Étape 1 : Spécification de la répartition du programme. L'application est d'abord décrite en SIGNAL comme un processus composé $P = P_1 \mid ... \mid P_n$. Celui-ci est représenté dans l'interface graphique de POLYCHRONY par un ensemble de boîtes interconnectées, contenant chacune le code SIGNAL du sous-processus P_i auquel elles correspondent. Le processus P reflète ainsi l'architecture logicielle de l'application. Des vérifications préalables peuvent être effectuées sur le programme obtenu pour s'assurer qu'il est correct vis-à-vis des spécifications d'origine. On définit ensuite l'architecture matérielle sur laquelle P sera déployé. Par exemple sur la FIG. 5.4, les processeurs sont représentés de façon macroscopique par des boîtes vides. Enfin, on procède au *placement* (*mapping*) du graphe logiciel spécifiant l'application (i.e. le processus P) sur le graphe matériel (i.e. l'architecture), avec réplication de code autorisée [4]. On obtient une description de l'application, $P' = P'_1 \mid ... \mid P'_m$, reflétant l'architecture cible (m dénote le nombre de processeurs). On remarquera que les P'_j ne sont pas forcément endochrones (par exemple, si P'_j est la composition de deux sous-processus dont les arbres d'horloges ont pour racines h_2 et h_3 comme sur la FIG. 5.2).

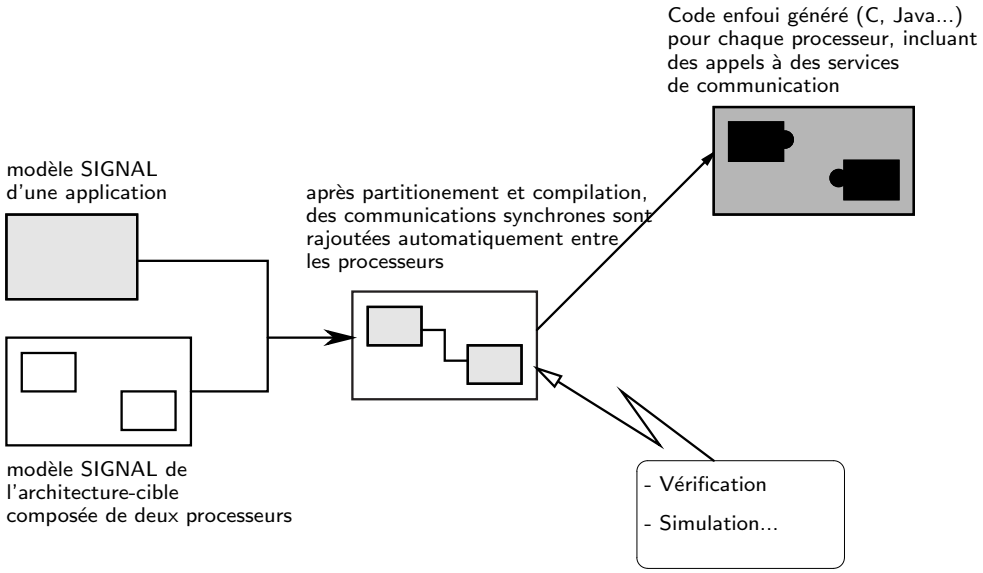


FIGURE 5.4 – Schéma de conception d'applications distribuées à l'aide de SIGNAL.

Étape 2 : Transformations automatisées garantissant la correction de la répartition. L'idée ici, consiste à transformer P' de sorte que les sous-processus SIGNAL associés aux processeurs (par exemple, P'_j qui correspond au j^{eme} processeur) soient endochrones et qu'ils communiquent de façon endo-isochrone. Ainsi, chaque "processeur" P'_j est compilé de façon modulaire dans le contexte du programme global. À partir de la hiérarchie d'horloges de P, celle de P'_j est complétée de sorte qu'elle constitue un seul arbre d'horloges, i.e. que P'_j soit endochrone [40]. On utilise pour cela des signaux booléens d'interface (à chaque racine de sous-arbre d'horloges caractérisant le processus non endochrone, correspond un signal booléen ; et tous ces signaux sont synchrones). D'autre part, des transformations sont effectuées pour faire communiquer les "processeurs" de façon endo-isochrone. Pour cela, des communications sont insérées entre eux, permettant à chaque processeur de recevoir les informations utilisées qu'il ne produit pas. Il

4. Le code SIGNAL associé à une boîte "logicielle" peut être dupliqué dans plusieurs boîtes représentant des processeurs.

en résulte alors le modèle de l'application sur l'architecture cible. Au niveau graphique, cela se traduit par un remplacement automatique de chaque P'_j par le résultat de sa compilation (sous forme d'un nouveau programme SIGNAL). Dans le modèle résultant, les échanges de signaux entre les "processeurs" sont représentés à l'aide de communications synchrones (autrement dit, instantanées). Des vérifications de propriétés peuvent alors être effectuées sur le nouveau programme. La mise en œuvre actuelle de la méthodologie dans POLYCHRONY ne prend en compte que des programmes endochrones P' à l'étape 2. Tout programme P' est d'abord compilé globalement, puis rendu endochrone s'il ne l'est pas. Ensuite, il est partitionné selon l'architecture cible. Puisqu'il est endochrone, chacune de ses parties P'_j peut aussi être endochrone. Le sous-arbre d'horloges correspondant à chaque P'_j est alors directement déduit de l'arbre d'horloges global de P'. Le fait d'imposer à P' d'être endochrone dans cette mise en œuvre garantit que l'on peut toujours en extraire des sous-parties endo-isochrones P'_j (de ce point de vue, le critère d'endochronie est "plus fort" que l'endo-isochronie). Néanmoins, ce choix peut se révéler restrictif. C'est pour cette raison qu'une amélioration de la mise en œuvre est en cours, visant à prendre en compte des programmes P' qui ne sont pas nécessairement endochrones mais qui satisfont le critère d'endo-isochronie.

Étape 3 : Génération du code distribué associé. La description obtenue à l'étape 2 contribue à un schéma modulaire de génération automatique de code. Puisque chaque "processeur" est endochrone, le code associé peut être généré indépendamment des autres. Il en est de même pour les communications. Cette génération se fait nécessairement sous forme de *lignées* qui sont des groupes d'instructions résultant d'un partitionnement de l'application [184]. Chaque groupe est identifié grâce à la dépendance du même sous-ensemble d'entrées des instructions qu'il comprend. Les lignées sont exécutées de façon atomique (nous reviendrons plus en détail sur les lignées au chapitre 8). Ce type de génération de code permet de garantir pour chaque processeur un schéma d'exécution optimisé (dans le sens où le système d'exploitation supporté par un processeur n'est sollicité que très raisonnablement ; cela réduit par exemple le nombre de commutations). En outre, si l'on considère différentes entrées qui doivent arriver sur un processeur à un instant logique, il est souhaitable que dès que l'une d'entre elles est disponible, le processeur commence les traitements qui dépendent de cette entrée. Ceci est dû à la non connaissance *a priori* de leur ordre d'arrivée dans un contexte multi-processeur, où celles-ci sont susceptibles de provenir de sites différents. Le code correspondant aux lignées est généré en C, C++ ou JAVA. Il peut être soit embarqué soit utilisé pour la simulation.

* * *

L'intérêt majeur de l'approche que nous venons de décrire réside dans le fait que toutes les transformations effectuées lors des différentes phases sont correctes par construction. Cela est garanti par le modèle sémantique du langage SIGNAL, sur lequel elles sont basées. Par ce moyen, les risques de modifications des spécifications initiales sont écartés. La validation est donc plus facile. L'amélioration de la méthodologie ci-dessus a été l'un des objectifs de l'équipe ESPRESSO dans le projet SAFEAIR. Il s'agit d'une part, de définir des modèles de mécanismes de communication asynchrones (cf. chapitre 6) qui serviront à décrire par exemple les communications entre les processeurs comme sur la FIG. 5.5, et plus généralement d'en donner une description à l'aide des concepts de mise en œuvre temps réel introduits au chapitre 2 (i.e. sous forme de tâches, ordonnanceur, etc.). Ces aspects sont notamment traités dans les chapitres 6 et 8. La technique d'évaluation de comportements temps réel, développée dans l'environnement POLYCHRONY (cf. chapitre 4), contribue également à améliorer l'approche.

5.2 Vers une amélioration de la méthodologie

Nous présentons d'abord en section 5.2.1, les grandes lignes concernant l'amélioration de la méthodologie qui vient d'être exposée. Il s'agit d'une part, de définir des modèles de composants permettant de représenter des mécanismes asynchrones et d'autre part, d'étendre les techniques de validation disponibles dans POLYCHRONY. Cette thèse traite en grande partie du premier point. La section 5.2.2 introduit ainsi les approches utilisées dans les chapitres suivants pour décrire des composants en SIGNAL.

5.2.1 Démarche proposée

Une observation générale est que le niveau de détail requis pour décrire l'architecture d'une application dépend souvent du degré de raffinement choisi pour le *mapping* sur l'architecture. En d'autres termes, dans les cas les plus simples, la quantité d'informations dont on a besoin est raisonnablement petite, et simple à estimer :

- L'ensemble des processeurs ou des tâches, et le *mapping* des opérations ou des sous-processus définis dans la spécification de l'application sur ces processeurs ou tâches. Ces informations permettent de partitionner le graphe représentant l'application en sous-graphes, selon le *mapping* ;
- La topologie du réseau de processeurs, l'ensemble des connexions entre processeurs et le *mapping* des communications entre processus sur ces liens de communication. Cela est utile notamment lorsque des signaux échangés par des processus situés sur des processeurs différents, ou dans des tâches différentes, doivent être acheminés via le même médium de communication ;
- Une définition d'un ensemble de primitives de niveau système, utilisées par exemple pour les communications (lectures et écritures sur un médium). D'une certaine façon, cela revient à définir les profils de la bibliothèque de fonctions auxquelles le code généré pour l'application devra être lié.

Des raffinements supplémentaires de la description peuvent être nécessaires pour une meilleure adaptation de l'architecture : par exemple, en ce qui concerne le type et la nature des liens de communication (ces dernières peuvent être mises en œuvre à l'aide de variables partagées, de communications synchrones ou asynchrones...). D'autre part, si l'architecture cible est caractérisée par un système d'exploitation, le modèle requis consistera typiquement en un profil des primitives correspondantes. Par exemple, selon le niveau d'utilisation du système d'exploitation, nous aurons besoin de modèles de portes de synchronisation, de communications (incluant éventuellement des routages entre processeurs) ou de fonctions de gestion de tâches (dans le cas de tâches non interruptibles : démarrer et arrêter une tâche ; pour des tâches interruptibles : suspendre et reprendre l'exécution d'une tâche, ou affecter une priorité à une tâche).

Définition d'un ensemble de composants. Dans le but de raffiner les descriptions obtenues à l'issue des étapes de la méthodologie, nous proposons un ensemble de composants, utilisables pour modéliser divers aspects au sein d'une application : mécanismes de communication asynchrone, supports d'exécution (exemples : tâches, processus), primitives de niveau système pour la gestion de tâches, pour la communication et la synchronisation entre tâches. Ces composants sont regroupés au sein d'une bibliothèque accessible dans POLYCHRONY. Le schéma de la FIG. 5.5 illustre la façon dont sont utilisés ces modèles dans le cadre d'une amélioration de la méthodologie : les descriptions obtenues après l'étape 2 deviennent instanciables dans un modèle de mise en œuvre constitué de composants provenant de notre bibliothèque. Chaque

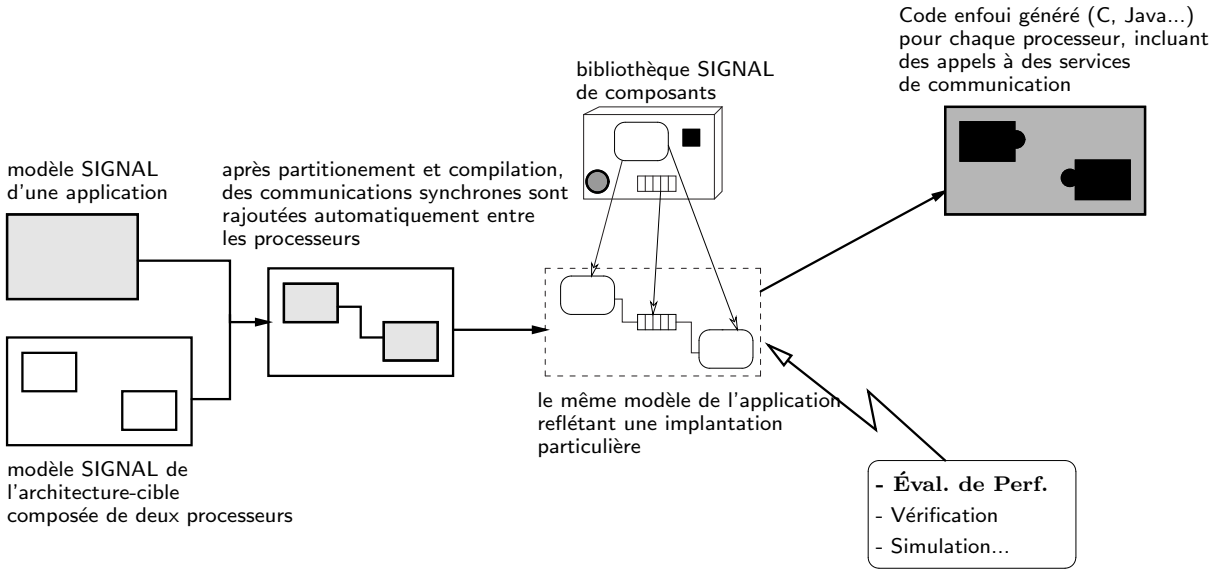


FIGURE 5.5 – Amélioration de la méthodologie de conception d'applications distribuées.

modèle de processeur peut désormais être explicité en faisant apparaître clairement les tâches s'exécutant sur celui-ci. L'exécutif temps réel qui prend en charge la gestion de ces tâches est également modélisable à l'aide des primitives de niveau système décrites dans la bibliothèque. Enfin, des modèles de mécanismes asynchrones sont proposés pour décrire les communications.

Extension des techniques de validation dans POLYCHRONY. La seconde amélioration apportée à la méthodologie concerne les techniques de validation qui y sont utilisées. Nous avons déjà souligné que l'avantage d'un formalisme tel que SIGNAL est incontestablement sa capacité à faciliter l'analyse des modèles décrits. Cela est possible grâce à sa sémantique formelle qui permet de raisonner sans ambiguïté sur ces modèles. Parmi les techniques classiques de vérification applicables à un programme, nous mentionnons l'interprétation abstraite ou le *model checking*. Des outils associés sont disponibles dans POLYCHRONY. Ici, la nouveauté consiste surtout en la mise en œuvre de la technique d'évaluation de comportements temps réel présentée au chapitre 4. Ce qui permet de disposer d'un moyen d'étudier quantitativement le comportement temporel des applications sur leurs architectures matérielles de déploiement. La répartition du graphe logiciel d'une application sur des processeurs étant effectuée suivant les décisions de l'utilisateur, ce dernier doit valider ses choix afin de s'assurer que les performances des processeurs satisfont bien les critères requis pour que l'application réalise correctement la mission qui lui est affectée. L'utilisation d'une interprétation temporelle de l'application comme observateur est un moyen permettant de vérifier les propriétés temporelles de celle-ci en vue de sa validation.

*
* *

Une contribution importante de cette thèse est la définition des modèles de composants. La présentation de ceux-ci est abordée dans les chapitres 6 et 8. Nous introduisons ici les approches suivies pour définir ces composants (section 5.2.2).

5.2.2 Définition de composants de description d'architectures

Pour définir nos composants, nous utilisons deux démarches qui sont classiques en conception. L'une est qualifiée de *descendante* tandis que l'autre est dite *ascendante*.

5.2.2.1 Approche descendante

La conception d'un système suivant une approche dite *descendante* consiste à dériver, à partir d'une description très générale, une mise en œuvre particulière de celui-ci. Généralement, cette démarche doit respecter un certain nombre d'étapes : d'abord l'identification des différents sous-systèmes et les mécanismes de communication nécessaires, ensuite la définition des opérations associées à ces sous-systèmes, enfin la mise en œuvre explicite des algorithmes.

En SIGNAL, une modélisation d'un composant consiste à considérer sa *description abstraite*, une description initiale de celui-ci, qui par la suite est *raffinée* pour dériver une description plus précise. On itère le processus jusqu'à l'obtention du modèle final souhaité. En se basant sur ce principe, les composants sont d'abord décrits de manière abstraite à travers certaines de leurs propriétés (par exemple, leurs propriétés d'interface). Le choix de celles-ci dépend généralement de l'utilisation ultérieure dont ces composants feront l'objet. Dans le chapitre 8, nous nous basons sur une approche descendante pour décrire des services d'un exécutif temps réel.

5.2.2.2 Approche ascendante

Dans une approche de conception qualifiée d'*ascendante*, on identifie d'abord les composants de niveaux inférieurs. Ces derniers sont alors définis, puis utilisés pour réaliser les composants du niveau au-dessus et ainsi de suite, jusqu'à la description du composant principal. C'est donc une approche incrémentale. La modularité du langage SIGNAL facilite une telle démarche. Supposons un composant c défini par le modèle SIGNAL suivant :

$$P = P_1 \mid ... \mid P_n$$

où chaque P_i ($i \in [1..n]$) dénote le programme associé à un sous-composant de c. On commence par définir l'un des P_i, par exemple P_1, on fait de même pour P_2, et on compose les deux modèles pour obtenir $P_{1,2}$. Ensuite, la composition de $P_{1,2}$ avec P_3 définit $P_{1,2,3}$, et ainsi de suite. On réitère le processus avec tous les sous-composants restants jusqu'à l'obtention de $P_{1,...,n}$, qui modélise le composant c. Nous présentons deux modèles d'abstraction définis en SIGNAL qui peuvent s'inscrire dans le cadre d'une démarche de conception ascendante : *abstractions boîte noire* et *boîte grise*. Ces modèles offrent des vues abstraites d'une description que nous pouvons qualifier de *boîte blanche* : où les comportements internes du composant décrit sont entièrement explicités.

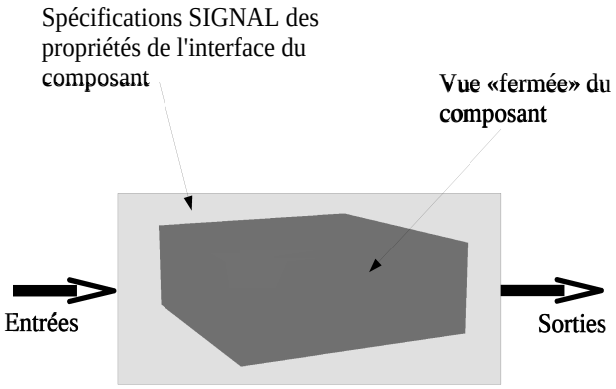


FIGURE 5.6 – Modèle *boîte noire* d'un composant.

Modèle boîte noire. Dans une conception à l'aide de composants, le niveau de détail requis

par la description de chacun des composants tient compte au minimum de la spécification des propriétés d'interface. Elles indiquent notamment comment un composant peut être connecté aux autres composants. Dans certains cas, cela est suffisant pour caractériser le composant, et permettre d'étudier des propriétés comportementales du système global formé par l'ensemble composants.

Une description SIGNAL d'un composant peut être obtenue en "encapsulant" celui-ci dans un modèle de processus, qui sera utilisé comme une "boîte noire" par la suite. Son interface décrit les entrées et les sorties. Elle spécifie notamment les dépendances et les relations d'horloges qui existent entre les entrées/sorties. Un tel modèle, comme le montre la FIG. 5.6, est en partie considéré comme étant *externe* lors de l'analyse statique, car sa mise en œuvre n'est pas connue *a priori*. Seules les propriétés d'interface sont traitées.

Modèle boîte grise. La principale limitation du modèle boîte noire est qu'il ne permet pas d'analyses fines du composant. Par conséquent, si on veut être capable de raisonner davantage sur les propriétés de celui-ci, il faut une abstraction moins "grossière" de la description. Pour ce faire, on procède à des analyses qui permettent de séparer le composant en sous-composants fonctionnellement cohérents. Ces analyses reposent essentiellement sur les relations de dépendance et les contraintes de synchronisation spécifiées dans le programme SIGNAL représentant le composant. Chaque sous-composant est représenté par un modèle boîte noire. Des interconnexions sont spécifiées entre les sous-composants résultants. Enfin, les horloges d'activation des sous-composants sont fournies. C'est ce que nous avons schématisé sur la FIG. 5.7, où le composant principal est dorénavant éclaté en quatre sous-composants.

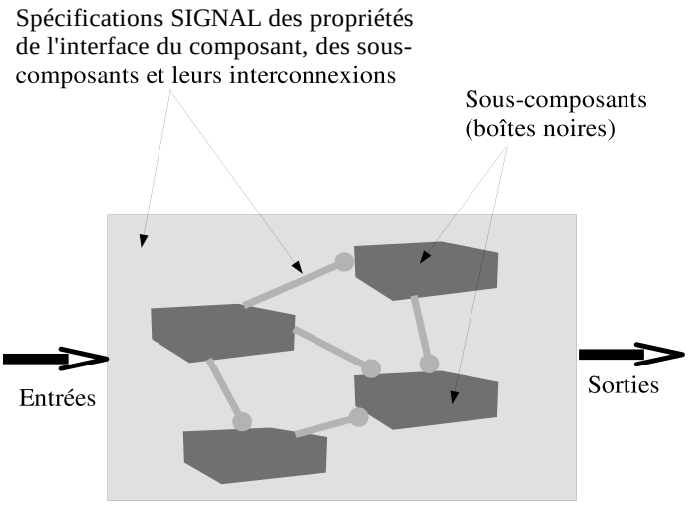


FIGURE 5.7 – Modèle *boîte grise* d'un composant.

Le modèle boîte grise offre ainsi une vue opérationnelle plus précise qu'un modèle boîte noire d'un composant (donc, plus de possibilités pour étudier des propriétés comportementales de celui-ci). Plus la granularité des sous-composants est faible, plus le modèle est précis. Cela veut dire que l'on peut à nouveau appliquer le principe à chaque sous-composant et ainsi de suite, jusqu'à ce que le niveau de détail souhaité soit atteint (par exemple, lorsqu'on n'obtient que des boîtes blanches).

Pour des systèmes définis à partir de composants provenant d'origines diverses, l'abstraction constitue un moyen approprié pour établir leur intégration. Le formalisme sous-jacent utilisé est parfois qualifié de "glue". La description de l'architecture d'un système à l'aide du langage MetaH [210] repose sur ce principe (cf. chapitre 1). Les modules externes sont encapsulés dans des boîtes noires spécifiées en MetaH. Les propriétés du système résultant sont étudiées à partir des propriétés extraites des modules encapsulés (exemples : durée d'exécution du code associé à un module, ordonnancements au sein du module). L'approche ascendante est illustrée dans le chapitre 6, où nous modélisons une file de messages utilisable dans un protocole de communication asynchrone.

5.3 Quelques travaux en relation avec l'approche

La méthodologie de conception présentée dans ce chapitre possède des points communs avec plusieurs autres approches parmi lesquelles nous citons celles qui ont été définies dans Sildex [207], SynDEx [105] ou Autofocus [124].

Sildex. C'est un environnement de conception développé et commercialisé par TNI-Valiosys. Sildex [207] repose sur le langage Signal mais contrairement à Polychrony, il n'exige pas forcément de l'utilisateur une maîtrise poussée du langage. Il propose un éditeur graphique bien fourni (bibliothèque de composants pour décrire une application), avec la possibilité de faire des simulations. L'approche à la conception d'applications distribuées dans Sildex est quasiment identique à celle de Polychrony. Cependant dans Sildex, l'accent est mis surtout sur l'aspect génération de code : le résultat des transformations effectuées au cours de l'étape 2 de la méthodologie n'est pas répercuté dans le programme Signal comme c'est le cas dans Polychrony. Il est pris en compte directement dans le code généré. En ce qui nous concerne, le fait d'avoir le programme transformé en Signal rend toujours possible l'utilisation de techniques et outils accessibles dans l'environnement pour des analyses. Cela n'est pas le cas dans Sildex.

SynDEx. La méthodologie AAA (*Adéquation Algorithme-Architecture*) [104], développée dans l'environnement SynDEx (introduit au chapitre 3), "vise le prototypage rapide et l'implantation optimisés d'applications distribuées temps réel embarquées devant être tolérantes aux fautes, telles celles rencontrées en contrôle-commande de systèmes complexes comprenant du traitement du signal et des images. Elle est fondée sur des modèles de graphes, autant pour spécifier les Algorithmes applicatifs et les Architectures matérielles distribuées comportant un certain niveau de redondance, que pour réduire les implantations possibles en termes de transformations de graphes. l'Adéquation revient à résoudre un problème d'optimisation consistant à choisir une implantation dont les performances, déduites des caractéristiques des composants matériels, respectent les contraintes temps réel et d'embarquabilité, et exploitent la redondance matérielle pour effectuer de la tolérance aux fautes par redondance logicielle automatique. Dans le cas du temps réel critique, les approches *hors ligne* sont privilégiées, et quand les approches *en ligne* sont utilisées, le nombre de décisions prises lors de l'exécution est minimisé, c'est-à-dire, uniquement quand elles sont inévitables. Tout cela permet de générer automatiquement d'une part des exécutifs distribués temps réel à faible surcoût pour les composants processeurs, et d'autre part des *net-lists* pour les composants circuits intégrés spécifiques, supportant tous ensemble (*co-design*) l'exécution tolérante aux fautes de l'algorithme sur l'architecture".

Nous pouvons donc qualifier la méthodologie AAA d'approche *quantitative* dans la mesure où elle détermine automatiquement la distribution suivant les performances des moyens de calcul et de communication disponibles. Dans Polychrony, la répartition est effectuée manuellement

indépendamment de toute contrainte quantitative *a priori*. Cette démarche se défend très bien quand on sait que généralement, la tâche de distribution est confiée à un expert qui connaît en principe quel type de processeur convient pour une exécution efficace d'un composant de l'application. L'approche est donc ici plutôt *qualitative*. On peut ensuite vérifier les performances obtenues par rapport aux exigences des spécifications grâce à la technique basée sur le morphisme de programmes SIGNAL. Nous reviendrons sur l'approche SYNDEX dans le chapitre 8, où nous discutons sa complémentarité avec celle de POLYCHRONY.

AutoFOCUS. C'est une méthodologie basée sur des modèles, qui permet de concevoir des systèmes embarqués distribués [124] à l'aide de la plate-forme AUTOFOCUS, développée au *Technische Universität* de Munich. Parmi les modèles de base, on distingue les *composants* qui communiquent par l'intermédiaire de *ports* reliés par des *canaux*. Ces modèles, qui permettent de donner une description structurelle d'un système, constituent les objets de base des *diagrammes de structure du système* (ou SSD). La description comportementale d'un composant est exprimée au moyen de *diagrammes à états-transitions* (ou STD). Ce sont des machines à états hiérarchiques comprenant un ensemble d'états de contrôle, des transitions et des variables locales. Les interactions entre composants sont données en termes de traces d'événements qui se rapprochent des diagrammes de séquences UML. Dans AUTOFOCUS, la sémantique comportementale est *synchrone*. On considère une horloge globale suivant laquelle les modèles évoluent. Cependant, une particularité est que les calculs effectués au sein des composants et les communications entre ces composants se déroulent de façon exclusive. La description d'un système distribué dans AUTOFOCUS prend en compte deux aspects, de même que l'approche proposée dans POLYCHRONY : le modèle *logique* qui représente le graphe logiciel, et le modèle *technique* qui symbolise l'architecture l'implantation. La description finale du système est obtenue en projetant le modèle logique sur le modèle technique. Pour cela, la sémantique est étendue avec une interprétation GALS [181] : la sémantique du modèle technique n'est pas synchrone ; pour chaque canal logique qui doit être projeté sur un canal technique, on supprime les synchronisations entre les composants qu'il relie, et on rajoute des *buffers* et un ordonnanceur global pour ces canaux. Une fois le modèle final obtenu, on peut faire de la vérification de propriétés (en utilisant des outils comme le *model checker* SPIN), ou bien de la simulation. AUTOFOCUS permet aussi de faire de la synthèse de séquences de tests.

* * *

D'autres travaux reposant sur une démarche comparable à la méthodologie présentée se situent dans le domaine du *co-design*, à l'image de [21] ou [107]. Le lecteur pourra se référer à [140] et [148] pour un résumé de ces approches.

5.4 Résumé

La méthodologie de conception exposée dans ce chapitre concerne les applications distribuées temps réel dans l'environnement POLYCHRONY du langage SIGNAL. Elle comporte trois étapes principales : *i*) la spécification par l'utilisateur de la répartition du programme SIGNAL représentant l'application, *ii*) des transformations formelles automatisées de programmes intermédiaires garantissant la correction du partitionnement, avec possibilité d'analyser le programme résultant ; *iii*) la génération de code distribué. Les transformations effectuées au cours de ces étapes reposent sur des propriétés bien identifiées dans le modèle polychrone, à savoir l'*endochronie* et l'*endo-isochronie*.

Une telle approche définit un cadre de conception unifié dans lequel l'unique support sémantique est le modèle de SIGNAL. Le travail présenté dans cette thèse s'insère naturellement dans la méthodologie, tout en visant une amélioration de celle-ci. Pour cela, nous proposons des modèles de mécanismes asynchrones, sous la forme de composants, pour décrire des communications ou des synchronisations entre les tâches qui composent une application temps réel. La modélisation de ces mécanismes repose sur des démarches soit ascendantes soit descendantes. Le chapitre suivant illustre la description de composants en SIGNAL suivant la démarche ascendante. Dans le chapitre 8, nous utilisons plutôt l'approche descendante pour modéliser les mécanismes de communication et de synchronisation définis dans la norme ARINC 653 [8].

Chapitre 6

Illustration d'une conception par composants à l'aide de SIGNAL

Le but de ce chapitre est d'illustrer l'approche ascendante décrite au chapitre précédent. Pour cela, nous considérons un cas très simple qui consiste en une file d'attente de messages gérée en *fifo* (*first in first out*). À travers cet exemple, nous montrons que le langage SIGNAL (et c'est le cas plus généralement des langages synchrones) possède des caractéristiques qui favorisent des descriptions modulaires et génériques [93]. Cela est indispensable dans une bonne approche orientée composants.

Pour décrire un composant, nous procédons de façon incrémentale : partant d'une spécification d'un *composant de base*, nous ajoutons de nouvelles contraintes à ce "composant" (grâce à la composition) ; et nous obtenons de nouveaux composants. En l'occurrence, nous décrivons par la suite plusieurs variantes d'une file d'attente de messages. Les unes ayant des fonctionnements plus restreints que les autres. Chacune peut avoir son utilité selon l'usage qu'on en fait. En ce qui nous concerne, ces composants serviront à modéliser des mécanismes de communications asynchrones, typiquement dans la description d'une application distribuée telle que présentée au chapitre précédent.

La section 6.1 présente la modélisation de deux files de messages bornées, gérées en *fifo*. La première, F_1, constitue le composant de base. Elle admet plus de comportements que la seconde file, F_2. Par exemple, F_1 autorise toujours l'insertion d'un message malgré sa taille bornée (i.e., il peut y avoir des pertes de messages). Dans certaines situations, ce type de comportement peut entraîner des problèmes (par exemple, dans les systèmes critiques). La file F_2 modélise un mécanisme de communication plus *fiable* dans le sens où elle n'engendre aucune perte de message. Elle résulte du raffinement de F_1. Dans la section 6.2, nous nous assurons que la file F_2 garantit bien la transmission de messages sans pertes. Nous utilisons notamment l'outil SIGALI, accessible dans POLYCHRONY, pour vérifier un certain nombre de propriétés. La démarche illustrée dans ce chapitre se veut avant tout générale, et donc ne se limite pas uniquement aux composants de communication.

6.1 Modélisation d'un composant de communication

Dans cette section, nous exposons une démarche incrémentale de description de composants à l'aide de SIGNAL. Nous modélisons un premier composant (une file de messages gérés en *fifo*) dans la section 6.1.1. Celui-ci est ensuite utilisé pour décrire un autre composant du même type

dans la section 6.1.2. Le second composant est caractérisé par un nombre de comportements plus restreint par rapport au premier.

6.1.1 Définition d'un composant de base

Le modèle de *fifo* considéré ici représente le composant de base. On lui associe ainsi un comportement "très peu contraint" [1]. Ce composant est appelé *basic_fifo*. On donne ci-après la spécification informelle et une spécification synchrone à l'aide de SIGNAL.

Spécification informelle de basic_fifo. On considère une file de messages bornée gérée suivant une politique *fifo*. Elle est destinée à la description ultérieure de mécanismes de communication similaires, où les opérations de *lecture* et *écriture* de messages sont réalisées de façon explicite. Le comportement de la file est le suivant :

- La file peut toujours accepter un message venant de l'extérieur (écriture), quel que soit le nombre courant de messages qu'elle contient. En particulier, si elle était déjà pleine, on choisit (arbitrairement) de perdre le message le plus ancien. On décale ainsi les autres messages de façon à pouvoir insérer le nouvel arrivé.
- La file peut toujours renvoyer un message (lecture), quel que soit son état. Dans le cas où elle se trouvait déjà vide, on distingue deux situations possibles : soit il n'y a pas encore eu d'écriture dans la file et elle rend alors un message "par défaut", soit il y a eu au moins une écriture et elle renvoie alors le tout dernier message sorti.

D'autre part, pour simplifier la présentation, on suppose qu'il n'y a jamais d'écritures et/ou lectures simultanées sur la *fifo*. Cette hypothèse n'est pas très restrictive. Par exemple, sur une architecture mono-processeur multi-tâche, on a à tout instant une seule tâche au plus qui s'exécute à la fois, et donc toute requête de la part de celle-ci sur la file est effectuée de façon exclusive. Sur une architecture multi-processeur, il peut y avoir des requêtes simultanées provenant de processeurs différents sur une même file. Plusieurs techniques sont possibles. Si la file est une ressource partagée, on utilise à un niveau donné (logiciel ou matériel) un mécanisme (par exemple un verrou) garantissant un accès exclusif à la file. Une autre solution consiste à considérer des copies de la file. Un indicateur de validité de copie indiquera à chaque tâche si l'état de la file est consistant lors d'un accès quelconque. Ici, nous nous plaçons plutôt dans le cadre de la première solution.

Spécification SIGNAL de basic_fifo. Le programme SIGNAL correspondant (appelé `basic_fifo`) est donné sur la FIG. 6.1.

Dans ce processus, les paramètres `message_type`, `fifo_size` et `default_mess` désignent respectivement le type des messages stockés [2], la capacité maximale de la *fifo* et le message renvoyé par défaut en cas de lecture initiale dans une file vide. L'unique entrée, dénotée par `mess_in`, correspond au flot des messages provenant de l'extérieur qui doivent être insérés dans la file. Quant à la sortie `mess_out`, elle représente le flot des messages renvoyés par la file. Ces deux signaux sont indispensables dans l'interface de `basic_fifo`. Les instants de lecture/écriture des messages seront fixés ultérieurement par les différents contextes d'utilisation. D'autre part, des sorties supplémentaires ont été prises en compte dans le but de rendre le composant plus général dans ses utilisations. Ainsi, les signaux `OK_write` et `OK_read` sont utiles pour savoir si

1. En comparaison aux contraintes habituelles imposées à une *fifo* utilisée dans un *buffer*, notamment la robustesse aux pertes de messages.
2. Cela assure la généricité du composant.

une opération quelconque sur la *fifo* est sans risque : `OK_write` (resp. `OK_read`) porte la valeur *vrai* lorsque la *fifo* n'est pas pleine (resp. vide), autrement il prend la valeur *faux*. Quant à `nbmess`, il représente le nombre courant de messages dans la file à tout instant. Dans certains mécanismes de communication basés sur les files d'attente, il peut être intéressant de connaître l'état courant de ceux-ci. C'est notamment le cas du *buffer*, défini dans la norme ARINC [8] (cf. chapitre 7) : parmi les services qui lui sont associés, l'un d'eux est chargé de fournir l'état du mécanisme.

```
process basic_fifo =
    { type message_type; integer fifo_size, message_type default_mess; }
    ( ? message_type mess_in;
      ! message_type mess_out; integer nbmess; boolean OK_write, OK_read;
    )
  (| nbmess := ((prev_nbmess+1) when (^mess_in) when OK_write) default
               ((prev_nbmess-1) when (^mess_out) when OK_read) default
               prev_nbmess                                                    (1.a)
   | prev_nbmess := nbmess$1 init 0                                           (1.b)
   | OK_write := prev_nbmess<fifo_size                                        (1.c)
   | OK_read := prev_nbmess>0                                                 (1.d)
   | queue := (mess_in window fifo_size) cell (^nbmess)                       (1.e)
   | mess_out := prev_mess_out when (not OK_read) when (^mess_out) default
                 queue[fifo_size - prev_nbmess] when (^mess_out)              (1.f)
   | prev_mess_out := mess_out $ 1 init default_mess                          (1.g)
   |)
  where
     integer prev_nbmess; [fifo_size]message_type queue;
     message_type prev_mess_out;
  end;
```

mess_in	:	$\perp$	4	6	$\perp$	$\perp$	$\perp$	5	7	8	$\perp$	$\perp$	...
mess_out	:	-1	$\perp$	$\perp$	4	6	6	$\perp$	$\perp$	$\perp$	7	8	...
nbmess	:	0	1	2	1	0	0	1	2	2	1	0	...
OK_write	:	t	t	t	f	t	t	t	t	f	f	t	...
OK_read	:	f	f	t	t	t	f	f	t	t	t	t	...

FIGURE 6.1 – Programme correspondant au modèle *basic_fifo* avec une trace d'exécution.

Dans le corps du processus, l'équation `(1.b)` définit le signal `prev_nbmess` comme le nombre de messages dans la file à l'instant précédent. Ce signal est utilisé dans `(1.c)` et `(1.d)` pour définir respectivement les booléens `OK_write` et `OK_read`. L'équation `(1.a)` indique que le nombre courant de messages évolue : sa valeur précédente est incrémentée de 1 lorsqu'on enregistre une entrée et que la capacité maximale de la file n'était pas atteinte ; sinon, il est décrémenté de 1 lorsqu'il y a une lecture alors que la file n'était pas déjà vide ; autrement, le nombre de messages dans la *fifo* reste inchangé. Sur la ligne `(1.e)`, deux nouveaux opérateurs du langage SIGNAL sont utilisés. Il s'agit de la fenêtre glissante (`window`) et de la mémoire (`cell`) :

– *Fenêtre glissante* : Dans le processus `y:= x window k init tab_init`, le signal `y` est un tableau de taille $k \geq 1$ dont les éléments sont du même type que le signal `x` ; et `tab_init` est un tableau de dimension $k' \geq k - 1$, contenant les valeurs d'initialisation.

Cette équation définit une fenêtre glissante sur `x`, de taille constante `k` telle que :

$$(\forall t \geq 0) \quad \Big(\begin{array}{l} (t+i \geq k) \Rightarrow (Y_t[i] = X_{t-k+i+1}) \quad \vee \\ (1 \leq t+i < k) \Rightarrow (Y_t[i] = tab\_init[t-k+i+2]) \end{array} \Big)$$

– *Mémoire* : L'équation `y := x cell b init c` permet de *mémoriser* les valeurs du signal `x`. Elle est définie comme `(| y := x default (y $ 1 init c) | y ^= x ^+ (when b) |)`. Le signal `y` prend la valeur de `x` lorsque `x` est présent. Sinon, si ce dernier est absent et que le signal booléen `b` est présent et possède la valeur *vrai*, `y` mémorise la dernière valeur de `x`. Dans le cas où `b` est présent et *vrai* avant la première occurrence de `x`, le signal `y` est initialisé avec la constante `c`. L'horloge de `y` est égale à l'union de celle de `x` et de l'ensemble des instants où `b` vaut *vrai*.

Ainsi, dans l'équation `(1.e)`, le signal `queue` est un tableau de dimension `fifo_size`. Il contient les `fifo_size` dernières valeurs du message `mess_in`, récupérées à l'aide de la construction `window`. L'opérateur `cell` quant à lui rend le signal `queue` disponible à l'horloge de `nbmess` (représentant les instants d'accès à la *fifo*). Enfin dans l'équation `(1.f)`, nous spécifions la relation suivante : aux instants où la file est vide, le signal `mess_out` prend la valeur du dernier message lu (défini par `(1.g)`) ; autrement, c'est le plus ancien des messages présents qui est choisi. On remarquera que l'horloge du signal `mess_out` (les instants où on effectue une lecture) n'est pas fixée dans ce processus.

Sur la trace d'exécution de la Fig. 6.1, la capacité maximale de la *fifo* est 2, les messages sont codés par des entiers et le message par défaut est -1. Ce premier modèle de *fifo* peut d'un certain point de vue être considéré comme étant inadéquat dans des protocoles où on ne souhaite pas perdre de messages pendant les échanges. Il faudrait donc une file qui restreigne les comportements de la première afin de satisfaire aux critères d'échanges de messages sans perte. Nous allons donc montrer comment cette nouvelle file est obtenue à partir de `basic_fifo`.

6.1.2 Un modèle raffiné du composant de base

Le modèle présenté dans cette section, appelé `safe_fifo`, a le même fonctionnement de base que `basic_fifo`. Cependant, la différence est qu'il n'admet d'écriture ou de lecture de message dans la file que lorsque cela est "possible". Ces deux opérations ne sont respectivement possibles que lorsque la file n'est pas déjà pleine ou vide. Par ailleurs, les lectures et écritures sont réalisées sur des demandes explicites. Ce nouveau modèle est ainsi caractérisé par un ensemble de comportements strictement inclus dans celui qui est associé au modèle précédent : $[\![\texttt{safe\_fifo}]\!] \subset [\![\texttt{basic\_fifo}]\!]$. Cette restriction des comportements de `basic_fifo` au sous-ensemble de comportements associés à `safe_fifo` fait de ce dernier un modèle raffiné de `basic_fifo`.

Le modèle `safe_fifo` est représenté sur la Fig. 6.2. Son interface est très proche de celle de `basic_fifo`. Dans ce nouveau contexte, l'entrée `mess_in` dénote à la fois le flot des messages à stocker et les demandes d'écriture. Une nouvelle entrée a été ajoutée pour représenter les demandes de lecture. Il s'agit de `get_mess`. Par ailleurs, nous choisissons de ne garder que les sorties `OK_write` et `OK_read` pour tester s'il est possible d'écrire ou de lire un message dans la file. Le signal `nbmess` est désormais local au processus (ce qui ne serait pas le cas par exemple dans l'optique d'une modélisation d'un *buffer* ARINC).

Dans l'équation `(2.a)`, on synchronise `nbmess` à l'union des instants où il y a une demande d'écriture ou de lecture (l'horloge la plus rapide). L'appel du processus `basic_fifo` est fait en

```
process safe_fifo =
    { type message_type; integer fifo_size; message_type default_mess; }
    ( ? message_type mess_in; event get_mess;
      ! message_type mess_out; boolean OK_write, OK_read;
           )
  (| nbmess ^= mess_in ^+ get_mess                                          (2.a)
   | new_mess_in := mess_in when OK_write                                   (2.b)
   | mess_out ^= get_mess when OK_read                                      (2.c)
   | (mess_out, nbmess, OK_write, OK_read) :=
       basic_fifo{ message_type, fifo_size, default_mess }(new_mess_in)     (2.d)
   |)
 where
    use basic_fifo;
    integer nbmess; message_type new_mess_in;
 end;
```

mess_in	:	⊥	4	6	⊥	⊥	⊥	5	7	8	⊥	⊥	...
get_mess	:	t	⊥	⊥	t	t	t	⊥	⊥	⊥	t	t	...
mess_out	:	⊥	⊥	⊥	4	6	⊥	⊥	⊥	⊥	5	7	...
OK_write	:	t	t	t	f	t	t	t	t	f	f	t	...
OK_read	:	f	f	t	t	t	f	f	t	t	t	t	...

FIGURE 6.2 – Programme correspondant au modèle *safe_fifo* avec une trace d'exécution.

`(2.d)` et l'entrée est `new_mess_in`. C'est un signal local qui représente les messages reçus à la suite de requêtes d'écriture, aux instants où la file n'est pas pleine. C'est ce que `(2.b)` exprime. De la même manière, l'équation `(2.c)` exprime une relation qui contraint les messages issus de `basic_fifo` à être présents sur une demande de lecture, seulement lorsqu'il y en a au moins un dans la file. Une trace d'exécution du modèle `safe_fifo` est donnée sur la FIG. 6.2 (avec les mêmes paramètres que pour la FIG. 6.1).

D'autres types de mécanismes de communication peuvent être décrits à l'aide de `basic_fifo`. Par exemple, le mécanisme *buffer* défini dans la norme ARINC [8] peut être facilement modélisé. Celui-ci est équivalent au composant `safe_fifo`, auquel on adjoint une couche de gestion de tâches bloquées en attente de lecture ou d'écriture dans la file des messages.

Nous pouvons aussi décrire le mécanisme *sampling port* de la norme ARINC. Celui-ci offre des possibilités de lecture et écriture de messages dans un tampon à une place. Bien entendu, une modélisation judicieuse évitera l'utilisation d'une *fifo* pour représenter un tampon à une place. Dans POLYCHRONY, le compilateur est capable d'optimisations qui provoqueront dans ce cas la suppression de l'opération `window` utilisée dans le processus `basic_fifo`. Dans un *sampling port*, à chaque écriture d'un message, une date lui est associée. Les messages lus sont toujours accompagnés d'un paramètre de validité qui indique la consistance de leur date avec une certaine période de rafraîchissement, associée au port. Les lectures ainsi que les écritures sont toujours possibles. Le programme SIGNAL correspondant à un *sampling port* ressemble à celui de `safe_fifo`, dans lequel les contraintes `(2.b)` et `(2.c)` disparaissent (i.e., les entrées / sorties ne sont pas contraintes). Le type des messages et la taille de la file étant des paramètres, il suffit de les instancier avec le bon type et la valeur "1". Enfin, on ajoutera une entrée pour récupérer la période de rafraîchissement, et une sortie pour indiquer la validité des messages lus.

Le composant `basic_fifo` peut aussi servir à la description du mécanisme *blackboard*, également défini dans la norme ARINC. Comme le *sampling port*, le *blackboard* est une mémoire à une cellule, dans laquelle on peut toujours écrire un message. Par contre, on ne peut lire de message que lorsque le *blackboard* n'est pas vide. Le programme SIGNAL associé est obtenu de façon similaire à celui de `safe_fifo`. Dans le programme de la FIG. 6.2, il suffit de supprimer l'équation `(2.b)` qui contraint l'entrée de `basic_fifo` (le signal `mess_in` est directement passé à ce processus). Le modèle résultant est ensuite instancié avec une taille égale à "1".

Par ailleurs, on pourrait définir d'autres mécanismes de communication similaires à `safe_fifo`, qui cette fois-ci contraindraient uniquement les entrées mais pas les sorties ou inversement, les sorties mais pas les entrées.

Plus généralement, en se basant sur une telle démarche, nous pouvons définir différents modèles de composants. Ici, il s'agissait particulièrement de mécanismes de communication utilisables dans une description d'applications distribuées, telle que nous l'avons présentée dans le chapitre précédent. De façon incrémentale, le composant `safe_fifo` défini ci-dessus pourra être utilisé à son tour dans la description de certains protocoles de communication (par exemple, le protocole LTTA [31]).

Nous venons donc d'illustrer la modélisation de composants (en particulier, il s'agit de mécanismes de communication) suivant une approche ascendante. L'approche est fondamentalement la même pour d'autres catégories de composants. On notera que la principale difficulté dans la définition des modèles concerne l'identification du composant de base. Le critère est que ce dernier doit être le moins contraint possible au niveau de ses comportements. Il suffit alors de lui adjoindre progressivement des contraintes jusqu'à obtenir le composant souhaité. Une autre observation importante est que le langage SIGNAL (et c'est le cas plus généralement des langages synchrones) possède des caractéristiques qui favorisent des descriptions modulaires et génériques. Cela est essentiel à une approche orientée composants.

6.2 Vérification de propriétés sur les composants

Pour vérifier les propriétés des composants modélisés, nous utilisons les outils de preuve accessibles dans POLYCHRONY. Le compilateur permet d'étudier la consistance des contraintes d'horloges exprimées dans un programme SIGNAL : absence de définitions contradictoires ou d'horloges nulles, etc. Quant à l'outil SIGALI [160], il permet d'appréhender les propriétés dynamiques de programmes. C'est un système interactif spécialisé dans les calculs algébriques sur le corps $\mathbb{Z}/3\mathbb{Z}$. Dans ce qui suit, nous nous intéressons essentiellement à ce dernier type de propriété. Il s'agit de garantir que les comportements du composant `safe_fifo` sont corrects (par exemple, il n'y a pas d'écrasements de messages dans la file, les messages lus sont obtenus dans l'ordre où ils ont été stockés). Nous commençons par rappeler les principes de base concernant la vérification de propriétés dynamiques à l'aide de SIGALI.

Dans le chapitre 4, nous avons évoqué le fait que l'abstraction par le contrôle d'un programme SIGNAL produit un autre programme, qui peut être encodé en équations polynomiales sur le corps $\mathbb{Z}/3\mathbb{Z}$, représenté par l'ensemble $\{-1, 0, 1\}$. Dans ce codage, les valeurs -1, 1 et 0 dénotent respectivement la "présence avec la valeur *faux*", la "présence avec la valeur *vrai*" et l'"absence". On peut remarquer que ce codage prend en compte complètement la valeur des signaux booléens, mais pas celle des signaux numériques. Ainsi, dans les équations portant sur

des signaux non booléens, seules les relations de synchronisation sont encodées. L'horloge d'un signal quelconque x est donnée par x^2 (elle vaut 1 si x est présent, et 0 sinon). Par exemple, la propriété qu'ont deux signaux x et y d'être synchrones est représentée par l'équation $x^2 = y^2$. Dans une équation sur des signaux booléens telle que `y := not x`, on a l'information $y = -x$ en plus de la synchronisation.
L'encodage des valeurs de signaux booléens définis à l'aide de l'opérateur de décalage introduit une variable d'état. Celle-ci est notée ϵ dans le système d'équations (6.1) ; elle modélise une transition d'état. Ainsi, le processus `y := x $ 1 init c`, où `y`, `x` et `c` sont de type booléen, est encodé en

$$\begin{cases} \epsilon' = x + \epsilon.(1 - x^2) & (1) \\ y = \epsilon.x^2 & (2) \\ \epsilon_0 = c & (3) \end{cases} \tag{6.1}$$

L'équation (1) décrit l'état suivant ϵ' ; il est égal à x si x est présent ; il est égal à ϵ sinon. Dans (2), on définit la valeur de y comme celle de ϵ lorsque x est présent. La dernière équation définit quant à elle la valeur initiale de l'état ϵ.

On obtient l'encodage suivant pour le filtrage `y:= x when b`, où `y`, `x` et `b` sont des signaux booléens :

$$y = x.(-b - b^2) \tag{6.2}$$

Enfin, le mélange déterministe `y:= u default v` est encodé comme suit (`y`, `u` et `v` sont de type booléen) :

$$y = u + (1 - u^2).v \tag{6.3}$$

La composition des équations élémentaires sur $\mathbb{Z}/3\mathbb{Z}$ (comme celles que nous venons d'illustrer ci-dessus pour les signaux booléens) forme un *système dynamique polynomial* associé au processus auquel appartiennent ces équations.

L'étude de la sémantique abstraite d'un programme SIGNAL se ramène alors à l'étude de systèmes dynamiques polynomiaux de la forme (6.4), sur lesquels il est possible de vérifier des propriétés ou bien de faire du contrôle.

$$\begin{cases} X' = P(X, Y) & (a) \\ Q(X, Y) = 0 & (b) \\ Q_0(X_0) = 0 & (c) \end{cases} \tag{6.4}$$

Dans un tel système, X et X' sont des vecteurs dans $(\mathbb{Z}/3\mathbb{Z})^n$ représentant respectivement l'état courant et l'état suivant du système ; Y quant à lui, est un vecteur dans $(\mathbb{Z}/3\mathbb{Z})^m$ regroupant des variables d'événements qui font évoluer le système (i.e. les entrées). L'équation (a) caractérise l'évolution des variables d'état dans le temps ; les invariants du système sont regroupés dans (b) - ce sont principalement les contraintes d'horloges spécifiant les synchronisations entre signaux ; enfin, l'initialisation des variables d'état est donnée par (c).

Ainsi, l'analyse dynamique de programmes SIGNAL s'appuie sur la représentation équationnelle d'un automate (les automates et leurs états, les événements et les trajectoires sont manipulés au travers des équations qui les décrivent). Cela permet de calculer les états *accessibles* du système

dynamique considéré, de vérifier des propriétés sur les trajectoires de celui-ci (exemple : absence de *blocage* pour des valeurs initiales particulières). Ces calculs sont basés sur des structures de données appelées TDD (*Ternary Decision Diagrams*) [80]. Ce sont des extensions des BDD [54] pour manipuler des équations dans $\mathbb{Z}/3\mathbb{Z}$ qui ont été mises en œuvre dans SIGALI. Des expériences ont montré que celui-ci peut effectuer en un temps raisonnable des preuves sur des automates comportant plusieurs millions d'états atteignables.

6.2.1 Quelques propriétés dynamiques

La FIG. 6.3 illustre l'évolution d'un système dynamique. Chaque E_i est une instance de l'ensemble E des états du système. Les événements y_j provoquent des transitions entre les E_i.

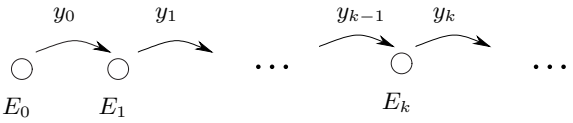


FIGURE 6.3 – Évolution du système dynamique.

Parmi les propriétés qu'on peut vérifier sur ce système [159], on distingue la *vivacité*, l'*invariance* ou l'*accessibilité*. Elles sont définies ci-après.

Définition 25 (accessibilité) *Un sous-ensemble F d'états est accessible pour un système dynamique, ssi chaque état $x \in F$ peut être atteint à partir des états initiaux du système dynamique considéré.* □

La propriété de vivacité d'un système garantit le fait que celui-ci n'atteindra jamais un état à partir duquel aucune transition n'est possible. Elle est définie de la façon suivante :

Définition 26 (vivacité) *La définition de la propriété de vivacité d'un système repose sur les notions suivantes :*

- *un état x est vivant ssi il existe un événement y tel que $Q(x, y) = 0$ (i.e., une transition peut être effectuée) ;*
- *un ensemble d'états V est dit vivant ssi chaque état de V est vivant.*

Un système est vivace ssi pour tout couple (x, y) tel que $Q(x, y) = 0$, $P(x, y)$ est un état vivant. □

La propriété d'invariance permet de caractériser la sûreté dans les comportements d'un système dynamique. En effet, cela revient à montrer que ce système reste toujours dans un ensemble d'états qui ne varie pas.

Définition 27 (invariance) *Un ensemble d'états E est invariant ssi pour tout état $x \in E$, et pour tout événement y admissible dans l'état x (i.e. $Q(x, y) = 0$), l'état $P(x, y)$ appartient à E.* □

SIGALI fournit un certain nombre de fonctions prédéfinies qui permettent de vérifier des propriétés types des systèmes dynamiques, comme celles qui ont été introduites ci-dessus (vivacité, invariance, accessibilité, etc.). Il offre aussi d'autres fonctions pour effectuer par exemple, des

calculs polynomiaux (somme, différence, produit...), des calculs de point fixe, etc.

Certaines propriétés d'un système dynamique peuvent être spécifiées à l'aide de relations instantanées sur des variables d'état. Par exemple, "une variable x du système, de type entier, vaut toujours 15" ou bien "une variable x' de type booléen, vaut toujours *faux*". Nous qualifions de ce type de propriété d'*invariants*. D'autres propriétés en revanche ne peuvent s'exprimer uniquement à l'aide de relations instantanées. C'est par exemple le cas d'une propriété qui consisterait à vérifier que les valeurs *vrai* et *faux* d'une variable d'état booléenne alternent (i.e. un *flipflop*) : $\forall i \in \mathbb{N}, x_i = \neg x_{i+1}$. De telles propriétés sont appelées *propriétés localement testables d'ordre k* [159], où k représente le nombre d'indices temporels intervenant dans l'expression de la propriété (c'est-à-dire de i à $i + k$).

Une technique permettant de vérifier ce genre de propriété consiste à utiliser un programme observateur (cf. chapitre 4). Ici, l'observateur contient des variables d'état qui mémorisent les k états sur lesquels porte chaque propriété localement testable d'ordre k. Un avantage déjà mentionné de cette technique est le fait qu'elle repose sur un unique formalisme, contrairement à certaines techniques qui en combinent plusieurs (par exemple, association de la logique temporelle aux automates pour faire du *model-checking*). Quelques-unes des propriétés considérées ci-dessous (en l'occurrence, des invariants) sont vérifiées en utilisant des observateurs.

6.2.2 Vérification de propriétés sur la file de messages

Nous voulons vérifier que la file modélisée par `safe_fifo` est bien robuste : lors des lectures, les messages sont reçus dans le même ordre que celui dans lequel ils ont été insérés lors des écritures ; de plus, il n'y a aucune perte de messages. Cette propriété est dénotée par R. Pour la vérification, nous allons procéder par étapes. Nous analysons d'abord les comportements de la file lorsqu'on tente d'y accéder alors qu'elle est dans un état critique (pleine ou vide). Elle doit alors satisfaire les propriétés suivantes :

- (S_1) : *sur une demande d'écriture dans une file déjà pleine, l'état de celle-ci reste inchangé (i.e., le message ne peut y être inséré)* ;
- (S_2) : *sur une demande de lecture dans une file déjà vide, son état demeure inchangé (i.e., aucun message ne peut en être retiré).*

Les propriétés (S_1) et (S_2) expriment des propriétés de sûreté [3] lors des accès à `safe_fifo`. Nous nous intéressons ensuite aux invariants ci-après :

- (I_1) : *sur une demande d'écriture, lorsque la file n'est pas déjà pleine, le message est bien inséré dans celle-ci* ;
- (I_2) : *sur une demande de lecture, lorsque la file n'est pas déjà vide, un message est bien reçu de celle-ci.*

Les propriétés (I_1) et (I_2) expriment le fait que toute requête sur la file, en dehors des situations critiques (c'est-à-dire, file pleine ou vide) selon le type d'opération, est "satisfaite".

La vérification de ces propriétés par `safe_fifo` écarte toute situation où il y aurait soit écrasement de messages, soit lecture d'un message "fantôme", dans le fonctionnement de celle-ci. Pour montrer que `safe_fifo` satisfait R, il suffira alors de montrer que le fonctionnement interne de la file garantit R. Les propriétés (S_1), (S_2), (I_1) et (I_2) vont être vérifiées à l'aide de

3. Ces propriétés seront vérifiées en considérant une reformulation de celles-ci en termes de propriétés d'accessibilité.

l'outil SIGALI. Nous montrons ensuite, en raisonnant sur la spécification SIGNAL de `safe_fifo`, qu'elle est robuste.

Nous allons considérer une abstraction du modèle qui prend en compte uniquement les variables d'état (i.e. les signaux définis à l'aide des opérateurs `$` et `cell`). Ce sont les signaux `nbmess`, `queue` et `mess_out`. Ceux-ci reflètent entièrement la dynamique du système d'équations associé à `safe_fifo`. Nous représentons l'état de ce système par st. Lorsque la file est pleine (resp. vide), la valeur de st est notée $full$ (resp. $empty$) ; autrement elle vaut $none$. Si $[\![safe\_fifo]\!]$ dénote l'ensemble des comportements associés au processus `safe_fifo` et b un comportement lui appartenant, les propriétés ci-dessus sont alors exprimées de la façon suivante :
$\forall b \in [\![\texttt{safe\_fifo}]\!]$, $\mathcal{T}_1 = tags(b(mess\_in))$, $\mathcal{T}_2 = tags(b(get\_mess))$,

$$\begin{array}{lcll} t \in \mathcal{T}_1 \wedge t \neq min(\mathcal{T}_1) \wedge b(st)(pred_{\mathcal{T}_1}(t)) = full & \Rightarrow & b(st)(t) = b(st)(pred_{\mathcal{T}_1}(t)) & (S_1) \\ t \in \mathcal{T}_2 \wedge t \neq min(\mathcal{T}_2) \wedge b(st)(pred_{\mathcal{T}_2}(t)) = empty & \Rightarrow & b(st)(t) = b(st)(pred_{\mathcal{T}_2}(t)) & (S_2) \\ t \in \mathcal{T}_1 \wedge t \neq min(\mathcal{T}_1) \wedge b(st)(pred_{\mathcal{T}_1}(t)) = none & \Rightarrow & t \in tags(b(new\_mess\_in)) & (I_1) \\ t \in \mathcal{T}_2 \wedge t \neq min(\mathcal{T}_2) \wedge b(st)(pred_{\mathcal{T}_2}(t)) = none & \Rightarrow & t \in tags(b(mess\_out)) & (I_2) \end{array} \quad (6.5)$$

On obtient ainsi des propriétés localement testables d'ordre 1. Les valeurs de st peuvent être exprimées à l'aide des signaux `OK_write` et `OK_read`, spécifiés au sein de `basic_fifo`. Leur définition est basée sur le nombre de messages présents dans la file, à savoir la "valeur retardée" de `nbmess`. Par conséquent, le nombre de messages courant suffit pour caractériser l'évolution de l'état de `safe_fifo`. Nous obtenons les correspondances ci-après : $\forall b \in [\![\texttt{safe\_fifo}]\!]\ \forall t \in tags(b)$

$$\begin{array}{lcl} b(st)(t) = full & \Leftrightarrow & b(nb\_mess)(t) = fifo\_size \\ b(st)(t) = empty & \Leftrightarrow & b(nb\_mess)(t) = 0 \\ b(st)(t) = none & \Leftrightarrow & 0 < b(nb\_mess)(t) < fifo\_size \end{array} \quad (6.6)$$

Enfin, puisque SIGALI ne permet que du *model-checking* booléen, nous devrons encoder `nbmess` en un signal booléen (car il est de type entier).

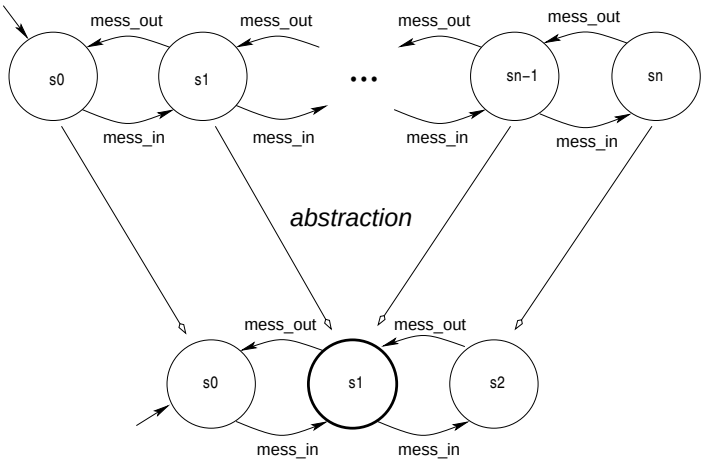


FIGURE 6.4 – Abstraction du comportement d'une *n-fifo* par une 2-*fifo*.

Abstraction de la file. On appelle *n-fifo*, une *fifo* pouvant contenir au maximum n messages.

Les différents états de celle-ci peuvent être représentés à travers un automate fini à $n+1$ états. Sur la FIG. 6.4, l'automate du haut représente les comportements possibles d'une *n-fifo*. Dans chaque état sk (représenté par un cercle), k dénote le nombre courant de messages dans la *fifo* :

$$\forall k \geq 0 \quad (nbmess = k \Rightarrow sk = true) \wedge (nbmess \neq k \Rightarrow sk = false)$$

L'état initial est $s0$. Les étiquettes *in* et *out* dénotent respectivement les écritures et lectures dans la file. Cet automate va servir d'abstraction pour une `safe_fifo` de taille n. D'autre part, pour simplifier l'analyse, nous pouvons abstraire cet automate en un second automate ayant trois états. Cela est justifié par la nature des propriétés considérées (c'est-à-dire (S_1), (S_2), (I_1) et (I_2)). En effet, le problème de la vérification de celles-ci sur les états extrêmes ($s0$ et sn) est identique pour une *n-fifo* et une 2-*fifo*. Quant aux états intermédiaires, ils admettent tous le même type d'actions. En particulier, nous pouvons nous contenter d'un seul état (i.e. $s1$ dans la 2-*fifo*).
Les analyses effectuées par la suite seront valables pour toute *n-fifo*, avec $n \geq 1$.

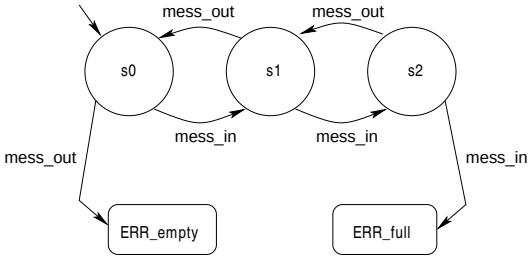


FIGURE 6.5 – Abstraction du comportement d'une 2-*fifo*.

En plus des trois états de la 2-*fifo*, nous avons ajouté deux états particuliers représentés par des rectangles (cf. FIG. 6.5). Ils caractérisent une tentative d'action "illégale" sur la 2-*fifo*. L'état *ERR_empty* est atteint lorsqu'on tente une lecture dans une file vide. De façon analogue, l'automate bascule dans l'état *ERR_full* sur une tentative d'écriture dans une file pleine.

La spécification d'un automate est très facile en SIGNAL. Chaque état est spécifié par une variable d'état booléenne. Pour illustration, nous donnons sur la FIG. 6.6 la définition de l'état $s0$ (le signal `prev_s1` décrit la valeur précédente de l'état `s1`).

```
(| s0 := (true when prev_s1 when (^mess_out)) default
         (false when prev_s0 when (mess_in ^+ mess_out)) default
         prev_s0
 | prev_s0 := s0$1 init true
 | ...
 |)
```

FIGURE 6.6 – Spécification SIGNAL de l'état s0 dans l'automate 2-*fifo*.

Le programme SIGNAL qui abstrait la 2-*fifo* dérive de la composition de l'automate ci-dessus, et du processus `safe_fifo` dans lequel la définition des signaux `OK_write` et `OK_read` change

comme indiqué sur la FIG. 6.7.

```
(| OK_write := false when (prev_err_full or prev_s2) default true
 | OK_read := false when (prev_err_empty or prev_s0) default true
 |)
```

FIGURE 6.7 – Spécification des conditions d'écriture et lecture suivant l'abstraction.

La première équation exprime le fait qu'il ne peut y avoir d'écriture lorsque l'automate était précédemment dans l'un des états $s2$ ou Err_full (i.e., il y avait déjà deux messages dans la *fifo*). Dans ce cas, `OK_write` vaut *faux*, sinon il prend la valeur *vrai*. L'autre équation définit de façon similaire le signal `OK_read`. Les signaux `s0`, `s1`, `s2`, `err_empty`, `err_full`, `OK_write` et `OK_read` sont définis à la même horloge, celle de `nbmess`, qui est la plus rapide.

L'outil SIGALI peut alors être utilisé pour vérifier les propriétés. Pour cela, il faut d'abord produire le modèle de l'abstraction dans le format d'entrée du *model-checker*. On l'obtient automatiquement en compilant le fichier source SIGNAL avec l'option correspondante (notée **- z3z**, en référence à l'encodage dans $\mathbb{Z}/3\mathbb{Z}$ mentionné au chapitre 4).
Un certain nombre d'opérateurs sont offerts par SIGALI, grâce auxquels on peut spécifier puis évaluer des formules. Nous utiliserons essentiellement quatre d'entre eux pour vérifier les propriétés (S_1), (S_2), (I_1) et (I_2). Les opérateurs `B_True` et `B_False` s'appliquent à une expression booléenne (définie à l'aide des variables du système). Ils permettent respectivement de tester si l'expression vaut *vrai* ou *faux*, dans un état donné du système (i.e., ce sont des opérateurs statiques). Les deux autres opérateurs sont `Always` et `Reachable`. Ils sont utilisés lorsqu'il s'agit de propriétés liées à l'évolution du système (i.e., ce sont des opérateurs dynamiques). Lorsqu'il est appliqué à une expression, l'opérateur `Always` permet de vérifier que la propriété ainsi exprimée est satisfaite par *tous les états* possibles du système pendant son évolution. Quant à `Reachable`, il teste l'existence d'*un état* satisfaisant la propriété pendant l'évolution du système.

```
Sigali;                                          (1)
read("safe_fifo.z3z");                           (2)
read("Creat_SDP.lib");                           (3)
read("Verif_Determ.lib");                        (4)
Always(B_False(Err_full));      →  True          (5)
Reachable(B_True(Err_empty));   →  False         (6)
```

FIGURE 6.8 – Script pour la vérification des propriétés de sûreté.

Considérons le script sur la FIG. 6.8 : on commence par invoquer SIGALI (ligne `(1)`), puis tous les fichiers nécessaires sont chargés (`Creat_SDP.lib` et `Verif_Determ.lib` contiennent des fonctions spécifiques à SIGALI). La formule de la ligne `(5)`, définie à l'aide des opérateurs `Always` et `B_False` sur la variable `ERR_full`, est évaluée à `True`. Elle exprime le fait que l'état ERR_full demeure toujours inaccessible. En d'autres termes, aucun écrasement ne peut être

provoqué, donc la propriété (S_1) est vérifiée. Quant à l'évaluation de la formule à la ligne (6)), elle exprime le fait que l'état *Err_empty* n'est jamais accessible, d'où la propriété (S_2). On vérifie ainsi les propriétés S_1 et S_2 par une simple analyse d'accessibilité.

```
(| (| actual_write := true when(^new_mess_in) default false                    (a)
   | inv1 := actual_write when(z_s0 or z_s1) when(^mess_in) default z_inv1     (b)
   | z_inv1 := inv1 $ 1 init true                                              (c)
 | (| actual_read := true when(^mess_out) default false
   | inv2 := actual_read when(z_s1 or z_s2) when pull_mess default z_inv2
   | z_inv2 := inv2 $ 1 init true
   |)
 |)
```

FIGURE 6.9 – Spécification des observateurs pour les propriétés (I_1) et (I_2).

Pour vérifier les propriétés (I_1) et (I_2), on considère deux *observateurs* qui vont scruter les comportements de la file lorsqu'on y accède. Sur la FIG. 6.9, le booléen `actual_write` indique si oui ou non il y a écriture dans la file (cf. équation (a)). Il est utilisé dans la définition de la variable d'état `inv1` qui dénote la propriété (I_1). Cette variable prend la valeur de `actual_write` lorsque la file n'est pas déjà pleine et qu'il y a une demande d'écriture, autrement elle garde sa valeur précédente (cf. équation (b)). Ainsi, l'observateur associé à (I_1) est constitué des équations (a), (b) et (c). Nous procédons de façon similaire pour I_2 (cf. FIG. 6.9).
Tous ces signaux sont à la même horloge que `nbmess`. Le script correspondant à la vérification des invariants est donné par la FIG. 6.10. L'évaluation des formules correspondantes montre que les variables `inv1` et `inv2` qui encodent respectivement les invariants (I_1) et (I_2) valent toujours `True`, et ne peuvent jamais devenir fausses.

```
Sigali;
read("safe_fifo.z3z");
read("Creat_SDP.lib");
read("Verif_Determ.lib");
Always(B_True(inv1));        →  True
Reachable(B_False(inv1));    →  False
Always(B_True(inv2));        →  True
Reachable(B_False(inv2));    →  False
```

FIGURE 6.10 – Script pour la vérification des invariants.

À présent, nous devons vérifier que le composant `safe_fifo` préserve bien l'ordre des messages stockés dans la file d'attente. L'idée consiste à considérer le flot de messages d'entrée dans l'appel à `basic_fifo` au sein de `safe_fifo` (i.e., le flot associé au signal `new_mess_in`). En effet, ce sont uniquement les instances de ce signal qui sont prises en compte et lues ensuite. Il s'agit donc de montrer que :

– (E_1) : *l'ordre des valeurs présentes dans le flot d'entrée est préservé par la définition de la file de messages* ; i.e., $\forall b \in [\![\texttt{safe\_fifo}]\!]\ \forall v, v' \in \mathcal{V}$ t.q. $\exists t, t'$ avec $v = b(new\_mess\_in)(t)$,

`mess_in` :	$\perp$	1	2	$\perp$	3	4	...
`queue` :	$[?,?]$	$[?,1]$	$[1,2]$	$[1,2]$	$[2,3]$	$[3,4]$	...

FIGURE 6.11 – Rangement de messages dans la file d'attente *queue* à partir du flot *mess_in*.

$v' = b(new\_mess\_in)(t')$ et i, i' les indices respectifs de v et v' présents dans `queue` [4], on a :

$$t \leq t' \Rightarrow i \leq i'$$

– (E_2) : *l'ordre des valeurs présentes dans le flot de sortie est identique à celui des valeurs stockées dans la file de messages* ; i.e., $\forall b \in safe\_fifo\ \forall v, v' \in \mathcal{V}$ t.q. $\exists t, t'$ avec $v = b(mess\_out)(t)$, $v' = b(mess\_out)(t')$ et i, i' comme ci-dessus, on a :

$$i \leq i' \Rightarrow t \leq t'$$

En vérifiant que l'ordre des valeurs est préservé dans les deux cas distingués ci-dessus, nous établissons que la propriété d'invariance de flot est bien satisfaite par `safe_fifo`.

1. **Vérification de** (E_1)**.**
   La vérification de cette propriété repose essentiellement sur la sémantique de l'opérateur `window`. En effet, la file des messages représentée par le signal `queue` est définie en appliquant cet opérateur au signal `mess_in` (cf. équation `(1.e)` dans `basic_fifo`). Le flot associé à ce dernier contient les messages d'entrée dans la file. La sémantique de l'opérateur `window` est telle que toute lecture de valeur sur le flot d'entrée provoque un décalage vers la gauche des valeurs déjà présentes dans le tableau. La nouvelle valeur est alors insérée par la droite. Par exemple, la trace représentée sur la FIG. 6.11 illustre le fonctionnement d'une file de taille égale à deux. Le symbole "?" désigne une valeur d'initialisation quelconque (des opérateurs comme `window` et `cell` requièrent généralement des valeurs d'initialisation ; lorsque celles-ci ne sont pas spécifiées explicitement, le compilateur choisit des valeurs initiales par défaut définies par le langage).
   Pour l'équation `queue := new_mess_in window fifo_size`, la sémantique de la fenêtre glissante est la suivante (pour alléger l'écriture, les valeurs initiales sont omises) :

   $$(\forall t \geq 0)\ \ (t + i \geq fifo\_size) \Rightarrow (queue_t[i] = new\_mess\_in_{t-fifo\_size+i+1})$$

   Grâce cette sémantique de l'opérateur `window`, les valeurs qui sont prises en compte dans le flot associé au signal d'entrée `mess_in` sont stockées dans `queue` (représentant la file) suivant leur ordre d'apparition. Donc la propriété (E_1) est vérifiée.

2. **Vérification de** (E_2)**.**
   Le flot de sortie est déterminé par l'équation `(1.f)` de `basic_fifo`. Seul le second argument de l'opérateur `default` est à considérer ici puisque le premier est écarté par (S_2). Sur une requête de lecture, on dénote par i l'indice du message récupéré du tableau `queue`. La valeur de i est donnée par la différence entre la taille maximale (`fifo_size`) de la file et le nombre précédent de messages de la file (`prev_nbmess`) :

   $$\forall t \geq 0\ \ i_t = fifo\_size - prev\_nbmess_t$$

4. Nous considérons que les valeurs sont insérées de la droite vers la gauche dans ce tableau défini à l'aide de l'opérateur `window`.

Nous considérons l'invariant suivant : *à tout instant t, l'indice i_t indique le message le plus ancien de la file lorsque celle-ci n'est pas vide*. Il s'agit alors de vérifier qu'il est toujours maintenu. Nous examinons les instants auxquels on accède à la file des messages.

(a) Cas 1 : *lecture d'un message à un instant k.*
Lorsqu'on lit un message de la file à un instant k, le signal `nbmess` est décrémenté de un (cf. `(1.a)`), c'est-à-dire :

$$nbmess_k = prev\_nbmess_k - 1$$

À l'instant suivant $k+1$, on a :

$$prev\_nbmess_{k+1} = nbmess_k$$

D'où,

$$\begin{aligned} i_{k+1} &= fifo\_size - prev\_nbmess_{k+1} \\ &= fifo\_size - nbmess_k \\ &= fifo\_size - (prev\_nbmess_k - 1) \\ &= (fifo\_size - prev\_nbmess_k) + 1 \\ &= i_k + 1 \end{aligned}$$

D'autre part, lors de chaque lecture, le tableau représenté par le signal `queue` reste inchangé (cf. équation `(1.e)`). Par conséquent, l'invariant est bien préservé car l'élément le plus ancien dans la file est dorénavant celui ayant pour indice i_{k+1} dans le tableau.

(b) Cas 2 : *écriture d'un message à un instant k.*
Lorsqu'un message est effectivement écrit dans la file à un instant k, le signal `nbmess` est incrémenté de un (cf. `(1.a)`), c'est-à-dire :

$$nbmess_k = prev\_nbmess_k + 1$$

À l'instant suivant, $k+1$, on a :

$$prev\_nbmess_{k+1} = nbmess_k$$

D'où,

$$\begin{aligned} i_{k+1} &= fifo\_size - prev\_nbmess_{k+1} \\ &= fifo\_size - nbmess_k \\ &= fifo\_size - (prev\_nbmess_k + 1) \\ &= (fifo\_size - prev\_nbmess_k) - 1 \\ &= i_k - 1 \end{aligned}$$

Contrairement à la lecture, ici le signal `queue` est modifié. En effet, la fenêtre glissante sur le signal `new_mess_in` va prendre en compte l'occurrence de celui-ci à l'instant k. Les éléments déjà présents dans `queue` sont décalés selon la sémantique de l'opérateur `window` (i.e., décalage d'une cellule vers les indices inférieurs du tableau et insertion du nouveau message dans la cellule ayant le plus grand indice). Ainsi, l'indice i_{k+1} désigne toujours le même message que précédemment. Donc, l'invariant est préservé.

Dans les deux cas présentés ci-dessus, l'invariant est toujours maintenu. Nous pouvons en déduire que les valeurs présentes dans la file sont restituées dans le flot de sortie suivant le même ordre où elles ont été stockées.

Le raisonnement ci-dessus nous permet d'assurer la robustesse du mécanisme de communication ainsi modélisé. Nous venons d'illustrer la vérification de quelque propriétés à l'aide d'outils et de raisonnements basés sur le modèle sémantique du langage SIGNAL.

6.3 Conclusion

L'approche ascendante présentée dans ce chapitre permet de décrire des composants complexes de façon incrémentale. L'idée consiste à identifier un composant de base à partir duquel d'autres composants peuvent être définis. Le critère de choix de celui-ci est qu'il doit être le moins contraint possible dans ses comportements. En lui adjoignant progressivement des contraintes, on obtient de nouveaux composants. La programmation à l'aide du langage SIGNAL favorise des descriptions modulaires et génériques. Pour des applications à grande échelle, cette démarche est bien adaptée. Elle permet de détecter et corriger plus facilement les erreurs au bon moment (i.e. durant la définition du composant concerné). Par ailleurs, la réutilisation réduit de façon significative l'effort de mise en œuvre car les composants nécessaires existent déjà. Une majeure partie de l'effort concerne leur intégration. Nous avons illustré l'approche à l'aide d'un exemple simple qui consiste en la modélisation d'un mécanisme de file d'attente, géré suivant la politique *fifo*. Nous avons montré ensuite comment des propriétés fonctionnelles d'un tel modèle peuvent être vérifiées à l'aide d'outils de preuve (nous avons utilisé ici l'outil SIGALI, accessible dans POLYCHRONY, pour faire du *model-checking*).

Nous noterons enfin que l'approche s'apparente à une forme d'héritage où le composant initial (i.e. le moins contraint) joue le rôle de la classe dont héritent les autres classes, représentées ici par les composants déduits par ajout de nouvelles contraintes au composant initial. Dans l'exemple traité, `safe_fifo` hérite de `basic_fifo`. Le mécanisme d'héritage a été étudié plus en détail par M. Kerbœuf dans le contexte polychrone [134]. La notion habituelle de classe dans un langage orienté objet tel que JAVA est représentée par un processus SIGNAL. Les méthodes quant à elles sont dénotées par les signaux du processus : les signaux d'entrée et de sortie constituent respectivement les méthodes importées et exportées par la classe ; les signaux locaux sont des méthodes privées. Enfin, les attributs sont définis par l'ensemble de valeurs des variables d'état du processus.

Troisième partie

Méthodologie de conception polychrone appliquée à l'avionique

Chapitre 7

Conception de systèmes avioniques et norme ARINC

La complexité croissante et la haute criticité des systèmes enfouis temps réel dans le domaine de l'avionique posent un certain nombre de défis concernant leur développement. Parmi ces défis on peut citer la correction des systèmes conçus vis-à-vis des exigences, l'effort de développement, la correction et la fiabilité de l'implantation, le temps de mise sur le marché du "produit". Il y a donc une grande nécessité de méthodologies de conception qui soient adéquates, autrement dit qui prendraient en compte les défis mentionnés. D'après Pnueli [175], de telles méthodologies doivent au minimum inclure la possibilité d'avoir des spécifications formelles, de faire de la vérification de propriétés et de l'analyse. D'autre part, la génération automatique de code (éventuellement distribué) doit être possible.

Dans ce chapitre, nous abordons de façon générale les systèmes avioniques ainsi que leur conception (section 7.1). Pour cela, nous commençons par observer quelques-unes des tendances récentes dans ce domaine (section 7.1.1). Ensuite, nous présentons les deux principales approches qui permettent de décrire les architectures de systèmes dans l'avionique (section 7.1.2). Enfin, nous introduisons les concepts sur lesquels nous nous basons pour définir les modèles décrits dans le chapitre suivant, définis par la norme ARINC 653 (section 7.2).

7.1 Les systèmes avioniques et leur conception

L'avionique occupe une place non négligeable dans le coût des avions modernes (par exemple, dans l'aéronautique civile, on estime ce coût jusqu'à 35% du coût total). On entend par ce terme l'ensemble des logiciels et matériels embarqués à bord de l'avion, qui assurent diverses fonctions telles que le traitement des informations provenant des capteurs, le pilotage automatique, la gestion du niveau de carburant, les échanges de messages avec l'opérateur au sol pendant le vol, etc.

7.1.1 Quelques tendances dans l'industrie de l'avionique

Les systèmes avioniques modernes [152] [183] [66] se distinguent des systèmes traditionnels par un certain nombre de caractéristiques liées notamment à leur complexité et donc à la façon de les concevoir.

– Les fonctionnalités ne cessent d'augmenter dans les systèmes avioniques : la maintenance et les diagnostics à bord, la simulation de missions, le besoin d'autonomie, etc. D'autre

part, le niveau d'intégration des fonctions pour une coopération efficace est de plus en plus élevé.
– Contrairement à l'approche traditionnelle où les fonctions sont très faiblement couplées (par exemple, le pilotage automatique et la navigation fonctionnent sur des dispositifs différents qui sont indépendants), les systèmes avioniques adoptent à l'heure actuelle une approche plutôt intégrée, où des fonctions de niveaux critiques différents peuvent s'exécuter sur un même dispositif.
– Par ailleurs, l'industrie de l'avionique utilise de plus en plus des composants matériels commerciaux qui *a priori* n'ont pas été conçus pour de tels systèmes. Si cela peut réduire les coûts de développement et augmenter les fonctionnalités, cela présente néanmoins l'inconvénient du manque de garantie suffisante de tels produits (ces derniers peuvent devenir vite obsolètes ou indisponibles).
– Les exigences de correction et de fiabilité imposent l'utilisation de méthodes formelles pour répondre à ces attentes.

7.1.2 Conception des systèmes avioniques : approches fédérées et intégrées

7.1.2.1 Les architectures fédérées

Habituellement dans les systèmes avioniques, chaque fonction de contrôle dispose de ses propres ressources matérielles (représentées par une machine ou un calculateur) pour son exécution, comme le montre la Fig. 7.1. Ces dispositifs, qui sont très souvent répliqués pour la tolérance aux fautes, peuvent varier d'une fonction à l'autre. Il en résulte ainsi une architecture hétérogène et faiblement couplée, où chaque fonction peut opérer de manière quasi indépendante vis-à-vis des autres fonctions. Ce qui n'est pas le cas localement, où les éléments constituant une fonction doivent beaucoup coopérer pour accomplir la tâche affectée à celle-ci. Une telle architecture est qualifiée de *fédérée*[1] [7] [183]. Par exemple, la construction des *Airbus* A330 et A340 repose sur ce type d'architecture. Les équipements numériques mettant en œuvre certaines des fonctions à bord sont reliés par un bus mono-émetteur.

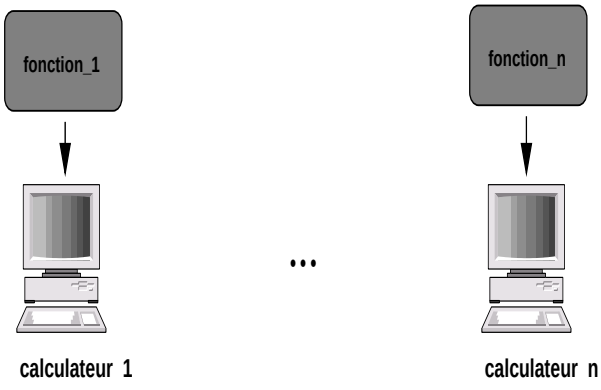


Figure 7.1 – Architecture fédérée.

Un avantage majeur des architectures fédérées, en plus de leur simplicité, est la minimisation des risques de propagation d'erreurs qui peuvent survenir lors de l'exécution d'une fonction au sein du système. Cela assure une meilleure disponibilité du système global dans la mesure

1. En anglais, *Federated Architecture*.

où une fonction défaillante peut éventuellement être isolée sans avoir recours à un arrêt complet du système. Cette caractéristique inhérente aux architectures fédérées est primordiale dans des systèmes critiques comme ceux qui sont embarqués dans les avions. Un autre avantage des architectures fédérées est leur hétérogénéité. En effet, les types de machines utilisées peuvent varier d'une fonction à l'autre au sein du même système. Cela permet l'utilisation de machines aux puissances variables.

Malheureusement, un gros inconvénient des architectures fédérées est le risque potentiel d'utilisation abusive de ressources matérielles. Le fait que chaque fonction nécessite sa propre machine (qui plus est, répliquée pour la tolérance aux fautes) peut conduire à un coût trop élevé : d'une part, il faut suffisamment d'espace à bord de l'avion pour stocker tous les calculateurs et leur poids doit être supportable ; d'autre part, il faut en assurer l'installation et la maintenance. De plus, on rappelle qu'un objectif essentiel dans la conception des systèmes en général, est justement la réduction du coût de réalisation. D'autre part, se pose le problème de la définition d'une approche claire concernant la séparation des différentes fonctions à répartir dans une architecture fédérée. Actuellement, en ce qui concerne les systèmes embarqués dans les avions, cette séparation repose hélas en grande partie sur des leçons tirées des nombreux accidents survenus par le passé (par exemple, l'expérience a montré que deux variables de contrôle qui interagissent l'une sur l'autre doivent être impérativement fortement couplées).

7.1.2.2 Les architectures modulaires intégrées

Plus récemment, une autre vision a émergé dans la conception des systèmes avioniques ; celle-ci a pour but de pallier les insuffisances des architectures fédérées en proposant une organisation du système dans laquelle plusieurs fonctions peuvent désormais partager des ressources de calcul et de communication communes (offertes par une machine tolérante aux fautes). C'est ce qui est illustré par la Fig. 7.2. Ce type d'architecture est qualifié de *modulaire intégré*, plus communément appelé *IMA* (*Integrated Modular Avionics*) [7] [8]. Des exemples d'avions adoptant la solution intégrée sont l'*Airbus* A380 ou le *Boeing* B777.

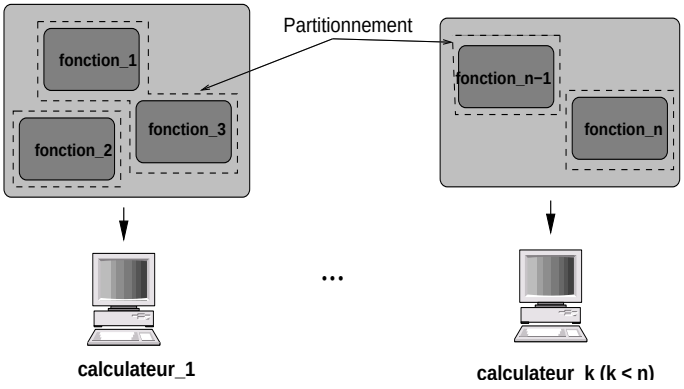


Figure 7.2 – Architecture modulaire intégrée (IMA).

Cette organisation du système permet d'économiser les ressources, contrairement aux architectures fédérées, et par la même occasion le coût de réalisation peut être limité de façon raisonnable. Cependant, elle introduit une probabilité assez élevée de propagation d'erreur qui n'existe pas dans les architectures fédérées. Par exemple, une fonction en dysfonctionnement peut monopoliser le système de communication ou envoyer des commandes inappropriées, et

pour chacune des fonctions il est difficile de se mettre à l'abri d'un tel comportement.

Une solution a été proposée à travers le mécanisme dit de *partitionnement* [7]. Par ce moyen, une ou plusieurs applications regroupées au sein d'un même *module* peuvent s'exécuter de manière "sûre" (deux fonctions quelconques prévues pour s'exécuter dans un même module ne peuvent en aucun cas interagir l'une sur l'autre). Pour cela, le partitionnement procède en un découpage fonctionnel du système qui prend en compte les ressources disponibles en temps et en mémoire. L'unité d'allocation résultant de ce découpage est appelée *partition* (c'est l'équivalent d'un programme dans le cas d'un environnement contenant une seule application). En pratique, le partitionnement spatial des applications repose souvent sur l'utilisation de composants matériels chargés de gérer la mémoire, pour empêcher toute corruption de zones mémoires adjacentes à une zone en cours de modification. Quant au partitionnement temporel, il dépend des fonctionnalités attendues de l'ensemble (par exemple, on aura besoin d'exécuter plus souvent des fonctions qui s'occupent du rafraîchissement de paramètres critiques). Un autre avantage du mécanisme de partitionnement réside dans le fait qu'il facilite la vérification, la validation et la certification des systèmes embarqués dans les avions.

La norme avionique ARINC (*Aeronautic Radio INC*) [7] [8] définit les principes de base du partitionnement ainsi qu'un ensemble de services permettant de décrire la mise en œuvre d'applications avioniques sur une architecture de type IMA.

7.2 Introduction au standard ARINC

Dans cette section, nous présentons les notions définies dans le document numéro 653 [8] de la série 600 publiée par l'organisme en charge la norme ARINC. Ces notions vont servir de base pour les modèles SIGNAL utilisés pour décrire des applications avioniques.

Les spécifications ARINC 653 [8] sont dédiées principalement à la présentation des caractéristiques d'une interface générique entre la couche applicative et le système d'exploitation dans une architecture IMA. Cette interface appelée APEX (APplication EXecutive) représentée sur la FIG. 7.3, permet de contrôler l'exécution des partitions, y compris les communications entre celles-ci. Elle est constituée d'un ensemble de services que nous appelons *services APEX* dans la suite.

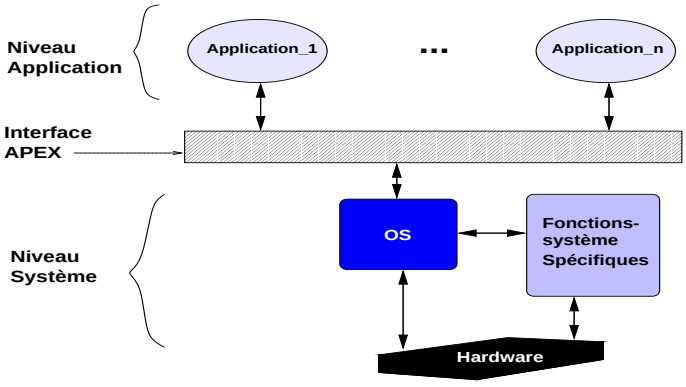


FIGURE 7.3 – Interface APEX de la norme ARINC.

Organisation des applications. Les applications (mettant en œuvre différentes fonctions) sont organisées de telle sorte qu'une ou plusieurs applications peuvent être regroupées au sein d'un *module-noyau* (en anglais, *core processing module* ou CPM) qui offre les ressources de calcul nécessaires. Celui-ci est un ensemble d'entités résultant du partitionnement des applications qu'il contient, appelées *partitions*. Ces dernières disposent chacune d'un espace mémoire propre et d'un budget temps pour s'exécuter. Elles sont gérées par un système d'exploitation (*module-level OS*) défini selon la norme ARINC 653. Celui-ci doit faire en sorte que le fonctionnement d'une partition ne soit pas entravé par celui d'une autre partition (ce problème est en majeure partie résolu lors du partitionnement). Les partitions sont quant à elles composées de *processus* représentant les entités élémentaires d'exécution qui accomplissent la mission affectée à la partition qui les contient. Leur gestion est placée sous la responsabilité du système d'exploitation (*partition-level OS*). La politique d'ordonnancement adoptée au sein de chaque partition peut varier. Par contre, l'ordonnancement doit être préemptif et basé sur les priorités. Ainsi, chaque fonction occupe les ressources de calcul de façon exclusive dans l'intervalle de temps qui lui est alloué ; et la stratégie d'ordonnancement interne à chaque fonction peut différer de celle des autres. Une analogie peut être faite entre partition/processus ARINC et processus/*thread* POSIX. Enfin, il n'existe aucune limite quant au nombre et la taille des partitions et processus (dans la limite de la mémoire physique disponible).

Les CPM sont répartis en *cabinets*. Les communications entre *cabinets* se font à travers des réseaux multiplexés formés de bus (par exemple de type ARINC 629) ou de brins Ethernet reliés par des commutateurs. La liaison des réseaux de communication IMA avec le monde externe (c'est-à-dire, les capteurs, les actionneurs ou le réseau de maintenance de l'appareil au sol) est réalisée grâce à des passerelles, appelées *gateway modules* (GWM).

Des exemples d'avioniques modulaires intégrées sont illustrés sur les Fig. 7.4 et 7.5 [25]. Le premier montre une architecture pour un Boeing B777. Les CPM regroupés d'un même *cabinet* échangent leurs données à travers un bus de type ARINC 659. Ce dernier est interne au *cabinet* IMA. Les communications avec l'extérieur se font à travers un bus de type ARINC 629, via les GWM. Ce bus est quant à lui, externe aux *cabinets* IMA dont il assure les échanges de messages. Dans le second exemple (i.e. Fig. 7.5), il s'agit d'une architecture pour un Airbus A380. Les différents CPM communiquent entre eux à travers un réseau Ethernet "switché" (au niveau des nœuds SW).

Une spécificité de la norme ARINC. Une particularité du standard APEX est sa dépendance vis-à-vis du type d'architecture de mise en œuvre [19]. Seules les architectures modulaires intégrées supportent l'interface APEX. En effet, celles-ci considèrent l'existence de deux bus particuliers : le *Backplane Data Bus* [6] et le *Data Bus* (représentant le réseau global) [5]. Le premier bus utilise un protocole basé sur une table qui donne de façon cyclique des fenêtres de messages. Chaque fenêtre a une taille prédéfinie. Elle peut contenir des messages entiers ou des portions de messages lorsque ces derniers sont de très grande taille. Le second bus fournit périodiquement des fenêtres de messages, avec des temps d'accès au bus adéquats qui permettent d'éviter tout conflit lors des envois. L'utilisation de ces bus influe notamment sur le schéma d'ordonnancement adopté par APEX. Cela implique l'adoption d'un schéma cyclique en ce qui concerne l'ordonnancement des partitions ainsi que les communications entre celles-ci. Lors de l'implémentation, les cycles d'exécution des processeurs doivent par conséquent être ajustés par rapport à ceux des bus. Les autres standards utilisés dans le domaine du temps réel [2] comme POSIX [205] et OSEK [172] n'imposent *a priori* aucune contrainte sur la nature

2. Nous reviendrons sur ces aspects dans la liste des perspectives qui figurent dans la conclusion de cette

de l'architecture de mise en œuvre d'un système. C'est donc un aspect qui les distingue de APEX.

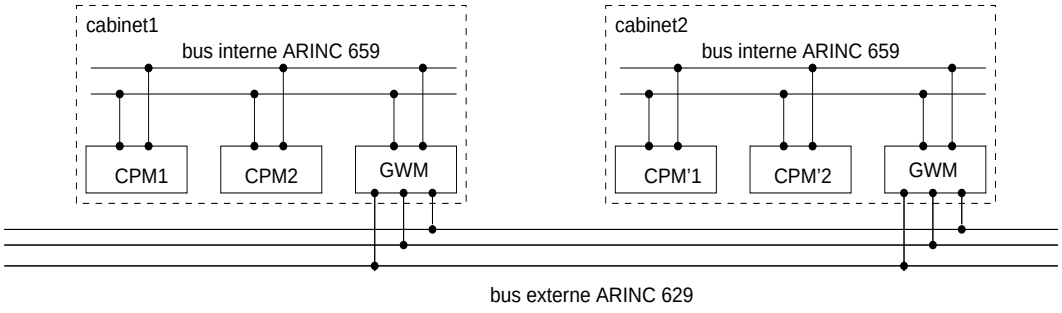


FIGURE 7.4 – Exemple d'avionique modulaire intégrée dans un Boeing B777.

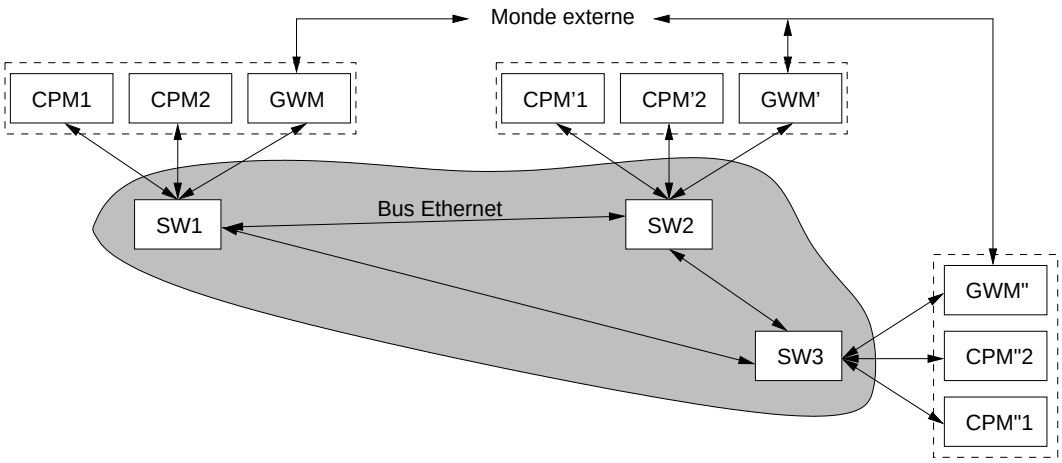


FIGURE 7.5 – Exemple d'avionique modulaire intégrée dans un Airbus A380.

7.2.1 Les partitions

Une partition représente l'entité de base qui doit satisfaire les contraintes spatiales et temporelles du partitionnement. La partition est constituée d'un ou plusieurs processus. Elle est caractérisée par un certain nombre de données telles que ses attributs de configuration et son contexte. Ces informations servent à gérer les partitions au sein d'un module.

Attributs. Parmi les attributs d'une partition, on distingue deux types : des attributs *fixes* et des attributs *variables*.
Attributs fixes :

1. **Identificateur** (type numérique) - Identificateur spécifique à chaque partition (il facilite l'activation des partitions et le routage des messages).
2. **Espace mémoire** (type dépendant du système d'adressage dans la plate-forme d'implantation) - Informations concernant l'espace mémoire alloué à la partition (zones de données et de code).

thèse, où nous discutons la modélisation de composants suivant POSIX et OSEK.

3. **Durée** (type numérique, dépendant de l'horloge considérée dans la plate-forme d'implantation ; par exemple, les unités de temps peuvent être en microsecondes ou en nanosecondes) - Temps d'occupation des ressources matérielles par la partition à chaque période d'activation.
4. **Période** (même type que la durée) - Période d'activation de la partition durant l'exécution globale.
5. **Criticité** (type dépendant de la convention choisie) - Niveau de criticité de la partition. Cette information reflète le fait qu'au sein d'une architecture IMA, un système peut être construit à partir de sous-systèmes (représentés par des partitions en l'occurrence) dont les niveaux de criticité varient, et que ceux-ci peuvent s'exécuter sur le même processeur.
6. **Communications** - Description des partitions et/ou autres entités avec lesquelles la partition communique.
7. **Point d'entrée** (type dépendant de la plate-forme) - Adresse de début pour l'exécution de la partition.
8. **Health Monitor** - Table dans laquelle sont spécifiés les traitements à faire en cas de mauvais fonctionnement de la partition.

Attributs variables :

1. **Niveau de verrouillage** (type numérique) - Niveau courant de verrouillage de la partition. Il indique si le ré-ordonnancement de processus est possible au sein d'une partition (le verrouillage de la partition empêche la préemption).
2. **Mode opératoire** - Mode de fonctionnement courant de la partition. Il existe quatre modes :
   - *idle* : mode d'arrêt (aucun processus n'est en train de s'exécuter dans la partition) ;
   - *cold_start* : mode de démarrage de la partition suite à une coupure longue, sans sauvegarde du contexte ;
   - *warm_start* : mode de démarrage de la partition suite à une coupure longue, avec sauvegarde du contexte ;
   - *normal* : mode fonctionnel (les ressources nécessaires à la partition ont été créées, et les processus sont activables).

Gestion. C'est le *module-level OS* qui contrôle la gestion (ordonnancement et communications) des partitions. Pour cela, il se base sur leurs attributs. D'autre part, le système d'exploitation s'appuie sur une horloge globale pour activer chaque partition durant le déroulement de l'application. Ainsi, chaque partition se voit attribuer une portion de temps au cours de laquelle elle a à sa disposition les ressources matérielles comme le processeur pour s'exécuter (cette attribution peut se faire de façon cyclique à intervalles fixes ou bien de manière arbitraire). Pendant cette tranche de temps, la partition est ininterruptible [3]. L'ordre d'activation des partitions est prédéterminé à la configuration (dans une table de configuration). Ainsi, l'ordonnancement est strictement déterministe dans le temps. De même, l'allocation de l'espace mémoire dont chacune des partitions aura besoin est préétablie à la configuration. Le droit en écriture dans une zone mémoire donnée est accordé statiquement à une partition au plus. Cela évite les conflits d'accès et par conséquent, une partition ne peut modifier une zone que lorsque cette dernière lui est réservée.

3. Néanmoins, ce comportement n'est pas toujours respecté. Les occurrences d'événements externes au module, même s'ils ne concernent pas la partition active dans le module, sont traitées prioritairement par le système d'exploitation.

Services	Description de la requête engendrée
get_partition_status	récupération du statut courant d'une partition
set_partition_mode	mise d'une partition dans un mode opérationnel donné

TABLE 7.1 – Services APEX pour la gestion des partitions.

Services	Description de la requête engendrée
create_sampling_port	création et initialisation d'un port fonctionnant en mode *sampling*
write_sampling_message	écriture d'un message dans un port fonctionnant en mode *sampling*
read_sampling_message	lecture d'un message dans un port fonctionnant en mode *sampling*
get_sampling_port_id	récupération de l'identificateur d'un port fonctionnant en mode *sampling*
get_sampling_port_status	récupération du statut courant d'un port fonctionnant en mode *sampling*
create_queuing_port	création et initialisation d'un port fonctionnant en mode *queuing*
send_queuing_message	envoi d'un message dans un port fonctionnant en mode *queuing*
receive_queuing_message	réception d'un message d'un port fonctionnant en mode *queuing*
get_queuing_port_id	récupération de l'identificateur d'un port fonctionnant en mode *queuing*
get_queuing_port_status	récupération du statut courant d'un port fonctionnant en mode *queuing*

TABLE 7.2 – Services APEX pour la communication entre partitions.

Le modèle d'ordonnancement des partitions présente les caractéristiques suivantes :

- L'unité d'ordonnancement est la partition ;
- Les partitions n'ont pas de priorité ;
- L'algorithme d'ordonnancement est prédéterminé, répétitif avec une périodicité fixe et n'est configurable que par l'intégrateur du système. Pendant chaque cycle d'activation des partitions, au moins une tranche de temps est allouée à chacune d'entre elles ;
- Seul le *module-level OS* contrôle l'allocation des ressources à une partition.

Toute exécution de partition, déclenchée par le système d'exploitation, commence lorsque celui-ci passe en mode opérationnel. Les ressources (canaux de communication, processus, mécanismes de synchronisation) utilisées par les partitions sont spécifiées et les instances correspondantes sont créées. Les partitions passent alors en mode *normal*.
La gestion des partitions est réalisée à l'aide des services APEX indiqués dans la table TAB. 7.1.

Communications entre partitions. Ces communications [8] impliquent :

- soit deux partitions au moins (au sein d'un même module ou non),
- soit des partitions et d'autres types d'équipements.

La communication est essentiellement réalisée à travers des messages. Ceux-ci sont constitués de données organisées en séquences de taille finie dans les espaces mémoires associés à la source et aux destinations. L'échange de messages entre partitions met en jeu à la fois un expéditeur et un ou plusieurs destinataires. L'interface APEX permet de transmettre des messages dans leur intégralité. Ils peuvent être envoyés en plusieurs morceaux lorsqu'ils sont trop longs. Les données contenues dans un message sont complètement transparentes au système de passage des messages. Parmi les types de messages distingués, on note :

- Des messages de *taille* soit *fixe* soit *variable*. Dans le second cas, l'expéditeur spécifie la longueur du message lors de l'envoi ;
- Des messages *périodiques* ou *apériodiques*. La périodicité des messages est implicite (elle repose sur la façon dont ils sont envoyés par les partitions. Le système lui-même ne fait

pas de distinction entre messages périodiques et apériodiques) ;
- Des messages *dirigés* (i.e., d'une source unique vers une destination unique) ou *diffusés* (i.e., d'une source unique vers deux destinations au moins) ;
- Des messages avec *accusé de réception*.

Pour un message quelconque, la combinaison de ces types doit se faire avec précaution : un message qui est diffusé, ou bien qui est dirigé et périodique, ne peut en aucun cas comporter d'accusé de réception. Autrement, toutes les autres combinaisons sont possibles. Par exemple, un message peut être à la fois de taille fixe, dirigé, apériodique et avec accusé de réception.

Le support de base pour la transmission des messages entre partitions est le *canal*. Il définit un lien logique entre une source et une ou plusieurs destinations, ainsi que le mode de transfert et les caractéristiques des messages à échanger. L'accès aux canaux se fait via des *ports* qui indiquent aux partitions les informations spécifiques à un canal par lequel elles échangeraient des messages (par exemple, nature de la taille des messages acheminés). Tous les canaux et ports sont définis à la configuration [4]. Cela a pour conséquence de fixer définitivement les choix de connexions entre les ports (i.e., les sources, destinations et modes de transfert associés aux canaux demeurent inchangés durant l'exécution complète de l'application). Le transfert de messages à travers les canaux se fait suivant deux modes : *sampling* et *queuing*. Dans le premier mode, aucune file d'attente de messages n'est autorisée. Pendant le transfert, les messages successifs contiennent des données identiques mais mises à jour. Ainsi, tout message écrit sur le port-source y reste jusqu'à ce qu'il soit transmis ou bien écrasé par une nouvelle occurrence de message. Ce mode offre la possibilité à l'expéditeur et au destinataire de respectivement envoyer et recevoir un message n'importe quand, sans être bloqués. Le mode *queuing*, à l'inverse de l'autre, autorise les files d'attente de messages au niveau des ports reliant un canal. La politique de gestion est en FIFO. Ce mode garantit l'absence de perte de messages durant les transferts. Lors des échanges de messages entre partitions, ni l'expéditeur ni le destinataire ne connaît l'identité ou l'emplacement physique de l'autre. Cela offre plus de flexibilité quant à la conception des fonctionnalités du système indépendamment de l'architecture choisie.

Les communications entre les partitions sont réalisées à l'aide des services APEX décrits dans la table Tab. 7.2. Le *statut* d'un port est obtenu à l'aide des services *get_sampling_port_status* et *get_queuing_port_status*. Il comprend les informations suivantes :

- *sampling* port : la taille, le sens dans les communications (source/destinataire), la période de rafraîchissement et l'indicateur de validité des messages ;
- *queuing* port : la taille, le sens dans les communications (source/destinataire), les nombres courants de messages et de processus en attente sur le port.

Le schéma représenté sur la Fig. 7.6 décrit une application qui s'exécute sur une architecture à deux processeurs. Elle est constituée de trois partitions : `partition_1`, `partition_2` et `partition_3`. La première partition s'exécute sur le premier processeur tandis que les deux autres se partagent le second processeur. Chaque partition est composée de processus. Nous présentons dans la section suivante les caractéristiques de ces processus.

4. Il revient à l'intégrateur du système de vérifier que les nœuds (exemples : modules-noyaux, modules d'E/S ou *gateway modules*) reliés par les canaux sont bien configurés. Cela n'est pas du ressort de l'OS.

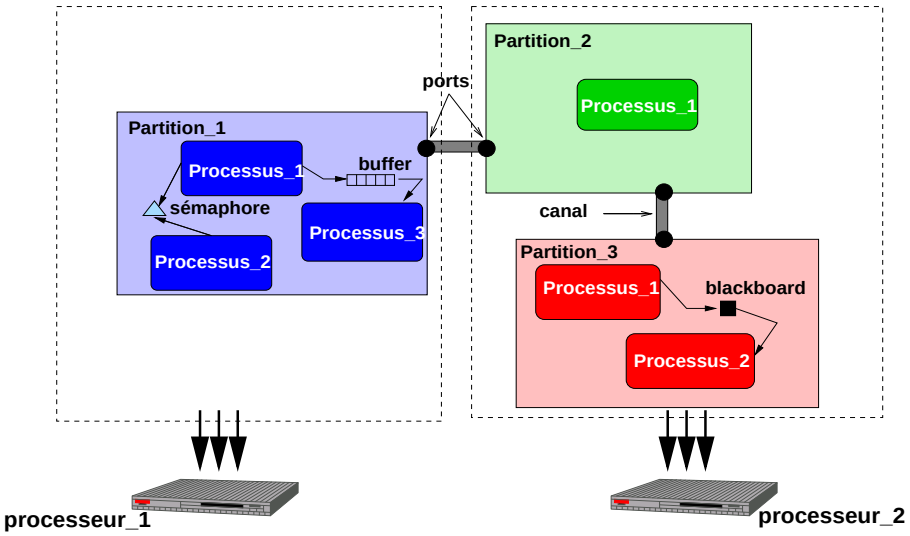


FIGURE 7.6 – Exemple d'application suivant une architecture IMA.

7.2.2 Les processus

Ce sont les entités élémentaires d'exécution. Les processus coopèrent pour accomplir la fonction affectée à la partition au sein de laquelle ils s'exécutent [5]. Comme pour la partition, un processus est caractérisé par diverses informations parmi lesquelles des attributs, un programme exécutable et un pointeur sur une zone de pile.

Attributs. Ici aussi, on distingue des attributs fixes et variables.
Attributs fixes :

1. **Nom** (chaîne de caractères) - Identificateur unique à chaque processus,
2. **Point d'entrée** (type dépendant de la plate-forme) - Point d'entrée pour l'exécution du processus,
3. **Taille de pile** (type numérique) - Taille de la pile associée au processus,
4. **Priorité de base** (type numérique) - Priorité à laquelle le processus est initialisé ;
5. **Temps alloué** (type numérique, dépendant de l'horloge de la plate-forme de mise en œuvre) - Durée de temps à ne pas excéder pendant l'exécution d'un processus. Cette information permet de déterminer la date d'échéance d'un processus ;
6. **Période** (même type que le temps alloué) - Période d'activation pour les processus périodiques. Autrement, cet attribut a une valeur constante particulière pour les processus apériodiques (ce qui permet de les distinguer des processus périodiques),
7. **Échéance** - Nature de l'échéance d'un processus : *hard* ou *soft*.

Attributs variables :

1. **Priorité courante** (type numérique) - Priorité courante du processus,
2. **Date d'échéance** (même type que la période) - Date à laquelle le processus doit avoir accompli sa mission ;
3. **État** - État courant du processus. Quatre états sont définis : *ready* (éligible pour l'accès au processeur), *running* (en cours d'exécution), *waiting* (bloqué en attente d'une ressource ou d'un événement) et *dormant* (inéligible pour recevoir le processeur).

Services	Description de la requête engendrée
get_process_id	récupération de l'identificateur d'un processus
get_process_status	récupération du statut courant d'un processus
create_process	création et initialisation d'un processus
set_priority	mise à jour de la priorité d'un processus
suspend_self	suspension d'un processus par lui-même
suspend	suspension d'un processus par un autre processus
resume	réactivation d'un processus suspendu
stop_self	arrêt d'un processus par lui-même
stop	arrêt d'un processus par un autre processus
start	activation d'un processus
lock_preemption	augmentation du niveau de verrouillage
unlock_preemption	diminution du niveau de verrouillage

TABLE 7.3 – Services APEX pour la gestion des processus.

Gestion. C'est le *partition-level OS* qui a la charge de la gestion des processus. Il doit prendre en compte la périodicité des processus pour ce qui concerne leur activation. Il se base sur la priorité et l'état des processus d'une partition pour les ordonnancer. D'autre part, grâce au niveau de verrouillage de chaque partition, la préemption peut être autorisée ou non (certaines portions de code associées aux processus peuvent nécessiter une exécution ininterruptible). Une partition doit pouvoir réinitialiser tout processus à tout moment (la détection de certaines erreurs dans le système peut requérir une réinitialisation ou même la terminaison du processus actif). Enfin, des moyens sont fournis afin de garantir l'exclusion mutuelle lors des accès aux ressources partagées par les processus. Nous résumons ci-après les principales caractéristiques du modèle d'ordonnancement au sein d'une partition :

- L'unité d'ordonnancement est le processus. L'accès aux ressources (par exemple, le processeur) par les processus se fait de manière exclusive ;
- Contrairement aux partitions, chaque processus a une priorité. Celle-ci est représentée par des valeurs positives. Une priorité est plus forte qu'une autre lorsque sa valeur est supérieure à celle de la seconde ;
- L'algorithme d'ordonnancement est préemptif. L'ordonnanceur se base sur la priorité et l'état courant des processus rangés dans une file d'attente : c'est le processus ayant la priorité la plus forte et se trouvant dans l'état *ready* qui se voit attribuer le processeur. Dans le cas où plusieurs processus ont la même priorité, c'est le plus ancien dans la file qui est sélectionné. Ainsi, le processus élu détiendra les ressources jusqu'à la prochaine demande d'ordonnancement. Des réordonnancements peuvent cependant survenir sur une demande explicite d'un processus, ou bien sur l'occurrence d'un événement interne à la partition ;
- Le système offre les moyens de gérer les processus périodiques et apériodiques (cette gestion est uniforme ; les structures utilisées pour décrire les attributs des processus sont identiques) ;
- Enfin, les processus d'une même partition partagent l'intégralité des ressources allouées à celle-ci.

Les services APEX servant à la gestion des processus sont donnés dans la table TAB. 7.3. Le *statut* d'un processus est composé de la valeur courante de ses attributs.

5. Un processus n'est visible qu'à l'intérieur de la partition où il est défini.

Services	Description de la requête engendrée
create_buffer	création et initialisation d'un *buffer*
send_buffer	envoi d'un message dans un *buffer*
receive_buffer	réception d'un message d'un *buffer*
get_buffer_id	récupération de l'identificateur d'un *buffer*
get_buffer_status	récupération du statut courant d'un *buffer*
create_blackboard	création et initialisation d'un *blackboard*
display_blackboard	écriture d'un message dans un *blackboard*
read_blackboard	lecture d'un message dans un *blackboard*
clear_blackboard	nettoyage d'un *blackboard*
get_blackboard_id	récupération de l'identificateur d'un *blackboard*
get_blackboard_status	récupération du statut courant d'un *blackboard*
create_event	création et initialisation d'un *event*
set_event	positionnement d'un *event*
reset_event	réinitialisation d'un *event*
wait_event	attente d'un *event*
get_event_id	récupération de l'identificateur d'un *event*
get_event_status	récupération du statut courant d'un *event*
create_semaphore	création et initialisation d'un sémaphore
wait_semaphore	attente d'un sémaphore
signal_semaphore	relâchement d'un sémaphore
get_semaphore_id	récupération de l'identificateur d'un sémaphore
get_semaphore_status	récupération du statut courant d'un sémaphore

TABLE 7.4 – Services APEX pour la communication et la synchronisation au sein d'une partition.

Communications et synchronisations. On distingue principalement quatre mécanismes pour réaliser les communications et les synchronisations entre processus au sein d'une partition.

Trois mécanismes de communication sont utilisés. Le *buffer* offre le moyen d'écrire et de lire des messages rangés dans une file selon une politique FIFO (*First In First Out*) ; la file a une taille bornée. L'*event* permet de notifier l'occurrence d'un événement à des processus qui seraient bloqués en attente de celui-ci. Enfin, le dernier mécanisme est le *blackboard*. Celui-ci permet d'écrire et de lire des messages dans un tampon à une place. Lorsqu'un message vient d'être écrit, il y reste jusqu'à ce qu'il soit écrasé par l'écriture d'un nouveau message ou bien que le *blackboard* soit "nettoyé". Le mécanisme de synchronisation utilisé par les processus est un *sémaphore* à compteur. La table TAB. 7.4 donne les services APEX utilisés pour les communications et les synchronisations au sein d'une partition. Les informations constituant le statut des différents mécanismes sont :

- *buffer* : le nombre courant de messages stockés dans le *buffer*, la capacité maximale de celui-ci, la taille des messages qu'on peut y stocker et le nombre courant de processus bloqués en attente d'envoi ou de réception de messages dans le *buffer* ;
- *blackboard* : l'état courant (vide ou non), la taille des messages qu'on peut y stocker et le nombre courant de processus bloqués en attente de lecture de messages dans le *blackboard* ;
- *event* : l'état courant et le nombre de processus actuellement bloqués sur l'*event* ;
- *sémaphore* : la valeur courante, la valeur maximale et le nombre courant de processus bloqués sur le sémaphore.

Services	Description de la requête engendrée
timed_wait	suspension pour une quantité de temps donnée
periodic_wait	suspension d'un processus périodique jusqu'à sa date d'activation suivante
get_time	récupération du temps affiché par l'horloge du système
replenish	décalage de la date d'échéance d'un processus à une date ultérieure

TABLE 7.5 – Services APEX pour la gestion du temps.

Services	Description de la requête engendrée
report_application_message	transmission d'un message à *health monitor* par une partition
create_error_handler	création du processus *error handler*
get_error_status	récupération d'informations concernant une erreur par *error handler*
raise_application_error	activation de *error handler* par une partition sur détection d'une erreur

TABLE 7.6 – Services APEX de gestion d'erreurs.

7.2.3 La gestion du temps

En plus des services évoqués ci-dessus, l'interface APEX offre aussi des services permettant la gestion du temps. Grâce à ces services, la date d'échéance d'un processus peut être modifiée si celui-ci n'est plus en mesure de respecter celle qui a été initialement spécifiée (par exemple, l'invocation du service *replenish* par un processus apériodique provoque un décalage de sa date d'échéance de la valeur de l'attribut "temps alloué" par rapport à l'instant courant). Un processus peut aussi suspendre son exécution pour une certaine quantité de temps à l'aide de ces services (par exemple, l'appel du service *periodic_wait* par un processus périodique entraîne sa suspension jusqu'à sa prochaine date d'activation, et la mise à jour de sa date d'échéance suivante). L'ensemble de ces services est donné dans la table TAB. 7.5.

7.2.4 La gestion des erreurs

Une table dite *health monitor* indique les différents traitements lors d'une détection d'erreur pendant l'exécution de l'application (arrêt d'une partition, redémarrage de celle-ci en mode *cold start* ou *warm start*). Il revient à un processus spécial d'exécuter ces traitements. Ce processus est appelé *error handler*. La gestion des erreurs au cours de l'exécution des applications est réalisée à l'aide des services APEX mentionnés dans la table TAB. 7.6.

7.3 Résumé

La conception des systèmes avioniques a reposé pendant plusieurs années sur le modèle d'*architecture fédérée*, où chaque fonction dispose de ses propres ressources de calcul. L'avantage d'une telle architecture réside dans le faible risque de propagation d'erreurs d'une fonction à l'autre. Cependant, vu l'espace limité dont on dispose dans un avion, des problèmes peuvent rapidement se poser quant à l'installation à bord des calculateurs requis par les fonctions. Le coût engendré par la maintenance vient s'y ajouter. Le modèle d'*architecture modulaire intégrée* (IMA) a été proposé dans le but de pallier ces inconvénients du modèle fédéré. Plusieurs fonctions peuvent ainsi partager des ressources de calcul communes sans interférences sur le fonctionnement des unes par les autres. Cela est garanti par un *mécanisme de partitionnement* défini dans la norme ARINC. Les fonctions sont réparties au sein de *modules*, eux-mêmes composés de *partitions*, elles-mêmes constituées de *processus*.

Le diagramme UML de la FIG. 7.7 donne une vision globale des différents niveaux de granularité pour une description IMA (*module*, *partition*, *processus*). Dans ce diagramme, les associations dont l'une des extrémités est représentée par un losange plein dénotent des *compositions*. C'est-à-dire qu'un objet de la classe qui n'est pas repérée par le losange (appelé aussi "composant") est *subordonné* pendant toute sa durée de vie à un objet de l'autre classe (appelé "composé"). C'est donc une relation de contenance forte. Tous les liens portent des cardinalités. Le symbole "*" peut être interprété comme "un certain nombre" : lorsqu'il est utilisé tout seul (exemple : dans l'association entre les classes *Module* et *Partition*.), il peut valoir zéro ; par contre, dans "1..*" il est au moins égal à un.

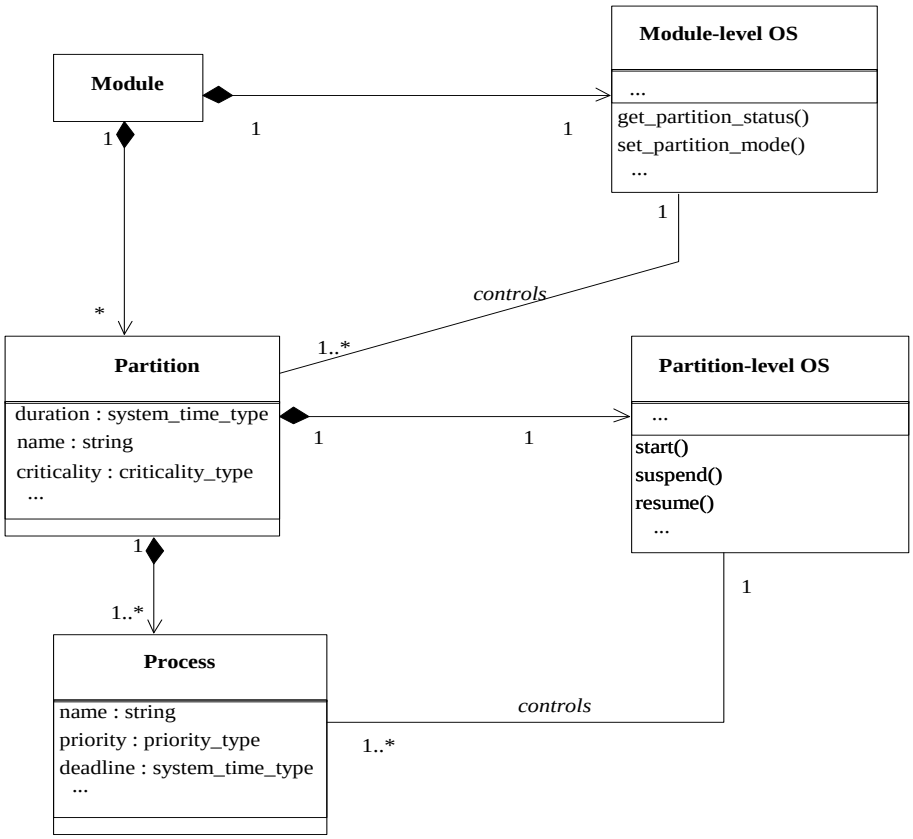


FIGURE 7.7 – Différents niveaux de granularité dans une description IMA.

Le document [8] présente une description complète des services permettant de gérer ces objets à travers l'interface APEX. Les modèles SIGNAL définis par la suite reposent sur les spécifications issues de ce document.

Chapitre 8

Description polychrone d'applications dans l'avionique

Nous présentons ici la modélisation des notions introduites dans le chapitre précédent. Il s'agit plus précisément des notions dont nous avons besoin pour décrire des applications avioniques suivant l'approche IMA. Le travail ainsi exposé a été introduit dans [91], puis développé dans [94]. L'on trouvera une version détaillée dans le rapport de recherche [92].

La modélisation d'une application suivant une architecture de type IMA requiert au préalable la description des composants de base (partitions, processus et services APEX) dont l'assemblage fournit le modèle de l'application. En particulier, nous partons du fait que le modèle exécutable d'une partition peut être représenté comme sur la FIG. 8.1. Trois aspects sont principalement distingués dans le modèle :

1. les *entités d'exécution*, représentées par les processus [1] ;
2. les *échanges* entre ces entités (communications et synchronisations entre processus), réalisés au moyen de services APEX ;
3. l'*exécution concurrente* des processus au sein de la partition, placée sous le contrôle de l'exécutif [2] (i.e., le *partition-level OS*).

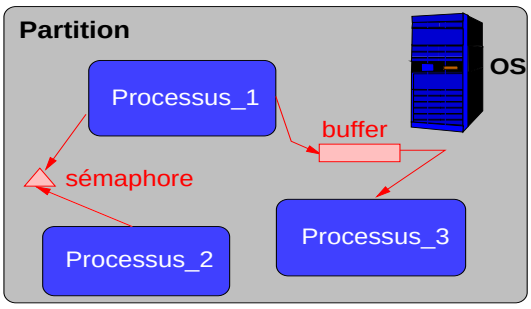


FIGURE 8.1 – Modèle Exécutable représentant une partition.

1. À ce stade du document, il conviendra de ne pas confondre "processus" défini comme entité d'exécution dans ARINC, et "processus" défini comme programme dans SIGNAL.

2. Il est important de distinguer clairement l'exécutif temps réel du système d'exploitation temps réel malgré l'appellation "*partition-level OS*" ici, qui désigne un exécutif constitué uniquement des fonctionnalités nécessaires à la gestion des processus ainsi que leurs communications au sein d'une partition. L'exécutif ne représente donc qu'une partie du système d'exploitation (cf. chapitre 2 - section 2.2.3.2).

Nous définissons les composants SIGNAL permettant de décrire chacun des aspects mentionnés ci-dessus. Nous débutons par la description des fonctionnalités de l'exécutif temps réel : la section 8.1 est consacrée à la modélisation des services APEX ; la section 8.2 quant à elle, concerne les autres fonctionnalités de l'exécutif, non spécifiées par la norme ARINC 653. Un modèle est ensuite proposé pour les processus dans la section 8.3. Enfin, l'approche adoptée est discutée dans la section 8.4.

8.1 Modélisation des services APEX à l'aide de SIGNAL

Pour illustrer la démarche adoptée pour modéliser les services APEX, nous considérons le cas particulier d'un service de l'interface. Nous commençons par présenter la façon dont le modèle SIGNAL correspondant est obtenu à partir de la spécification informelle donnée dans [8]. La description des modèles des autres services repose sur le même principe. Le service considéré ici s'appelle *read_blackboard* ; il permet de lire un message dans un *blackboard*.

8.1.1 Spécification informelle

La FIG. 8.2 illustre une spécification algorithmique de *read_blackboard* d'après la norme ARINC 653 [8]. La description des autres services APEX est donnée sous une forme analogue dans la norme.

Les paramètres d'interface du service sont clairement spécifiés. Deux entrées sont distinguées : l'*identificateur* du *blackboard* représenté par *board_ID* et une quantité de temps associée au *timeout*, dénotée par *timeout*. Cette dernière entrée indique, lors d'une requête utilisant le service, la durée d'attente du processus appelant dans le cas où il n'y aurait aucun message à lire. Parmi les paramètres de sortie, on note l'*adresse*[3] et la *taille* du message lu, respectivement représentées par *message* et *length*. De plus, un paramètre dénoté par *return_code*, qui indique le diagnostic d'une requête, est renvoyé. Les valeurs possibles de ce paramètre dans le service *read_blackboard* sont *INVALID_PARAM*, *NOT_AVAILABLE*, *INVALID_MODE*, *TIMED_OUT* et *NO_ERROR* (une description détaillée de tous les types considérés pour nos modèles se trouve dans [92]).

L'algorithme du service est donné dans un pseudo langage impératif de façon à faciliter la compréhension de son fonctionnement. La principale difficulté à ce stade réside dans le passage d'une description impérative telle que celle de la FIG. 8.2 à une spécification basée sur le style flot de données synchrone de SIGNAL. Pour cela, nous adoptons une démarche descendante pour définir le modèle de *read_blackboard*. Nous partons d'abord d'une spécification SIGNAL très abstraite du service (i.e., un modèle "boîte noire"), qui sera détaillée par la suite (i.e. par dérivations successives de modèles "boîtes grises"). En procédant ainsi, nous obtenons des descriptions successives jusqu'à ce que le niveau de détail souhaité soit atteint (par exemple, jusqu'à ce que l'on n'obtienne que des descriptions "boîtes blanches", i.e., tout le fonctionnement interne entièrement explicité).

8.1.2 Spécification à l'aide de SIGNAL

Dans cette section, nous présentons les différentes étapes suivies pour décrire un service APEX en SIGNAL. Après avoir introduit un modèle initial correspondant à *read_blackboard* (cf.

3. Appelée aussi *zone*.

```
Service read_blackboard IN : board_ID ; timeout ; OUT : message ; length ; return_code ;

  if les entrées sont invalides (c-à-d, l'identificateur du blackboard n'est pas connu du
  système et/ou la valeur du time-out n'est pas "licite") then
     return INVALID_PARAM ;
  else if il y a un message disponible dans le blackboard spécifié then
     récupérer celui-ci ; return NO_ERROR ;
  else if la valeur du time-out est nulle then
     return NOT_AVAILABLE ;
  else if la préemption n'est pas autorisée dans la partition courante, ou bien le processus
  appelant est le gestionnaire d'erreurs then
     return INVALID_MODE ;
  else
     mettre le processus appelant dans l'état "waiting" ;
     if la valeur du time-out n'est pas infinie then
        déclencher un compteur avec comme valeur initiale celle du time-out ;
     end if ;
     demander un ré-ordonnancement (le processus appelant est ainsi bloqué ; il ne re-
     tournera à l'état "ready" que lorsqu'un autre processus aura fait un appel au service
     display_blackboard, ou sur expiration du time-out) ;
     if expiration du time-out then
        return TIMED_OUT ;
     else
        récupérer le message le plus récent dans le blackboard ; return NO_ERROR ;
     end if
  end if
```

FIGURE 8.2 – Spécification informelle du service *read_blackboard* selon [8].

section 8.1.2.1), nous raffinons progressivement celui-ci jusqu'à obtenir un modèle final (cf. section 8.1.2.2).

8.1.2.1 Un premier modèle abstrait

Nous avons vu que dans une démarche de conception descendante, on commence par une description abstraite d'un composant, qui est raffinée par la suite. Dans un premier temps, cette description peut se limiter à la spécification des propriétés d'interface. Ce niveau de détail suffit par exemple pour vérifier la conformité du composant ainsi défini, lors de son intégration au sein d'un système décrit dans le même formalisme.

```
(| (| { {board_ID, timeout} --> return_code } when C_return_code      (d.1)
   | { {board_ID, timeout} --> {message, length} }
            when (return_code = #NO_ERROR)                           (d.2)
   |)
 | (| board_ID ^= timeout ^= C_return_code                           (s.1)
   | return_code ^= when C_return_code                               (s.2)
   | message ^= length ^= when (return_code = #NO_ERROR)             (s.3)
   |)
 |) / boolean C_return_code
```

FIGURE 8.3 – Description abstraite du service *read_blackboard*.

Le processus SIGNAL de la FIG. 8.3 spécifie des propriétés d'interface du service *read_blackboard*. Il précise notamment la condition sous laquelle un message est renvoyé sur une requête utilisant le service (i.e., **quand** un message peut être récupéré suite à une demande de lecture). Par contre, le processus ne décrit pas exactement **comment** le message est récupéré.

Si on examine les propriétés exprimées dans le processus, `(s.2)` décrit les instants de présence d'un code retour. Ce dernier est reçu uniquement lorsqu'un certain signal local `C_return_code` de type booléen, porte la valeur *vrai*[4]. Ce booléen apparaît pour l'instant dans l'abstraction comme un signal local dont la définition sera explicitée lors des raffinements. Cela peut être interprété comme "il existe un signal `C_return_code` tel que les propriétés exprimées dans l'abstraction sont satisfaites". Dans `(s.1)`, on spécifie que ce signal est synchrone avec les entrées du service (i.e. chaque fois qu'il y aura une demande de lecture, `C_return_code` indiquera si un code retour doit être renvoyé ou non). La propriété `(s.3)` exprime le fait que les messages ne sont récupérés que lorsque le code retour vaut *NO_ERROR*. Les lignes `(d.1)` et `(d.2)` donnent des relations de dépendance entre les entrées et les sorties. En SIGNAL, la notation `x --> y` exprime une relation de dépendance entre deux signaux `x` et `y` à l'intérieur d'un instant logique (on dit que `x` *précède* `y` à cet instant). Par exemple dans la propriété `(d.2)`,

4. Lors de l'appel du service *read_blackboard*, la réception d'un code retour n'est pas forcément immédiate. Par exemple, lorsque le *blackboard* est vide et que la valeur du paramètre d'entrée *timeout* (cf. interface du service sur la FIG. 8.2) n'est pas l'infini (identifié dans [8] par la constante *INFINITE_TIME_VALUE*), le processus appelant doit attendre : soit l'écriture d'un message dans le *blackboard*, soit l'expiration du compteur de temps déclenché (le code retour prend alors la valeur *TIMED_OUT*). Le booléen `C_return_code` est introduit ici pour représenter les instants logiques auxquels un code retour est produit immédiatement, suite à un appel. Ce signal sera défini ultérieurement lorsqu'on affinera la description du service.

les signaux `timeout` et `board_ID` précèdent les signaux de sortie `message` et `length` lorsque le diagnostic de la requête est `NO_ERROR`.

Si la spécification informelle de *read_blackboard* (cf. FIG. 8.2) semble assez facile à comprendre, elle peut néanmoins comporter quelques ambiguïtés qui ne sont pas perceptibles de prime abord. C'est d'ailleurs un inconvénient des spécifications données dans [8]. Celles-ci ne sont pas toujours précises. Elles nécessitent donc un choix de la part du développeur. En l'occurrence, il existe deux mises en œuvre possibles du service *read_blackboard*. Elles sont dues aux interprétations possibles concernant la valeur d'un message récupéré par un processus qui se trouvait précédemment bloqué en attente de lire un message sur le *blackboard* (cf. dernier "else" sur la FIG. 8.2) :

1. Certaines implémentations peuvent considérer que ce message est *celui qui vient d'être écrit* dans le *blackboard* par le processus qui provoque le déblocage du processus en attente.
2. Pour d'autres implémentations, il s'agit plutôt du message présent dans le *blackboard* au moment où le processus débloqué reprend son exécution. Ce qui veut dire qu'il ne recevra pas forcément le message du processus à l'origine de sa libération, puisqu'entre temps il peut y avoir eu d'autres messages dans le *blackboard*.

Ces ambiguïtés justifient la nécessité d'une démarche descendante, où les descriptions abstraites permettent d'avoir des spécifications assez générales qu'on pourra raffiner selon l'interprétation voulue. Comme nous pouvons le remarquer, le programme SIGNAL de la FIG. 8.3 englobe les deux interprétations du service *read_blackboard*. Il ne précise pas la valeur d'un message reçu lors d'une lecture. Il indique seulement la condition sous laquelle ce message peut être lu. En raffinant le programme par la suite, on pourra alors faire un choix d'interprétation.

8.1.2.2 Vers un raffinement du modèle initial

Nous procédons à présent au raffinement du premier modèle proposé pour le service *read_blackboard*. La nouvelle description prend en compte la définition des messages lus à la suite de requêtes utilisant ce service. Avant de présenter le modèle SIGNAL correspondant, nous étudions les comportements induits par la spécification informelle afin de faciliter une description flot de données plus détaillée.

Situations typiques. Afin de mieux cerner les aspects à prendre en compte dans une modélisation SIGNAL de l'algorithme informel donné sur la FIG. 8.2, on considère une exécution concurrente de deux processus P1 and P2 au sein d'une même partition. On suppose que P1 a une priorité plus forte que P2. Les deux processus communiquent à travers un *blackboard* qu'on considère vide pour le moment. La FIG. 8.4 montre deux scénarios d'exécution possibles, appelés *situation A* et *situation B*.

Dans les deux cas, P1 tente de lire le *blackboard* avant P2. Il voit son exécution suspendue puisqu'aucun message n'est encore disponible dans le *blackboard*. Cela provoque un ré-ordonnancement qui va entraîner une activation du processus P2. Le processus P1 doit dorénavant attendre soit une notification d'expiration de son *time-out*[5] (*situation A* sur la figure), soit la disponibilité d'un message dans le *blackboard* (*situation B*).

Analyse des situations en vue du raffinement. En examinant ce qui se passe du point de vue temporel, on voit que dans les deux situations l'intervalle de temps concerné par l'exécution

5. Dans la spécification informelle, cela correspond à l'émission du code retour *TIMED_OUT*.

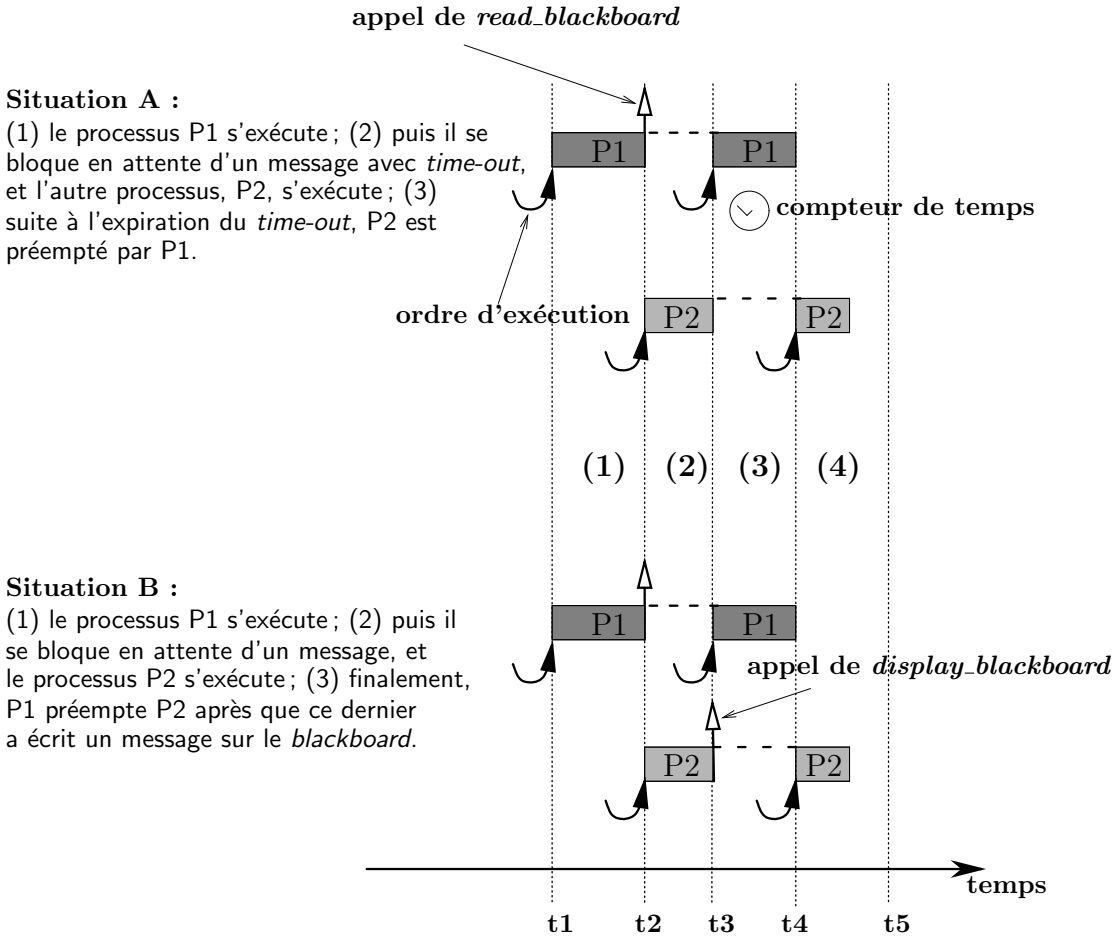


FIGURE 8.4 – Exécution de deux processus P1 et P2 au sein d'une même partition.

du service *read_blackboard* est [`t2, t3`]. Il contient partiellement des exécutions de P1 et P2. D'autre part, on rappelle qu'au sein d'une partition, un processus et un seul peut s'exécuter à tout instant. Du point de vue synchrone, cela va être représenté par le fait qu'à chaque instant logique, au plus un modèle de processus ARINC est activé au sein d'une partition.

Partant de cette observation, les actions décrites dans le service *read_blackboard* doivent donc être découpées en différentes parties puisqu'elles ne peuvent être exécutées entièrement pendant la seule exécution de P1. On distingue ainsi deux types d'actions :

- les actions qui sont exécutées de façon instantanée lorsque le processus appelant est actif (exemple : contrôler la validité des paramètres d'entrée, déclencher un compteur de temps), appelées *actions locales* ;
- les actions accomplies durant la suspension du processus appelant (exemple : dans la *situation A*, ces actions consistent à décrémenter le compteur de temps déclenché et à notifier son expiration), appelées *actions globales*.

En fait, les actions globales sont sous le contrôle du noyau de l'exécutif qui est responsable de la gestion des processus, ainsi que des ressources partagées (exemples : *blackboard*, *buffer*) dans la partition. La mise à jour des compteurs de temps est effectuée au sein de l'exécutif. Lorsque la valeur d'un compteur déclenché par un processus suspendu arrive en butée, un événement correspondant à la notification de l'expiration du *time-out* est alors produit par l'exécutif. Dans le service *read_blackboard*, cela correspond au cas où le code retour vaut *TIMED_OUT*. Ce type d'information est calculé pendant la suspension du processus à l'origine de l'appel du service. Il est donc nécessaire de distinguer une telle information de celles qui sont déterminées lors de l'exécution de ce processus (comme les actions que nous qualifions de locales dans un service). Il en découle le principe suivant :

Principe 1 *La modélisation d'un service APEX consiste en un découpage temporel suivant lequel on distingue deux types d'actions : les actions locales, effectuées lorsque le processus appelant est encore actif, et les actions globales, effectuées pendant la suspension de celui-ci.*

Modélisation des actions locales dans un service APEX. Nous présentons ici la modélisation des actions locales d'un service APEX. Les actions globales sont abordées dans la section suivante. Elles sont traitées au sein d'autres services du noyau.

On considère la *situation B* de la FIG. 8.4, où le processus P1 reprend son exécution après écriture d'un message dans le *blackboard* par P2. Les actions locales sont exécutées par P1 exactement aux instants `t2` (par exemple, le contrôle de la validité des paramètres d'entrée, le déclenchement du compteur de temps) et `t3` (par exemple, la récupération du dernier message disponible dans le *blackboard*). Soient `L` et `L'`, les sous-ensembles d'actions locales exécutées respectivement à ces instants. On les regroupe au sein d'un même processus SIGNAL qui représente un modèle partiel du service *read_blackboard*. D'autre part, puisque `L` et `L'` ne sont pas exécutés en même temps, il faut définir les conditions de sélection du sous-ensemble approprié, à chaque activation du modèle global. Cela est décrit facilement au moyen d'une variable d'état interne au processus SIGNAL correspondant. Celle-ci sera positionnée en fonction des actions locales à exécuter. Typiquement, cette variable d'état est représentée par un signal booléen. C'est ce que le modèle de la figure 8.5 exprime, le signal booléen `blocked` (initialement à *faux*) jouant le rôle de cette variable d'état.

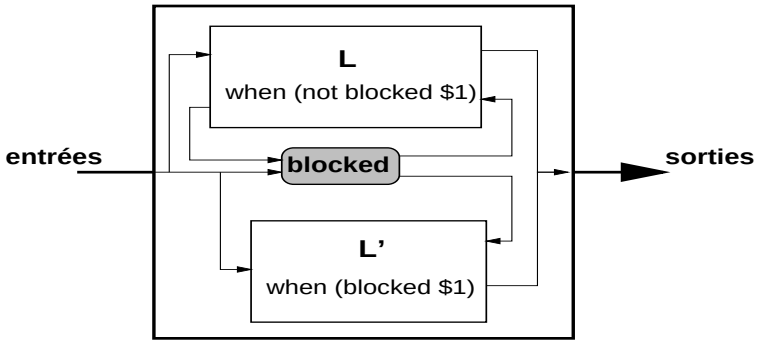


FIGURE 8.5 – Modèle des actions locales dans un service bloquant.

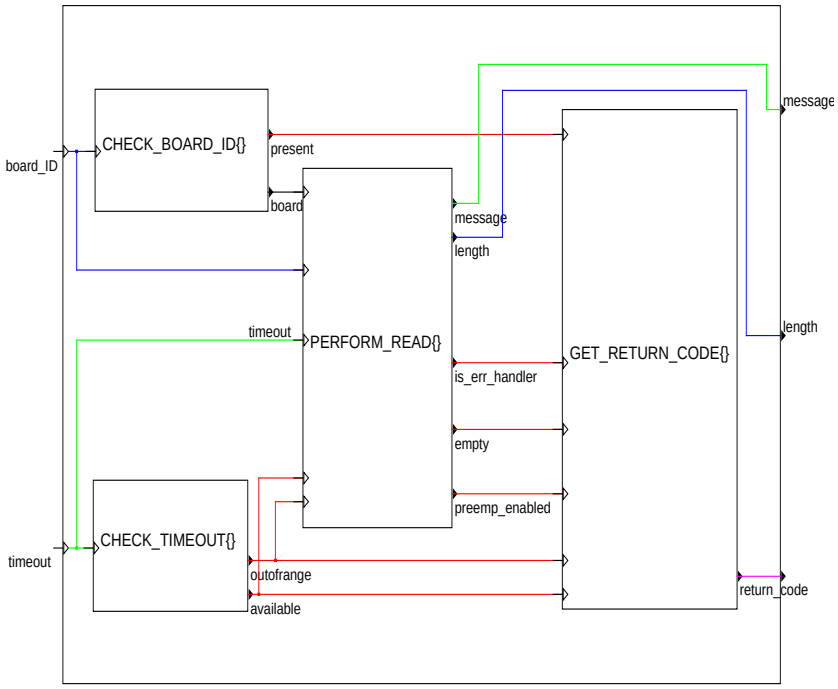
Le sous-ensemble `L` est exécuté quand l'appelant n'était pas bloqué précédemment sur un appel du service (dénoté par la condition `when (not blocked $ 1)` sur la figure). Suite à cet appel, si le *blackboard* est vide, le booléen `blocked` est mis à *vrai*. Cela est réalisé grâce à une information provenant de `L` (représentée par la flèche allant de `L` vers `blocked` sur le schéma). Lorsque la variable d'état portait précédemment la valeur *vrai* (i.e. l'appelant est resté bloqué), c'est `L'` qui est exécuté et le booléen `blocked` devient *faux*.

Application au service *read_blackboard*. Dans le processus SIGNAL décrit sur la FIG. 8.6, on considère que le message lu par un processus débloqué est celui qui est présent au moment où il reprend son exécution (i.e., la deuxième interprétation de *read_blackboard* - voir la discussion à ce sujet dans la section 8.1.2.1). Ce modèle reflète les actions locales du service *read_blackboard* (i.e. `L` $\cup$ `L'`). On y distingue principalement quatre sous-processus. Cette décomposition respecte le *deuxième principe de conception*[6] mentionné dans la méthodologie de programmation à l'aide de SIGNAL [98]. Les sous-processus `CHECK_BOARD_ID` et `CHECK_TIMEOUT` vérifient la validité des paramètres d'entrée `board_ID` et `timeout`. Lorsque ces derniers sont valides, `PERFORM_READ` tente alors de lire un message dans le *blackboard* spécifié. Après quoi, il renvoie le dernier message disponible dans le *blackboard* (son adresse mémoire et sa taille sont respectivement spécifiées par `message` et `length`) et transmet toutes les informations nécessaires à `GET_RETURN_CODE` qui doit déterminer le diagnostic de la requête.

Dans cette nouvelle description du service qui est plus détaillée que la précédente, les sous-processus dégagés mettent clairement en évidence la structure globale du modèle associé. Chacun d'entre eux est caractérisé ici par ses signaux d'interface. De plus, les relations d'horloges entre ces signaux sont explicitées. Nous pouvons notamment remarquer que le signal local `C_return_code`, introduit dans le modèle initial, est ici défini. Ainsi, lors de l'invocation de ce modèle, un code retour est immédiatement produit dans l'une des situations suivantes :

- l'identificateur du *blackboard* et/ou la valeur du *timeout* sont invalides (déterminé à l'aide respectivement des signaux booléens `present` et `outofrange`) ;
- le *blackboard* est vide (indiqué par le booléen `empty`) et la valeur initiale du *timeout* est nulle (détecté lorsque le booléen `available` porte la valeur *false*) ;
- le *blackboard* est vide, et le ré-ordonnancement n'est pas autorisé ou le processus appelant est celui qui est chargé de la gestion des erreurs (respectivement déterminé à l'aide des booléens `preemp_enabled` et `is_err_handler`) ;
- le *blackboard* n'est pas vide.

6. *Un processus est décomposé en composants fonctionnellement cohérents.*




```
(| board_ID ^= timeout ^= present ^= outofrange ^= available ^= C_return_code
 | board ^= empty ^= when present
 | message ^= length ^= when (not empty)
 | is_err_handler ^= when empty when available
 | preemp_enabled ^= when (not is_err_handler)
 | C_return_code := when ((not present) or outofrange)) default
                    when empty when (not available)) default
                    when ((not preemp_enabled) default is_err_handler)) default
                    when (not empty) default
                    false
 | return_code ^= when C_return_code
 |) / boolean C_return_code
```

FIGURE 8.6 – Raffinement de *read_blackboard* et relations d'horloges entre les signaux mis en évidence.

Par exemple, lorsque les signaux `empty` et `preemp_enabled` portent respectivement les valeurs *vrai* et *faux*, `GET_RETURN_CODE` émet *INVALID_MODE* comme valeur du paramètre `return_code` (cela signifie que l'appelant du service doit attendre qu'un message soit disponible dans le *blackboard*, et aucun autre processus ARINC ne peut s'exécuter en attendant parce que le mode opératoire courant n'autorise pas la préemption). Dans le cas où les entrées sont invalides (exemple : `board_ID` est un identificateur inconnu du système, ou `timeout` n'est pas "licite"), des informations sont toujours envoyées à `GET_RETURN_CODE` par `CHECK_BOARD_ID` et `CHECK_TIMEOUT` pour élaborer le code retour approprié qui est *INVALID_PARAM*. On trouvera la définition précise du signal `return_code` dans le modèle SIGNAL complet du service *read_blackboard*, donné en annexe A.

Chacun des sous-processus SIGNAL distingués sur la FIG. 8.6 est à son tour explicité de façon analogue au modèle global du service. La description finale résultant de ce processus de raffinement est donnée sous forme textuelle en annexe A.

Remarque 1 *Le modèle de la* FIG. *8.6 n'est pas une transposition directe du schéma général de la* FIG. *8.5 à cause de l'interprétation considérée pour le service read_blackboard. Celle-ci impose à un processus précédemment en attente de message de réitérer sa requête lorsqu'il redevient actif. Donc, les ensembles* `L` *et* `L'` *qui dénotent respectivement les actions à exécuter avant et après la suspension du processus appelant représentent les mêmes actions. Le schéma de la* FIG. *8.5 s'applique tel quel lorsque les actions distinguées dans* `L` *et* `L'` *ne sont pas identiques. Par exemple, pour l'autre interprétation de read_blackboard, où il n'y a pas besoin de relire nécessairement le blackboard, le processus récupère simplement une copie du message à l'origine de sa libération. Dans ce cas,* `L` *et* `L'` *ne sont pas les mêmes. D'autres exemples de services définis à l'aide du schéma général sont receive_buffer, wait_event et wait_semaphore.* □

*
* *

La modélisation des autres services APEX suit la même démarche que celle adoptée pour le service *read_blackboard*. Par la suite, nous verrons que la décomposition des spécifications informelles en actions locales et globales convient parfaitement pour le modèle d'exécution adopté pour les processus ARINC (cf. section 8.3).

À l'aide des services APEX, nous pouvons à présent décrire la gestion des processus ARINC ainsi que les communications et synchronisations entre ceux-ci. D'autre part, ces services comprennent quelques primitives de gestion du temps. Ces services (énumérés dans le chapitre 7) sont cependant insuffisants pour modéliser entièrement l'exécutif temps réel au sein d'une partition. Parmi les fonctionnalités supplémentaires requises, il faut aussi disposer de modules chargés des tâches suivantes :

- module de gestion des informations sur tous les objets d'une partition (processus, mécanismes de communication et synchronisation, compteurs de temps), c'est-à-dire, l'ensemble des *descripteurs* ;
- module de choix des processus à exécuter dans une partition ; le cœur de ce module est l'*ordonnanceur* ;
- module de sauvegarde et restitution du contexte[7] des processus d'une partition, appelé *dispatcher*.

7. Cet aspect ne sera pas traité explicitement au niveau où nous nous plaçons dans la modélisation.

La section suivante présente la modélisation de ces modules, c'est-à-dire des primitives "non APEX" nécessaires à la description complète de l'exécutif au sein d'une partition.

8.2 Modélisation des autres fonctionnalités de l'exécutif

Nous présentons la description des services de gestion des informations sur les objets d'une partition : descripteurs de processus et de mécanismes de communication et synchronisation, ainsi que compteurs de temps. Nous modélisons aussi les services utilisés pour l'ordonnancement des processus et la mise à jour des compteurs de temps.

8.2.1 Gestion des descripteurs

Les descripteurs d'objets sont des structures de données globales reflétant ces objets, créées au sein de l'exécutif. Ils sont constitués d'attributs à travers lesquels sont obtenues les informations descriptives et dynamiques qui permettent de gérer les objets. Nous appelons *managers* les collections de descripteurs sur lesquelles s'appuient les modules de gestion des objets (cf FIG. 8.7). Des opérations typiques offertes par ces modules sont : le test d'existence d'un objet (c'est le cas du service `CHECK_BOARD_ID` utilisé dans le modèle de la FIG. 8.6) ; la récupération de l'état courant d'un objet (nécessaire pour définir des services comme *get_blackboard_status*) ; ou la mise à jour des attributs d'un objet (consistant parfois en des effets de bord sur les structures de données).

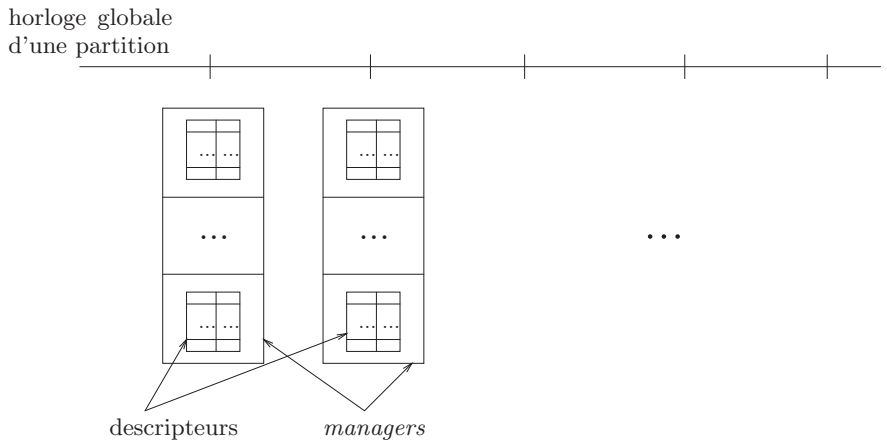


FIGURE 8.7 – Gestion des objets sous forme de descripteurs.

Nous avons construit une première version de la modélisation SIGNAL des services réalisant les opérations mentionnées ci-dessus dans notre bibliothèque de composants. Une seconde version est en cours de mise en œuvre. Dans la version terminée, nous avons utilisé des mécanismes d'abstraction pour décrire une partie des services. Les structures de données ont été définies dans le langage C++. Les méthodes des classes correspondantes mettent alors en œuvre les accès aux attributs. Ainsi, le processus SIGNAL représentant une méthode donnée est défini comme une abstraction boîte noire de celle-ci. La version en cours est une évolution de la première. L'idée de base reste inchangée en ce qui concerne le découpage des structures de données, ainsi que les services d'accès aux structures. Seul le niveau de description SIGNAL change.

```
process CHECK_BOARD_ID =
    ( ? blackboardID_type board_ID;
      ! boolean present;
        APEX_Blackboard_type board_OUT;
    )
    spec (| (| board_ID --> present                                    (d.1)
            | {board_ID --> board_OUT} when present                    (d.2)
            |)
          | (| board_ID ^= present                                     (s.1)
            | board_OUT ^= when present                                (s.2)
            |)
          |)
    pragmas
       CPP_CODE "&o1 = GLOBAL_BLACKBOARD_MANAGER->BlackboardCheckID(&i1, &o2)"
    end pragmas
```

FIGURE 8.8 – Modèle SIGNAL d'un service externe avec *pragma* de liaison au code cible.

Pour illustrer les modèles de la première version, considérons la spécification SIGNAL de la figure 8.8 qui représente le processus SIGNAL correspondant au service `CHECK_BOARD_ID`. Ce service teste la présence d'un *blackboard* dans le manager associé aux *blackboards*. Ce *blackboard* est identifié par le signal d'entrée `board_ID`. En sortie, le booléen `present` indique s'il a été effectivement enregistré dans le *manager*. L'autre sortie, `board_OUT`, représente la structure de données correspondant au *blackboard* si celui-ci est présent.

Dans `CHECK_BOARD_ID`, la contrainte `(s.1)` traduit le fait qu'à chaque invocation de ce service (indiquée par la réception du signal `board_ID`), le service retourne une réponse sous la forme d'un signal booléen `present`. La propriété `(s.2)` exprime que `board_OUT` est présent uniquement aux instants où `present` porte la valeur *vrai*. Le signal `board_OUT` dénote la structure de données correspondant au *blackboard* recherché. Les dépendances entre les entrées/sorties sont décrites à travers `(d.1)` et `(d.2)`. Les *pragmas* sont introduits par le mot-clé `pragmas` dans un processus SIGNAL. Ici, le pragma est utilisé dans une optique de génération de code. L'identificateur `CPP_CODE` indique que ce pragma est interprété lorsque le langage d'implantation est C++. Les notations `&i1`, `&o1` et `&o2` encodent respectivement la première entrée, les première et seconde sorties. Enfin, tout le morceau de code entre guillemets (invocation d'une méthode `BlackboardCheckID` sur `GLOBAL_BLACKBOARD_MANAGER`) représente le fragment considéré lors de la génération du code, et après substitution des paramètres encodés.

L'ensemble des services d'accès aux descripteurs a été défini suivant ce schéma. Ces services sont présentés de façon détaillée dans [92]. D'un point de vue programmation, le langage C++ est bien adapté pour l'implantation des descripteurs et leurs fonctions d'accès. De par son orientation objet, il offre de bons concepts pour la définition des structures de données en question. Une autre caractéristique utilisée est la possibilité de réaliser des effets de bord sur les structures de données. Par exemple, si on considère un pointeur qui indique à chaque instant d'une exécution le descripteur du processus actif, sa valeur peut être modifiée facilement. Ce genre de manipulation n'est pas possible en SIGNAL, qui n'offre pas de pointeurs.

Par ailleurs, ce choix de modélisation qui repose en partie sur un langage externe (ici C++) est lié à la non disponibilité, au moment de la réalisation, d'une mise en œuvre des constructions SIGNAL adaptées dans le compilateur. Les opérations de manipulation de descripteurs nécessitent un moyen d'exprimer le parcours des collections de structures de données qu'on souhaite consulter ou modifier. Ce genre de manipulation requiert la notion d'itération en SIGNAL. Dans la version actuelle du compilateur, l'itération est mise en œuvre par une construction spéciale, appelée *tableau de processus*. Au sein de chaque instant logique, il y a une itération spatiale qui traite tous les éléments du tableau (en l'occurrence, des processus SIGNAL). Cela est caractérisé à l'aide de la construction `array` [41]. L'exemple suivant définit [8] le signal `vect` comme un vecteur de dimension `n` dont les éléments représentent la diagonale de la matrice carrée `mat`, de même dimension :

```
array i to N
    vect[i] := mat[i,i]
end
```

Grâce à cette construction, nous pouvons définir une nouvelle description des services de gestion des objets, entièrement en SIGNAL. Nous rappelons qu'un des objectifs de cette modélisation est de pouvoir accéder aux outils et techniques formels disponibles dans POLYCHRONY pour la vérification et la validation. Dans des descriptions basées sur des abstractions boîte noire comme celle de la FIG. 8.8, le nombre d'informations que le compilateur peut traiter est relativement limité. En particulier, le fragment de code C++ spécifié dans le `pragma` est vu comme une information externe. Ce qui veut dire qu'il ne sera pas concerné par les vérifications effectuées. Par exemple, pour le service `CHECK_BOARD_ID` tel que donné sur la FIG. 8.8, seules les propriétés des signaux d'entrée/sortie sont prises en compte par le compilateur lors du calcul d'horloges (rien n'est connu du calcul effectué pour déterminer la valeur du signal `present` par exemple). Dans de pareilles situations, on doit donc s'assurer au préalable de la correction du fragment de code utilisé dans le `pragma`. Ici, les méthodes de classes C++ mises en œuvre sont en général courtes et faciles. Cela réduit les risques d'erreurs de programmation même si cela reste insuffisant pour garantir entièrement leur correction.

Toutes ces raisons motivent une évolution de la première solution vers la nouvelle, où davantage d'aspects seront appréhendés en utilisant les outils et techniques formels de POLYCHRONY.

8.2.2 Ordonnancement des processus

On rappelle que la politique d'ordonnancement est préemptive et basée sur les priorités. Chaque fois qu'un ordonnancement est déclenché, le nouveau processus actif dans la partition est celui qui a la priorité la plus forte et qui se trouve dans l'état *ready*. Pour décrire l'ordonnancement et le choix du processus actif (cf. FIG. 8.9), nous considérons deux services supplémentaires : `PROCESS_GETACTIVE` et `PROCESS_SCHEDULINGREQUEST` (les autres primitives requises font partie des services APEX). Ces services sont décrits, dans la première version de la bibliothèque, de façon analogue aux services de gestion de descripteurs (c'est-à-dire, en utilisant des abstractions). Leurs interfaces sont données sur la FIG. 8.10 (le code complet se trouve en annexe).

- Le service `PROCESS_SCHEDULINGREQUEST` comporte un signal d'entrée `sched_req` et un signal booléen en sortie, appelé `sched_ok`. Chaque réception du signal `sched_req` provoque un ré-ordonnancement des processus de la partition. C'est un événement qui est susceptible d'avoir pour origine soit une demande directe d'un processus ARINC, soit

8. Le calcul est réalisé à travers une itération spatiale.

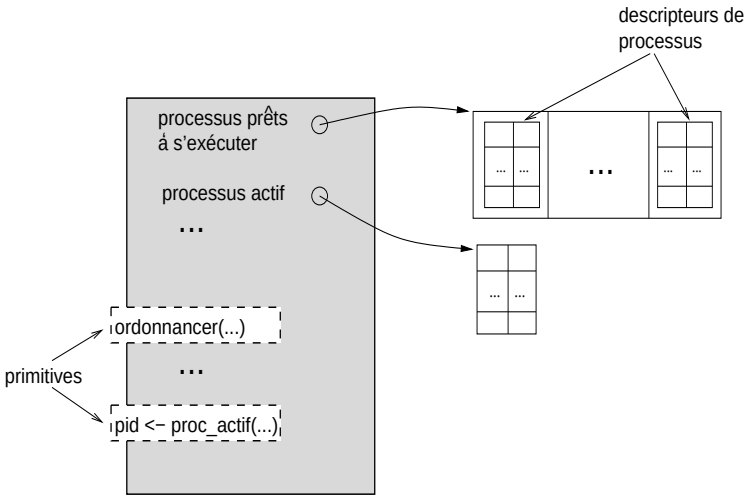


FIGURE 8.9 – Module d'ordonnancement des processus.

un événement quelconque interne à la partition (par exemple, une tentative d'accès à une ressource peut entraîner un ré-ordonnancement). La sortie booléenne `sched_ok` est produite instantanément. Elle est mise à *vrai* lorsque le ré-ordonnancement est effectué avec succès. Sinon, le signal porte la valeur *faux* (cela se produit notamment lorsque la partition interdit tout ré-ordonnancement, ce qui est indiqué par la valeur de son niveau de verrouillage).
On peut observer que seul ce service nécessite des modifications si on décide de changer de politique d'ordonnancement (exemples : R.M.S, E.D.F). Les services APEX modélisés ne sont pas affectés. Par conséquent, leur réutilisation est préservée.

- Le service `PROCESS_GETACTIVE` comporte quatre signaux d'interface : une entrée `request` et trois sorties `process_ID`, `status` et `valid`. Ce service est invoqué après chaque demande de ré-ordonnancement pour récupérer les informations sur le nouveau processus actif de la partition. La présence de l'événement `request` dénote une demande de récupération des informations. Celles-ci sont représentées par les signaux `process_ID` et `status` qui dénotent respectivement l'identificateur et le statut (état, priorité, période, etc.) du processus actif. La dernière sortie, `valid`, indique la validité des deux autres sorties (pour le cas où aucun processus ne serait prêt à s'exécuter - par exemple, dans le mode *idle*).

8.2.3 Gestion des compteurs de temps

La gestion du temps dans les exécutifs temps réel repose souvent sur la gestion d'un temps local. Elle est réalisée à travers des services, qui varient selon les exécutifs. Ces services permettent généralement de : maintenir à jour l'heure (sur une machine) ; effectuer des synchronisations (par exemple, suspension d'une tâche pour une certaine durée - service APEX *timed_wait*) ; attendre des communications (en utilisant un *time-out*).

La mise à jour des compteurs de temps déclenchés par les appels de services est réalisée au sein de l'exécutif. Ainsi, à chaque exécution d'actions dans le processus actif, tous les compteurs ayant une valeur strictement positive sont décrémentés de la durée consommée par cette exécution. Lorsque la valeur d'un compteur devient nulle, le processus déclencheur est alors

```
process PROCESS_SCHEDULINGREQUEST =
     ( ? event sched_req;
       ! boolean sched_ok;
     )
     spec (| (| sched_req --> sched_ok |)
           | (| sched_req ^= sched_ok |)
           |)
;

process PROCESS_GETACTIVE =
     ( ? event request;
       ! ProcessID_type process_ID;
         ProcessStatus_type status;
         boolean valid;
     )
    spec (| (| request --> valid
             | request --> {process_ID, status} when valid
             |)
          | (| request ^= valid
             | process_ID ^= status ^= when valid
             |)
          |)
;
```

FIGURE 8.10 – Interfaces des services *process_schedulingrequest* et *process_getactive*.

averti. Tous ces traitements sont réalisés à travers le service `UPDATE_COUNTERS`. Son interface, qui est illustrée sur la FIG. 8.11, comprend un seul signal d'entrée, `dt`. Celui-ci représente la quantité de temps servant à mettre à jour les compteurs de la partition. En sortie, le service fournit le signal `timedout`. Il s'agit d'un tableau de booléens dont la taille est au moins égale au nombre de processus de la partition. Pour chaque processus dans la partition, identifié par un certain entier *pid*, la valeur *timedout*[*pid*] indique si oui ou non le compteur de temps associé au processus est arrivé en butée (dans ce cas, *timedout*[*pid*] est mis à *vrai*, sinon il vaut *faux*). Tout expiration de temps concernant un compteur associé à *pid* peut être alors détectée aux instants où on observe un basculement de *timedout*[*pid*] de la valeur *faux* à la valeur *vrai*. Le signal `timedout` est émis régulièrement par le noyau en direction des processus pour leur rendre compte de l'état courant des compteurs de temps actifs.

*
* *

Une liste complète des services modélisés est donnée dans l'annexe A. On trouvera par ailleurs une présentation de ces services dans [92], où les propriétés d'interfaces sont données pour chaque service.

8.3 Modélisation des processus ARINC

Après une très brève présentation de la façon dont nous procédons pour décrire un processus ARINC, nous revenons plus en détail sur la construction et le fonctionnement du modèle.

```
process UPDATE_COUNTERS =
     ( ? SystemTime_type dt
       ! [MAX_NUMBER_OF_PROCESSES]boolean timedout;
     )
   spec (| (| dt --> timedout |)
         | (| dt ^= timedout |)
         |)
;
```

FIGURE 8.11 – Interface du service *update_counters*.

8.3.1 Présentation générale

Les processus représentent les entités élémentaires d'exécution au sein d'une application définie selon une architecture de type IMA. Pour modéliser un processus, nous le décomposons en fragments appelés *blocs*. Ces derniers représentent les actions atomiques effectuées par le processus. Leur exécution est instantanée et le processus ne peut être interrompu qu'entre l'exécution de deux blocs. Le fonctionnement global du modèle d'un processus est décrit ainsi par l'enchaînement d'exécutions de blocs résultant de son découpage. Ce choix de représentation est justifié par les propriétés du modèle polychrone introduit au chapitre 4. Nous présentons d'abord en section 8.3.2 le principe de construction d'un modèle SIGNAL de processus ARINC. Ensuite, nous montrons en section 8.3.3 comment un tel modèle fonctionne dans une exécution concurrente (comme celle de la FIG. 8.4 par exemple).

8.3.2 Construction du modèle

La démarche de construction du modèle repose sur une idée très simple et bien connue dans la conception des systèmes en général : *séparation du contrôle et du calcul*. Par exemple, c'est ce qu'on peut observer dans la *conception matérielle*, où un système est composé d'unités mettant en œuvre sa combinatoire [9] d'une part, et ses calculs de l'autre. Cette séparation des deux aspects est rappelée dans le *troisième principe de conception* [10] d'applications à l'aide de SIGNAL [98]. Le modèle d'un processus ARINC est constitué ainsi de deux sous-parties appelées *CONTROL* (qui représente le flot de contrôle) et *COMPUTE* (l'ensemble des actions exécutées). C'est ce qui est illustré sur la FIG. 8.12.

Le processus peut être vu comme un composant réactif dans lequel, chaque fois qu'un ordre d'activation est reçu de l'ordonnanceur, des actions sont sélectionnées pour être exécutées. Ainsi, le signal d'entrée `Active_process_ID` identifie le processus actif dans la partition. Il est reçu par tous les processus. Ceux-ci consultent la valeur qu'il porte de façon à déterminer s'ils peuvent s'exécuter ou pas. Ce signal joue donc le rôle d'ordre d'activation. L'entrée `timedout` quant à elle, sert à notifier à un processus bloqué sur l'appel d'un service APEX avec *time-out* (*read_blackboard*, par exemple), l'expiration du délai spécifié. Nous avons vu dans la modélisation des services que le décompte du temps pendant l'exécution d'un service bloquant fait partie des *actions globales*. Ces actions sont traitées au niveau de la partition, plus précisément par le *partition-level OS*. Ainsi, pendant la suspension d'un processus avec *time-out*, le compteur de

9. Autrement dit, la partie contrôle.
10. *Un processus est éclaté en deux sous-parties : les calculs et le contrôle.*

temps associé est décrémenté dans le *partition-level OS* jusqu'à ce qu'il devienne nul. Le processus en attente est alors averti via `timedout`. La sortie `dt` dénote la quantité de temps écoulée à chaque exécution d'actions par un processus. Cette information est utilisée au sein du *partition-level OS* pour la mise à jour des compteurs de temps.

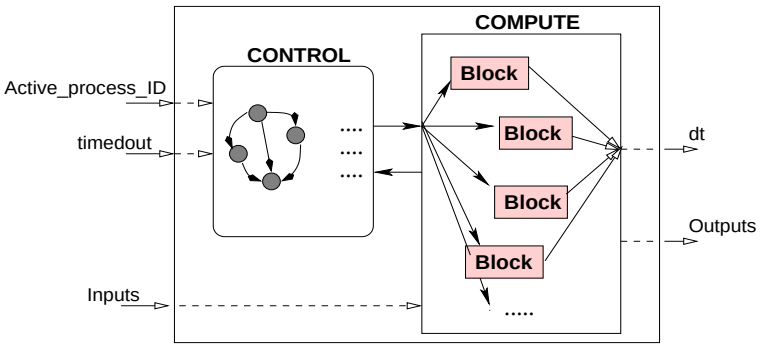


FIGURE 8.12 – Modèle générique d'un processus ARINC.

Les trois signaux mentionnés ci-dessus (`Active_process_ID`, `timedout` et `dt`) sont communs à toutes les interfaces de modèles de processus ARINC. En plus de ceux-là, il faut distinguer ceux qui représentent les entrées (resp. sorties) requises pour (resp. produites par) les calculs effectués dans la partie *COMPUTE*. Ces signaux varient d'un processus à un autre.

8.3.2.1 Description du contrôle

Le composant CONTROL décrit le flot d'exécution du processus. Typiquement, il est représenté par un système de transition qui, selon son état courant indique les actions à exécuter lorsque le processus est actif. La spécification d'automates en SIGNAL se fait de façon assez naturelle et simple (elle pourrait être produite automatiquement à partir d'une description graphique). Une interface générique du modèle SIGNAL correspondant au sous-composant *CONTROL* est donnée sur la FIG. 8.13.

Dans le processus SIGNAL de la FIG. 8.13, le paramètre `PID` dénote l'identificateur du processus ARINC associé (d'autres paramètres, comme le temps d'exécution au pire cas, peuvent aussi être spécifiés...). Ainsi, chaque fois que l'entrée `Active_process_ID` (qui est diffusée sur tous les processus) vaut `PID`, le processus ARINC identifié s'exécute. Le signal `timedout` (également diffusé à tous les processus) est représenté ici par un tableau de booléens tel que `timedout`$[i]$ vaut *vrai* lorsque le compteur de temps associé au i^{eme} processus vient d'expirer ; sinon, `timedout`$[i]$ est à *faux*. La sortie `Active_block` spécifie le sous-ensemble d'actions que le processus ARINC doit exécuter. Les signaux d'entrée `control_info_1`, ..., `control_info_n` font partie du dialogue entre *CONTROL* et le sous-composant *COMPUTE*. Ils dénotent des informations locales reçues de *COMPUTE* pour effectuer les transitions appropriées dans l'automate encodé dans *CONTROL*.

8.3.2.2 Description des actions exécutées

Le composant *COMPUTE* décrit les actions exécutées par le processus. Il est composé de *blocs* qui sont des fragments de code exécutés sans interruption comme des *filaments* [88] [11] ou encore les *lignées* [184] (cf. section 8.4.2). Dans le modèle, leur exécution est instantanée.

11. Des modèles de processus légers utilisés dans certaines architectures exploitant du parallélisme à grain très fin. Leur exécution est ininterruptible.

```
process CONTROL =
         { ProcessID_type PID; ... }
         ( ? ProcessID_type Active_process_ID;
             [NB_PROC]boolean timedout;
             info_type control_info_1 ... control_info_n;
           ! BlockID_type Active_block;
         )
    spec (| (| { {Active_process_ID, timedout, control_info_1 ... control_info_n}
                  --> Active_block } when (Active_process_ID = PID)
             |)
          | (| Active_process_ID ^= timedout
             | control_info_1 ^= when C_control_info_1
             | ...
             | control_info_n ^= when C_control_info_n
             | Active_block ^= when C_control_info_1 ^= ... ^= when C_control_info_n
                 ^= when (Active_process_ID = PID)
             |)
          |)
    where boolean C_control_info_1 ... C_control_info_n;
```

FIGURE 8.13 – Abstraction de la partie CONTROL d'un processus.

La figure 8.14 montre une interface générique du processus SIGNAL associé au sous-composant *COMPUTE* (un modèle complet est donné dans le chapitre 9).

Le signal d'entrée `Active_block` indique le bloc courant devant être exécuté. Les signaux `control_info_1`, ..., `control_info_n3` sont envoyés à *CONTROL* tandis que `IN_1`, ..., `IN_n1` et `OUT_1`, ..., `OUT_n2` sont des signaux requis ou produits par l'exécution des blocs. La sortie `dt` indique la durée d'exécution du bloc courant.

Types d'actions associées aux blocs. On distingue :

- D'une part, les actions susceptibles d'interrompre l'exécution du processus actif. Typiquement, l'appel à un service APEX (par exemple, *read_blackboard*). Elles sont qualifiées d'*appels systèmes*[12].
- D'autre part, les actions qui ne peuvent provoquer l'interruption du processus qui les exécute (c'est le cas des fonctions qui font juste un calcul sur des données). Celles-ci ne peuvent affecter que le contrôle local d'un processus : par exemple, le résultat obtenu suite à l'exécution d'un bloc b_i, et envoyé au composant *CONTROL*, peut être utilisé dans le choix du prochain bloc à exécuter. Ce genre d'action est appelé tout simplement *fonction*.

Construction des blocs. Le schéma d'exécution considéré pour les modèles de processus ARINC nécessite la prise en compte de quelques précautions, en particulier lorsqu'il s'agit de définir les actions à regrouper au sein d'un bloc. Nous pouvons faire les observations suivantes :

1. En ce qui concerne la nature des actions associées à un bloc : *au plus, un seul appel système peut être mis dans un bloc, et aucune autre instruction ne doit être mise dans le même bloc*. Supposons une situation contraire, c'est-à-dire, au sein d'un bloc, un appel système

12. En référence au fait qu'elles impliquent le modèle du système d'exploitation lors de leur invocation.

```
process COMPUTE =
  ( ? BlockID_type Active_block;
      Input_type IN_1 ... IN_n1;
    ! info_type control_info_1 ... control_info_n3;
      SystemTime_type dt;
      Output_type OUT_1 ... OUT_n2;
  )
spec (| (| {{Active_block, IN_1, ..., IN_n1} --> OUT_1} when C_OUT_1
         | ...
         | {{Active_block, IN_1, ..., IN_n1} --> OUT_n2} when C_OUT_n2
         | {{Active_block, IN_1, ..., IN_n1} --> control_info_1} when C_control_info_1
         | ...
         | {{Active_block, IN_1, ..., IN_n1} --> control_info_n3} when C_control_info_n3
         | Active_block --> dt
         |)
      | (| Active_block ^= C_OUT_1 ^= ... ^= C_OUT_n2  ^= C_control_info_1 ^= ... ^=
                C_control_info_n3 ^= dt
         | OUT_1 ^= when C_OUT_1 | ... | OUT_n2 ^= when C_OUT_n2
         | control_info_1 ^= when C_control_info_1
         | ...
         | control_info_n3 ^= when C_control_info_n3
         |)
      |)
 where boolean C_OUT_1, ..., C_OUT_n2, C_control_info_1, ..., C_control_info_n3;
```

FIGURE 8.14 – Abstraction de la partie COMPUTE d'un processus.

suivi d'autres actions. Puisque l'exécution d'un bloc est atomique (instantanée), on voit bien qu'il peut y avoir une incohérence lorsque l'appel système interrompt le processus appelant. En effet, les actions qui suivent sont exécutées aussi malgré l'interruption ; ce qui n'est pas correct. D'autre part, si un bloc doit être ré-exécuté après une interruption du processus provoquée par l'appel système contenu dans ce bloc, il n'est pas forcément nécessaire de ré-exécuter les autres actions présentes dans le bloc. En définitive, pour être sûr d'une exécution correcte des blocs, on mettra dans un bloc un seul appel système et aucune autre action en plus de cet appel [13].

2. Pour garantir un comportement temps réel du modèle de processus proposé, il est important d'imposer que la durée d'exécution des actions associées à chaque bloc soit bornée. On pourrait ainsi considérer *a priori* les temps d'exécution au pire cas obtenus par une technique particulière. Par exemple dans POLYCHRONY, on utilisera une technique d'évaluation de performance d'un programme SIGNAL pour déterminer ces informations. Nous reviendrons plus en détail sur cette technique dans la troisième partie de ce document.

8.3.3 Fonctionnement du modèle

Les sous-composants *CONTROL* et *COMPUTE* coopèrent de la même façon que dans un automate de mode [158]. Un processus s'exécute chaque fois qu'il est désigné comme étant actif. Lorsque c'est le cas, sa partie CONTROL vérifie que le signal `Active_process_ID` reçu vaut bien `PID`. À ce moment seulement, et selon l'état courant du système de transition qui encode le flot d'exécution du processus, un bloc est choisi pour être exécuté instantanément. Ensuite, le processus attend un nouvel ordre d'exécution de la part de l'ordonnanceur, et ainsi de suite. L'unité d'exécution au sein du modèle d'un processus ARINC est donc le bloc.

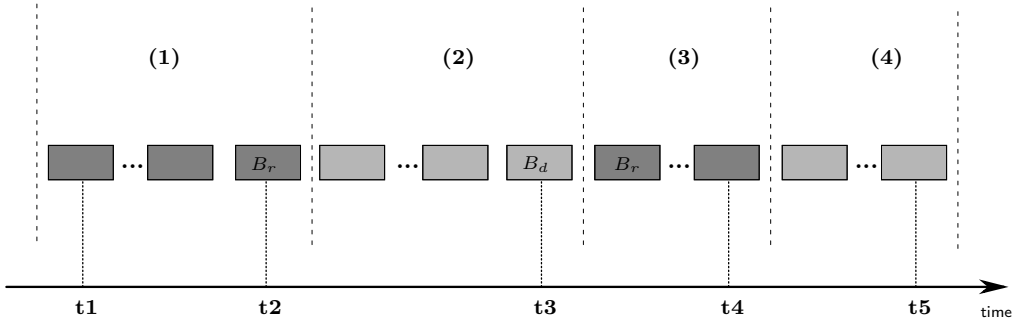


FIGURE 8.15 – Trace résultant de la simulation des modèles de P1 et P2.

Ce schéma d'exécution est parfaitement cohérent avec la décomposition présentée en section 8.1 concernant la modélisation des services APEX (décomposition en actions locales et globales). Pour s'en convaincre, considérons encore les processus $P1$ et $P2$ de la FIG. 8.4, qui communiquent via un *blackboard*. Parmi les blocs du modèle de processus ARINC associé à $P1$, il existe un bloc B_r contenant un appel au service *read_blackboard*. Ce bloc est entouré d'autres blocs, qui contiennent d'une part, les instructions exécutées par $P1$ avant l'appel du service (situées dans la tranche `(1)` sur la FIG. 8.4) ; et d'autre part, celles qui sont exécutées après (i.e., dans la tranche `(3)` sur la FIG. 8.4). De façon similaire, le modèle correspondant à $P2$ inclut un bloc B_d qui contient un appel au service *display_blackboard* (utilisé pour écrire un

13. Cependant, pour des raisons d'optimisation du nombre de blocs, on peut dans certains cas faire précéder un appel système d'autres actions qui seront forcément des fonctions.

message dans un *blackboard*). Sur la FIG. 8.15, nous décrivons la trace d'exécution des modèles de processus ARINC associés à $P1$ et $P2$ (c'est une autre façon d'observer le scénario de la *situation B* de la FIG. 8.4). Les boîtes représentent les blocs. Les plus foncées appartiennent à $P1$ et les autres correspondent aux blocs de $P2$. Les traces d'exécution (en termes de blocs) des deux processus sont superposées pour montrer comment elles s'entrelacent. On rappelle que chaque exécution d'un bloc résulte de la réception d'un ordre de l'ordonnanceur. Considérons la partie suivante dans la trace : P1 s'exécute d'abord jusqu'à l'instant `t2` ; puis, B_r est exécuté et le nouveau processus actif devient P2 jusqu'à l'exécution de B_d. L'exécution de B_r correspond à celle des actions locales uniquement (comme la vérification de la validité des paramètres d'entrée du service, le déclenchement d'un compteur pour l'attente). Les actions globales (telles que la décrémentation des compteurs non nuls, la notification aux processus concernés lorsque les compteurs deviennent zéro) spécifiées dans le service sont traitées au sein du *partition-level OS*. En l'occurrence, celles qui ont été engendrées par l'appel B_r sont prises en charge durant l'exécution de $P2$, dans la tranche `(2)` sur la FIG. 8.15. Le processus $P1$ ne récupère le processeur qu'après l'instant `t3`, suite à l'écriture d'un message dans le *blackboard* par $P2$ (bloc B_d). Il ré-exécute le bloc B_r et lit enfin un message.

Écoulement du temps. La durée d'exécution de chaque bloc du processus actif est obtenue via son signal de sortie `dt`. Cette quantité de temps résulte d'une estimation qui peut être obtenue avant l'exécution ou bien pendant celle-ci. Dans les deux cas, la technique d'évaluation de performances basée sur le morphisme de programmes SIGNAL, présentée au chapitre 4, est applicable :

- Pour une estimation avant l'exécution, des simulations peuvent être réalisées isolément pour chaque morceau de code associé à un bloc pour déterminer son temps d'exécution au pire cas (ce qu'on pourrait qualifier d'estimation hors ligne). Dans un modèle d'exécution où on considère *a priori* un *quantum de temps* d'utilisation des ressources, il faut être capable d'estimer les durées des calculs avant l'exécution. Cela permet de garantir que le code associé à chaque bloc pourra être exécuté sans excéder le *quantum* choisi.
- On peut aussi composer le programme SIGNAL représentant l'interprétation temporelle de chaque bloc avec celui qui décrit les actions associées au bloc. La co-simulation de l'ensemble (cf. chapitre 4) fournira dans ce cas les temps d'exécution en ligne. Cette seconde possibilité fournit des durées de calcul égales aux temps de latence des actions qui sont effectivement exécutées. Ces durées reflètent mieux l'exécution en cours, comparées à celles qui sont obtenues à l'aide d'une estimation au pire cas.

Bien entendu, une combinaison de ces deux techniques est possible, permettant ainsi de profiter des avantages de chacune.

Description de la préemption. Sur la trace schématisée sur la FIG. 8.15, nous remarquons que suivant le modèle de processus proposé, tout processus actif est interruptible après l'exécution de chacun de ses blocs. C'est donc à ce niveau que nous choisissons [14] de faire intervenir la préemption : à la fin de chaque exécution de bloc, un ré-ordonnancement de processus est systématiquement effectué pour désigner le prochain processus à activer (cf. section 8.2). Ainsi, si un processus de priorité plus forte que celui qui détenait le processeur précédemment venait à passer dans l'état *ready*, c'est lui qui est choisi. L'exécution du processus actif au pas d'avant est alors gelée. Celle-ci reprendra à partir du bloc suivant de sa partie *COMPUTE* si

14. Selon la norme ARINC 653, tout processus est susceptible d'être préempté à tout moment par un autre processus de priorité plus grande et prêt à s'exécuter.

le bloc courant n'est pas à la base de sa suspension. Sinon, une ré-exécution du bloc courant peut être nécessaire (typiquement, sur un appel d'un service APEX bloquant, non suivi d'une réception de code retour).

*
* *

La FIG. 8.16 illustre une vue globale d'un modèle de partition. Nous pouvons observer le lien entre le modèle du *partition-level OS* (qui joue le rôle du noyau de l'exécutif temps réel) et les processus ARINC.

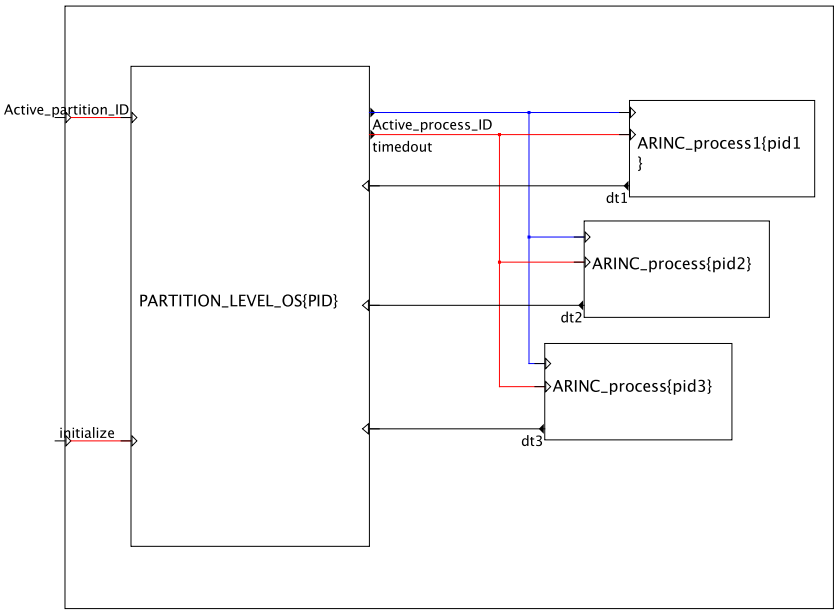


FIGURE 8.16 – Modèle d'une partition.

L'entrée `Active_partition_ID` représente l'identificateur de la partition active, sélectionnée par l'OS de niveau supérieur, c'est-à-dire le niveau module [15]. Elle dénote un ordre d'exécution lorsqu'elle identifie la partition courante. La réception du signal `initialize` correspond à la phase d'initialisation de la partition : création de l'ensemble des mécanismes et processus contenus dans celle-ci. Les entrées `Active_partition_ID` et `initialize` sont externes à la partition : elles proviennent du niveau supérieur (c'est-à-dire le module, non représenté ici).

Les signaux `dt1`, `dt2` et `dt3` indiquent les quantités de temps écoulées lors des exécutions de blocs. Contrairement aux signaux `Active_partition_ID` et `initialize`, ces entrées du modèle de l'OS sont internes à la partition (elles proviennent toutes des processus de la même partition). Le signal `Active_process_ID` identifie le processus actif désigné à chaque ré-ordonnancement dans la partition. Il est déterminé par l'OS en fonction de la politique d'ordonnancement. Il est envoyé à tous les processus de la partition.

*
* *

15. À l'instar du modèle de processus, l'activation de chaque partition dépend de l'entrée *Active_partition_ID*, qui identifie la partition active courante. C'est un signal qui est fourni par le *module-level OS* qui est en charge de la gestion des partitions dans un module (cf. chapitre 7).

Nous venons de présenter la modélisation SIGNAL des différents composants requis pour décrire des applications temps réel sous forme de partitions. Ces modèles comprennent les services APEX, définis dans la norme ARINC 653 [8], auxquels nous avons ajouté des services complémentaires nécessaires à une modélisation complète. L'ensemble de ces services permet de décrire les fonctionnalités habituellement offertes par un système d'exploitation ou un noyau temps réel au développeur : gestion de processus, communications et synchronisations entre processus, ordonnancement et gestion du temps. Un modèle de processus ARINC a été proposé. Celui-ci permet de prendre en compte les caractéristiques d'une exécution temps réel.
Ainsi, une partition ARINC est représentée par un modèle SIGNAL dont le fonctionnement est le suivant :

- l'enchaînement des actions effectuées par chaque processus de la partition (i.e. l'exécution des blocs) est déterminé *statiquement*, puis encodé au sein du sous-composant *CONTROL* ;
- le choix d'activation de chacun des processus est lui réalisé *dynamiquement*. Il dépend des décisions de l'ordonnanceur qui se trouve dans le *partition-level OS*.

La modélisation d'un module est quasi identique à celle de la partition. La principale différence concerne la définition du *module-level OS*. La politique d'ordonnancement adoptée dans celui-ci repose sur du temps partagé. On définit *a priori* un ordonnancement statique suivant lequel les partitions appartenant au module sont activées. En SIGNAL, cela peut se décrire simplement à l'aide de *compteurs modulo* pour une stratégie en tourniquet par exemple.

8.4 Discussion

Nous faisons d'abord quelques observations générales par rapport à l'approche de description qui vient d'être exposée (cf. section 8.4.1). Ensuite, nous abordons l'intégration de celle-ci dans la méthodologie de conception d'applications distribuées temps réel au sein de POLYCHRONY (cf. section 8.4.2). Enfin, nous mentionnons d'autres approches en rapport avec la nôtre (cf. section 8.4.3).

8.4.1 Quelques observations

L'approche décrite dans ce chapitre s'inscrit dans une démarche orientée composants. Nous avons défini une bibliothèque SIGNAL de services APEX, ainsi que des modèles correspondants aux processus et partitions ARINC. Ces éléments sont suffisants pour décrire une application avionique, suivant l'architecture modulaire intégrée. Ces modèles peuvent facilement être adaptés pour d'autres types d'applications temps réel. Par ailleurs, le fait qu'ils soient définis dans un formalisme de haut niveau tel que SIGNAL garantit leur indépendance vis-à-vis de toute plate-forme de mise en œuvre.

Pour des applications de grande taille, il est important de disposer de mécanismes adéquats pour en définir un modèle. Ici, nous nous basons essentiellement sur deux caractéristiques de la programmation à l'aide du langage SIGNAL : la modularité et l'abstraction. Le modèle d'une application consiste, d'une façon générale, en une abstraction où on ne prend en compte que les propriétés pertinentes pour l'étude souhaitée sur ce modèle. Une démarche consiste à modéliser soit complètement, soit partiellement (par abstraction) des parties de l'application considérée. Celles-ci sont ensuite composées pour donner de nouvelles parties, elles-mêmes abstraites éventuellement puis composées et ainsi de suite jusqu'à obtention du modèle final. À chaque étape, des analyses sont possibles sur les modèles intermédiaires.

La vérification de propriétés est un aspect important dans la conception d'applications critiques telles que celles visées par nos modèles. L'environnement POLYCHRONY fournit un certain nombre d'outils formels pour l'analyse. Nous avons déjà illustré une utilisation de SIGALI pour faire du *model checking* (cf. chapitre 6). La technique d'évaluation de performances mise en œuvre permet d'examiner le comportement temporel d'une application. En outre, la possibilité de générer automatiquement du code permet de faire des simulations pour des tests. Pour une application décrite à l'aide des composants ARINC, nous observons différents niveaux de génération de code :

- Si on dispose d'un exécutif de type ARINC, on peut faire le lien avec cet exécutif au niveau de l'appel des services APEX (la modélisation complète en SIGNAL peut cependant être utilisée pour la simulation).
- Si on n'en dispose pas ou si on souhaite s'en passer, on génère le code à partir du modèle SIGNAL complet. Dans ce cas, le principal avantage réside dans la possibilité d'utiliser les outils et techniques disponibles dans POLYCHRONY pour la vérification. D'autre part, il faudra s'assurer que les contraintes temps réel seront bien respectées. Pour cela, la technique de co-simulation de programmes SIGNAL avec leurs interprétations temporelles, présentée au chapitre 4, pourra être utilisée.

Ces niveaux de génération de code ont des conséquences sur la façon d'effectuer l'analyse temporelle. Tous ces moyens contribuent à la mise à disposition d'un cadre d'étude adéquat pour la validation.

Enfin, l'utilisation des modèles définis ici s'inscrit dans la méthodologie générale de conception d'applications distribuées temps réel, présentée au chapitre 5. Celle-ci prône une démarche basée le plus possible sur le langage SIGNAL. Ainsi, toutes les transformations effectuées peuvent être prouvées valides dans le modèle sémantique du langage. Dans la section suivante, nous proposons une base de réflexion vers une intégration des composants définis dans la méthodologie.

8.4.2 Vers une approche générale de conception dans POLYCHRONY

Dans cette section, nous examinons comment les modèles ARINC définis peuvent être utilisés dans une approche similaire à la méthodologie qui a été définie dans POLYCHRONY, pour la conception d'applications distribuées temps réel (cf. chapitre 5). Suivant la démarche que nous adoptons ici, les composants ne serviront dans la description d'un programme SIGNAL qu'après un certain nombre de transformations de celui-ci. Nous décrivons d'abord ces transformations. Ensuite, nous montrons comment est réalisé le passage à une description de type ARINC.

8.4.2.1 Transformations préliminaires de programmes

Une caractéristique de SIGNAL est de permettre une décomposition modulaire des programmes spécifiant une application. Cette séparation est obtenue à travers des transformations formelles valides (présentées au chapitre 4). Il en résulte une description distribuée de l'application. L'utilisateur peut alors choisir l'architecture d'implantation souhaitée. Celle-ci peut être *multi-tâche mono-processeur*, ou plus généralement, *multi-tâche multi-processeur*. L'avantage dans cette démarche réside dans la capacité à faciliter la validation grâce au modèle sémantique de SIGNAL.

Les notions présentées ci-après sont issus de résultats obtenus lors du projet SACRES [97]. Le but était de définir des schémas de génération de code distribué à partir de spécifications

synchrones, et plus particulièrement, de programmes décrits en SIGNAL.

On considère les hypothèses suivantes :

1. les programmes considérés sont initialement **endochrones** (donc déterministes) ;
2. ils ne contiennent **aucune définition circulaire** ;
3. il existe une fonction *allouer*, qui définit la répartition du modèle fonctionnel sur celui de l'architecture matérielle (l'allocation peut être manuelle ou automatique).

On dispose :

– d'un ensemble fini de processeurs $q = \{q_1, q_2, ..., q_m\}$;
– de la fonction *allouer*, définie de la manière suivante :

$$allouer : \{P_i\} \longrightarrow \mathcal{P}(q)$$

Elle affecte chaque processus P_i à un ensemble non vide de processeurs, avec possibilité de répliquer du code.

Enfin, pour présenter les transformations, nous considérons des programmes SIGNAL de la forme $P = P_1 \mid P_2 \mid ... \mid P_n$, où chaque processus P_i peut lui-même être composé récursivement d'autres processus (i.e., $P_i = P_{i1} \mid P_{i2} \mid ... \mid P_{im}$).

Première transformation. Soit un programme SIGNAL représenté par le processus $P = P_1 \mid P_2$, illustré sur la FIG. 8.17. Chaque sous-processus P_i (décrit par un ovale transparent) est lui-même composé de quatre sous-processus P_{i1}, P_{i2}, P_{i3} et P_{i4}. On souhaite répartir P sur deux processeurs, q_1 et q_2. La fonction d'allocation est définie comme suit :

$$\begin{array}{ll} \forall i \in \{1,2\}\ \forall k \in \{1,2\}, & allouer(P_{ik}) = \{q_1\}, \quad et \\ \forall i \in \{1,2\}\ \forall k \in \{3,4\}, & allouer(P_{ik}) = \{q_2\} \end{array}$$

Le processus P peut alors se récrire en :

$$P = Q_1 \mid Q_2$$

où

$$\begin{array}{ll} Q_1 = & P_{11} \mid P_{12} \mid P_{21} \mid P_{22}, \quad et \\ Q_2 = & P_{13} \mid P_{14} \mid P_{23} \mid P_{24} \end{array}$$

Les sous-processus Q_1 et Q_2, issus de ce partitionnement de P, sont appelés *s-tasks* [97]. Cette transformation fournit une description multi-processeur strictement équivalente au programme initial d'un point de vue sémantique (ensembles de comportements identiques) car il s'agit d'une simple récriture de programme.

Seconde transformation. On souhaite raffiner le niveau de granularité de la décomposition effectué lors de la première transformation. Pour cela, on se place au niveau de chaque processeur et on procède à un partitionnement des *s-tasks*. Celles-ci sont vues alors comme un ensemble de *nœuds*, représentant les entités d'exécution atomique. Deux choix sont envisageables concernant le niveau de granularité de ces nœuds : soit tout nœud consiste en un processus primitif de SIGNAL ; soit un nœud peut être constitué de plusieurs processus primitifs. On obtient une description très fine en adoptant la première possibilité. Cependant, la seconde peut être plus

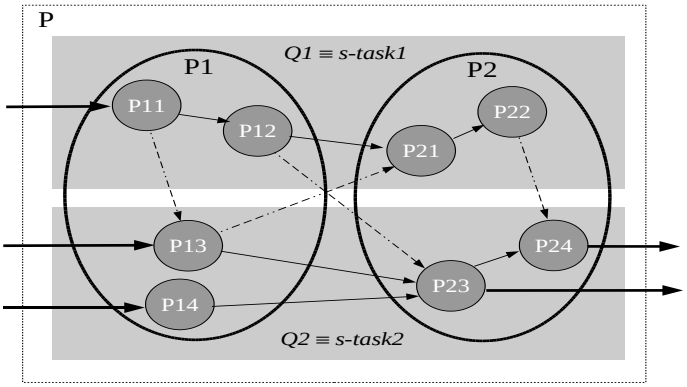


FIGURE 8.17 – Décomposition d'un programme SIGNAL P en deux *s-tasks*.

judicieuse du point de vue de l'exécution car davantage d'actions peuvent être effectuées de façon atomique. Une façon efficace de construire de tels nœuds est de choisir comme critère le fait que les expressions regroupées au sein d'un nœud dépendent d'un même ensemble d'entrées. Cela repose essentiellement sur une analyse de *sensitivité*, effectuée sur le programme SIGNAL.

Définition 28 *Deux nœuds N_1 et N_2 sont sensitivement équivalents ssi pour chaque entrée e, les propriétés 1. et 2. sont équivalentes :*

1. *il existe un chemin de causalité entre e et N_1,*
2. *il existe un chemin de causalité entre e et N_2.*

□

La définition ci-dessus spécifie que des nœuds sensitivement équivalents requièrent les mêmes signaux d'entrée. Ils peuvent donc être regroupés au sein d'un même sous-ensemble de nœuds qui est exécutable de façon atomique : ces sous-ensembles sont appelés *lignées* (ou *clusters*). Une propriété importante de celles-ci est que toute entrée précède toute sortie. D'autre part, dans la mise en œuvre de POLYCHRONY, des transformations supplémentaires sont effectuées pour rendre chaque lignée endochrone. Ce qui rend leur exécution déterministe. La FIG. 8.18 montre la décomposition de Q_1 en deux lignées $L1$ et L_2. L'entrée du sous-processus $P11$ (flèche en gras) est à l'origine une entrée de P. Les autres flèches mises en évidence sur le schéma sont des communications locales à P. Elles représentent des signaux échangés par les *s-tasks*. Nous pouvons remarquer qu'après cette seconde transformation, l'équivalence sémantique stricte du processus initial et du processus résultant est toujours préservée.

Les deux transformations présentées ci-dessus décrivent un partitionnement de programmes SIGNAL selon une architecture générique multi-tâche multi-processeur. L'instanciation d'une telle représentation dans le modèle ARINC consiste à utiliser les composants définis dans ce chapitre (services APEX, processus ARINC, partitions) pour modéliser les concepts correspondants.

8.4.2.2 Instanciation de programmes SIGNAL dans le modèle ARINC

Pour aborder la modélisation à l'aide des composants ARINC, nous nous plaçons d'abord au niveau d'un processeur. La démarche pourra ensuite être généralisée à un ensemble quelconque

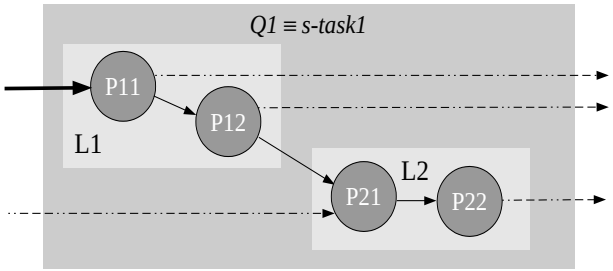


FIGURE 8.18 – Décomposition de Q_1 en deux nœuds.

de processeurs. Ainsi, d'après les transformations ci-dessus, un processeur peut être vu comme un graphe où les nœuds sont des lignées. Le partitionnement d'un programme SIGNAL suivant le modèle d'architecture IMA est réalisable à travers les étapes suivantes :

Étape 1 : Transformation d'un programme SIGNAL en graphe de lignées. Cette tâche est réalisable de façon automatique grâce au compilateur.

Étape 2 : Regroupement des lignées en modèles de partitions et de processus ARINC. La transformation du graphe de lignées associé à un processeur commence par le choix d'une représentation en termes de partitions ARINC. Il s'agit précisément d'identifier les lignées (i.e. les sous-ensembles du graphe) qui vont s'exécuter au sein d'une même partition. On rappelle qu'une partition s'exécute sur un processeur à la fois. Celui-ci peut se voir affecter plusieurs partitions. Sur notre exemple, on décide de modéliser le graphe associé à $Q1$ (cf. FIG. 8.18) par une seule partition. Les partitions étant choisies, le graphe correspondant à chacune d'entre elles est à son tour décomposé en sous-graphes. Ces derniers contiennent les lignées devant s'exécuter dans un même processus ARINC. Ici, toutes les lignées associées à la partition correspondant à $Q1$ sont mises dans un processus ARINC unique. La distribution d'un graphe de lignées en partitions et processus se fait de façon *a priori* arbitraire. Cependant, il est plus judicieux de regrouper au sein d'une même partition les lignées qui dépendent fortement les unes des autres. On peut alors procéder à l'instanciation à proprement parler dans le modèle ARINC.

		$p - OS$	(OS de niveau partition)
		$cont_i$	(partie *control* d'un processus p_i)
		b_{ij}	(*bloc* contenant soit un *appel système* soit une *fonction*)
p	::=	$p - OS \mid p_1 \mid ... \mid p_n$	(*partition* ARINC)
p_i	::=	$cont_i \mid comp_i$	(*processus* ARINC)
$comp_i$	::=	$b_{i1} \mid ... \mid b_{im_i}$	(partie *compute* d'un processus p_i)

FIGURE 8.19 – Description des modèles ARINC.

Étape 3 : Instanciation dans le modèle ARINC. On procède en deux phases : on instancie d'abord les processus ARINC et ensuite, on fait de même avec les partitions. Les principaux composants ARINC modélisés sont rappelés brièvement sur la FIG. 8.19. Le symbole "|" dénote la composition synchrone. On définit les transformations suivantes :

1. Description du processus ARINC associé à un ensemble de lignées :
   - La partie *CONTROL* du processus associé est construite en se basant sur les dépendances entre les lignées. On choisit un ordonnancement basé sur une exécution séquentielle des lignées ;
   - Chaque lignée est "plongée" dans un bloc de type fonction ;
   - Les communications internes entre les lignées d'un même sous-graphe associé à un processus sont modélisées à l'aide de variables d'état locales au processus. Elles servent à mémoriser les données échangées. En ce qui concerne les communications entre sous-graphes de lignées, on utilise les services APEX. Il s'agit de communications entre processus. Pour chaque entrée (resp. sortie) d'un sous-graphe, on ajoute dans la partie *COMPUTE* du processus ARINC associé, un bloc de type appel système (service de communication ou de synchronisation) qui lui correspond. Lorsque le processus ARINC devient actif, le bloc appel système ajouté doit être exécuté juste avant (resp. après) le bloc qui contient la lignée concernée par l'entrée (resp. la sortie).

2. Description de la partition ARINC associée à un ensemble de lignées :
   - On ajoute aux processus définis ci-dessus le composant correspondant au système d'exploitation de niveau partition (contenant entre autres l'ordonnanceur qui fixe l'ordre d'exécution des processus) ;
   - On crée les mécanismes de communication et de synchronisation utilisés dans les services APEX ajoutés au sein des processus (par exemple, pour un appel du service *send_buffer*, il faut créer le *buffer* dans lequel les messages sont stockés).

Exemple 3 *Un modèle de processus ARINC résultant de la transformation de $Q1$ est représenté de façon non détaillée sur la* FIG. *8.20. Il comprend six blocs dont deux qui contiennent les lignées $L1$ et $L2$. Les autres blocs ont été ajoutés pour les communications : r, s et w dénotent respectivement une lecture (par exemple, receive_buffer ou read_blackboard), une notification d'un événement (comme set_event), et une requête d'attente d'un événement (par exemple, wait_event). Enfin, l'automate de la partie contrôle décrit un ordre d'exécution des blocs (avec possibilité de boucler sur certains états - même si cela n'apparaît pas sur la figure). Cet ordre respecte les contraintes de précédence entre les lignées du graphe.*

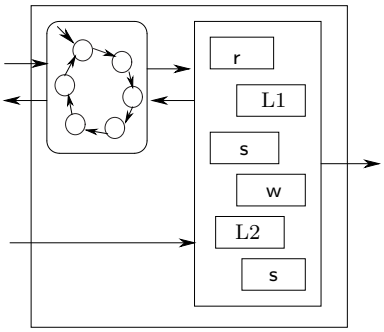


FIGURE 8.20 – Modèle de processus ARINC associé à $Q1$.

Pour obtenir la partition correspondante, nous nous rapportons à la deuxième phase d'instanciation (ajout du modèle de l'OS de niveau partition et création des mécanismes de communication requis par le processus ARINC).

Sur un processeur où il y a plusieurs partitions, on adopte la même démarche pour chacune d'entre elles. Un modèle d'ordonnanceur de partitions est nécessaire au niveau du processeur.

La politique de gestion à adopter est alors basée sur du temps partagé (comme dans une politique *round robin*). On ajoutera donc aux partitions un composant correspondant au système d'exploitation de niveau module. Ce dernier est similaire à l'OS de niveau partition dans le sens où sa définition repose sur les services APEX. Par contre au sein du module, l'ordonnancement est basé sur une politique à temps partagé, au lieu d'une politique préemptive comme c'est le cas dans une partition.

*
* *

Nous venons d'esquisser une démarche suivant laquelle des programmes SIGNAL, spécifiés indépendamment d'un choix de mise en œuvre sont décrits selon une architecture modulaire intégrée. Les composants ARINC définis sont utilisés pour décrire le modèle final. Cette démarche s'intègre parfaitement à la méthodologie générale de POLYCHRONY pour la conception d'applications distribuées temps réel, présentée au chapitre 5. Par ailleurs, il convient de mentionner que ce n'est pas l'unique voie de représentation de programmes SIGNAL sous forme d'architecture de type IMA. Dans l'optique d'une modélisation de plus bas niveau, on peut vouloir exprimer un calcul complexe provenant d'une lignée, avec du sur-échantillonnage temporel (i.e., l'exécution d'un bloc n'est plus instantanée comme c'est le cas ici). Dans ce cas, l'horloge la plus rapide d'un processus ARINC n'est plus nécessairement celle qui est définie par le *partition-level OS*, mais plutôt celle qui permet de raffiner temporellement les exécutions de blocs.

8.4.3 Lien avec d'autres travaux

Dans le chapitre 1, nous avons évoqué un certain nombre d'approches à la conception d'applications temps réel. Il s'agit à présent de voir en quoi la nôtre se distingue de celles-ci.

Taxys. C'est une approche qui a été définie pour la conception et la validation d'applications temps réel enfouies [65]. Elle combine les langages ESTEREL et C pour décrire respectivement les parties contrôle et calcul d'une application. Conceptuellement, une telle description est constituée de trois composants exprimés en ESTEREL : l'application elle-même, l'environnement d'interaction de celle-ci et un gestionnaire externe des événements échangés par l'application et son environnement (cela est nécessaire pour éviter des pertes durant les échanges). Le code ESTEREL est ensuite annoté avec des contraintes temporelles (durées d'exécution des calculs et dates d'échéance). L'ensemble est transformé sous forme d'automates temporisés. L'outil KRONOS [219] est alors utilisé pour vérifier que les contraintes temporelles spécifiées dans l'application sont bien respectées. Le compilateur SAXO-RT [215] génère le code C enfoui correspondant. Un point commun de l'approche TAXYS avec celle que nous avons proposée est qu'elles utilisent toutes les deux la technologie synchrone. Dans un cas, c'est ESTEREL qui sert à la description et dans l'autre, c'est plutôt SIGNAL. Elles proposent ainsi une démarche basée sur des outils formels. La principale différence entre ces approches réside dans la façon de représenter les applications. En effet, notre modélisation propose un niveau de détail plus fin d'une mise en œuvre d'application temps réel. Elle est donc de plus bas niveau que celle qui est offerte par TAXYS. Ce qui permet d'aborder davantage d'aspects d'une description d'application.

Giotto. Cette approche a déjà été introduite au chapitre 1. Elle repose sur le modèle temporisé pour la conception de systèmes de contrôle enfouis, à contraintes temps réel strictes [120]. GIOTTO est aussi une approche formelle. Une spécificité de celle-ci est la sémantique *time-triggered* dont l'avantage est la prédictibilité des comportements temporels du système décrit. Un modèle GIOTTO d'une application est donné sous forme de tâches avec un ordonnanceur.

Il est donc proche de celui que nous avons défini. Cependant, les tâches sont essentiellement périodiques alors que dans notre cas, cette restriction n'existe pas.

MetaH. Nous avons également présenté le formalisme MetaH [210] au chapitre 1. Une approche basée sur ce formalisme a été définie pour la conception de systèmes temps réel embarqués. Celle-ci figure parmi les plus connues dans le domaine de l'avionique. MetaH constitue notamment la base du standard Aadl (*Avionics Architecture Description Language*) [86] dédié exclusivement à la description d'architectures de systèmes avioniques. La description d'un système suivant MetaH ressemble à celle adoptée dans Giotto (un ensemble de processus avec un module chargé d'ordonnancer ceux-ci). Cependant, dans la modélisation MetaH, il existe à la fois des processus périodiques et apériodiques, comme c'est le cas aussi dans la nôtre. MetaH pose quelques restrictions au niveau des communications entre processus. Ces derniers ne peuvent échanger de messages qu'en début ou en fin d'exécution. Si ce choix facilite l'analyse des comportements de l'application, il est restrictif. Dans le modèle proposé ici pour les processus ARINC, les communications peuvent avoir lieu à n'importe quel moment de l'exécution.

SynDEx. Lors de la conception d'applications distribuées temps réel dans l'environnement SynDEx, c'est l'outil qui se charge de produire un exécutif dédié [106], défini selon les besoins de l'application sans faire appel à un exécutif résident. Le noyau de cet exécutif est générique et facile à porter sur d'autres composants matériels. Dans notre cas, l'exécutif est résident et conforme à un type spécifique d'architecture qui est IMA. Cela est dû au standard APEX qui requiert l'utilisation des bus ARINC 629 [5] et ARINC 659 [6] (cf. chapitre 7). Pour permettre l'utilisation d'exécutifs basés sur d'autres standards (par exemple, POSIX ou OSEK), il faut étendre la bibliothèque avec des modèles respectant les spécifications issues de ces standards comme nous l'avons fait pour la norme APEX. Nous développerons ce point dans les perspectives données dans la conclusion de cette thèse. L'approche SynDEx et les techniques utilisées dans la méthodologie de conception développée dans Polychrony sont complémentaires.

D'une part, dans SynDEx, la distribution et l'ordonnancement d'une application sur une architecture matérielle multi-processeur tient compte des critères de performance de l'architecture cible (temps de réponse déterminé à partir d'un graphe logiciel de l'application, valué par des temps d'exécution des processus sur le type de processeur considéré, et les durées des communications). Ces critères sont typiquement le genre d'informations qu'on peut calculer à l'aide de la technique d'évaluation de performances basée sur le morphisme de programmes Signal.

D'autre part, le partitionnement des applications dans SynDEx garantit une "adéquation algorithme achitecture" : c'est-à-dire, la distribution et l'ordonnancement obtenus automatiquement permettent de respecter des contraintes temps réel et de minimiser les ressources matérielles. Cela peut donc orienter vers des choix de partitionnement intéressants dans la méthodologie générale de conception dans Polychrony. Étant donnée les spécifications d'une application et de son architecture d'implantation, l'outil SynDEx fournit une répartition optimisée de l'application qui prend en compte les performances de la plate-forme de mise en œuvre. Ce partitionnement est alors utilisable pour instancier l'application à l'aide des modèles ARINC (partition, processus et services APEX). Ensuite, la phase d'évaluation de performances prévue dans la méthodologie de Polychrony interviendra pour la validation temporelle finale, mais également pour des analyses suivant d'autres métriques comme l'énergie.

*
* *

Pour terminer, nous soulignons l'existence de nombreux travaux dédiés à la conception de

systèmes *globalement asynchrones, localement synchrones* (GALS) basée sur le modèle synchrone. En particulier dans [110], Halbwachs et Baghdadi étudient une modélisation synchrone de systèmes partiellement asynchrones. Ils illustrent quelques modèles ESTEREL de mécanismes élémentaires de communication et de synchronisation (mémoire partagée et rendez-vous). Leur travail a pour but de "systématiser et populariser" l'utilisation des langages synchrones pour la description de systèmes asynchrones à des fins de simulation et de validation. Notre approche s'inscrit également dans cette démarche. De plus, elle va plus loin en modélisant des mécanismes plus complexes que ceux qui sont considérés dans [110]. Enfin, grâce à la définition d'une bibliothèque de modèles de mécanismes standards, notre travail répond à l'une des perspectives relevées par les auteurs.

8.5 Conclusion

Nous avons présenté la modélisation d'un certain nombre de composants permettant de décrire des applications distribuées temps réel. Il s'agit d'une part, de composants utilisables pour représenter des concepts d'une architecture de type IMA (à savoir, partitions et processus) ; et d'autre part, d'un ensemble de primitives permettant de modéliser les fonctionnalités d'un exécutif temps réel. La définition de ces primitives repose pour la majeure partie sur les spécifications de la norme ARINC 653. Ces composants permettent ainsi de disposer d'un modèle SIGNAL de support d'exécution (c'est-à-dire, l'ensemble des protocoles et services requis par des applications temps réel pour réaliser leurs fonctionnalités).

Les modèles définis ici offrent des descriptions auxquelles des outils et techniques formels peuvent être appliqués ; en particulier, ceux de l'environnement POLYCHRONY. La vérification de propriétés comportementales d'une application peut être effectuée en adoptant une démarche similaire à celle qui a été illustrée au chapitre 6. En outre, puisque la simulation est possible dans POLYCHRONY, le fonctionnement d'une application est observable. La validation d'un modèle de système temps réel comporte toujours une analyse de ses propriétés non fonctionnelles. En particulier, il faut être en mesure de garantir le respect de ses contraintes temporelles. Dans cette optique, la technique d'évaluation de performances mise en œuvre dans POLYCHRONY est bien adaptée pour nos modèles. Enfin, à toutes ces possibilités vient s'ajouter celle de générer automatiquement du code optimisé.

Sur le plan méthodologique, la définition de ces composants s'inscrit dans le cadre d'une approche de conception générale qui est développée dans POLYCHRONY. Elle concerne plus précisément les applications distribuées temps réel (cf. chapitre 5). L'idée fondamentale derrière l'approche consiste à utiliser le modèle sémantique du langage SIGNAL comme support de raisonnement formel, à la fois pour des aspects logiciels (i.e., fonctionnalités d'une application) et matériels (i.e., propriétés de la plate-forme d'implantation). Par exemple, un raffinement des descriptions obtenues suivant cette approche consistera en l'instanciation de certaines de leurs parties à l'aide des composants modélisés.

Enfin, l'utilisation de ces composants dépasse le cadre initial pour lequel ils ont été développés (c'est-à-dire, la modélisation d'applications avioniques suivant ARINC). Ainsi, ils ont servi de base dans la mise en œuvre d'un schéma de traduction de programmes écrits en REAL-TIME JAVA vers SIGNAL. Cela consiste d'une part, à décrire le support d'exécution de programmes JAVA et d'autre part, à représenter l'architecture fonctionnelle d'une application également écrite en JAVA. Cela est possible en raison de la similitude des services et concepts utilisés en REAL-TIME JAVA avec ceux décrits dans la norme ARINC. Nous reviendrons sur

cette étude dans le chapitre suivant, après avoir présenté la modélisation d'une application à l'aide des composants ARINC.

Chapitre 9

Cas d'étude : conception d'une application à l'aide des modèles ARINC

Ce chapitre a pour but d'illustrer comment les composants ARINC permettent de modéliser une application de taille réaliste. Une étude similaire portant sur un exemple très simple a été abordée dans [95]. La démarche qui est présentée ici s'applique donc à n'importe quelle application distribuée temps réel à modéliser selon une architecture définie en termes de partitions. Le cas d'étude auquel nous nous sommes naturellement intéressé consiste en une application avionique. Cependant pour des raisons de confidentialité, nous nous contenterons plutôt dans ce chapitre d'une analogie avec les satellites. Leur fonctionnement comporte en effet quelques aspects communs avec l'avionique. L'utilisation de l'informatique embarquée en est un. Par conséquent, la maintenance à bord des appareils est l'une des tâches sensibles. Par ailleurs, ce sont des appareils qui sont en contact permanent avec des centres de contrôle au sol. Ce qui leur permet de solliciter divers types de services sans être nécessairement au sol : des informations sur leur position courante pour la navigation, des réparations mécaniques, des ravitaillements en carburant [1], etc. Ainsi, la norme ARINC sert aussi pour la construction d'équipements pour satellites.

L'exemple que nous considérons dans la suite est une application qui est chargée d'acquérir et de traiter des données de maintenance reçues à partir d'un satellite en orbite. Nous illustrons dans la section 9.1 la démarche globale sur laquelle repose la modélisation de cette application. Nous montrons dans la même section comment on peut procéder pour évaluer ses performances dans Polychrony. Ensuite, dans la section 9.2, nous mentionnons brièvement une étude en cours reposant sur une utilisation des composants. Celle-ci a pour objectif de réaliser une traduction de programmes écrits en Real-Time Java vers le langage Signal [202]. Cela permet d'accéder aux outils et techniques formels de Polychrony pour analyser ces programmes.

1. Par exemple, dans les satellites de type *Météosat* (qui sont des satellites météorologiques géostationnaires de l'organisation européenne pour l'exploitation des satellites météorologiques - l'EUMETSA), six réacteurs propulsés par le carburant *hydrazine* sont utilisés pour le contrôle de l'orbite et de l'altitude. Ces réacteurs permettent le maintien de l'orientation du satellite dans l'espace, pour effectuer des ajustements sur orbite et pour déplacer le satellite vers un nouvel emplacement. Les réservoirs d'hydrazine sont dimensionnés pour contenir le carburant nécessaire à six ans minimum de maintenance sur orbite dans des conditions opérationnelles normales.

9.1 Modélisation d'une application temps réel

L'application que nous décrivons dans cette section s'appelle SATMAINT. Les fonctionnalités très variées de celle-ci ne permettent pas un traitement séquentiel. Elle est donc décomposée en plusieurs processus regroupés au sein d'une partition unique (ce choix a été fixé dans le cahier des charges). La partition SATMAINT fait partie d'un module qui contient d'autres partitions avec lesquelles elle coopère. Les échanges entre partitions sont effectués via des *sampling* et *queuing* ports. Ce module est quant à lui relié, via un bus Ethernet, à un périphérique assurant l'interface homme / machine de l'application SATMAINT.

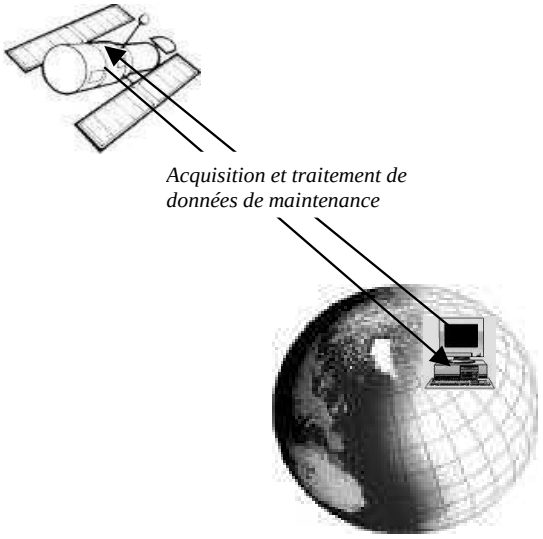


FIGURE 9.1 – Rôle de l'application SATMAINT.

Dans ce qui suit, nous nous concentrons principalement sur la partition SATMAINT pour illustrer l'utilisation de nos modèles. Avant de présenter informellement les éléments qui la composent, nous indiquons d'abord ses différentes fonctionnalités (FIG. 9.1) :

- calcul et émission, par SATMAINT vers les autres partitions du module, de paramètres généraux et des informations correspondant à la phase de maintenance (cela est fait de façon cyclique) ;
- acquisition et traitement par SATMAINT de messages de maintenance transmis par les autres partitions lorsque le satellite est en orbite ;
- communication avec un périphérique Q afin de permettre à un opérateur de visualiser les données de maintenance stockées en mémoire par l'application ;
- surveillance et émission périodique d'un rapport de maintenance.

La description globale de l'architecture de l'application est donnée dans la section ci-après. Elle présente notamment les différents composants qui constituent la partition correspondante : processus, mécanismes de communication et synchronisation et autres ressources partagées.

9.1.1 Spécification informelle de l'application

La FIG. 9.2, qui illustre l'application SATMAINT, met en évidence les interactions entre les différents processus. Les objets suivants sont distingués dans la spécification initiale :

- Huit *processus*, identifiés chacun par *Proc_i*, $i \in 1..8$ (ce découpage en processus est défini dans la spécification initiale) :

– *Proc_1* : il surveille et synthétise les défauts qui peuvent affecter l'application et son environnement puis transmet les messages associés ;
– *Proc_2* : il collecte des messages transmis lorsque le satellite est en orbite ;
– *Proc_3* : il met à jour une ressource *Message_table* qui contient les messages de panne susceptibles d'être traités ;
– *Proc_4* : il récupère des informations transmises par le périphérique Q ;
– *Proc_5* : il traite les paramètres généraux et met à jour des compteurs de temps utilisés lors des traitements de messages de pannes ;
– *Proc_6* : il gère un protocole avec le périphérique Q et transmet le contenu du rapport demandé par l'opérateur ;
– *Proc_7* : il met en mémoire les informations de maintenance ;
– *Proc_8* : il traite les pannes stockées en mémoire afin de fournir les données de maintenance à sauvegarder ou à présenter à l'opérateur.

- Cinq *buffers* notés chacun *buff_i*, où *i* dénote le numéro du processus qui reçoit ses messages dans ce *buffer*.

- Trois *événements* représentés sous la forme *evt_ij*, où *i* et *j* dénotent la paire de processus qui communiquent via cet événement.

- Enfin, un *sémaphore* noté *sema_38* qui est utilisé par les processus *Proc_3* et *Proc_8* pour accéder à la ressource *Message_table* de façon exclusive.

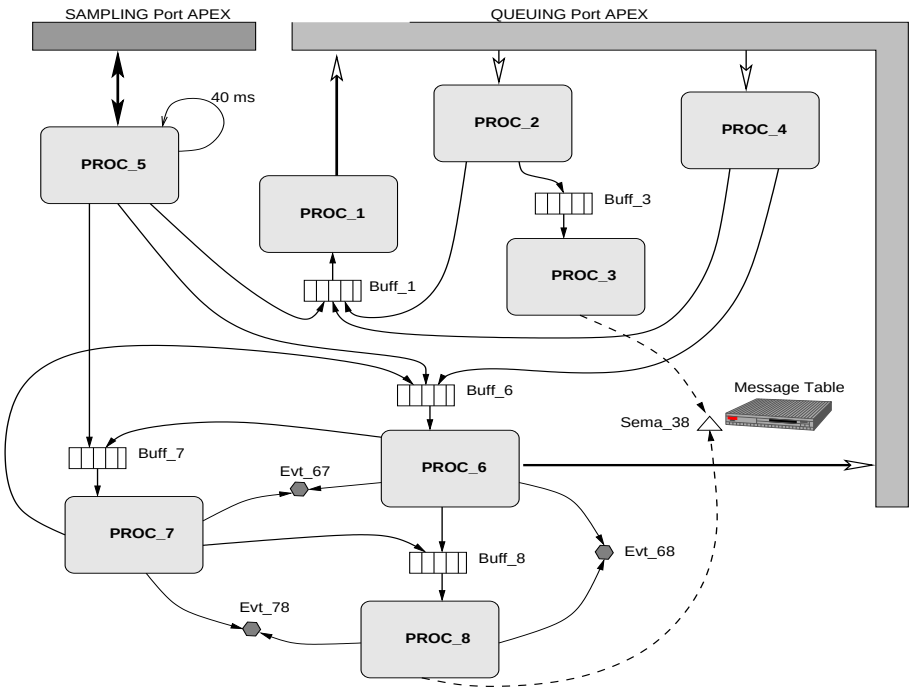


FIGURE 9.2 – Architecture de l'application SATMAINT.

La partition communique avec son environnement via des *sampling ports* et des *queuing ports*. À partir de cette spécification informelle de l'application, nous allons définir le modèle SIGNAL correspondant, en utilisant les composants présentés dans le chapitre précédent. Pour simplifier, nous ne détaillerons pas le modèle entier. En particulier, en ce qui concerne les

processus de la partition, nous ne développerons que la description du processus *Proc_8* qui est un cas emblématique (la modélisation des autres processus suit le même principe).

9.1.2 Spécification à l'aide de SIGNAL

Description de la partition. Le modèle SIGNAL de l'application est donné sur la FIG. 9.3. En entrée de la partition, deux signaux sont distingués : `Active_partition_ID` qui dénote l'activation de celle-ci (il provient du niveau supérieur, c'est-à-dire du module qui contient la partition) ; et `initialize` qui est reçu uniquement lors de la phase d'initialisation de la partition. Ces signaux apparaissent toujours dans le modèle d'une partition. Ce qui n'est pas le cas des entrées `queuing_port` et `sampling_port` via lesquelles sont réalisées les communications inter-partitions à l'intérieur du module contenant SATMAINT (ce sont des tableaux d'identificateurs de ports, respectivement de types *queuing* et *sampling*).

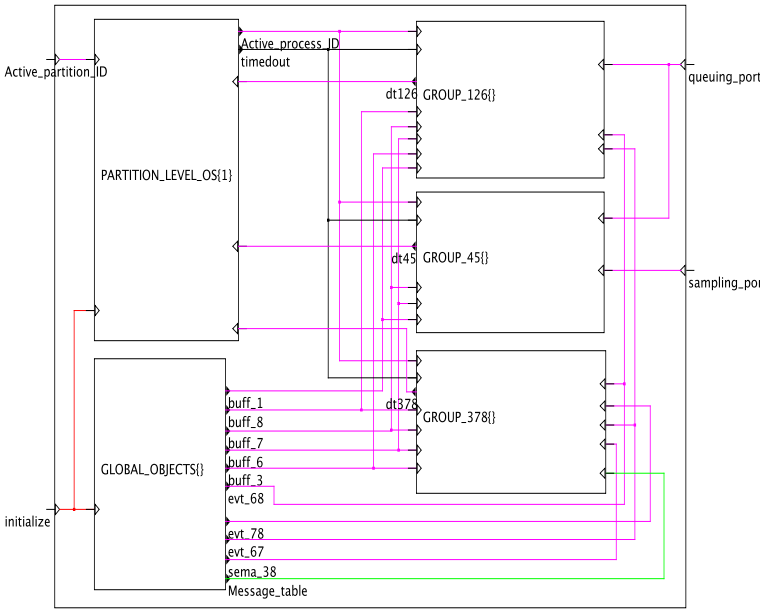


FIGURE 9.3 – Modèle SIGNAL de la partition SATMAINT.

Le composant *partition-level OS* est identifiable sur la FIG. 9.3 en haut à gauche. Il est chargé notamment de l'ordonnancement des processus de la partition. Pour cela, il leur fournit les signaux `Active_process_ID` et `timedout`, représentant respectivement l'identificateur du processus actif et les informations sur les compteurs de temps actifs. Le *partition-level OS* reçoit de la part des processus les signaux `dt126`, `dt45` et `dt378`. Ceux-ci indiquent la quantité de temps consommée à chaque activation d'un processus dans la partition. Les compteurs de temps sont mis à jour à l'aide de ces informations. Nous reviendrons plus en détail sur le contenu du composant *partition-level OS*.

Les ressources internes à la partition (mécanismes de communication et synchronisation, et *Message_table*) sont créées et initialisées au sein du composant `GLOBAL_OBJECTS` (en bas à gauche sur la FIG. 9.3). À la création de celles-ci, un identificateur unique leur est affecté. Ainsi, l'accès à chacune d'entre elles se fait via son identificateur associé. Par exemple sur la FIG. 9.3, le signal `buff_8` est utilisé pour identifier le *buffer buff_8* lors de l'appel d'un service APEX

sur ce dernier. Quant à la ressource *Message_table*, qui n'est pas un mécanisme APEX, le signal `Message_table` produit par `GLOBAL_OBJECTS` dénote simplement son adresse.

Enfin, les composants restants, `GROUP_126`, `GROUP_45` et `GROUP_378`, contiennent les modèles des processus. Chaque chiffre dans le suffixe de leur nom dénote l'identificateur d'un processus inclus. Par exemple, `GROUP_378` comprend *Proc_3*, *Proc_7* et *Proc_8*, comme cela est illustré sur la Fig. 9.4. La raison de cette répartition des processus en sous-groupes est essentiellement liée au besoin de structuration. Par ailleurs, une telle démarche favorise des descriptions modulaires, offrant ainsi une solution aux problèmes dus à la modélisation d'applications à grande échelle.

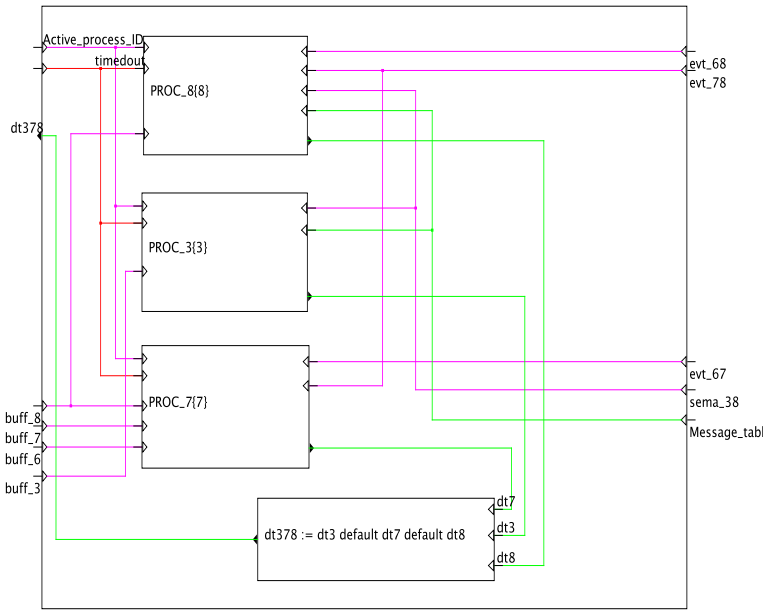


Figure 9.4 – Regroupement de processus dans `GROUP_378`.

Description du noyau de l'exécutif. Le contrôle des ressources de la partition est représenté au sein de `PARTITION_LEVEL_OS` dans le modèle de la Fig. 9.3. Ce composant est décrit essentiellement à l'aide de services APEX, combinés avec les services supplémentaires requis pour décrire l'ordonnancement et la gestion des compteurs de temps.

La Fig. 9.5 illustre partiellement le contenu de `PARTITION_LEVEL_OS`. À la réception du signal `initialize` (i.e., ce qui correspond à la phase d'initialisation de la partition), les attributs des processus [2] sont d'abord définis (cf. équation `(a)`). Ensuite, ils sont créés et mis dans l'état "prêt". Cela est fait en utilisant respectivement les modèles de services APEX `CREATE_PROCESS` (cf. ligne `(b)`) et `START` (cf. ligne `(d)`). D'autre part, la partition est placée dans le mode *normal*, qui permet d'exécuter les processus (cf. ligne `(c)`).

La première réception du signal `Active_partition_ID` démarre l'exécution à proprement parler de la partition. Ce signal identifie la partition active, choisie par le *module-level OS*. Lorsqu'il dénote la partition courante, le signal `is_running`, qui est de type événement, devient présent (cf. équation `(e)`). À chaque activation de la partition (i.e., lorsque le signal `is_running`

2. Remarquer le nombre de processus qui est neuf au lieu de huit (0..8). Un processus servant de tâche de fond a été ajouté.

```
(| (att0,...,att8) := GET_PROCESSES_ATTRIBUTES{}(when initialize)            (a)
 | (pid0,return_code0) := CREATE_PROCESS{}(att0 when initialize)             (b)
 | ...
 | (pid8,return_code8) := CREATE_PROCESS{}(att8 when initialize)
 | return_code9 := SET_PARTITION_MODE{}(#NORMAL when initialize)             (c)
 | return_code10 := START{}(pid0)                                           (d)
 | ...
 | return_code18 := START{}(pid8)
 | is_running := when (Active_partition_ID = Partition_ID)                  (e)
 | diagnostic := PROCESS_SCHEDULINGREQUEST{}(is_running)                    (f)
 | (Active_process_ID,status,valid) := PROCESS_GETACTIVE{}(is_running)      (g)
 | timedout := UPDATE_COUNTERS{}(dt126 default dt45 default dt378)          (h)
 | partition_switching := when ((Active_partition_ID$1) = Partition_ID)
            when (not (Active_partition_ID = (Active_partition_ID$1)))     (i)
 | return_code19 := SET_PARTITION_MODE{}(#IDLE when partition_switching)    (j)
 |)
```

FIGURE 9.5 – Description du *partition-level OS*.

est présent), un ré-ordonnancement de processus est systématiquement effectué afin de désigner le processus actif dans la partition (cf. ligne `(f)`). Comme nous l'avons mentionné à la section 8.3.3 du chapitre 8, chaque activation du modèle d'une partition correspond à l'exécution atomique d'un bloc du processus actif. Grâce au ré-ordonnancement effectué avant chaque exécution de bloc, la préemptibilité des processus se situe ainsi au niveau des blocs. D'autre part, des ré-ordonnancements peuvent également survenir lors de l'exécution d'un bloc contenant un service APEX (cf. code SIGNAL de *read_blackboard* en annexe A). Sur la FIG. 9.5, l'identificateur du nouveau processus actif, le signal `Active_process_ID`, est récupéré à l'horloge d'activation de la partition (c'est-à-dire, à l'horloge de `is_running`), via l'appel du service `PROCESS_GETACTIVE` (cf. ligne `(g)`). Il est transmis aux processus de la partition, qui le consulteront et s'exécuteront s'ils sont désignés par le signal. Les compteurs de temps actifs sont mis à jour à la réception des signaux `dt126`, `dt45` et `dt378`, qui indiquent le temps consommé par l'exécution des blocs des processus dans la partition (cf. ligne `(h)`).

Enfin, lorsque le temps d'exécution alloué à la partition au moment de la configuration arrive à son terme (ce temps est géré au niveau du *module-level OS* qui n'est pas représenté ici), celle-ci bascule en mode *arrêt* (cf. équation `(j)`). Cela est reflété à travers le signal-événement `partition_switching`. Celui-ci est défini aux instants où l'entrée identifiant la partition active change alors que la partition courante était active juste avant (cf. équation `(i)`).

Description des processus. Pour illustrer la modélisation des processus, nous considérons *Proc_8*. Le modèle SIGNAL associé à ce processus est défini en s'appuyant sur la description suivante que nous avons extraite de la spécification informelle initiale. C'est une représentation simplifiée qui fait ressortir surtout l'enchaînement des actions effectuées par le processus :

surveiller le buffer buff_8 et récupérer les messages stockés ;
if *un message est récupéré* ***then***
  effectuer les actions A1 ;
  réquisitionner le sémaphore sem_38 ;
  effectuer les actions A2 (requièrent un accès à la ressource Message_Table) ;
  restituer le sémaphore sem_38 ;
  effectuer les actions A3 ;
  if *le message reçu dénote une demande d'un opérateur* ***then***
    positionner l'événement evt_68 ;
  else if *le message dénote une demande provenant du satellite* ***then***
    positionner l'événement evt_78 ;
  end if
end if

Le modèle associé à *Proc_8* est donné sur la FIG. 9.6. La définition de ses parties *CONTROL* et *COMPUTE* repose d'une part, sur la connaissance de l'ensemble des actions qu'il effectue ; et d'autre part, sur la façon dont s'enchaînent les exécutions de ces actions.

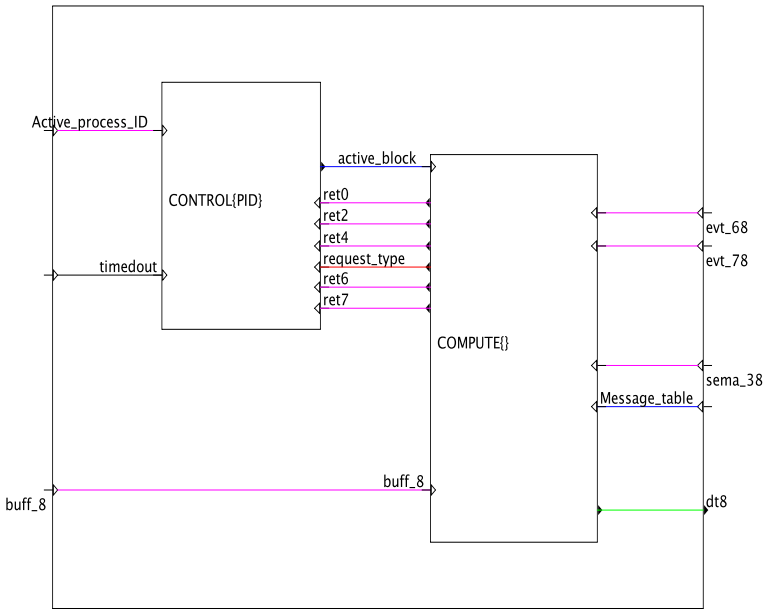
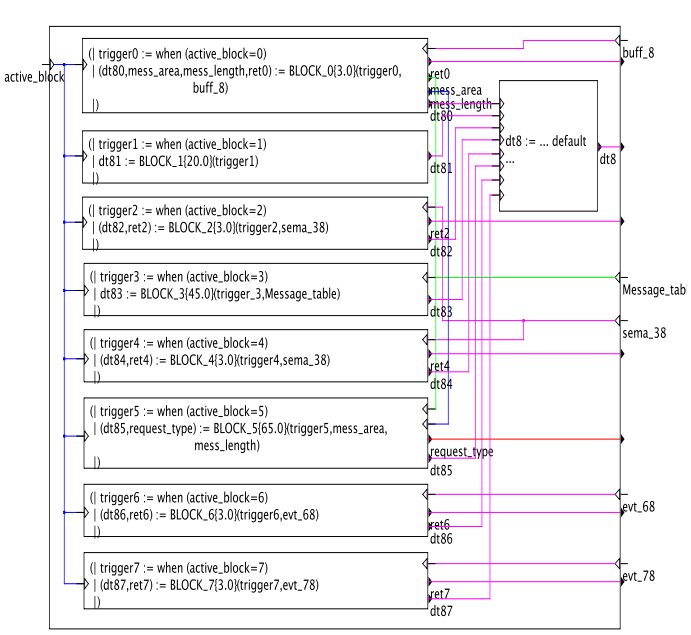


FIGURE 9.6 – Modèle du processus `PROC_8`.

Nous avons effectué un premier travail consistant à mettre la spécification informelle du processus sous une forme [3] permettant de dégager le contrôle du processus. Nous avons également identifié les points de communication du processus qui sollicitent des mécanismes APEX. Les actions effectuées entre ces points peuvent être vues comme étant "locales" au processus. Ainsi, pour chaque point de communication, nous associons un bloc dans la partie *COMPUTE*. Quant aux actions locales, elles peuvent être représentées à l'aide d'un nombre quelconque de blocs.

3. Ici, nous avons choisi une forme impérative pour décrire informellement le fonctionnement de *Proc_8*. Ce choix nous semble naturellement justifié par notre vision du fonctionnement du modèle d'un processus ARINC tel que nous l'avons défini au chapitre 8 (à savoir, une exécution séquentielle de blocs d'instructions). La forme impérative permet de distinguer d'une part, les actions à répartir en blocs, et d'autre part, l'enchaînement des exécutions de ces blocs. Néanmoins, l'on aurait pu envisager une spécification informelle de *Proc_8* sous une forme non impérative, par exemple dans un style déclaratif. Dans ce cas, la définition du modèle d'un processus nous paraît moins immédiate.

Ici pour simplifier, nous utilisons un seul bloc pour chaque sous-ensemble d'actions effectuées entre deux points de communication. L'enchaînement des exécutions de blocs (déduit de la spécification informelle) est alors encodé sous forme d'un système de transition dans *CONTROL*.




```
process BLOCK_0 =
  { SystemTime_type wcet; }
  (? event trigger0;
     Comm_ComponentID_type buff_8;
   ! SystemTime_type dt80;
     MessageArea_type mess_area;
     MessageSise_type mess_length;
     ReturnCode_type ret0;
  )
 (| (mess_area, mess_length, ret0) :=
      RECEIVE_BUFFER{8}((var buff_8)
                        when trigger0, 500.0)
  | dt80 := wcet when trigger0
  |)

process BLOCK_1 =
  { SystemTime_type wcet; }
  (? event trigger1;
   ! SystemTime_type dt81;
  )
 (| Actions1{}(when trigger1)
  | dt81 := wcet when trigger1
  |)
  ...
```

FIGURE 9.7 – Partie `COMPUTE` du processus `PROC_8`.

Le modèle `COMPUTE` qui est décrit sur la FIG. 9.7 reflète exactement les actions de la spécification informelle de *Proc_8*. Le découpage considéré conduit à huit blocs. À la réception du signal d'entrée `Active_block` identifiant le bloc à exécuter, des actions sont effectuées : si la valeur de `Active_block` vaut k, l'événement `triggerk` devient présent et provoque l'activation du k^{ieme} bloc dans la partie *COMPUTE*. Ici, l'exécution des blocs se fait séquentiellement du haut vers le bas. Elle est contrôlée par l'automate de la FIG. 9.8 (vu en quelque sorte comme un séquenceur). Son état initial indique le premier bloc (numéroté 0). Celui-ci contient un appel au service APEX *receive_buffer* (cf. FIG. 9.7 à droite). D'autre part, chaque appel de bloc est paramétré par le temps d'exécution au pire cas des actions qui lui sont associées. Par exemple, l'exécution du service APEX dans le premier bloc consomme trois unités de temps[4]. Ces informations peuvent être supposées connues *a priori* comme dans les modèles temporisés. Cependant, comme nous l'avons indiqué précédemment, il est aussi possible de les déterminer "en ligne" sous POLYCHRONY en adoptant le schéma de co-simulation illustré dans le chapitre 4, pour l'évaluation de performances. Dans ce cas, au sein de chaque bloc, on composera les processus SIGNAL représentant les actions et leurs interprétations temporelles. Chaque signal `dtk` exprimera alors la latence observée suite à l'activation du k^{ieme} bloc.

Dans l'appel au service *receive_buffer* réalisé dans le premier bloc, les paramètres d'entrée sont l'identifiant du *buffer* (`buff_8`) et une durée correspondant au *time-out* (`500.0`). Un nouvel opérateur du langage SIGNAL est utilisé. Il s'agit de la *mémoire dépendant du contexte* (une variante de l'opérateur de mémoire (`cell`) :

– Cet opérateur permet d'obtenir la dernière valeur mémorisée d'un signal à l'horloge du contexte d'utilisation. La principale différence avec l'opérateur `cell` réside dans le fait

4. Ici, ces valeurs ne sont pas significatives. Elles servent juste pour le test du modèle.

que les horloges des signaux impliqués ne sont pas liées *a priori*. Ainsi dans `y := (var x) init c`, le signal `y` prend la valeur portée par `x` aux instants où l'environnement rend `y` disponible. La constante `c` sert de valeur d'initialisation.

Les deux premiers paramètres de sortie (`mess_area` et `mess_length`) dénotent les adresses et tailles des messages récupérés. Le diagnostic de la requête est obtenu à travers la dernière sortie (`ret0`). Le temps consommé par cet appel est égal à la valeur du paramètre `wcet`. Le second bloc correspond à des actions locales (appelées *A1* dans la spécification informelle). Il est programmé de façon analogue au premier bloc. De façon générale, tous les autres blocs sont définis suivant ces schémas.

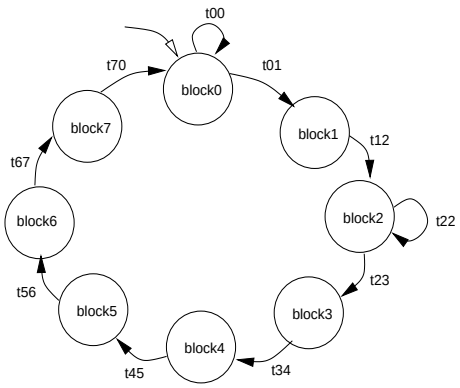


FIGURE 9.8 – Automate encodé dans la partie *CONTROL* de `PROC_8`.

Au sein de l'automate encodé dans la partie *CONTROL* (cf. FIG. 9.8), nous distinguons deux types de transitions : celles qui sont étiquetées `tij` (`i` $\neq$ `j`), qui expriment simplement un passage du bloc `i` au bloc `j` ; et celles dont l'étiquette est `tii`, qui dénotent le fait que certains blocs peuvent parfois nécessiter des exécutions consécutives. Ceci est spécifique aux blocs contenant un appel à un service bloquant. Par exemple, c'est le cas du premier bloc qui réalise des lectures de messages dans *buff_8*. Si le *buffer* est vide lors de la première tentative de lecture, le code retour `ret0` qui doit être envoyé au *CONTROL* est absent. Dans ce cas, *Proc_8* est suspendu jusqu'à l'envoi d'un message dans *buff_8*. Il reprendra son exécution plus tard à partir du bloc ayant entraîné sa suspension pour récupérer le message stocké dans le *buffer*. En fait, ces exécutions consécutives d'un même bloc correspondent en général à l'exécution des deux sous-ensembles d'actions locales à un service APEX (cf. chapitre 8). Cependant, il existe des exceptions à cette règle telles que le service *read_blackboard*, dont les actions locales forment un ensemble indivisible, pouvant nécessiter parfois plus de deux exécutions consécutives.

9.1.3 Analyse du modèle

À présent que nous avons défini le modèle de `SATMAINT` en SIGNAL, plusieurs types d'analyse peuvent être effectués sur celui-ci. Par exemple, on peut étudier ses propriétés comportementales : le compilateur de POLYCHRONY se charge de l'analyse statique (par exemple, vérifier que le programme ne contient pas de définitions cycliques ou d'horloges nulles) ; les propriétés dynamiques quant à elles, peuvent être étudiées à l'aide de l'outil SIGALI comme cela a été illustré dans le chapitre 6 (par exemple, étudier l'accessibilité de certains états du système dynamique défini par le programme).

Ici, nous nous intéressons surtout à l'étude des caractéristiques temporelles de l'application.

Pour cela, nous utilisons la technique d'évaluation de performances mise en œuvre dans POLYCHRONY (cf. chapitre 4). Celle-ci repose sur la dérivation d'un autre programme SIGNAL (servant d'observateur), qui représente l'interprétation temporelle du programme correspondant à l'application. Grâce à la co-simulation des deux programmes, on obtient des informations de latence qui permettent par exemple, de déterminer des temps d'exécution au pire cas.

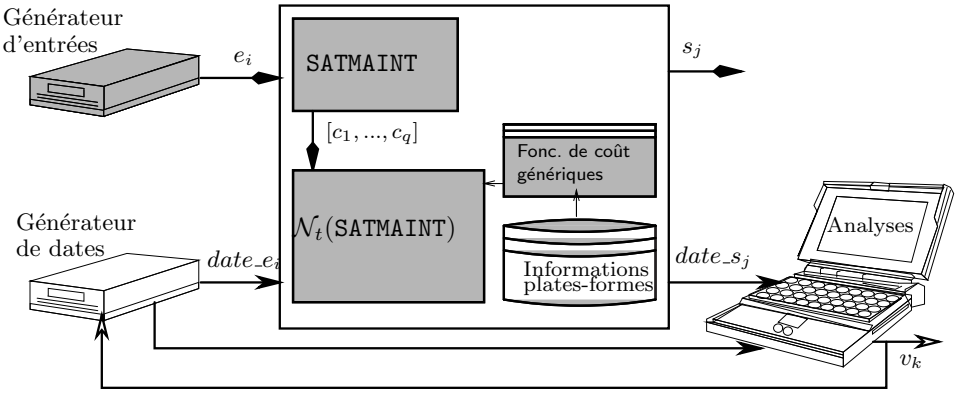


FIGURE 9.9 – Co-simulation du modèle de SATMAINT et son interprétation temporelle.

Nous considérons que le modèle utilisé pour l'évaluation temporelle ci-après est décrit en SIGNAL, avec en plus des appels à des services externes. Par contre, nous ne supposons aucun exécutif (de type ARINC) avec lequel le lien sera fait lors de la génération de code (cf. discussion en section 8.4.1 du chapitre 8). Ainsi, le temps d'exécution de chaque appel à un service APEX est déterminé de la même façon que celui des autres traitements (c'est-à-dire, à partir de son interprétation temporelle). Le schéma global de mise en œuvre pour la co-simulation est facilement déduit de celui que nous avons illustré au chapitre 4. Il est rappelé sur la FIG. 9.9, où $\mathcal{N}_t$(SATMAINT) représente l'interprétation temporelle de SATMAINT. Grâce à la co-simulation des deux programmes, on peut par exemple observer les temps de latence calculés à chaque activation de la partition. La mise en œuvre effective de ce schéma de co-simulation soulève une difficulté principale. Celle-ci est liée à une question de passage à l'échelle qui se pose à la compilation dès lors que la taille de l'application étudiée est importante (la recompilation du programme issu de la composition de l'application et de son interprétation peut tout simplement échouer). La solution retenue pour résoudre le problème consiste à utiliser un schéma [5] modulaire d'évaluation que nous présentons ci-après. Par ailleurs, nous pouvons souligner d'autres avantages à la modularité comme la possibilité d'adopter une démarche par composants, d'où la réutilisation.

Un schéma modulaire pour l'évaluation. Le caractère modulaire du langage SIGNAL permet de construire un programme à partir d'autres programmes par composition. Ce principe s'applique donc à la construction du modèle temporel d'une application telle que SATMAINT qui a une taille importante. Ainsi, nous considérons un découpage de la partition en différentes parties de taille raisonnable, pour lesquelles nous définirons d'abord le modèle temporel associé (par exemple, une partie peut être un processus ARINC ou le *partition-level OS*). Ensuite, ce modèle sera composé avec la partie à laquelle il correspond. Le programme résultant sera abstrait dans le but de le simplifier (en prévision de la réduction de la taille du programme final considéré pour la simulation). Enfin, le modèle global correspondant à la composition de l'application et

5. Ce schéma a servi de base pour des expérimentations dans le cadre de SAFEAIR.

de son interprétation est obtenu en composant les programmes abstraits.

Si $\texttt{P} \equiv \texttt{P}_1 \texttt{ | ... | } \texttt{P}_n$ dénote l'application à analyser, nous cherchons donc à définir le programme $\texttt{P'} \equiv \texttt{P'}_1 \texttt{ | ... | } \texttt{P'}_n$ qui sera utilisé pour la simulation en vue de l'évaluation temporelle (i.e., le programme qui équivaut à $\texttt{(| P |} \mathcal{N}_t(\texttt{P}) \texttt{ |)}$). Chaque $\texttt{P'}_i$ dénote la composition du sous-processus $\texttt{P}_i$ de P, avec son interprétation temporelle $\mathcal{N}_t(\texttt{P}_i)$. Nous distinguons alors les étapes suivantes pour la réalisation des expérimentations :

1. *Définition partielle d'interprétations temporelles* : pour chaque sous-processus $\texttt{P}_i$ ($i \in \{1, ..., n\}$) de P, on définit le modèle temporel associé $\mathcal{N}_t(\texttt{P}_i)$.

2. *Composition de chaque partie avec son modèle temporel puis abstraction* : cette étape définit les parties $\texttt{P'}_i$ ($i \in \{1, ..., n\}$) formant le programme simulé P' qu'on cherche à construire. L'abstraction a pour but de transformer les programmes résultants de façon à diminuer leur taille notamment. On a ainsi : $\texttt{P'}_i \equiv \alpha(\texttt{P}_i \texttt{ |} \mathcal{N}_t(\texttt{P}_i))$.

3. *Construction du programme global pour la simulation* : celui-ci résulte de la composition des $\texttt{P'}_i$ obtenus à l'étape précédente, autrement dit : $\texttt{P'} \equiv \texttt{P'}_1 \texttt{ | ... | } \texttt{P'}_n$.

Il convient de remarquer que le schéma ci-dessus peut aussi être récursif. Dans un programme représenté par $\texttt{P} \equiv \texttt{P}_1 \texttt{ | ... | } \texttt{P}_n$, lorsque les sous-processus $\texttt{P}_i$ ($i \in \{1, ..., n\}$) sont de taille non négligeable, nous pouvons également leur appliquer le même schéma pour définir les sous-processus $\texttt{P'}_i$ ($i \in \{1, ..., n\}$) correspondants. La composition de ces derniers donne alors le programme P', utilisé pour la simulation en vue de l'évaluation temporelle.

*
* *

Par ailleurs, nous mentionnons l'existence d'un autre schéma possible pour l'évaluation temporelle. Il est décrit par le processus SIGNAL suivant :

```
(|  step :: (| α(P1) | ... | α(Pn) |)
 |  latency :: (| Nt(P1) | ... | Nt(Pn) |)
 |  step --> latency  |)
```

Ici, le symbole "::" permet d'affecter une *étiquette* à un processus. Par exemple, l'étiquette `step` désigne le processus qui consiste à composer des abstractions de sous-processus du programme P introduit ci-dessus. Quant à `latency`, elle correspond à la construction du modèle temporel de P en composant les modèles temporels associés à ses sous-processus. Ce schéma, qui est aussi modulaire, distingue ainsi deux phases. L'enchaînement de ces dernières est exprimé à travers la contrainte de précédence spécifiée entre leurs étiquettes associées, au sein du processus SIGNAL.

Le programme global utilisé pour la simulation en vue de l'évaluation temporelle est obtenu en composant les programmes issus des deux phases : d'une part, une abstraction [6] de l'application (phase `step`), et d'autre part, l'interprétation temporelle de celle-ci (phase `latency`). Ces deux programmes sont décrits séparément. C'est une première différence avec le schéma introduit précédemment. En outre, ici, on n'abstrait que le programme représentant l'application contrairement à ce qui est fait dans le premier schéma, où on peut aussi abstraire les modèles

6. Par exemple, la partie contrôle du programme décrivant l'application peut suffire pour effectuer sa co-simulation avec le modèle temporel associé. Dans ce cas, α représente l'abstraction par le contrôle (cf. section 4.3.2 au chapitre 4).

temporels. Ce qui permet de réduire davantage la taille du programme global utilisé pour la simulation. Par conséquent, le schéma précédent est mieux adapté pour des programmes de taille importante.

Application du schéma à SATMAINT. Nous adoptons le premier schéma pour l'évaluation temporelle. Pour appliquer celui-ci à la partition SATMAINT, nous devons d'abord choisir le niveau de granularité pour le découpage de celle-ci en parties. En l'occurrence, si on se place au niveau des blocs (constituant chaque modèle de processus ARINC), les parties sont plus fines que lorsqu'on est au niveau des processus (puisque ces derniers sont constitués de blocs). Lorsque les parties sont de taille non négligeable, nous pouvons toujours procéder de façon incrémentale pour décrire leur modèle temporel. Par exemple dans le processus PROC_8, nous commencerons par déterminer l'interprétation de la partie COMPUTE :

$$\mathcal{N}_t(\texttt{COMPUTE}) = \mathcal{N}_t(\texttt{BLOCK\_0}) \mid ... \mid \mathcal{N}_t(\texttt{BLOCK\_7})$$

Dans cette composition, nous supposons que le modèle temporel de chaque bloc est obtenu sans avoir besoin de considérer une décomposition de celui-ci comme c'est le cas pour les niveaux supérieurs (exemples : processus, partition). Nous déterminons de même l'interprétation de la partie CONTROL du processus. En composant les interprétations des parties COMPUTE et CONTROL, on en déduit celle de PROC_8 :

$$\mathcal{N}_t(\texttt{PROC\_8}) = \mathcal{N}_t(\texttt{COMPUTE}) \mid \mathcal{N}_t(\texttt{CONTROL})$$

On procède ainsi pour toutes les autres parties de la partition (processus ARINC, *partition-level OS*). À l'étape suivante, chacune d'entre elles est composée avec son modèle temporel. Pour PROC_8, on obtient le processus suivant :

$$\texttt{(| PROC\_8 |} \; \mathcal{N}_t(\texttt{PROC\_8}) \; \texttt{|)}$$

Ce processus peut être simplifié en considérant des approximations (ou abstractions) de sa partie "interprétation temporelle". Cela a pour conséquence de réduire le coût de la compilation du programme global. Comme exemple, prenons le cas d'une fonction qui effectue un calcul complexe [7] nécessitant un temps constant δ sur une architecture matérielle cible (exemple : produit de deux matrices). Son modèle temporel associé peut se ramener à l'ajout de la constante δ aux dates de disponibilité des entrées pour déterminer celles des sorties. Il en résulte donc une interprétation nettement plus simple que celle qui est formée par composition des modèles temporels de chaque opération intermédiaire effectuée à l'intérieur de la fonction (à savoir, des additions et multiplications principalement). Une autre simplification consiste à considérer des modèles temporels "au pire cas" de sous-programmes [67] [177]. Par exemple, supposons au sein d'une fonction, des calculs parallèles n'ayant pas *a priori* le même temps d'exécution. Un modèle temporel "au pire cas" de cette fonction est obtenu en choisissant comme durée d'exécution associée, le plus grand délai engendré par les calculs.

Les approximations jouent aussi un rôle important dans la définition de l'interprétation temporelle de certaines constructions du langage. La bibliothèque des fonctions de calcul de coûts d'exécution, dans son état actuel, prend en compte surtout des opérations sur des types de données simples comme les entiers, les booléens ou les réels (par exemple, le coût d'une addition de deux entiers ou de deux réels). Une amélioration de celle-ci est en cours afin de pouvoir traiter des opérations sur des structures plus complexes (par exemple, les tableaux de processus mis en œuvre par la construction array, dont on aura besoin lorsque la version de

7. Par "complexe", nous sous-entendons un calcul nécessitant un grand nombre d'opérations élémentaires.

la bibliothèque en cours de développement sera terminée). Cela est nécessaire pour avoir une meilleure estimation du coût des opérations effectuées par l'ordonnanceur des processus d'une partition. Une solution consiste à adopter des modèles définis sous forme d'abstractions. Par ailleurs, nous avons déjà mentionné le fait que certains services APEX ont été spécifiés au moyen d'abstractions dans la première version de notre bibliothèque de composants (i.e. celle qui est actuellement opérationnelle). Les modèles temporels associés à ces services peuvent eux-aussi être décrits à l'aide d'abstractions. À ce propos, diverses solutions sont envisageables. On peut attribuer à un service une valeur constante comme coût d'exécution. On peut également lui associer un modèle de fonction simulant son comportement temporel.

L'ultime étape dans la construction du modèle global de l'application pour la simulation consiste à composer les descriptions résultant de l'étape 2. Il en résulte alors le programme qui sera utilisé pour la simulation (c'est-à-dire `(| SATMAINT |` $\mathcal{N}_t$`(SATMAINT) |)`) suivant le schéma illustré sur la Fig. 9.9. Ainsi, en combinant la modularité et les abstractions, nous parvenons à résoudre les problèmes que pose la compilation d'applications de taille importante pour l'évaluation temporelle. Cette démarche non monolithique s'applique également pour des évaluations basées une métrique différente.

* * *

En l'état actuel des expérimentations, nous avons d'abord travaillé sur des programmes Signal de petite taille, sur lesquels des tests ont été effectués avec succès. Parmi ceux-ci, nous mentionnons l'exemple simple de la partition présentée dans [95] (appelée `ON_FLIGHT`) dont la spécification est similaire à celle de `SATMAINT`. Cela permet de tirer des enseignements préliminaires avant d'appliquer le même schéma à `SATMAINT` pour son évaluation temporelle.

9.2 Réalisation d'une passerelle Real-Time Java vers Signal

Le travail présenté dans cette section concerne la réalisation d'une passerelle permettant une traduction vers Signal de programmes développés en Real-Time Java [202] [203]. L'objectif consiste à étudier des transformations en vue d'optimisations de tels programmes, en se basant sur la plate-forme Polychrony. En effet, cette dernière, par les fonctionnalités qu'elle offre, facilite l'activité de conception : vérification et validation ; mais également, certaines optimisations inaccessibles avec des outils de compilation classique (par exemple, élimination de tests inutiles ou de code mort par l'inférence d'un flot de contrôle canonique), qui permettent la mise en œuvre de transformations globales d'applications.

L'étude s'inscrit dans le cadre du projet RNTL Expresso [8] dont l'objectif est de réaliser un atelier comprenant les composants nécessaires à un environnement de développement d'applications temps réel Java, pour des systèmes critiques enfouis ou embarqués, et susceptibles d'être certifiés au sens de standards tels que le DO-178B [9]. Dans cette étude, nous nous sommes restreints à un sous-ensemble fonctionnel de Real-Time Java, basé sur la spécification appelée *Ravenscar High Integrity Profile* [141]. Dans un tel cadre, un programme Real-Time Java est constitué d'un programme principal, effectuant l'allocation et l'initialisation des données et la création des tâches avant de passer en mode nominal (le nombre de tâches temps réel est fixe). Ces tâches sont soit périodiques, soit sporadiques. Enfin, la synchronisation entre tâches pour l'accès aux données partagées s'effectue au moyen de *moniteurs*.

8. `http://www.telecom.gouv.fr/rntl/FichesA/Expresso.htm`
9. `http://www.rtca.org/`

9.2.1 Présentation globale de l'approche

Le modèle de programmation supporté par les classes d'extension temps réel considérées dans Expresso se prête bien à une modélisation dans la plate-forme POLYCHRONY. En effet, les concepts utilisés peuvent être facilement représentés à l'aide des composants ARINC, décrits en SIGNAL. Le principe retenu pour cette modélisation est résumé par la FIG. 9.10.

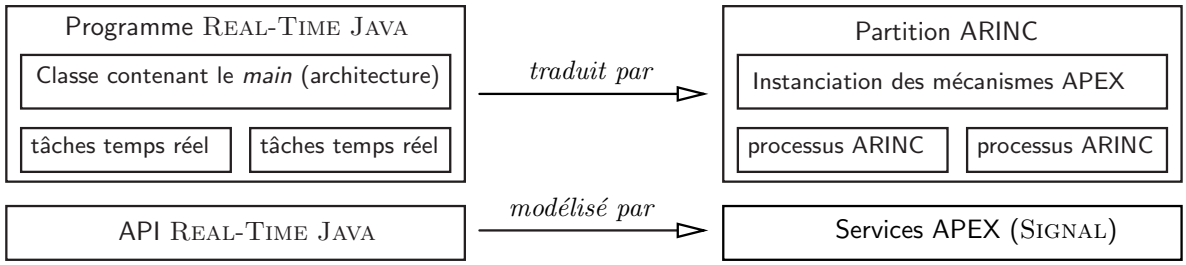


FIGURE 9.10 – Passerelle REAL-TIME JAVA - SIGNAL.

La démarche globale consiste d'une part, à modéliser le support d'exécution de l'application considérée (c'est-à-dire, l'ensemble des protocoles et services ne faisant pas partie des fonctionnalités de l'application) et d'autre part, à abstraire les fonctionnalités de l'application.

Le support d'exécution considéré ici est représenté par la machine virtuelle de JAVA (JVM - JAVA *Virtual Machine*) au-dessus de laquelle une couche de services a été ajoutée. Cette couche est en fait une API REAL-TIME JAVA définie par [141] (elle est constituée de primitives offertes par les classes *Ravenscar* d'extension temps réel). Cette API comporte des fonctionnalités proches de celles que nous avons déjà définies en SIGNAL, dans le cadre des spécifications ARINC (par exemple, par rapport à la gestion des tâches, à la communication à travers des événements, à la synchronisation à l'aide de sémaphores). Par contre, les aspects liés à la gestion mémoire ne sont pas traités dans cette étude [10]. L'utilisation de la bibliothèque de services APEX trouve donc naturellement sa place dans la modélisation en SIGNAL du support d'exécution des applications visées.

Pour décrire les fonctionnalités de l'application, on analyse le code JAVA associé afin d'inférer l'architecture de celle-ci. Les informations requises sont reflétées au sein du programme principal de l'application : paramètres temps réels, allocation et initialisation des ressources partagées et autres données, création des tâches à activer en mode nominal. L'analyse de ce programme permet ainsi de dégager la structuration globale de l'application (les tâches qui la composent, les données partagées par ces tâches, et les mécanismes de communication et de synchronisation). Au moyen de ces informations, les services servant à modéliser le support sont alors instanciés avec les paramètres appropriés pour exécuter l'application. Si cette dernière est représentée par une partition, cette phase correspond à la définition du *partition-level OS* et du nombre de processus contenus dans la partition (c'est le "socle" du modèle SIGNAL de l'application considérée). Enfin, le modèle de chacune des tâches temps réel est obtenu grâce à une analyse du code associé, basée sur un format intermédiaire en code à 3 adresses, appelé JIMPLE [208].

La transformation JIMPLE du code source (JAVA) de la classe mettant en œuvre une tâche

10. Les aspects se rapportant à la gestion mémoire ont été étudiés par le laboratoire Verimag, qui est un des participants au projet Expresso.

est réalisée à l'aide de l'outil *open-source* SOOT [11]. Celui-ci permet de mettre en œuvre des optimisations efficaces, et surtout offre une représentation intermédiaire très simple de *byte code*. L'outil SOOT est la base d'un logiciel d'importation automatique de code JAVA, qui a été mis en œuvre dans le cadre du projet Expresso. À partir d'une classe JAVA représentant une tâche temps réel, ce logiciel produit un modèle SIGNAL correspondant. Cela est réalisé en découpant le code JAVA associé à la tâche en sections critiques : une section critique est délimitée soit par un accès à une ressource partagée, soit par l'activation d'une exception, soit par une boucle interne. Le contrôle de la tâche est ainsi décomposé en blocs atomiques que l'on pourra ensuite recomposer très facilement suivant l'architecture cible souhaitée.

Le modèle que nous avons proposé pour les processus ARINC convient parfaitement pour représenter les tâches temps réel issues des programmes REAL-TIME JAVA considérés. Pour chacune d'entre elles, le logiciel d'importation produit un modèle de processus ARINC dont la partie *COMPUTE* est formée par les blocs issus du découpage de la tâche, tandis que la partie *CONTROL* décrit simplement l'enchaînement de l'exécution des blocs.

L'idée centrale de l'approche présentée ici rejoint celle de la méthodologie présentée au chapitre 5, à savoir la séparation des aspects purement fonctionnels et des aspects liés au support d'exécution, pour une application qu'on souhaite modéliser (la spécification de tous ces aspects étant à la charge de l'utilisateur). Les composants que nous avons définis ont servi dans la description de chacun de ces aspects : les partitions et les processus ont permis de représenter l'architecture fonctionnelle des programmes étudiés, tandis que les services APEX ont permis de modéliser les fonctionnalités du support d'exécution.

9.2.2 Premières expérimentations

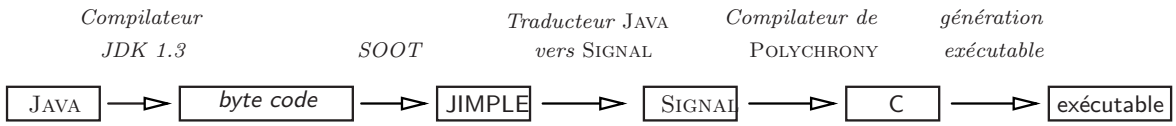


FIGURE 9.11 – Chemin de compilation selon la passerelle REAL-TIME JAVA - SIGNAL.

La démarche présentée ci-dessus offre un vaste champ d'investigation pour la vérification, la validation et l'optimisation globale d'applications REAL-TIME JAVA. Les premières expérimentations concernent l'étude de l'impact sur le chemin de compilation résultant de cette traduction [70]. Celui-ci est donné sur la FIG. 9.11, où les formats des fichiers sont représentés dans les rectangles et les outils mentionnés en étiquettes des flèches. Des programmes de petite taille, écrits en REAL-TIME JAVA ont été considérés pour des tests. Cela a permis d'observer que des optimisations dues au compilateur de POLYCHRONY permettent d'améliorer les performances de calcul des programmes traduits suivant l'approche, tout en préservant leurs fonctionnalités. Néanmoins, si ces résultats sont encourageants, ils restent à confirmer notamment sur des programmes de tailles plus significatives.

9.3 Conclusion

Ce chapitre a permis de confronter les composants ARINC définis dans POLYCHRONY à leur utilisation effective dans des applications. Ils ont servi en premier lieu à modéliser une application

11. The SOOT project (Mc Gill University) `http://www.sable.mcgill.ca/soot`.

temps réel suivant une architecture de type modulaire intégré. L'application considérée consiste en une partition unique, dont les principaux éléments (processus, mécanismes APEX pour la communication et la synchronisation) sont modélisées à l'aide des composants. Sur le modèle SIGNAL correspondant, des vérifications et analyses peuvent être effectuées à l'aide des outils et techniques accessibles dans l'environnement POLYCHRONY. En particulier, nous avons insisté sur l'estimation des temps d'exécution qui est importante pour la validation du modèle. Pour l'instant, la mise en œuvre non achevée de certaines bibliothèques utilitaires ne nous permet d'avoir encore que des résultats préliminaires. Un travail est en cours pour obtenir des résultats plus poussés. D'autre part, nous avons mentionné une étude récente dont l'objectif est la mise en œuvre d'une passerelle entre les langages REAL-TIME JAVA et SIGNAL. Ce travail permet d'avoir un modèle polychrone de programmes initialement écrits dans un sous-ensemble fonctionnel de REAL-TIME JAVA, adéquat pour une traduction polychrone, afin d'étudier certaines propriétés de ces programmes. Ce modèle est décrit en partie à l'aide des composants ARINC.

Conclusion

Les systèmes temps réel sont souvent critiques à cause du rôle qu'ils jouent (par exemple, le contrôle d'une centrale nucléaire ou celui du train d'atterrissage d'un avion). C'est pour cette raison que leur développement requiert des méthodologies de conception suffisamment élaborées pour satisfaire leurs exigences. L'approche synchrone a été proposée pour répondre à cette attente. Elle s'appuie sur des bases mathématiques, offrant ainsi un cadre formel pour le raisonnement, servant ainsi à vérifier les propriétés des systèmes décrits. Dans ce contexte, des outils et techniques ont été développés pour la spécification, la conception, la vérification, la validation, l'optimisation et la génération automatique de code.

Le modèle polychrone qui repose sur les concepts de l'approche synchrone permet de décrire des comportements typiques d'un système temps réel. Il est particulièrement adapté pour représenter un système au sein duquel les composants peuvent avoir leur propres fréquences de fonctionnement (i.e., leur propre notion d'horloge d'activation locale), et aussi concevoir de nouveaux composants à partir de ceux existant déjà sans avoir à modifier ces derniers. Ce modèle sert de support sémantique pour la méthodologie de conception d'applications distribuées temps réel, définie dans l'environnement de développement Polychrony associé au langage Signal.

Dans cette thèse, nous avons d'abord passé en revue un certain nombre de méthodologies de conception de systèmes temps réel. Ensuite, nous nous sommes attachés à faire ressortir les avantages du modèle synchrone de façon générale, et du modèle polychrone en particulier, pour le développement de tels systèmes. Nous avons mis l'accent d'une part, sur des aspects liés à la modélisation et d'autre part, sur l'utilisation de techniques et outils formels pour étudier les propriétés de ces systèmes.

Contributions

Modélisation de l'asynchrone à l'aide du synchrone. Dans [58], P. Caspi a relevé certains aspects sensibles en ce qui concerne la description de concepts asynchrones à l'aide du modèle synchrone. À l'issue de son analyse, il pose la question suivante : *comment peut-on combiner les paradigmes synchrones et asynchrones afin d'obtenir des systèmes distribués robustes par construction, tout en gardant les avantages de la programmation synchrone ?* Le travail exposé dans ce document apporte plusieurs éléments de réponse à cette question. D'une part, il est possible de déduire, à partir d'une description synchrone, une description asynchrone équivalente en effectuant des transformations valides sur l'unique support sémantique considéré (celui du langage Signal). D'autre part, nous disposons de modèles synchrones de mécanismes asynchrones de communication et de synchronisation ; et de façon générale, des modèles de services permettant de simuler les fonctionnalités d'un exécutif temps réel. Cela permet ainsi de concevoir aisément des systèmes globalement asynchrones, localement synchrones (GALS). Quelques

travaux dédiés à la conception de systèmes GALS sont [37], [28] ou [110]. Dans le dernier notamment, les auteurs illustrent des modèles synchrones de mécanismes asynchrones simples. Une perspective de leur travail est la définition d'une bibliothèque de mécanismes standards de communication et de synchronisation afin de faciliter la conception de systèmes GALS. Le travail réalisé dans cette thèse répond à cette perspective.

Amélioration de la méthodologie de conception dans POLYCHRONY**.** L'approche proposée dans cette thèse se place dans le contexte d'une méthodologie de conception d'applications distribuées temps réel, définie dans POLYCHRONY. Le souci est de dégager un cadre unifié pour le développement. Ainsi, l'idée consiste à considérer le modèle sémantique de SIGNAL comme l'unique support de raisonnement lors des transformations successives de la spécification initiale d'une application vers une architecture d'implantation. Notre approche propose des raffinements à différents niveaux en partant d'une spécification polychrone (elle permet une représentation fine du support d'exécution grâce à un ensemble de modèles de services d'un exécutif temps réel). Elle cadre bien avec les recherches actuelles sur la modélisation de systèmes temps réel. Celles-ci tendent vers la prise en compte de plus en plus en amont de caractéristiques de bas niveau dans la mise en œuvre des systèmes (exigences du support d'exécution, contraintes de déploiement). Ici, en permettant d'aborder de tels aspects dans un formalisme de haut niveau (c'est-à-dire, le modèle polychrone), on rend aussi accessible toute la technologie associée (i.e., les outils et techniques de conception, de vérification, de validation et génération automatique de code optimisé). Cela facilite la tâche du développeur et contribue à une réduction significative du coût global de conception.

Une bibliothèque de services APEX. Des composants de description d'applications ont été définis en SIGNAL sous forme d'une bibliothèque : mécanismes de communication et de synchronisation, entités d'exécution, et services d'un exécutif temps réel. Une large partie de ces composants est basée sur le standard ARINC 653, très utilisé dans l'avionique. Ce sont les services appartenant à l'interface dite APEX dont le rôle est la séparation de la couche applicative et du système d'exploitation dans l'architecture globale d'un système. Grâce à ces modèles, des applications temps réel peuvent être décrites en SIGNAL selon une architecture modulaire intégrée (sur laquelle repose la norme ARINC). Par ailleurs, cette bibliothèque a permis d'alimenter d'autres travaux de modélisation basés sur le modèle polychrone. Une illustration a été donnée au travers d'une étude qui a consisté à traduire des programmes REAL-TIME JAVA en SIGNAL.

Perspectives

Extension à d'autres standards. Une perspective immédiate et intéressante du travail qui a été effectué dans cette thèse est l'extension de la bibliothèque avec des composants basés sur des spécifications issues d'autres normes telles que POSIX (*Portable Operating System Interfaces*) [205] ou OSEK (*Open systems and the corresponding interfaces for automotive electronics*) [172]. Cela offrira davantage de choix quant aux types d'architectures modélisables. L'impact des choix de mise en œuvre pourra ainsi être évalué très tôt. La définition de composants qui reposent sur d'autres standards est évidemment envisageable en suivant une démarche similaire à celle qui a été adoptée dans cette étude. Pour cela, il est indispensable d'effectuer au préalable une comparaison entre les autres normes et APEX (par exemple, la synthèse donnée dans [3] peut servir de base). Nous pouvons brièvement mentionner les éléments suivants en ce qui concerne POSIX et OSEK :

- **POSIX.** Il s'agit surtout de l'extension temps réel du standard, appelée RT-POSIX [204]. Elle propose un ensemble de services pour le développement de systèmes temps réel au sein desquels l'ordonnancement des entités exécutives repose sur des priorités fixes. Ces entités sont les *processus* et les *threads* (ou processus légers). Un processus est formé d'un ou plusieurs *threads*. Les services proposés permettent la synchronisation (par exemple, à l'aide de sémaphore) ; l'exclusion mutuelle avec utilisation de protocoles évitant l'inversion de priorité ; la communication entre processus et *threads* à l'aide de files de messages ; la gestion du temps. L'ordonnancement des entités exécutives est basé sur diverses stratégies : *sched_fifo* (politique en *First In First Out*), *sched_rr* (politique en *Round Robin*), ou *sched_other* (politique spécifique à une mise en œuvre donnée). Les *threads* sont ordonnançables à différents niveaux : *global* (tous les *threads* du système sont traités au même niveau, indépendamment de leur appartenance aux processus dont les paramètres d'ordonnancement sont ignorés) ; *local* (l'ordonnancement des *threads* se restreint au processus qui les contient) ; *mixte* (c'est une combinaison des deux premiers choix). Des exemples d'exécutifs temps réel basés sur POSIX sont *VxWorks*, *RTLinux* ou *LynxOS*.

- **OSEK.** Ce standard s'adresse à l'électronique embarquée dans le domaine de l'automobile [172]. Il définit un ensemble de services de système d'exploitation, analogues à ceux proposés dans POSIX ou APEX. On distingue deux types d'entités exécutives : les *routines d'interruption* et les *tâches*. Leur ordonnancement est basé sur des priorités statiques. Les routines d'interruption ont une priorité supérieure aux tâches. La gestion des routines d'interruption est prise en charge par le matériel (leur priorité dépend donc de la plate-forme d'implantation). Les tâches quant à elles sont gérées à l'aide d'un ordonnanceur "classique". L'exécutif OSEK/VDX (*Vehicle Distributed eXecutive*) repose sur ce standard. Lors du développement d'un système, tous les objets (aussi bien de l'application que du système d'exploitation) doivent être alloués statiquement. Il existe aussi une autre spécification d'OSEK, qui est plutôt basée sur un schéma d'exécution de type *time-triggered*, compatible avec OSEK/VDX et appelée *OSEKtime*.

Deux aspects principaux doivent être clairement distingués pour une application de notre approche dans le cas des deux standards mentionnés ci-dessus :

- *Modélisation des services.* Par rapport à cet aspect, la démarche que nous avons adoptée pour décrire les services APEX reste *a priori* valable pour la définition des services proposés dans POSIX et OSEK, puisqu'ils sont de même nature.
- *Modélisation des entités d'exécution.* On remarque que chaque standard a sa propre notion d'entités d'exécution. Cependant, une analogie peut être faite entre la décomposition partition/processus d'APEX et la décomposition processus/*thread* de POSIX. Dans ce cas, les modèles que nous avons définis sont adaptables à POSIX. Une différence majeure à prendre en compte concerne la politique d'ordonnancement des processus POSIX. En effet, celle-ci est basée sur des priorités alors que ce n'est pas le cas pour les partitions APEX (où l'ordonnancement est temporel). D'autre part, il faut permettre un ordonnancement des *threads* POSIX à différents niveaux (global, local et mixte) alors que pour les processus APEX, l'ordonnancement est strictement local à une partition. Enfin, contrairement aux processus APEX dont l'ordonnancement repose sur des priorités dynamiques, la gestion des *threads* POSIX est basée sur des priorités fixes. Le rapprochement entre les entités d'exécution APEX et celles d'OSEK paraît moins immédiat. La décomposition hiérarchique d'APEX (qui est à deux niveaux imbriqués) est complètement différente de celle d'OSEK (qui comprend des niveaux indépendants : interruptions et

tâches). Néanmoins, nous pensons que le modèle que nous avons proposé pour les processus APEX peut servir de base pour décrire les tâches et les routines d'interruption d'OSEK.

Finalement, la question de la définition de modèles génériques de services d'exécutifs temps réel, basée sur notre approche, pourrait être examinée. Par exemple, l'on pourra se baser sur les travaux existants comme SynDEx [104].

Mise en œuvre effective dans la méthodologie de conception de Polychrony. L'utilisation automatique des composants dans le cadre de la méthodologie de conception d'applications distribuées temps réel dans Polychrony, reste à mettre en œuvre. Dans cette optique, nous avons présenté dans le chapitre 8 une possibilité reposant sur une décomposition d'un programme Signal en lignées, suivie de l'instanciation avec les modèles ARINC.

Analyse de programmes Real-Time Java et SystemC [201, 200] dans Polychrony. Nous mentionnerons également que le travail entamé concernant l'importation de programmes Real-Time Java dans Polychrony conduit naturellement à diverses études :

- la vérification et la validation de ces programmes dans un cadre formel ;
- l'optimisation globale du contrôle des programmes : une des capacités intéressantes du compilateur Signal est de pouvoir synthétiser une représentation canonique du graphe de flot de contrôle d'une spécification Signal. Cette fonctionnalité peut être profitable aux applications embarquées car les morceaux de code inutiles pourront être facilement éliminés ;
- la restructuration partielle ou totale de l'architecture originale des programmes : il est possible d'étudier plusieurs répartitions possibles d'une application en utilisant le schéma de transformation de programmes Java défini dans Polychrony, puis en l'instanciant selon les architectures qui peuvent être décrites à l'aide de la bibliothèque de composants.

Transfert industriel. Cette thèse s'étant déroulée dans le cadre d'un projet européen (IST SafeAir - *Advanced Design Tools for Aircraft Systems and Airborne Software*), regroupant plusieurs acteurs du monde industriel, il est légitime d'envisager une exploitation des résultats obtenus dans le contexte industriel. La bibliothèque développée dans ce travail sera fournie prochainement en accès libre dans la plate-forme Polychrony. D'autre part, une mise à disposition de celle-ci est également possible dans l'environnement Sildex (la version commerciale de la plate-forme de conception associée à Signal, développée par Tni-Valiosys).

Annexes

Annexe A

Spécifications SIGNAL de services APEX

Nous donnons d'abord le code SIGNAL des services suivants présentés au chapitre 8 (selon la première version de la bibliothèque APEX de POLYCHRONY) : *process_getActive* (section A.1.1), *process_schedulingRequest* (section A.1.2), *read_blackboard* (section A.1.3) et *update_counters* (section A.1.4). Ensuite, une liste exhaustive des services modélisés est fournie (section A.2).

A.1 Descriptions SIGNAL de quelques services

A.1.1 Modèle du service process_getActive

Ce service permet de récupérer le processus actif dans une partition (ce service ne fait pas non plus partie des services APEX).

```
process PROCESS_GETACTIVE =
    ( ? event request;
      ! ProcessID_type process_ID;
        ProcessStatus_type status;
        boolean valid;
    )
   (| (pid,pstatus,valid) := prag_PROCESS_GETACTIVE{}()
    | process_ID := pid when valid
    | status := pstatus when valid
    | valid ^= request
    |)
  where
    ProcessID_type pid;
    ProcessStatus_type pstatus;
    process prag_PROCESS_GETACTIVE =
        ( ! ProcessID_type process_ID;
            ProcessStatus_type status;
            boolean valid;
        )
      spec (| process_ID ^= status ^= when valid |)
      pragmas
         C_CODE "&o1 = SCHEDULER->ProcessGetActive(&o2,&o3)"
      end pragmas;
    end;
```

A.1.2 Modèle du service process_schedulingRequest

L'invocation de ce service provoque un ré-ordonnancement des processus dans une partition lorsque celle-ci l'autorise (ce service ne fait pas partie des services APEX).

```
process PROCESS_SCHEDULINGREQUEST =
    ( ? event sched_req;
      ! boolean sched_ok;
    )
  spec (| (| sched_req --> sched_ok |)
        | (| sched_req ^= sched_ok |)
        |)
  pragmas
     C_CODE " &o1 = SCHEDULER->ProcessSchedulingRequest(&i1)"
  end pragmas;
```

A.1.3 Modèle du service read_blackboard

La spécification textuelle du service APEX *read_blackboard* présenté au chapitre 8 est donnée ci-après. Ce service permet de lire un message dans un *blackboard*.

```
process READ_BLACKBOARD =
     { ProcessID_type process_ID; }
     ( ? Comm_ComponentID_type blackboard_ID;
         SystemTime_type timeout;
       ! MessageArea_type message;
         MessageSize_type length;
         ReturnCode_type return_code; )
   (| (present,blackboard_in) := BLACKBOARD_CHECKID{}(blackboard_ID)
    | (| outofrange := timeout>MAX_TIMEOUT_VALUE
       | available := timeout>0.0
       |)
    | (| (| empty := blackboard_in.Empty_Indicator when (not outofrange)
          | err_handler := PROCESS_CHECKERRORHANDLER{}(process_ID when empty when available)
          | enabled := PROCESS_CHECKPREEMPTION{}(when (not err_handler))
          |)
       | (| PROCESS_SETSTATE{}(process_ID when enabled,#WAITING)
          | INSERT_BLACKBOARDQUEUE{}(process_ID when enabled,blackboard_in when enabled)
          | START_COUNTER{}(process_ID when enabled when (not (timeout=INFINITE_TIME_VALUE)),
                            timeout when enabled when (not (timeout=INFINITE_TIME_VALUE)))
          | diagnostic := PROCESS_SCHEDULINGREQUEST{}(when enabled)
          | mess := blackboard_in.Message when (not empty)
          |)
       |)
    | (| message := mess.Message_Area
       | length := mess.Message_Size
       | return_code := (#INVALID_PARAM when ((not present) or outofrange)) default
                        (#NOT_AVAILABLE when empty when (not available)) default
                        (#INVALID_MODE when ((not enabled) default err_handler)) default
                        #NO_ERROR when (not empty)
       | blackboard_ID ^= timeout
       |)
    |)
   where
   APEX_Blackboard_type blackboard_out, blackboard_in; event blocked;
   boolean diagnostic, present, outofrange, available, enabled, empty, err_handler;
   Message_type mess;
   end;
```

A.1.4 Modèle du service update_counters

Ce service est invoqué régulièrement pour mettre à jour la valeur des compteurs de temps (ce service ne fait pas partie des services APEX).

```
process UPDATE_COUNTERS =
    ( ? SystemTime_type dt;
      ! [MAX_NUMBER_OF_PROCESSES]boolean timedout;
    )
   spec (| (| dt --> timedout |)
         | (| dt ^= timedout |)
         |)
   pragmas
      C_CODE "SCHEDULER->ProcessUpdateTimeCounters(&i1,&o1)"
   end pragmas;
end;
```

A.2 Liste complète des services de la bibliothèque

Nous commençons par indiquer les services APEX modélisés (section A.2.1). Ce sont les services spécifiés dans [8]. Ensuite, nous énumérons ceux qui ont été ajoutés pour permettre une description complète des applications (section A.2.2). La description détaillée de tous ces services se trouve dans [92].

A.2.1 Services de la norme ARINC 653

Les services mentionnés ici ont été déjà introduits dans le chapitre 7.

- Gestion des partitions
  - *get_partition_status*
  - *set_partition_mode*

- Communication entre partitions
  - *create_sampling_port*
  - *write_sampling_message*
  - *read_sampling_message*
  - *get_sampling_port_id*
  - *get_sampling_port_status*
  - *create_queuing_port*
  - *send_queuing_message*
  - *receive_queuing_message*
  - *get_queuing_port_id*
  - *get_queuing_port_status*

- Gestion des processus
  - *get_process_id*
  - *get_process_status*
  - *create_process*
  - *set_priority*
  - *suspend_self*
  - *suspend*
  - *resume*
  - *stop_self*
  - *stop*
  - *start*

- *lock_preemption*
- *unlock_preemption*

- Communication et synchronisation entre processus
  - *create_buffer*
  - *send_buffer*
  - *receive_buffer*
  - *get_buffer_id*
  - *get_buffer_status*
  - *create_blackboard*
  - *display_blackboard*
  - *read_blackboard*
  - *clear_blackboard*
  - *get_blackboard_id*
  - *get_blackboard_status*
  - *create_event*
  - *set_event*
  - *reset_event*
  - *wait_event*
  - *get_event_id*
  - *get_event_status*
  - *create_semaphore*
  - *wait_semaphore*
  - *signal_semaphore*
  - *get_semaphore_id*
  - *get_semaphore_status*

- Gestion du temps
  - *timed_wait*
  - *periodic_wait*
  - *get_time*
  - *replenish*

A.2.2 Services complémentaires

Les services énumérés dans cette section permettent de manipuler les descripteurs des différents objets considérés dans une description d'application selon APEX (mécanismes de communication et de synchronisation, processus, etc.). Ils comprennent également des services simulant d'autres fonctionnalités d'un exécutif temps réel telles que l'ordonnancement, gestion des compteurs de temps.

- Services d'accès au module de gestion des descripteurs de processus.
  - *process_checkId* : détermine la présence d'un processus à partir de son identificateur.
  - *process_checkAperiodic* : détermine si un processus est apériodique.
  - *process_checkErrorHandler* : détermine si un processus est celui qui est chargé de la gestion des erreurs.
  - *process_checkCreated* : détermine si un processus a été effectivement créé.
  - *process_checkOperatingMode* : retourne le mode opératoire de la partition.
  - *process_checkPreemption* : détermine si la préemption est autorisée.
  - *process_checkWaitingIndicator* : détermine si un processus est en attente d'une ressource.
  - *process_getPeriod* : retourne la période d'un processus.
  - *process_getState* : retourne l'état d'un processus.
  - *process_getActive* : retourne les informations sur le processus actif.
  - *process_getBasicPriority* : retourne la priorité de base d'un processus.
  - *process_getTimeCapacity* : retourne le temps d'exécution alloué à un processus.

- *process_setAllState* : met à jour l'état d'un groupe de processus.
- *process_setAttributes* : met à jour les attributs d'un groupe de processus.
- *process_setDeadlineTime* : met à jour la date d'échéance d'un processus.
- *process_setState* : met à jour l'état d'un processus.
- *process_setCreated* : positionne l'attribut qui indique qu'un processus a été effectivement créé.
- *process_resetStack* : ré-initialise la zone de pile d'un processus.
- *process_resetContext* : ré-initialise le contexte d'un processus.
- *process_resetErrorStatus* : ré-initialise les informations liées aux erreurs.
- *process_resetLocklevel* : ré-initialise le niveau de verrouillage d'une partition.
- *process_record* : permet d'allouer un descripteur à un processus.
- *process_releaseResources* : relâche toutes les ressouces détenues par un processus.
- *process_schedulingRequest* : permet d'ordonnancer les processus.
- *process_previousPreempted* : détermine si un processus est celui qui vient d'être préempté.
- *process_copymessage* : copie un message dans une zone spéciale pour un processus bloqué en attente de recevoir de message.
- *process_retrieveMessage* : permet à un processus, bloqué en attente de recevoir un message, de récupérer un message après sa libération.

- Services de gestion des files d'attente de processus
  - *insert_blackboardQueue* : insère un processus dans la file d'attente des processus bloqués, associée à un *blackboard*.
  - *removall_blackboardQueue* : retire tous les processus de la file d'attente associée à un *blackboard*.
  - *insert_bufferReceiveQueue* : insère un processus dans la file d'attente des processus bloqués en attente de recevoir un message, associée à un *buffer*.
  - *remove_bufferReceiveQueue* : retire un processus de la file d'attente des processus bloqués en attente de recevoir un message, associée à un *buffer*.
  - *insert_bufferSendQueue* : insère un processus dans la file d'attente des processus bloqués en attente d'envoyer un message, associée à un *buffer*.
  - *remove_bufferSendQueue* : retire un processus de la file d'attente des processus bloqués en attente d'envoyer un message, associée à un *buffer*.
  - *insert_eventQueue* : insère un processus dans la file d'attente des processus bloqués, associée à un *event*.
  - *removall_eventQueue* : retire tous les processus de la file d'attente associée à un *event*.
  - *insert_semaphoreQueue* : insère un processus dans la file d'attente des processus bloqués, associée à un sémaphore.
  - *remove_semaphoreQueue* : retire un processus de la file d'attente des processus bloqués sur un sémaphore.
  - *insert_queuigPortReceiveQueue* : insère un processus dans la file d'attente des processus bloqués en attente de recevoir un message, associée à un *queuing port*.
  - *remove_queuingPortReceiveQueue* : retire un processus de la file d'attente des processus bloqués en attente de recevoir un message, associée à un *queuing port*.
  - *insert_queuigPortSendQueue* : insère un processus dans la file d'attente des processus bloqués en attente d'envoyer un message, associée à un *queuing port*.
  - *remove_queuingPortSendQueue* : retire un processus dans la file d'attente des processus bloqués en attente d'envoyer un message, associée à un *queuing port*.

- Services d'accès aux descripteurs de *blackboard*, *buffer*, *event* et *sémaphore*.
  - *blackboard_checkId* : détermine la présence d'un *blackboard* à partir de son identificateur.
  - *blackboard_checkName* : détermine la présence d'un *blackboard* à partir de son nom.
  - *blackboard_checkCapacity* : détermine si un nouveau *blackboard* peut être créé.
  - *blackboard_record* : permet d'allouer un descripteur à un *blackboard*.
  - *blackboard_update* : met à jour un descripteur de *blackboard* avec des informations passées en entrée.
  - *buffer_checkId* : détermine la présence d'un *buffer* à partir de son identificateur.
  - *buffer_checkName* : détermine la présence d'un *buffer* à partir de son nom.

- *buffer_record* : permet d'allouer un descripteur à un *buffer*.
- *buffer_checkCapacity* : détermine si un nouveau *buffer* peut être créé.
- *buffer_update* : met à jour un descripteur de *buffer* avec des informations passées en entrée.
- *event_checkId* : détermine la présence d'un *event* à partir de son identificateur.
- *event_checkName* : détermine la présence d'un *event* à partir de son nom.
- *event_checkCapacity* : détermine si un nouveau *event* peut être créé.
- *event_record* : permet d'allouer un descripteur à un *event*.
- *event_update* : met à jour un descripteur d'*event* avec des informations passées en entrée.
- *semaphore_checkId* : détermine la présence d'un sémaphore à partir de son identificateur.
- *semaphore_checkName* : détermine la présence d'un sémaphore à partir de son nom.
- *semaphore_record* : permet d'allouer un descripteur à un sémaphore.
- *semaphore_update* : met à jour un descripteur de sémaphore avec des informations passées en entrée.
- *semaphore_checkCapacity* : détermine si un nouveau sémaphore peut être créé.

- Services d'accès aux descripteurs de *queuing_port* et *sampling_port*.
  - *queuingport_checkId* : détermine la présence d'un *queuing port* à partir de son identificateur.
  - *queuingport_checkName* : détermine la présence d'un *queuing port* à partir de son nom.
  - *queuingport_checkCapacity* : détermine si un nouveau *queuing port* peut être créé.
  - *queuingport_checkCreated* : détermine si un *queuing port* a été effectivement créé.
  - *queuingport_record* : permet d'allouer un descripteur à un *queuing port*.
  - *queuingport_update* : met à jour un descripteur de *queuing port* avec des informations passées en entrée.
  - *samplingport_checkId* : détermine la présence d'un *sampling port* à partir de son identificateur.
  - *samplingport_checkName* : détermine la présence d'un *sampling port* à partir de son nom.
  - *samplingport_checkCapacity* : détermine si un nouveau *sampling port* peut être créé.
  - *samplingport_checkCreated* : détermine si un *sampling port* a été effectivement créé.
  - *samplingport_record* : permet d'allouer un descripteur à un *sampling port*.
  - *samplingport_update* : met à jour un descripteur de *sampling port* avec des informations passées en entrée.

- Services de gestion des compteurs de temps.
  - *start_counter* : déclenche un compteur de temps.
  - *stop_counter* : arrête un compteur de temps.
  - *reset_runtime* : ré-initialise le *runtime*.
  - *stopall_counter* : arrête tous les compteurs de temps.
  - *update_counters* : met à jour les compteurs de temps actifs.

Table des figures

Bibliographie

[1] AADL Coordination Commitee. Avionics architecture description language. In *AADL seminar, Toulouse, France*, Octobre 2002.

[2] P.A. Abdulla and A. Nylén. Timed Petri Nets and BQOs. In *22nd International Conference on application and theory of Petri Nets (ICATPN)*, 2001.

[3] ARTIST Advanced Real-Time Systems Information Society Technologies. Selected topics in embedded systems design : Adaptive real-time systems for Quality of Service (QoS) management. In *(draft version at* `http://www.artist-embedded.org/Roadmaps`, Mai 2004.

[4] ARTIST Advanced Real-Time Systems Information Society Technologies. Selected topics in embedded systems design : Component-based design and integration platforms. In *(draft version at* `http://www.artist-embedded.org/Roadmaps`, Mai 2004.

[5] Airlines Electronic Engineering Committee. ARINC 629 : IMA Multi-transmitter databus parts 1-4. In *Aeronautical radio, Inc., Annapolis, Maryland*, Octobre 1990.

[6] Airlines Electronic Engineering Committee. ARINC 659 : Backplane data bus. In *Aeronautical radio, Inc., Annapolis, Maryland*, December 1993.

[7] Airlines Electronic Engineering Committee. ARINC report 651-1 : Design guidance for integrated modular avionics. In *Aeronautical radio, Inc., Annapolis, Maryland*, Novembre 1997.

[8] Airlines Electronic Engineering Committee. ARINC specification 653 : Avionics application software standard interface. In *Aeronautical radio, Inc., Annapolis, Maryland*, Janvier 1997.

[9] M. Akian, G. Cohen, S. Gaubert, R. Nikoukhah, and J.-P. Quadrat. Linear systems in (max,+) - algebra. In *29th Conference on Decision and Control*, Décembre 1990.

[10] G. Alefeld and J. Hertzberger. *Introduction to Interval Computation.* Academic Press, NY, 1983.

[11] A. Alkhodre, J.-P. Babau, and J.-J. Schwarz. Modeling of real-time embedded systems by using SDL. In *Proceedings of FDL'2001 Lyon, France*, Septembre 2001.

[12] R.J. Allen. A formal approach to software architecture. In *Ph.D. Thesis, Carnegie Mellon University, Technical Report Number : CMU-CS-97-144*, Mai 1997.

[13] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking for real-time systems. In *5th IEEE Symposium on Logic In Computer Science (LICS'90)*, pages 414–425, 1990.

[14] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. Hybrid automata : An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems I, Lecture Notes in Computer Science 736, Springer-Verlag*, pages 209–229, 1993.

[15] R. Alur and D.L. Dill. A theory of timed automata. In *Theoretical Computer Science*, volume 126, pages 183–235, 1994.

[16] T.P. Amagbégnon. Forme canonique arborescente des horloges de SIGNAL. In *Thèse de l'Université de Rennes I, IFSIC, France*, Décembre 1995.

[17] C. André. Computing syncharts reactions. In *SLAP'03, Electronic Notes in Theoretical Computer Science, vol 88, Porto, Portugal*, Juillet 2003.

[18] P. Aubry. Mises en œuvre distribuées de programmes synchrones. In *Thèse de l'Université de Rennes I, IFSIC, France*, Octobre 1997.

[19] N.C. Audsley and A.J. Wellings. Analysing apex applications. In *Real Time Systems Symposium*, 1996.

[20] J.C.M. Baeten and J.A. Bergstra. Real-time process algebra. In *Formal Aspects of Computer Science*, 1991.

[21] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems : The Polis Approach.* Kluwer Academic Press, Juin 1997.

[22] J. Barnes. *Programming in* ADA *95.* 2nd édition, Addition-Wesley, Harlow, 1998.

[23] J. Barnes. *High Integrity Software - The* SPARC *Approach to Safety and security.* Addition-Wesley, 2003.

[24] S. Basumallick and K. Nilsen. Cache issues in real-time. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, Juin 1994.

[25] G. Bel, F. Boniol, G. Durrieu, C. Fraboul, J. Foisseau, and V. Wiels. Modèles comportementaux pour l'avionique modulaire intégrée. In *1er Congrès plurisectoriel - Logiciel Temps Réel Embarqués - Météo France, Toulouse*, pages 677–691, Janvier 2002.

[26] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *4th DIMACS Workshop on Verification and Control of Hybrid Systems, New Brunswick, New Jersey*, pages 22–24, Octobre 1995.

[27] A. Benveniste. Safety critical embedded systems : the SACRES approach. In *proceedings of Formal techniques in Real-Time and Fault Tolerant Systems, FTRTFT'98 school, Lyngby, Denmark*, Septembre 1998.

[28] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In *proceedings of CONCUR'99, Concurrency Theory, 10th International Conference*, volume 1664 of LNCS, J.C.M. Baeten, S. Mauw (eds.), Springer Verlag, pages 162–177, Août 1999.

[29] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages : specification and distributed code generation. In *Information and Computation, vol. 163*, pages 125–171, 2000.

[30] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. In *Special issue on Embedded Systems, IEEE*, 2003.

[31] A. Benveniste, P. Caspi, P. Le Guernic, H. Marchand, J.-P.. Talpin, and S. Tripakis. A protocol for loosely time-triggered architectures. In *Conference on Embedded Software (EMSOFT'02), J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS vol 2491, Springer Verlag*, 2002.

[32] A. Benveniste, P. Le Guernic, and C. Jacquemot. Programming with events and relations : the SIGNAL language and its semantics. In *Science of Computer Programming, 16 :103-149*, 1991.

[33] L. Bernardinello and F. de Cindio. A survey of basic net models and modular net classe. In *Lecture Notes in Computer Science*, 1992.

[34] G. Berry. Real time programming : Special purpose or general purpose languages. In *Information Processing, G. K. Ritter, editor, vol. 89, North Holland. Elsevier Science Publishers B. V.*, pages 11–18, 1989.

[35] G. Berry. ESTEREL on hardware. In *Philosophical transactions of the Royal Society of London, series A, 19(2) :87-152*, 1992.

[36] G. Berry and G. Gonthier. The ESTEREL synchronous programming language : design, semantics, implementation. In *Science of Computer Programming, vol. 19(2)*, pages 87–152, Novembre 1992.

[37] G. Berry and E. Sentovich. Embedding synchronous circuits in GALS-based systems. In *Sophia-Antipolis conference on Micro-Electronics (SAME'98)*, Octobre 1998.

[38] B. Berthomieu, J.P. Bodeveix, J.M. Farines, M. Filali, P. Michel, P.-O. Ribet, and F. Vernadat. Développement de logiciels dans les systèmes temps-réel pour l'avionique. In *Journées FAC, IRIT-Toulouse*, Mars 2003.

[39] L. Besnard. Compilation de SIGNAL : horloges, dépendances, environnement. In *Thèse de l'Université de Rennes I, IFSIC, France*, Septembre 1992.

[40] L. Besnard, P. Bournai, T. Gautier, N. Halbwachs, S. Nadjm-Tehrani, and A. Ressouche. Design of a multi-formalism application and distribution in a data-flow context : An example. In *Intensional Programming II, Based on the Papers at ISLIP '99, M. Gergatsoulis and P. Rondogiannis, Eds, World Scientific*, 2000.

[41] L. Besnard, T. Gautier, and P. Le Guernic. SIGNAL v4 - INRIA version : Reference manual. In *http ://www.irisa.fr/espresso/Polychrony*, Décembre 2002.

[42] F. Besson, T. Jensen, and J.-P. Talpin. Timed polyhedra analysis for synchronous languages. In *10th International Conference on Concurrency Theory (CONCUR'99), LNCS volume 1664, Springer Verlag*, Août 1999.

[43] J. Bézivin, N. Farcet, J.-M. Jézéquel, B. Langlois, and D. Pollet. Reflective model driven engineering. In *UML2003, San Francisco, California, USA*, Octobre 2003.

[44] B. Boehm. A spiral model of software development and enhancement. In *IEEE Computer, vol.21, n.5*, pages 61–72, Mai 1988.

[45] G. Bollela, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Harding, M. Turnbull, and R. Belliardi. The real-time specification for java. In *The RealTime for Java expert group, final specification*, Décember 2001. http ://www.rtj.org.

[46] F. Boulanger. Intégration de modules synchrones dans un langage à objets. In *Thèse de l'Université Paris XI - Centre d'Orsay*, Décembre 1993.

[47] S. Boulme and G. Hamon. Certifying syncrony for free. In *2nd International Workshop on Implementation of Logics (LPAR'01)*, Havana, Cuba, 2001.

[48] F. Boussinot. Reactive C : An extension of C to program reactive systems. In *Software Practice and Experience, vol. 21(4), 401-428,*, 1991.

[49] F. Boussinot and R. de Simone. The ESTEREL language. In *Another Look at Real Time Programming, IEEE, vol. 79*, pages 1293–1304, 1991.

[50] F. Boussinot and C. Laneve. Two semantics for a language of reactive objects. In *Rapport de recherche INRIA n. 2511*, Mars 1995. Accessible à l'adresse `http://www.inria.fr/rrrt/rr-2511.html`.

[51] P. Brémond-Grégoire and I. Lee. A process algebra of communicating shared resources with dense time and priorities. In *Theoritical Computer Science 189*, 1997.

[52] B.M. Brosgol. A comparison of the object-oriented features of ada 95 and java. In *ACM TRI-Ada Conference (TRI-Ada'97), St. Louis, Missouri, USA,*, pages 213–229, Novembre 1997.

[53] B.M. Brosgol and B. Dobbing. Real-time convergence of ada and java. In *ACM SIGAda Annual International Conference (SIGAda'01), Bloomington, MN, USA.*, pages 11–26, Septembre 2001.

[54] R.E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE transactions on computers, C-35(8)*, pages 677–691, Août 1986.

[55] A. Burns and A. Wellings. Real-time systems and their programming languages. In *Addison-Wesley*, 1997.

[56] J.V. Busquets-Mataix, J.J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Real-Time technology and Applications Symposium, IEEE Computer Society Press*, Juin 1996.

[57] G. Buttazzo. Hard real-time computing systems : Predictable scheduling algorithme and applications. In *Kluwer Academic Publishers*, 2002.

[58] P. Caspi. Embedded control : From asynchrony to synchrony and back. In *1st International Workshop on Embedded Software (EMSOFT'01), Th. A. Henzinger and Ch. M. Kirsch, Eds., LNCS 2211*, 2001.

[59] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *International Conference on Functional Programming*, pages 226–238, 1996.

[60] F. Cassez and O.H. Roux. Traduction structurelle des réseaux de petri temporels vers les automates temporisés. In *4ème Colloque Francophone sur la Modélisation des Systèmes Réactifs, (MSR'03), France*, Octobre 2003.

[61] L. Chen. Timed processes : Models, axioms and decidability. In *PhD thesis, University of Edinburgh*, 1992.

[62] B. Chéron. Transformations syntaxiques de programmes SIGNAL. In *Thèse de l'Université de Rennes I, IFSIC, France*, Septembre 1991.

[63] D. Clarke, I. Lee, and H. Xie. VERSA : A tool for the specification and analysis of resource-bound real-time systems. In *Journal of Computer and Software Engineering, 3(2)*, Avril 1995.

[64] Paul C. Clements. A survey of architecture description languages. In *8th International Workshop on Software Specifications and Design, Paderborn, Germany*, Mars 1996.

[65] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys : a tool for the development and verification of real-time embedded systems. In *Proceedings of Computer Aided Verification, CAV'01. Paris, France. Lecture Notes in Computer Science 2102, Springer-Verlag*, Juillet 2001.

[66] D. Cofer and M. Rangarajan. Formal modeling and analysis of advanced scheduling features in an avionics RTOS. In *procedings of Conference on Embedded Software (EMSOFT'02), J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS 2491, Springer Verlag, p. 138-152*, 2002.

[67] A. Colin, I. Puaut, C. Rochange, and P. Sainrat. Calcul de majorants de pire temps d'exécution : état de l'art. *Techniques et Sciences Informatiques (TSI), 22(5)*, pages 651–677, 2003.

[68] Commission Électrotechnique Internationale. Language de spécification grafcet pour diagrammes fonctionnels de séquences. In *Rapport technique n. IEC 60848, CEI/IEC (International Electrotechnique Commission)*, 1988.

[69] E. Conquet and J.-L. Marty. Formal design for automatic coding and testing : The essi/spaces project. In *Proceedings of World Congress on Formal Methods (FM'99)*, pages 57–75, Septembre 1999.

[70] D. Costard. évaluation d'une chaîne de compilation JAVA temps-réel. In *Rapport de stage de fin d'étude,* ESIEE, Juin 2003.

[71] J.-P. Courtiat and R.C. de Oliveira. On RT-LOTOS and its application to the formal design of multimedia protocols. In *Annals of Telecommunications, vol. 50, no. 11-12*, Novembre 1995.

[72] P. Cousot. Abstract interpretation. In *Symposium of on Models of programming languages and computations, ACM Computing Surveys*, volume 28(2), 1996.

[73] L. Cucu. Ordonnancement non préemptif et condition d'ordonnançabilitṕour systèmes embarqués à contraintes temps réel. In *Thèse de l'Université de Paris-Sud, U.F.R. Scientifique d'Orsay*, Mai 2004.

[74] M. Daniels. Modelling real-time behaviour with interval time calculus. In *LNCS, 571*, pages 53–72, 1991.

[75] T. de Marco. *Structured Analysis and System Specification.* Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[76] J.B. Dennis. First version of a data flow procedure language. In *Programming Symposium, LNCS 19, Springer Verlag*, pages 362–376, 1974.

[77] J.B. Dennis, G. Andrew Boughton, and C.K. Leung. Building blocks for data flow prototypes. In *Proceedings of the 7th Annual Symposium on Computer Architecture (ISCA)*, pages 1–8, 1980.

[78] E.W. Dijkstra. The structure of the t.h.e. multiprogramming system. In *Communications of the ACM 18(8)*, pages 453–457, 1968.

[79] B.P. Douglass. *Doing Hard Time : Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns.* Addison Wesley, 1999.

[80] B. Dutertre. Spécification et preuve de systèmes dynamiques. In *Thèse de l'Université de Rennes 1*, Décembre 1992.

[81] S.A. Edwards. The specification and execution of heterogeneous synchronous reactive systems. In *PhD thesis, University of California, Berkeley*, Mars 1997.

[82] J.-P. Elloy. Les contraintes du temps-réel dans les systèmes industriels répartis. In *RGE(2)*, pages 26–30, Février 1991.

[83] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL - Formal Object-oriented Language for Communicating Systems.* Prentice Hall Europe, ISBN 0-13-621384-7, 1997.

[84] J. Ermont and F. Boniol. TPAP : an algebra of preemptive processes for verifying real-time systems with shared resources. In *proceedings of Workshop on Theory and Practice of Timed Systems (ETPAS'02), Grenoble, France*, Avril 2002.

[85] Eureka Synchron Project. The common formats of synchronous languages : The declarative code dc. In *deliverable of the Eureka Synchron Project*, 1995. Accessible à l'adresse `ftp://ftp.imag.fr/pub/labo-VERIMAG/LUSTRE/DC/dcdoc.ps.gz`.

[86] P.H. Feiler, B. Lewis, and S. Vestal. The SAE avionics architecture description language (aadl) standard : A basis for model-based architecture-driven embedded systems engineering. In *Workshop on Model-Driven Embedded Systems (MDES'03), Washington DC - USA*, Mai 2003.

[87] G. Fohler. Flexibility in statically scheduled hard real-time systems. In *Phd Thesis, Technical University of Vienna*, 1994.

[88] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed filaments : Efficient fine-grain parallelism on a cluster of workstations. In *1st Symposium on Operating Systems Design and Implementation, p. 201-212, Monterey, CA*, Novembre 1994.

[89] D. Gaffé. Le modèle GRAFCET : réflexion et intégration dans une plate-forme multiformalisme synchrone. In *Thèse de Doctorat de l'Université de Nice-Sophia Antipolis*, Janvier 1996.

[90] Abdoulaye Gamatié. Abstraction de domaines totalement ordonnés et calcul d'horloges. In *Rapport de DEA, IFSIC, Université de Rennes 1*, Juin 2000.

[91] Abdoulaye Gamatié and Thierry Gautier. Modeling of modular avionics architectures using the synchronous language SIGNAL. In *14th Euromicro Conference on Real Time Systems (ECRTS'02), Work In Progress session*, pages 25–28. Vienna, Austria, Juin 2002.

[92] Abdoulaye Gamatié and Thierry Gautier. Synchronous modeling of modular avionics architectures using the SIGNAL language. In *INRIA Research Report number 4678*, Décember 2002. Accessible à l'adresse `http://www.inria.fr/rrrt/rr-4678.html`.

[93] Abdoulaye Gamatié and Thierry Gautier. The SIGNAL approach to the design of system architectures. In *10th International Conference and Workshop on the Engineering of Computer-based Systems, Huntsville - Alabama*, Avril 2003.

[94] Abdoulaye Gamatié and Thierry Gautier. Synchronous modeling of avionics applications using the SIGNAL language. In *proceedings of 9th IEEE Real-time/Embedded technology and Applications symposium (RTAS'03)*. Washington D.C., USA, Mai 2003.

[95] Abdoulaye Gamatié, Thierry Gautier, and Loic Besnard. Modeling of avionics applications and performance evaluation techniques using the synchronous language signal. In *proceedings of Synchronous Languages, Applications, and Programming (SLAP'03, An ECRST'2003 Satellite Event)*. Porto - Portugal, Juillet 2003.

[96] D. Garlan, R.T. Monroe, and D. Wile. Acme : An architecture description interchange language. In *Proceedings of CASCON '97*, Novembre 1997.

[97] T. Gautier and P. Le Guernic. Code generation in the SACRES project. In *Safety-critical Systems Symposium, SSS'99, Springer.* Huntingdon, UK, Février 1999.

[98] T. Gautier, P. Le Guernic, and O. Maffeïs. For a new real-time methodology. In *INRIA Research Report number 2364*, Octobre 1994. Accessible à l'adresse `http://www.inria.fr/rrrt/rr-2364.html`.

[99] S. Gérard. Modélisation UML exécutable pour les systèmes embarqués de l'automobile. In *Thèse de l'Université d'Évry, France*, Octobre 2000.

[100] A. Girault. Sur la répartition de programmes synchrones. In *Thèse de l'Institut National polytechnique de Grenoble, France*, Janvier 1994.

[101] G. Gonthier. Sémantique et modèles d'exécution des langages réactifs synchrones : application à ESTEREL. In *Thèse de l'université d'Orsay, Paris, France*, Mars 1988.

[102] D. Goshen-Meskin, V. Gafni, and M. Winokur. SAFEAIR : An integrated development environment and methodology. In *INCOSE 2001, Melbourne*, Juillet 2001.

[103] T. Grandpierre. Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés. In *Thèse de l'Université de Paris-Sud, U.F.R. Scientifique d'Orsay*, Novembre 2000.

[104] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real time embedded heterogeneous multiprocessors. In *CODES'99 7th International Workshop on Hardware/Software Co-Design, Rome*, Mai 1999.

[105] T. Grandpierre, Ch. Lavarenne, and Y. Sorel. Modèle d'exécutif distribué temps réel pour SynDEx. In *Rapport de Recherche INRIA no 3476*, Août 1998.

[106] T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives : a seamless flow of graphs transformations. In *MEMOCODE2003, Formal Methods and Models for Codesign Conference, Mont Saint-Michel, France*, Juin 2003.

[107] R. K. Gupta. Co-synthesis of hardware and software for digital embedded systems. In *Kluwer Academic Publishers, Boston*, 1995.

[108] O. Hainque. Etude d'un environnement d'exécution temps-réel, distribué et tolérant aux pannes pour le modèle synchrone. In *Thèse de l'École Nationale Supérieure des Télécommunications - Paris*, Juin 2000.

[109] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Publications, 1993.

[110] N. Halbwachs and S. Baghdadi. Synchronous modelling of asynchronous systems. In *Conference on Embedded Software (EMSOFT'02), J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS 2491, Springer Verlag*, 2002.

[111] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. In *IEEE, vol.79(9)*, pages 1305–1320, Septembre 1991.

[112] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre. In *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, Septembre 1992.

[113] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology*, pages 83–96, 1993.

[114] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *ptoceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

[115] D. Harel. Statecharts : A visual formalism for complex systems. In *Science of Computer Programming, 8(3)*, pages 231–274, Juin 1987.

[116] D. Harel and A. Naamad. The Statemate semantics of statecharts. In *ACM Transactions. on Software Engineering and Methodology, 5(4)*, pages 293–333, 1996.

[117] D. Harel and A. Pnueli. On the development of reactive systems. In *K. R. Apt, editor, Logics and Models of Concurrent Sysstems, volume F-13 of NATO ASI Series, New-York, Springer Verlag*, pages 477–498, 1985.

[118] D. J. Hatley and I. Pirbhai. *Strategies for Real Time System Specification.* Dover Press, New York, 1987.

[119] T.A. Henzinger and P.-H. Ho. Hytech : The Cornell hybrid technology tool. In *Hybrid Systems II, Lecture Notes in Computer Science 999, Springer-Verlag*, pages 265–294, 1995.

[120] T.A. Henzinger, B. Horowitz, and Ch.M. Kirsch. Embedded control systems development with giotto. In *Proceedings of LCTES. ACM SIGPLAN Notices*, 2001.

[121] T.A. Henzinger, B. Horowitz, and Ch.M. Kirsch. Giotto : A time-triggered language for embedded programming. In *First International Workshop on Embedded Software*

(EMSOFT), Lecture Notes in Computer Science 2211, Springer-Verlag, pages 166–184, 2001.

[122] C.A.R. Hoare. Communicating sequential processes. In *Prentice Hall*, 1985.

[123] Honeywell Technology Center. MetaH user's manual. In *Version 1.27*, 1998. http ://www.htc.honeywell.com/metah.

[124] F. Huber, B. Schätz, A. Schmidt, and K. Spies. Autofocus - a tool for distributed systems specification. In *proceedings of FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems, P. 467-470*. Springer Verlag, LNCS 1135, 1996.

[125] International Telecommunication Union. Z.100 series. In *Z-series ITU-T Recommendations*, 2003. Voir `http://www-comm.itsi.disa.mil/itu`.

[126] J-Consortium. Real-time core extensions. In *Real-Time Java[tm] Working Group*, 2002. http ://www.j-consortium.org/rtjwg/index.shtml.

[127] F. Jahanian and A.K. Mok. Modechart : A specification language for real-time systems. In *IEEE Transactions on Software Engineering, vol. 20, no. 12*, Décembre 1994.

[128] J.-M. Jézéquel, N. Plouzeau, and Y. Le Traon. Développement de logiciels à objets avec Uml. In *IFSIC - Université de Rennes I*, Novembre 2002.

[129] F. Jiménez Fraustro. Conception sûre d'automatismes industriels : modélisation synchrone de langages d'automates programmables de la norme CEI-61131-3. In *Thèse de l'Université de Rennes I, IFSIC, France*, mars 2001.

[130] Jini Community. Jini.org. In *The Community Resource for Jini Technology*, 2003. http ://www.jini.org/.

[131] M. Joseph and P.K. Pandya. Finding response times in a real-time system. In *The Computer Journal 29(5)*, pages 390–395, 1986.

[132] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress, vol 74 of Information Processing*, pages 471–475, 1974.

[133] D.K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. Timed i/o automata : A mathematical framework for modeling and analyzing real-time systems. In *24th IEEE International Real-Time Systems Symposium, Cancun, Mexico (RTSS'03)*, Décembre 2003.

[134] M. Kerbœuf. Orientation objet d'un calcul de processus synchrones. In *Thèse de l'Université de Rennes I, IFSIC, France*, Décembre 2002.

[135] Ch. M. Kirsch. Principles of real-time programming. In *2002 Conference on Embedded Software, EMSOFT'02, J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS vol 2491, Springer Verlag*, 2002.

[136] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. A practitioner's handbook for real-time analysis : Guide to rate monotonic analysis for real-time systems. In *Kluwer Academic Publishers*, 1993.

[137] M.H. Klein, J.P. Lehoczky, and R. Rajkumar. Rate-monotonic analysis for real-time industrial computing. In *IEEE Computer 27(1)*, pages 24–33, 1994.

[138] R. Kocik and Y. Sorel. A methodology to design and prototype optimized embedded robotic systems. In *Computational Engineering in Systems Applications CESA'98, Tunisia*, Avril 1998.

[139] H. Kopetz. Real-time systems : Design principles for distributed embedded applications. In *Kluwer Academic Publishers*, 1997.

[140] A. Kountouris. Outils pour la validation temporelle et l'optimisation de programmes synchrones. In *Thèse de l'Université de Rennes I, IFSIC, France*, Octobre 1998.

[141] J. Kwon, A.J. Wellings, and S. King. A high integrity profile for real-time java. In *Joint ACM Java Grande Conference. Acm press*, 2002.

[142] N. Landin. The formal approach for real-time design. In *Real-time Magazine : Development Tools for real-Time Systems (99q1)*, page 25, 1999. Voir `http://www.realtime-info.be/magazine/99q1/index991.htm`.

[143] L. Lavagno and E. Sentovich. Ecl : A specification environment for system-level design. In *36th Conference on Design Automation (DAC'99), New Orleans, LA, USA*, 1999.

[144] A. Le Guennec. Génie logiciel et méthodes formelles avec UML, spécification, validation et génération de tests. In *Thèse de l'Université de Rennes I, IFSIC, France*, Juin 2001.

[145] P. Le Guernic and T. Gautier. Data-flow to von Neumann : the signal approach. In *Advanced Topics in Data-Flow Computing, J.-L. Gaudiot and L. Bic, Eds, Prentice-Hall*, pages 413–438, 1991.

[146] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. In *IEEE, 79(9)*, pages 1321–1336, Septembre 1991.

[147] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. In *Journal for Circuits, Systems and Computers. Special Issue on Application Specific Hardware Design. (c) World Scientific*, Avril 2002. Accessible aussi à l'adresse `http://www.inria.fr/rrrt/rr-4715.html`.

[148] J.C. Le Lann. Simulation et synthèse de circuits s'appuyant sur le modèle synchrone. In *Thèse de l'Université de Rennes I, IFSIC, France*, Mars 2002.

[149] E.A. Lee and al. Overview of the Ptolemy project. In *Technical Report UBC/ERL M01/11, University of California, Berkeley*, Mars 2001.

[150] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. In *IEEE Transactions on CAD, Vol. 17, No. 12*, Décembre 1998.

[151] J.Y. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. In *Performance Evaluation, No.2*, pages 237–250, 1982.

[152] B. Lewis, S. Vestal, and D. McConnell. Modern avionics requirements for the disbuted system annex. In *Ada-Europe'98, L. Asplund, Ed, LNCS 1411, Springer Verlag, p. 201-212*, 1998.

[153] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. In *Journal of the ACM, vol. 20, No. 1*, pages 46–61, Janvier 1973.

[154] N. Lynch, R. Segala, and F. Vaandraager. Hybrid i/o automata. In *Information and Computation, 185(1)g*, pages 105–157, Août 2003.

[155] N. Lynch and M. Tuttle. An introduction to input/output automata. In *CWI-Quarterly, 2(3)*, pages 219–246, Septembre 1989. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology.

[156] O. Maffeïs. Ordonnancements de graphes de flots synchrones : application à la mise en œuvre de SIGNAL. In *Thèse de l'Université de Rennes I, IFSIC, France*, Janvier 1993.

[157] F. Maraninchi. The ARGOS language : graphical representation of automata and description of reactive systems. In *IEEE Workshop on Visual Languages*, Octobre 1991.

[158] F. Maraninchi and Y. Rémond. Mode-automata : About modes and states for reactive systems. In *proceedings of European Symposium On Programming, Lisbon, Portugal, Springer-Verlag*, Mars 1998.

[159] H. Marchand. Méthodes de synthèse d'automatismes décrits par des systèmes à événements discrets finis. In *Thèse de l'Université de Rennes I, IFSIC, France*, Octobre 1997.

[160] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the SIGNAL environment. In *Discrete Event Dynamic System : Theory and Applications, 10(4)*, pages 325–346, Octobre 2000.

[161] N. Medvidovic, D.S. Rosenblum, D.F. Redmiles, and J.E. Robbins. Modeling software architectures in the unified modeling language. In *ACM Transactions On Software Engineering and Methodology (TOSEM'02), 11(1)*, pages 2–57, 2002.

[162] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Transactions on Software Engineering, vol. 26, no. 1*, Janvier 2000.

[163] Microsoft. .NET development. In *MSDN (Microsoft .NET)*, 2000. http ://www.msdn.microsoft.com/net/.

[164] R. Milner. Communication and concurrency. In *Prentice Hall*, 1989.

[165] F. Mosnier and J. Bortolazzi. Prototyping car-embedded applications. In *Advances in Information Technologies : The Business Challenge, IOS Press*, pages 744–751, 1997.

[166] M. Nebut. Réactions synchrones : spécification et analyse. In *Thèse de l'Université de Rennes I, IFSIC, France*, Novembre 2002.

[167] M. Nebut. An overview of the SIGNAL clock calculus. In *Synchronous Languages, Applications and Programming (SLAP 2003, An ECRST'2003 Satellite Event)*. Porto - Portugal, Juillet 2003.

[168] X. Nicollin, J.L. Richier, J. Sifakis, and J. Voiron. Atp : an algebra for timed processes. In *IFIP Working Conference on Programming Concepts and Methods*, Avril 1990.

[169] Object Management Group. Minimum corba - joint revised submission. In *OMG Document orbos/98-08-04 ed.*, Août 1998.

[170] Object Management Group. OMG unified modeling language specification version 1.4. In *(http ://www.omg.org/technology/documents/formal/uml.htm).*, Septembre 2001.

[171] Object Management Group. Real-time corba specification (version 1.1). In *OMG Document formal/02-08-02*, Août 2002.

[172] OSEK Group. OSEK/VDX Operating System Specification 2.2.1. In *http://www.osek-vdx.org*, 2003.

[173] D. Parnas. On the criteria for decomposing systems into modules. In *Communications of the ACM 15(12)*, pages 1053–1058, Décembre 1972.

[174] R.G. Pettit IV. Using ada 95 for the design of distributed real-time systems. In *TRI-Ada '96 Conference Proceedings, Philadephia, PA*, Décembre 1996.

[175] A. Pnueli. Embedded Systems : Challenges in Specification and Verification. In *2002 Conference on Embedded Software, J. Sifakis and A. Sangiovanni-Vincentelli, Eds, LNCS 2491, Springer Verlag, p. 252-265*, 2002.

[176] D. Potop-Butucaru and R. de Simone. Optimizations for faster execution of ESTEREL programs. In *First ACM and IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE '03), Mont Saint-Michel, France*, Juin 2003.

[177] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3) :115–128, Mai 2000.

[178] C. Ramchandani. Analysis of asynchronous concurrent systems by Petri Nets. In *Technical report, MIT, Laboratory of Computer Science, Cambridge, Massachusetts*, 1974.

[179] M. Riesco and J. Tuya. Synchronous Estelle : Just another synchronous language ? In *proceedings of SLAP'03, Electronic Notes in Theoretical Computer Science, vol 88, Porto, Portugal*, Juillet 2003.

[180] F. Rocheteau and N. Halbwachs. Pollux, a LUSTRE-based hardware design environment. In P. Quinton and Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II*, Château de Bonas, Juin 1991.

[181] J. Romberg. Model-based deployment with autofocus : a first cut. In *14th Euromicro Conference on Real Time Systems (ECRTS'02), Work In Progress session*, pages 41–44. Vienna, Austria, Juin 2002.

[182] J. Rushby. Formal methods and the certification of critical systems. In *INSTITUTION Computer Science Laboratory, SRI International*, number SRI-CSL-93-7, Menlo Park, CA, Décembre 1993. Accessible aussi comme rapport de la NASA n.4551, sous le titre "Formal Methods and Digital Systems Validation for Airborne Systems", à l'adresse `http://www.csl.sri.com/papers/csl-93-7/`.

[183] J. Rushby. Partitioning in avionics architectures : Requirements, mechanisms, and assurance. In *tech. report, Computer Science Laboratory SRI International, Menlo Park CA 94025 USA*, Mars 1999.

[184] Sacres Consortium. The semantic foundations of SACRES. In *EP 20897 Deliverable report*, Mars 1997.

[185] Safeair Consortium. Advanced design tools for aircraft systems and airborne software. In SAFEAIR *I*, 2002. Voir `http://www.safeair.org`.

[186] C. Scheidler, P. Puschner, S. Boutin, E. Fuchs, G. Gruensteidl, Y. Papadopoulos, M. Pisecky, J. Rennhack, and U. Virnich. System engineering of time-triggered architectures - the setta approach. In *DCCS-2000, 16th Int. Workshop on Distributed Control Systems, Sydney*, Novembre 2000.

[187] S. Schneider, J. Davies, D.M. Jackson, G.M. Reed, J.N. Reed, and A.W. Roscoe. Timed CSP : Theory and practice. In *REX Workshop*, pages 640–675, 1991.

[188] B. Selic. Model-driven development of real-time software using OMG standards. In *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003), 14-16 May 2003, Hakodate, Hokkaido, Japan*, pages 4–6, 2003.

[189] L. Sha, R. Rajkumar, and S. Sathaye. Generalized rate-monotonic scheduling theory : A framework for developing real-time systems. In *IEEE*, Janvier 1994.

[190] J. Sifakis. Use of petri nets for performance evaluation. In *Measuring, Modelling and Evaluating Computer Systems, H. Beilner and E. Gelenbe, Eds*, pages 75–93, 1977.

[191] J. Sifakis. Modeling real-time systems - challenges and work directions. In *EMSOFT'01, Tahoe City. Lecture Notes in Computer Science 2211*, Octobre 2001.

[192] F. Simonot-Lion and J.-P. Elloy. An architecture description language for in-vehicle embedded system development. In *15th Triennial World Congress of the International Federation of Automatic Control, Barcelona 2002, Elsevier Science*, Juillet 2002.

[193] R. Soley and the OMG Staff Strategy Group. Model driven architecture (draft 3.2). In *OMG, white paper*, Novembre 2000. Voir `http://www.omg.org/mda/presentations.htm`.

[194] I. Sommerville. Le génie logiciel. In *Addison-Wesley*, 1992.

[195] Y. Sorel. Massively parallel computing systems with real time constraints, the 'algorithm architecture adequation' methodology. In *Proceeding of the Massively Parallel Computing Systems, Ischia, Italy*, Mai 1994.

[196] J. Stankovic. Misconceptions about real time computing : A serious problem for next generation systems. In *IEEE Computer, Vol. 21, No. 10*, pages 10–19, 1988.

[197] J. Stankovic. The many faces of multi-level real-time scheduling. In *Real-Time Computing Systems and Applications, Tokyo*, Octobre 1995.

[198] J. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. In *IEEE Computer, Vol. 28, No. 6*, pages 16–25, Juin 1995.

[199] C. Sziperski. Component software : Beyond object-oriented programming. In *ACM Press and Addison-Wesley, New York*, 1998.

[200] Jean-Pierre Talpin, David Berner, Paul Le Guernic, Abdoulaye Gamatié, Rajesh Gupta, and Sandeep Shukla. Behavioral type inference for compositional system design. Rapport de recherche RR-5141, INRIA, 2004.

[201] Jean-Pierre Talpin, David Berner, Sandeep Shukla, Paul Le Guernic, Abdoulaye Gamatié, and R.K. Gupta. A Behavioral Type Inference System for Compositional System-on-Chip Design. In *Proceedings of the Fourth International Conference on Application of Concurrency to System Design (ACSD'04)*, pages 47–56, Hamilton, Ontario, Canada, 2004.

[202] Jean-Pierre Talpin, Abdoulaye Gamatié, David Berner, Bruno Le Dez, and Paul Le Guernic. Hard real-time implementation of embedded systems in java. In *International Workshop on Scientific Engineering of Distributed JAVA Applications. Lectures Notes in Computer Science, Springer Verlag*, Novembre 2003. Accessible aussi sous forme de rapport à l'adresse `http://www.inria.fr/rrrt/rr-5020.html`.

[203] Jean-Pierre Talpin, Bruno Le Dez, Abdoulaye Gamatié, Paul Le Guernic, and David Berner. Component-based engineering of real-time java applications on a polychronous design platform. In *INRIA Research Report number 4744*, Février 2003. Accessible à l'adresse `http://www.inria.fr/rrrt/rr-4744.html`.

[204] The Institute of Electrical and Electronics Enginneers. POSIX.13 - IEEE Std 1003.13 :2001 -Standard for Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP). In *`http://standards.ieee.org/catalog/olis/posix.html`*, 1998.

[205] The Institute of Electrical and Electronics Enginneers. POSIX.1 - IEEE Std 1003.1 :2001 -Standard for Information Technology - Portable Operating System Interface. In *`http://standards.ieee.org/catalog/olis/posix.html`*, 2001.

[206] Corporation TimeSys. The concise handbook of real-time systems. In *`http://www.timesys.com`*, 2000.

[207] TNI-Valiosys. SILDEX. In *Software Products - SILDEX V5*, 2002. Voir `http://www.tni.fr/tni/offre/sildex/index.eng.html`.

[208] R. Vallee-Rai and L.J. Hendren. Jimple : Simplifying java bytecode for analyses and transformations. In *Technical report, McGill University*, 1999.

[209] J.J. Vereijken. Discrete-time process algebra. In *PhD thesis, Eindhoven University of Technology*, 1997.

[210] S. Vestal. MetaH support for real-time multi-processor avionics. In *IEEE Workshop on Parallel and Distributed Real-Time Systems*, Avril 1997.

[211] M. von der Beeck. A comparison of statecharts variants. In *H. Langmaack, W.P. de Roever, and J. Vytopil, eds, Formal Techniques in Real-Time and Fault-Tolerant Systems, Third International Symposium, vol. 863 of LNCS, Springer Verlag, Lübeck, Germany*, pages 128–148, 1994.

[212] W.W. Wadge and E.A. Ashcroft. LUCID, the dataflow programming language. In *Academic Press*, 1985.

[213] Y. Wang. UML et technologie synchrone pour les systèmes réactifs distribués. In *Thèse de l'Université de Rennes I, IFSIC, France*, Décembre 2001.

[214] P. Ward and S.J. Mellor. *Structured Development for Real-time Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1985.

[215] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of ESTEREL for real-time embedded systems. In *Proceedings of CASES'2000, San Jose*, pages 2–8, Novembre 2000.

[216] N. Wirth. Embedded systems and real-time programming. In *Embedded Software, 1st International Workshop EMSOFT 2001, Th. A. Henzinger and Ch. M. Kirsch, Eds., LNCS 2211*, pages 486–492, 2001.

[217] J. Xu and D.L. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. In *IEEE Transactions on Software Engineering, 16(3)*, pages 360–369, Mars 1990.

[218] V. Yodaiken. Against priority inheritance. In *Copyright Finite State Machine Labs*, Octobre 2002. http ://hq.fsmlabs.com/ yodaiken.

[219] S. Yovine. KRONOS : A verification tool for real-time systems. In *Software Tools for Technology Transfer, 1(1+2)*, pages 123–133, Décembre 1997.

Modélisation polychrone et évaluation de systèmes temps réel

Résumé. Les *systèmes temps réel* sont des dispositifs constitués de matériels et de logiciels soumis à des contraintes à la fois fonctionnelles et temporelles pour réaliser des traitements, et agir sur leur environnement. Des exemples de domaines où on rencontre de tels systèmes sont les télécommunications, le nucléaire, l'avionique ou le médical. Ces systèmes sont souvent critiques à cause d'enjeux humains et économiques importants. Leur développement exige donc des méthodes très fiables. L'*approche synchrone* a été proposée dans le but de répondre à cette attente. Ses fondements mathématiques offrent un cadre formel propice à la description et la validation des systèmes temps réel. Parmi les modèles de spécification synchrone, le modèle *multi-horloge* ou *polychrone* se distingue par le fait qu'il permet de décrire des systèmes où chaque composant peut avoir sa propre horloge d'activation. Outre la validation formelle, il favorise des approches orientées composants et le développement modulaire de systèmes à grande échelle.

Cette thèse propose une méthodologie de conception de systèmes temps réel en utilisant comme formalisme de description le langage synchrone SIGNAL fondé sur le modèle polychrone. Elle utilise les outils et techniques formels basés sur ce modèle pour vérifier des propriétés comportementales portant sur des aspects tant fonctionnels que non fonctionnels. Cette démarche facilite l'évaluation des choix de conception. La méthodologie proposée permet de concevoir des systèmes comportant des mécanismes asynchrones à l'aide de l'approche synchrone. Elle illustre la caractérisation de comportements temps réel dans le modèle polychrone. L'avionique est le domaine d'application privilégié par la thèse. Cette dernière s'inscrit dans le cadre du projet européen IST SAFEAIR (*Advanced Design Tools for Aircraft Systems and Airborne Software*), regroupant plusieurs industriels. Nous nous sommes particulièrement intéressés à la conception d'applications suivant le modèle d'architecture IMA (*Integrated Modular Avionics*) sur lequel repose la norme avionique ARINC. Cela a conduit à la réalisation d'une bibliothèque SIGNAL de composants, constituée en majeure partie de services d'un exécutif temps réel définis par ARINC.

Mots clés : *systèmes temps réel, modèle polychrone, évaluation,* SIGNAL*, méthodologie de conception, norme ARINC, avionique, validation.*

Polychronous modeling and evaluation of real-time systems

Abstract. *Real-time systems* are devices consisting of hardware and software with functional and timing constraints for the interaction with their environment. Telecommunication, nuclear power plants, avionics, and medical technology are typical domains where such systems are encountered. These systems are often critical because of the high human and economic stakes. Therefore, the development of these systems requires highly reliable methods. The *synchronous approach* has been proposed in order to answer this demand. Its mathematical basis provides a formal framework that favors the description and validation of real-time systems. The *multi clock* or *polychronous* model stands out from other synchronous specification models by the fact that it allows the description of systems where each component can have its own activation clock. Besides formal validation, it favors component-based approaches and modular development of large-scale systems.

This document suggests a methodology for the design of real-time systems that uses as formalism the synchronous language SIGNAL which relies on the polychronous model. It uses the formal tools and techniques based on this model in order to verify both behavioral and non functional properties. This facilitates design space exploration. The methodology presented here allows to design systems that contain asynchronous mechanisms using the synchronous approach. It illustrates the characterization of real-time behaviors in the polychronous model. Avionics is considered as privileged application domain in the current document. This work has been supported by the european project SAFEAIR (*Advanced Design Tools for Aircraft Systems and Airborne Software*) in which many industrials are involved. We especially are interested in the design of applications following the *Integrated Modular Avionics* (IMA) architecture model that is based on the avionic standard ARINC. This leads to the implementation of a library in SIGNAL containing real-time executive services defined by ARINC.

Keywords : *real-time systems, polychronous model, evaluation,* SIGNAL*, design methodology, ARINC standard, avionics, validation.*