N° d'ordre :

UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS
École Doctorale Sciences Pour l'Ingénieur

Thèse

présentée pour obtenir le titre de

Docteur en Sciences

mention

Sciences de l'Ingénieur

par

Hédi Boufaïed

Machines d'exécution pour langages synchrones

Soutenue le 9 Novembre 1998 devant le jury composé de :

Messieurs	Jean-Paul RIGAULT	Président
	Lionel MARCÉ	Rapporteur
	Guy VIDAL-NAQUET	Rapporteur
	Jean-Paul MARMORAT	Examinateur
	Charles ANDRÉ	Directeur de thèse

Table des matières

Chapitre 1

Introduction

Les programmes *transformationnels* ont longtemps constitué l'essentiel des applications informatiques. L'introduction de l'informatique dans le contrôle des procédés industriels puis la généralisation des contrôleurs dans les applications grand public ont conduit au concept de programmes *réactifs*. Les programmes *interactifs* sont intermédiaires entre les programmes réactifs et transformationnels. Ils occupent une place grandissante en particulier dans les applications réseaux et multimédia. Les systèmes réactifs et interactifs lient intimement le programme et son environnement, le couplage le plus serré étant celui des systèmes réactifs. Harel et Pnueli [HP85] ont donné des systèmes réactifs l'image de boîtes noires qui réagissent de manière réflexe aux stimuli de l'environnement en changeant d'état et en produisant à leur tour des stimuli.

Les systèmes réactifs sont présents dans de nombreux domaines. Dans l'industrie manufacturière par exemple, des calculateurs sont chargés de contrôler et de superviser les procédés industriels. Dans les domaines de la robotique ou de l'aéronautique, ils prennent la forme de systèmes embarqués. D'autres types d'applications comme les protocoles de communication dans les réseaux ou les interfaces homme-machine sont composées de parties transformationnelles et de parties réactives.

Au caractère réactif de ces applications sont souvent associées des contraintes temps réel. Dans un contrôleur de processus temps réel, par exemple, le système doit impérativement réagir aux événements dans des délais prédéfinis. Quand les contraintes temporelles sont inviolables, on parle d'application temps réel à contraintes temporelles strictes ou de système temps réel "dur". Dans une interface homme machine, les contraintes de temps sont généralement plus laches, on parle alors de système temps réel à contraintes temporelles relatives ou de *"soft real-time system"*. Outre les aspects temporels, le terme temps réel implique aussi des contraintes de prévisibilité et de sûreté de fonctionnement : on doit pouvoir prévoir le comportement du système dans toutes les situations, même dans des situations extrêmes.

Nous nous intéressons pour notre part aux systèmes réactifs dans lesquels l'aspect contrôle est prépondérant. Ces systèmes doivent répondre à des stimuli nombreux et indépendants. D'autre part, leur comportement doit s'adapter à l'évolution de l'environnement. Ces systèmes posent des problèmes d'expression de comportements réactifs complexes. Ils mettent en œuvre des concepts tels que parallélisme, communication, synchronisation et préemption.

Approches des systèmes réactifs temps réel

La complexité croissante des applications a conduit les concepteurs à s'éloigner du matériel pour adopter des approches de plus haut niveau. Des méthodologies de développement basées sur des approches fonctionnelles comme SA/RT[1] [HP88] tentent de répondre aux exigences des systèmes temps réel en associant décomposition structurelle et diagrammes d'états pour la description des comportements. Plus récemment des approches du type ROOM[2] [SGW94] ou UML[3] *temps réel* [Dou98] prônent l'utilisation du concept objet depuis l'analyse jusqu'à l'implémentation. Les STATECHARTS [Har87] y ont remplacé les diagrammes d'états et permettent d'exprimer le comportement interne des objets. Toutefois il n'existe pas de règles établies pour composer les objets entre eux ce qui rend le comportement global des systèmes difficile à mettre au point et à analyser.
Au niveau implémentation, les solutions les plus répandues font appel à la notion de tâche soit directement au niveau du langage comme c'est le cas pour le langage ADA [Boo87], soit par l'intermédiaire d'un exécutif temps réel. Ce type de réalisation pose le problème du non déterminisme des réactions. L'asynchronisme entre les différentes tâches en est la cause essentielle.

Les applications réactives temps réel ont des besoins bien spécifiques auxquels les méthodes de développement traditionnelles n'apportent pas de réponse totalement satisfaisante. L'**approche synchrone** a été introduite dans le but de mieux répondre à ces exigences. Cette approche est basée sur l'hypothèse synchrone qui stipule que les réactions des systèmes se font en temps nul. Cette hypothèse simplificatrice permet d'exprimer des comportements sophistiqués qui font intervenir parallélisme, communication, synchronisation et préemption tout en restant déterministes. L'hypothèse synchrone est à la base des langages ESTEREL[BG92], LUSTRE [CHPR91] et SIGNAL [LGLL91]. Ces langages ont des sémantiques mathématiques rigoureuses sur lesquelles s'appuient les compilateurs pour produire un code déterministe et efficace à l'exécution. Ces fondements théoriques ont aussi permis de construire des outils de preuves formelles.

Au niveau implémentation, des travaux visant à concilier approche synchrone

1. Structured Analysis/Real Time.
2. Real-Time Object Oriented Modelling.
3. Unified Modelling Language.

et approche à objets ont donné lieu au concept d'objets synchrones [Bou93]. Les objets synchrones sont des modules synchrones dotés d'une interface standard qui facilite la communication avec l'extérieur et qui favorise ainsi l'intégration de comportements réactifs dans des environnements d'objets.

Problématique

L'approche synchrone est d'un grand intérêt pour les phases de conception et de vérification des systèmes réactifs temps réel. Toutefois, en raison des hypothèses synchrones, les programmes issus de cette approche n'ont qu'une vision abstraite du monde réel :

- Leurs réactions sont considérées comme instantanées.
- Les événements asynchrones de l'environnement sont perçus sous la forme de vecteurs de signaux logiques.
- Le temps considéré est un temps logique constitué d'une suite discrète d'instants.

En raison de l'écart qui existe entre les exigences du modèle synchrone et la réalité asynchrone, les programmes synchrones ne sont pas directement exécutables. Leur implantation effective pose le problème du **passage du monde synchrone au monde réel asynchrone**.

Par ailleur, rares sont les applications temps réel purement réactives. Un grand nombre d'applications sont constituées d'un cœur réactif et de traitements transformationnels s'exécutant en parallèle. Il apparaît alors nécessaire de faire coopérer sur une même cible d'exécution, le processus synchrone et les traitements transformationnels. En ESTEREL, les parties transformationnelles sont reportées dans le langage cible, support d'exécution. La notion de tâche de ce langage permet de représenter des traitements transformationnels qui ne pourraient en aucun cas être considérés comme instantanés. Les programmes ESTEREL n'ont qu'une vision logique de ces tâches. Leur exécution effective est considérée, au niveau du langage, comme un détail d'implémentation et laissée aux soins de l'utilisateur. Un moyen d'exécuter ces tâches est de faire appel aux services d'un exécutif multi-tâches.

Pour toutes ces raisons, la mise en œuvre de processus synchrones nécessite une **machine d'exécution** dont le rôle est d'enfouir le processus dans son environnement en assurant une approximation satisfaisante des hypothèses synchrones. La machine d'exécution doit remplir les fonctions suivantes :

- Réaliser l'interface entre le processus synchrone et la cible d'exécution.

- Garantir l'atomicité des réactions. L'hypothèse d'instantanéité étant irréalisable, la solution consiste à assurer l'atomicité des réactions et à vérifier a postériori que le système est "assez rapide" vis-à-vis des évolutions de son environnement.

- Établir la correspondance entre la vision idéale synchrone et l'environnement asynchrone en connectant les signaux logiques aux signaux réels et en déterminant les instants d'activations du processus.

- Supporter l'exécution des traitements asynchrones.

L'implantation des processus synchrones, dernière phase du cycle de développement synchrone est souvent considérée comme un détail d'implémentation et traitée au cas par cas.

> *Dans ce mémoire, nous nous intéressons à l'implantation d'applications synchrones centralisées à faibles contraintes de temps sur des exécutifs temps réel.*

L'objectif de nos travaux est l'élaboration d'une **architecture générique** de machine d'exécution. Cette architecture doit être assez **générale** pour s'adapter par configuration aux besoins spécifiques de chaque application. Elle doit être **indépendante de la cible d'exécution**. Un des buts visés étant l'**intégration de traitements asynchrones** dans du code synchrone, cette machine doit gérer l'exécution des tâches ESTEREL. Enfin, pour faciliter le travail du concepteur, nous proposons des outils pour la **génération automatique** de machines sur une cible particulière (Solaris 2).

Plan du mémoire

Le chapitre 2 est consacré aux *systèmes réactifs temps réel*. Nous y précisons les notions de réactivité et de temps réel. Nous rappelons brièvement les spécificités de ces systèmes et mettons en évidence les points faibles des approches traditionnelles. Deux exemples typiques de systèmes réactifs sont décrits.

L'*approche synchrone* est présentée dans le chapitre 3. Les hypothèses de base de l'approche sont énoncées. Étant donné notre intérêt pour les systèmes réactifs à contrôle dominant nous présentons les concepts du langage ESTEREL ainsi que le formalisme graphique SYNCCHARTS[And96] qui peut être considéré comme un ESTEREL graphique. Nous introduisons aussi la notion de tâche ESTEREL. Nous sommes ensuite amenés à détailler le concept d'objets synchrones introduit par F. Boulanger car il est d'un grand intérêt pour la mise en œuvre de notre machine d'exécution. Enfin nous mettons en évidence les problèmes posés par l'implantation des processus synchrones.

Le chapitre 4 est consacré à l'étude de l'*existant en matière de machines d'exécution.* Un certain nombre de réalisations sont étudiées. La diversité des solutions met en évidence la nécessité d'une architecture générique pour l'implantation des processus synchrones.

Le chapitre 5 est entièrement consacré à l'*élaboration d'une architecture générique de machines d'exécution.* Celle-ci est construite de manière progressive à partir d'un exemple simple. Cette architecture est souple et permet de prendre en compte différentes stratégies d'entrée/sortie, d'activation et de traitement des anomalies. Le modèle de boîtes réactives est introduit et donne un caractère récursif à l'architecture ce qui permet de construire des machines plus élaborées. Dans un deuxième temps, nous présentons les problèmes posés par l'exécution des tâches ESTEREL. La gestion des tâches est intégrée dans l'architecture de la machine et deux sémantiques sont proposées.

Dans le chapitre 6, nous traitons un exemple d'application réactive : il s'agit d'un distributeur automatique de billets (*Automatic Teller Machine* ou ATM). Le contrôleur est conçu en SYNCCHARTS. Nous utilisons ensuite la technique des observateurs pour établir des propriétés de sûreté sur cette application. Nous présentons également les outils de génération automatique qui ont été développés pour le système SOLARIS2. Ces outils sont utilisés pour produire une machine exécutable de l'exemple avec gestion des tâches. À ce stade nous voulons prouver que les propriétes vérifiées par le module synchrone restent vérifiées par la machine d'exécution. On modélise pour cela les distorsions introduites à l'acquisition par des modules ESTEREL. Cette analyse met en évidence des problèmes dûs à la discrétisation du temps.

Enfin, dans le dernier chapitre nous tirons les conclusions de nos travaux. Les perspectives et les limitations sont dégagées.

Des aspects plus techniques concernant la mise en œuvre effective et les logiciels dédiés sont reportés dans les annexes de ce mémoire.

Chapitre 2

Les systèmes réactifs temps réel

Objectifs

- Caractériser la classe des systèmes auxquels on s'intéresse: les systèmes réactifs temps réel à contrôle dominant.
- Mettre en évidence les problèmes posés par la conception de ces systèmes.
- Présenter brièvement les approches traditionnelles et leurs points faibles.
- Donner deux exemples typiques de systèmes réactifs temps réel que nous avons été amenés à développer.

2.1 Présentation

Avant de définir les systèmes concernés par nos travaux, nous devons préciser les notions de système temps réel et de système réactif.

2.1.1 Définitions

Systèmes temps réel

Tous les systèmes temps réel ont une caractéristique commune qui est la présence de **contraintes de temps**. Pour Levy et Agrawala [LA90], un système temps réel est défini non seulement par un comportement logique, mais aussi par un **comportement temporel**. Dans un système informatique classique, on considère que le fonctionnement est correct si le résulat fournit est bien celui attendu et en général "*plus le système est rapide, plus il est performant*". Au contraire, un système temps-réel a un fonctionnement correct s'il fournit le résultat attendu et si ce résultat arrive bien *dans les temps*. Ainsi, un bon résulat arrivé trop tard (ou trop tôt) est considéré comme un dysfonctionnement du système.

Les temps de réponse ne sont donc plus de simples mesures de performances mais des contraintes faisant partie intégrante du cahier des charges.

Une caractérisation des systèmes temps réel basée exclusivement sur l'aspect temporel nous semble trop restrictive. La définition donnée par J.P Elloy [Ell91] fait apparaître un élément aussi déterminant qui est la relation système/environnement :

> "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 le cahier des charges de l'application."

Cette définition insiste sur la relation particulière qui lie système et environnement dans une application temps réel. Le système est "connecté" à un environnement qu'il doit être capable de suivre ou de contrôler. Cet aspect prendra un sens encore plus précis avec la notion de système réactif. En ce qui concerne les contraintes de temps, celles-ci sont relatives à un environnement donné. On distingue de ce point de vue les systèmes temps réel à contraintes strictes inviolables (*hard real-time*) définies par des intervalles du type *(minimum, maximum)* des systèmes temps réels à contraintes faibles (*soft real-time*) définies par des temps de réponse *moyens*.

Halang et Stoyenko [HS91] donnent des systèmes temps réel une définition encore plus large en plaçant les notions **prévisibilité** et de **sûreté de fonctionnement** au centre de leurs préoccupations. L'idée sous-jacente est que l'on a affaire à des **systèmes critiques** dans lesquels tout dysfonctionnement peut avoir des conséquences dramatiques. Un système sûr doit avoir un comportement prévisible non seulement dans les conditions normales de fonctionnement mais aussi dans des situations extrêmes. Cette contrainte de prévisibilité porte aussi bien sur le comportement temporel que sur le comportement logique :

- Le respect des contraintes de temps doit être assuré y compris dans des situations de surcharge.

- La réaction du système à des situations fatales de type panne de composant doit être prévisible et donner lieu à un traitement particulier.

Systèmes réactifs

Indépendamment des aspects temporels et de sûreté de fonctionnement, on peut classer les systèmes selon le type d'interaction qu'ils ont avec leur environnement. On distingue de ce point de vue 3 types de systèmes :

- Les *systèmes transformationnels* sont typiquement des systèmes qui prennent des données en entrée, effectuent un traitement, produisent des sorties et terminent, le rythme d'évolution étant imposé par le programme. Dans ces systèmes, l'interaction avec l'environnement se limite aux phases d'acquisition et de production des résultats.

- Les *systèmes interactifs* sont eux, constamment en interaction avec leur environnement. Au cours de leur évolution, ils réagissent à des sollicitations de l'environnement en rendant des services. La prise en compte des sollicitations et la production des services se fait à l'initiative du système si bien que c'est lui qui impose son propre rythme.

- Un *système réactif* est lui aussi en interaction permanente avec son environnement, mais à un rythme imposé par l'environnement lui-même. Harel et Pnueli [HP85] ont donné de ces systèmes l'image de boîtes noires qui réagissent de manière **réflexe** à des stimuli externes en produisant à leur tour des sorties. L'évolution d'un système réactif est donc une suite de réactions ponctuelles provoquées par l'environnement.

Un système temps réel n'est temps réel que vis-à-vis d'un environnement donné. Il en est de même pour un système réactif. La notion de réactivité est une image *idéale* pour représenter des systèmes qui, vis-à-vis de leur environnement, réagissent de manière réflexe. Ainsi la plupart des systèmes réactifs sont implémentés par des systèmes interactifs suffisamment rapides pour prendre en compte tous les stimuli et y répondre à temps. Ce sont des solutions de ce type, basées sur une *"interactivité maîtrisée"*, que nous avons retenu pour les implémentations.

Dans les systèmes réactifs, on trouve à la fois du traitement des données et du contrôle. Certaines applications comme les filtres numériques sont nettement plus "orientées données" : le flot de données est complexe alors que le contrôle est souvent réduit à un séquencement d'opérateurs. À l'opposé, on trouve des systèmes réactifs dans lesquels c'est le contrôle qui est prépondérant : dans un contrôleur de bus par exemple, la complexité réside dans le comportement et les manipulations de données sont souvent triviales. Nous nous intéressons pour notre part aux systèmes réactifs où le **contrôle** joue un rôle primordial. Même pour ce type de systèmes, les deux aspects coexistent et sont intimement liés. Nous verrons dans le chapitre 6, qu'il n'est pas toujours évident de distinguer traitement des données et contrôle.

Dans les systèmes à contrôle dominant, les événements ont lieu de manière sporadique et leurs sources sont multiples. Pour cette raison, ils sont souvent composés de sous-systèmes interagissants. Les notions de parallélisme, de communication et de synchronisation y sont donc essentielles.

Par ailleur, ces systèmes étant la plupart du temps assujétis à l'évolution de l'environnement, leur comportement doit pouvoir évoluer et s'adapter dynamiquement. Ces changements de comportement consistent souvent à *interrompre* ou *suspendre* un traitement au profit d'un autre. La notion de *préemption* qui correspond à l'interruption définitive ou provisoire d'une activité est aussi un aspect essentiel au même titre que le parallélisme ou la communication.

Les systèmes réactifs temps réel à contrôle dominant sont présents dans de nombreux secteurs de l'industrie. Les applications typiques sont les commandes de procédés industriels (industrie manufacturière, robotique) et les contrôleurs embarqués (dans l'aéronautique, l'automobile,...) ou encore les pilotes de périphériques. D'autres applications ont un caractère critique moins prononcé mais sont tout autant concernées. On citera par exemple les interfaces homme/machine. De nombreuses applications encore, mêlent aspects transformationnels et aspects réactifs. C'est la cas notamment des protocoles de communication ou de certaines applications de traitement du signal.

2.1.2 Critères de conception

Pour répondre pleinement aux problèmes posés par la conception des systèmes réactifs, les approches proposées doivent satisfaire à un certain nombre de critères.

Premièrement, les systèmes réactifs ont des spécificités bien prononcées qui doivent être prises en compte à un **haut niveau d'abstraction**. Il s'agit en particulier de la capacité à exprimer des **comportements complexes**. Le parallélisme, la communication, la synchronisation et la préemption doivent être des concepts de premier plan. Les traitements exceptionnels et les contraintes de temps sont aussi à prendre en compte.

Étant données les contraintes de prévisibilité et de sûreté de fonctionnement évoquées plus haut, une caractéristique hautement souhaitable est le **déterminisme**. Les comportements des systèmes doivent être parfaitement déterminés par leurs histoires d'entrée et ne doivent pas subir l'influence de choix internes arbitraires. Étant donnée la complexité de certains systèmes, le déterminisme n'est souvent garanti que sur les parties les plus critiques.

Un autre aspect important concerne la **vérification** des comportements. La simulation est souvent insuffisante pour des systèmes critiques. Une approche adaptée devrait permettre de vérifier de manière plus ou moins formelle que les systèmes conçus sont bien conformes aux spécifications.

Pour minimiser les sources d'erreur humaine et augmenter la fiablilité, le concepteur doit bénéficier d'un maximum d'outils de **traitements automatiques**. Dans le cas idéal, ces outils devraient permettre d'éliminer les discontinuités qui existent entre les phases de spécification, de conception et d'implémentation. Dans le domaine des systèmes temps réel, plus qu'ailleurs, les implémentations finales doivent être sûres et efficaces.

D'autres aspects essentiels tels que la portabilité et la maintenabilité devraient être favorisés par une approche modulaire et hiérarchique.

2.2 Approches traditionnelles

Les premières réalisations de systèmes temps réel ont successivement fait appel aux technologies de l'éléctronique analogique puis numérique. Avec les mini-ordinateurs industriels sont apparues les premières solutions informatiques, programmées directement en assembleur. La complexité croissante des systèmes et l'augmentation de la durée de vie des logiciels ont forcé les concepteurs à s'éloigner du matériel pour adopter des approches de plus haut niveau. Nous rappelons ici les approches les plus couramment rencontrées :

Automates : Programmer un système directement sous forme d'automate permet d'obtenir un comportement déterministe et une exécution efficace. De plus des outils d'analyse permettent de vérifier formellement le comportement programmé. En revanche, le parallélisme et la hiérarchie ne sont pas directement exprimables et leur prise en compte conduit rapidement à une "explosion" de l'espace d'états. Un autre inconvénient des automates est qu'ils sont difficilement maintenables : une petite modification des spécifications peut entraîner un bouleversement de la structure.

Langages de haut niveau + exécutif temps réel multi-tâche : Cette approche est largement répandue dans l'industrie. Le système est conçu comme un ensemble de tâches séquentielles qui coopérent. L'exécutif fournit les services nécessaires au parallélisme, à la communication et à la synchronisation entre les tâches. Bien qu'efficace à l'exécution, cette solution souffre de l'asynchronisme entre les différentes tâches qui rend l'analyse et la mise au point difficiles et le comportement global indéterministe. D'autre part, le recours direct aux services de l'exécutif rend les programmes peu portables. Des travaux de normalisation comme SCEPTRE [Bur84] ou POSIX *temps réel* visent précisément à résoudre ce problème.

Langages parallèles asynchrones : Dans ces langages, la gestion du temps, du parallélisme, de la communication et de la synchronisation font l'objet de primitives spécifiques. C'est le cas notamment pour des langages comme ADA avec le concept de tâche et de rendez-vous. Ces langages sont un progrès du point de vue de la portabilité et du confort de programmation. Toutefois le problème fondamental de l'asynchronisme subsiste.

2.3 Exemples d'applications

Nous présentons ici deux applications réactives typiques. La première est caractéristique des systèmes de contrôle de procédés industriels et a été traitée dans un article [ABD98] (voir annexe B). La deuxième application mêle des aspects interface homme/machine et des aspects protocole de communication. Elle sera traitée dans le chapitre 6.

2.3.1 Contrôle d'un atelier de production automatisé

Il s'agit d'un atelier d'usinage de pièces métalliques. Cet exemple a été proposé par les chercheurs de la FZI[1] comme modèle pour l'évaluation des différentes méthodes formelles de développement des systèmes réactifs [LL95]. L'atelier existe réellement, mais nous avons utilisé pour le test un logiciel de simulation qui a été développé à la FZI. C'est une fenêtre graphique TCL-TK (figure 2.1) qui simule les différents dispositifs de l'atelier et leurs évolutions.

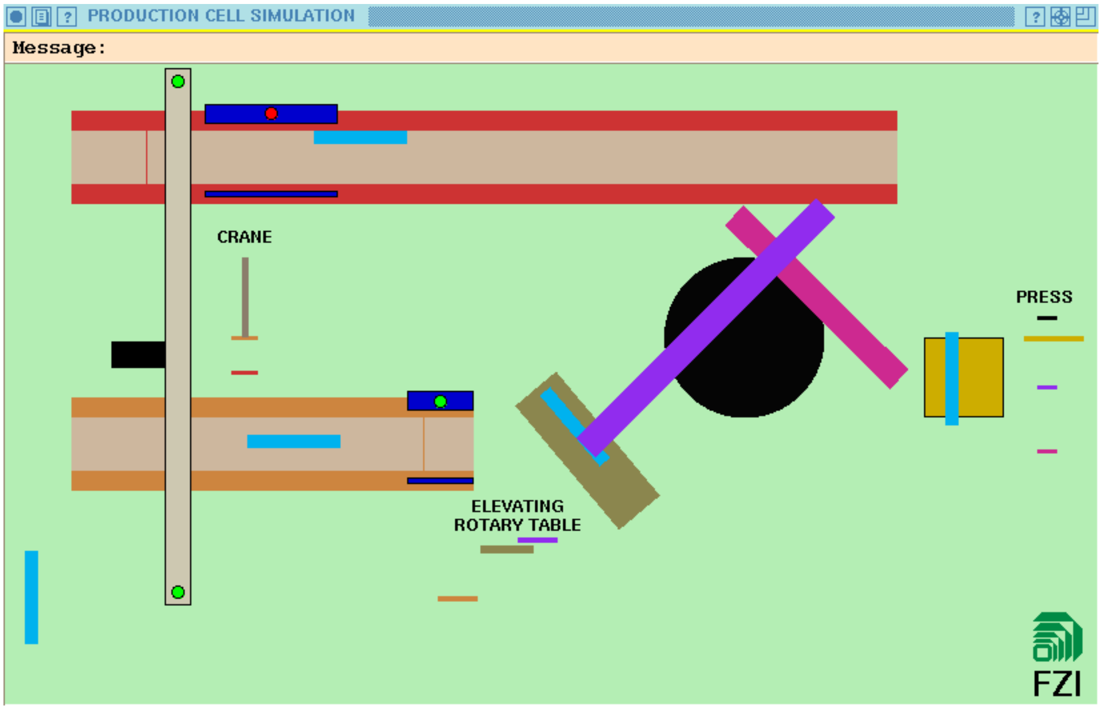


FIG. 2.1 - *Représentation graphique de l'atelier*

L'atelier est constitué de 6 éléments : un convoyeur d'alimentation, une table rotative-élévatrice, un robot à deux bras, une presse, un convoyeur de dépôt et une grue mobile. Le processus d'usinage se déroule de la manière suivante :

1. Forschungszentrum Informatik de Karlsruhe

chaque pièce est acheminée par le premier convoyeur vers la table rotative. Le robot est ensuite chargé de la transférer vers la presse. Une fois la pièce pressée, le robot effectue le transfert vers le deuxième convoyeur. Cet atelier étant conçu pour l'expérimentation, le processus doit pouvoir évoluer sans l'intervention d'un opérateur extérieur. Ainsi, la presse ne modifie pas réellement la forme des pièces et celles-ci sont réintroduites dans la chaîne en fin de parcours : la grue mobile déplace les pièces usinées du convoyeur de dépôt vers le convoyeur d'alimentation. La tâche du contrôleur est de commander les différents dispositifs et d'enchaîner les différentes opérations.

Le contrôleur connaît l'état du système grâce à un ensemble de capteurs (14 au total). Ce sont soit des capteurs *tout-ou-rien* (contacts ou cellules photo-électriques), soit, pour certains dispositifs, des potentiomètres ou codeurs (angle de rotation de la table, extension des bras du robot,...).
Les mouvements des dispositifs sont commandés par 13 actionneurs.

Parmi les problèmes posés par la conception du contrôleur, on notera les points suivants :

- Contraintes de temps : dans ce type de système, le non respect des contraintes temporelles peut avoir de graves conséquences et ne peut pas être toléré.
- Parallélisme : il est évident qu'on a intérêt à pouvoir exprimer le contrôleur comme un ensemble d'activités qui s'exécutent en parallèle.
- Synchronisation : ces différentes activités doivent se synchroniser pour réaliser le comportement global désiré.
- Modularité : l'idéal serait d'associer à chaque dispositif un module dont il serait exclusivement en charge.
- Réutilisabilité : certains comportements typiques, comme la commande d'un appareil dont la position est repérée par un capteur numérique, apparaissent à plusieurs reprises. Ils devraient faire l'objet de modules génériques.

2.3.2 Contrôle d'un ATM

Dans cette application, les aspects temporels sont moins présents mais les comportements à exprimer sont plus complexes et font intervenir des traitements transformationnels. Il s'agit d'un **système réparti** composé de distributeurs automatiques de billets (ou *Automatic Teller Machine*, en américain) reliés à une banque (figure 2.2). Chaque ATM peut servir un client à la fois et communique avec la banque qui centralise les comptes de tous les clients. L'ATM est une interface entre un client et la banque : elle interagit localement avec le client et de manière distante avec la banque.

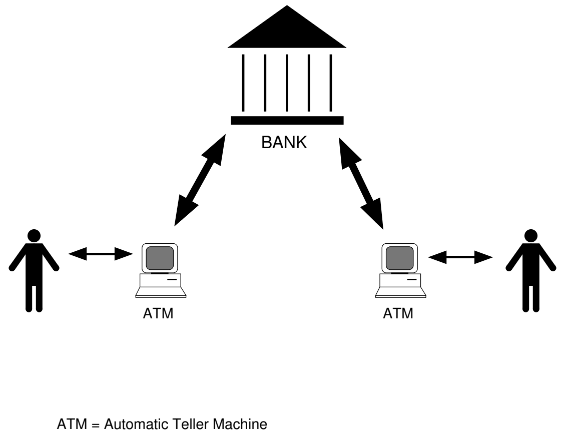


FIG. 2.2 - *Distributeur automatique de billets*

Même si les contraintes de temps ne sont pas au centre des préoccupations, cette application présente les caractéristiques d'un système réactif temps réel :

- **Comportement réactif complexe** : Un ATM doit être capable de gérer 2 activités en parallèle : le dialogue avec le client et le dialogue avec la banque. Si on peut qualifier l'interaction ATM-banque de processus interactif, l'interaction ATM-client est quant à elle permanente : à tout moment, le client peut par exemple annuler une transaction en cours. Le comportement global d'un ATM est celui d'un système réactif dans lequel parallélisme et préemption jouent un rôle majeur.

- **Sûreté de fonctionnement** : étant donné les sommes qui peuvent être en jeu, le comportement d'un ATM doit être prévisible. L'idéal est d'avoir des **garanties formelles** assurant certaines propriétés de sûreté. Il s'agit de garantir par exemple que si le client entre 3 fois un code erroné alors la carte est confisquée ou bien, si une transaction est annulée, alors aucune somme n'est délivrée.

- **Traitements transformationnels** : la gestion des comptes des clients et les algorithmes de vérification des codes secrets sont des traitements typiquement transformationnels. On a donc affaire à un exemple réaliste qui mêle, comme c'est souvent le cas, parties réactives et parties transformationnelles.

Cette application a initialement été introduite comme exemple d'illustration de la méthode OMT [RBP$^+$91]. Elle a ensuite été reprise pour illustrer le concept de Processus Réactifs Communicants dans lequel des modules synchrones indépendants communiquent par rendez-vous [BRS93]. Nous l'utiliserons ici pour illustrer les travaux effectués sur les machines d'exécution. L'exemple complet est traité dans le chapitre 6.

2.4 Conclusion

Dans ce chapitre, nous avons caractérisé les systèmes auxquels nous nous intéressons. Il s'agit des **systèmes réactifs temps réel à contrôle dominant**. Nous avons vu que les spécificités de ces systèmes étaient mal prises en compte par les approches traditionnelles. Deux exemples type de systèmes réactifs qui nous ont servi de support d'expérimentation ont été présentés.

Dans le chapitre suivant, nous présentons l'approche synchrone qui est spécialement dédiée au développement de ce type de systèmes.

Chapitre 3

Une méthode de développement basée sur l'approche synchrone

Objectifs

- Présenter les principes de base et l'intérêt de l'approche synchrone pour la conception des systèmes réactifs.
- Dégager les concepts de base du langage ESTEREL et donner un aperçu de ses instructions.
- Introduire le formalisme graphique SYNCCHARTS.
- Présenter le concept d'Objets Synchrones.
- Mettre en évidence le problème de l'implantation des programmes issus de l'approche synchrone.

3.1 Fondements de l'approche synchrone

Les solutions traditionnelles n'apportent pas de solutions satisfaisantes à la programmation des systèmes réactifs temps réel. Une autre façon d'aborder le problème est de mettre en avant les spécificités des systèmes réactifs temps réel et développer des langages dédiés, plutôt que d'*adapter* les langages à usage général. Cette approche est exposée dans un article de Gérard Berry [Ber89].

Hypothèses synchrones : Les systèmes temps réel sont par nature asynchrones dans leurs évolutions. L'asynchronisme les rend difficiles à décrire et à analyser. Les hypothèses synchrones conduisent à une vision abstraite des interactions et permettent ainsi de simplifier l'expression des comportements réactifs complexes. Les hypothèses de base sont les suivantes :

- Les **signaux** sont une abstraction des communications, entre le système et son environnement d'une part, et entre les différentes parties constituant le système, d'autre part.
- Les programmes synchrones considèrent un **temps logique** constitué d'une suite d'instants au cours desquels le système évolue. Cette vision idéalisée permet de faire abstraction du temps physique. Avec la notion d'instant, la simultanéité d'événements prend un sens et peut conduire à des comportements spécifiques.
- Les réactions se font en **temps nul**. Cette hypothèse est la plus caractéristique de l'approche synchrone. Les signaux émis pendant une réaction sont donc simultanés avec les signaux qui ont provoqué la réaction.
  Cette hypothèse est tout à fait acceptable au niveau d'un modèle. Elle peut poser des problèmes lors de l'implémentation. Nous verrons qu'il est en fait essentiel d'assurer l'*atomicité* des réactions.
- La communication à l'intérieur du système se fait par **diffusion instantanée** des signaux. Toutes les parties du système voient donc les mêmes signaux aux mêmes instants.

La principale conséquence de ces hypothèses est que l'on peut composer des systèmes synchrones pour former d'autres systèmes synchrones : le comportement résulant est parfaitement défini et déterministe. On peut ainsi utiliser à volonté le parallélisme, la séquence et la préemption pour exprimer des comportements sophistiqués. Ces 3 concepts (parallélisme, séquence et préemption) sont orthogonaux : ils peuvent être utilisés à tous les niveaux et dans n'importe quel ordre. Un point remarquable est que la composition parallèle n'introduit aucun indéterminisme.

L'hypothèse synchrone est à la base des langages ESTEREL[BG92], LUSTRE[CHPR91] et SIGNAL[LGLL91]. Ces 3 langages ont des sémantiques mathématiques rigoureuses et puissantes qui n'autorisent aucune ambiguïté d'interprétation comportementale. Ces fondements théoriques sont essentiels car ils sont à la base des techniques de compilation. Le code produit par les compilateurs de langages synchrones ne s'exécute bien sûr pas en temps nul mais il est efficace et déterministe. Un autre avantage de ces fondements théoriques est que les systèmes peuvent faire l'objet de vérifications formelles.

3.2 Formalismes pour l'expression du contrôle

Des langages synchrones comme LUSTRE et SIGNAL sont bien adaptés pour les systèmes réactifs dans lesquels le traitement des données joue un rôle essentiel.

C'est le cas notamment dans les applications d'automatique et de traitement du signal. Pour notre part, nous nous intéressons aux applications réactives à dominante contrôle. Le langage ESTEREL et le formalisme graphique SYNCCHARTS sont mieux adaptés à la programmation de ce type de systèmes.

3.2.1 Le langage Esterel

3.2.1.1 Présentation

ESTEREL est un langage synchrone qui adopte un style de programmation impératif. Ses caractéristiques essentielles sont :

- **Réactivité :** C'est une caractéristique intrinsèque au modèle liée aux hypothèses synchrones. Un système réactif évolue sous l'influence de stimuli externes qui provoquent des réactions ou *instants*. ESTEREL propose des primitives dites "réactives" faisant référence à ces stimuli et permettant d'exprimer la sensibilité aux événements sporadiques. De plus, les réactions à ces événements sont instantanées.

- **Instantanéité des réactions :** Ce principe interdit toute possibilité de réactions partielles concurentes d'un système, éliminant ainsi des sources d'indéterminisme, ce qui permet de "mettre à plat" le parallélisme.

- **Diffusion instantanée des signaux :** Le signal est l'entité de base du langage. Il permet communication et synchronisation. Les signaux sont *diffusés* : à moins d'avoir été déclarés locaux, ils sont visibles dans toutes les composantes d'un même programme. De plus, cette diffusion est *instantanée* : un signal émis par une composante est reçu par toutes les autres composantes dans le même instant. Ainsi, à un instant donné, toutes les parties d'un programme ont une vision globale commune.

- **Déterminisme :** La notion de déterminisme est souvent incompatible avec celle de parallélisme. La sémantique de l'opérateur `||` d'ESTEREL est, elle, parfaitement déterministe. Les compilateurs, se basent sur la sémantique mathématique du langage et rejètent tout programme indéterministe.

3.2.1.2 Instructions du langage

Nous utiliserons le langage ESTEREL dans la suite du mémoire pour exprimer certains comportements. Une présentation complète du langage est disponible dans le *"primer"* [Ber97]. Nous présentons ici quelques unes des instructions caractéristiques sans nous soucier de leur qualité de primitive ou de dérivée. On distinguera :

- Les instructions impératives classiques,

- Les instructions réactives : celles qui font référence aux instants

Instructions impératives classiques

- L'unité de compilation est le `module`. Il est constitué d'une interface définissant les signaux échangés avec l'environnement et d'un comportement.

- Traitement des données : appel de procédure par `call`, appel de fonction (par son nom), affectation (opérateur `:=`). Notons que l'exécution de ces instructions est supposée instantanée.

- Structures de contrôle classiques : la séquence (opérateur `;`), l'itération (`loop ... end loop`) et l'alternative (`if ... then ... else ... end if`).

- L'opérateur de composition parallèle `||`. Sa sémantique est du type *fork-join* : les différentes composantes démarrent simultanément et l'ensemble termine dès que toutes les composantes ont terminé. Cet opérateur a la propriété remarquable de conserver le déterminisme (les variables partagées ne sont pas autorisées).

- Un mécanisme d'échappement et de traitement des exceptions (`trap ... in ... end trap`).

Toutes ces instructions s'exécutent en temps nul.

Instructions réactives Ces instructions sont caractéristiques du langage ESTEREL. Elles font référence aux instants.

Émission/Attente de signaux : Le signal est l'entité de communication et de synchronisation. Un signal peut être purement événementiel ou peut porter une valeur (signal valué). Les signaux sont émis soit par l'environnement, soit par le programme lui-même. L'émission d'un signal `S` se fait par l'instruction :

```
emit S
```

L'attente d'un signal `S` se fait par l'instruction :

```
await S
```

Le flot de contrôle appelant se bloque alors jusqu'à la prochaine occurrence de `S` à un instant ultérieur. Le mot clé `immediate` indique que l'occurrence attendue peut appartenir à l'instant courant. Ainsi la séquence :

```
emit S; await immediate S
```

termine instantanément.

Préemption : La préemption est un concept de premier plan du langage, au même titre que le parallélisme ou la communication. Les primitives de préemption sont puissantes et permettent l'expression concise de comportements sophistiqués. La préemption consiste à interrompre un traitement en présence d'événements particuliers. Trois types de préemption existent en ESTEREL : l'avortement fort (`abort`), l'avortement faible (`weak abort`) et la suspension (`suspend`).

```
abort
  await DEBUT;
  emit CONSEQUENCES
when ARRET do
  emit TERMINE
end
```

Dans cet exemple, si le signal `ARRET` survient avant ou en même temps que `DEBUT`, l'émission du signal `CONSEQUENCES` n'a pas lieu. Le traitement d'exception consiste alors à émettre le signal `TERMINE`.
Avec un avortement faible (`weak abort`), l'activité préemptée est autorisée à accomplir ses *"dernières volontés"* :

```
weak abort
  await DEBUT;
  emit CONSEQUENCES
when ARRET do
  emit TERMINE
end
```

Si `ARRET` arrive en même temps que `DEBUT`, le signal `CONSEQUENCES` est quand même émis.
Le 3ème type de préemption est la suspension par la primitive `suspend`. Contrairement à l'avortement pour lequel l'activité est interrompue de manière définitive, une activité suspendue peut reprendre au point où elle a été préemptée une fois que les conditions de suspension ont disparu.

Tâches asychrones : Tous les traitements qui ont lieu au cours d'une réaction d'un programme ESTEREL sont considérés comme instantanés. Le besoin d'intégrer des traitements du type transformationnel dont la durée ne peut pas être négligée a donné lieu à la notion de tâche ESTEREL [Par92]. Les tâches s'exécutent en parallèle au programme. La primitive `exec` permet de gérer ces exécutions d'un point de vue *logique*. Une partie importante de notre travail porte sur l'exécution effective de ces tâches (voir 5.4).

3.2.2 Les SyncCharts

Les SyncCharts sont un modèle synchrone graphique : ils intègrent les concepts de haut niveau des langages synchrones dans un formalisme graphique expressif. L'aspect graphique est fortement inspiré des StateCharts [Har87] mais ils offrent une plus grande richesse au niveau des préemptions et surtout, ils bénéficient d'une sémantique plus stricte basée sur celle du langage Esterel.

Pour présenter ce modèle, nous allons prendre l'exemple du chien de garde de la figure 3.1. Il s'agit d'une alarme que l'on peut armer, désarmer ou geler.

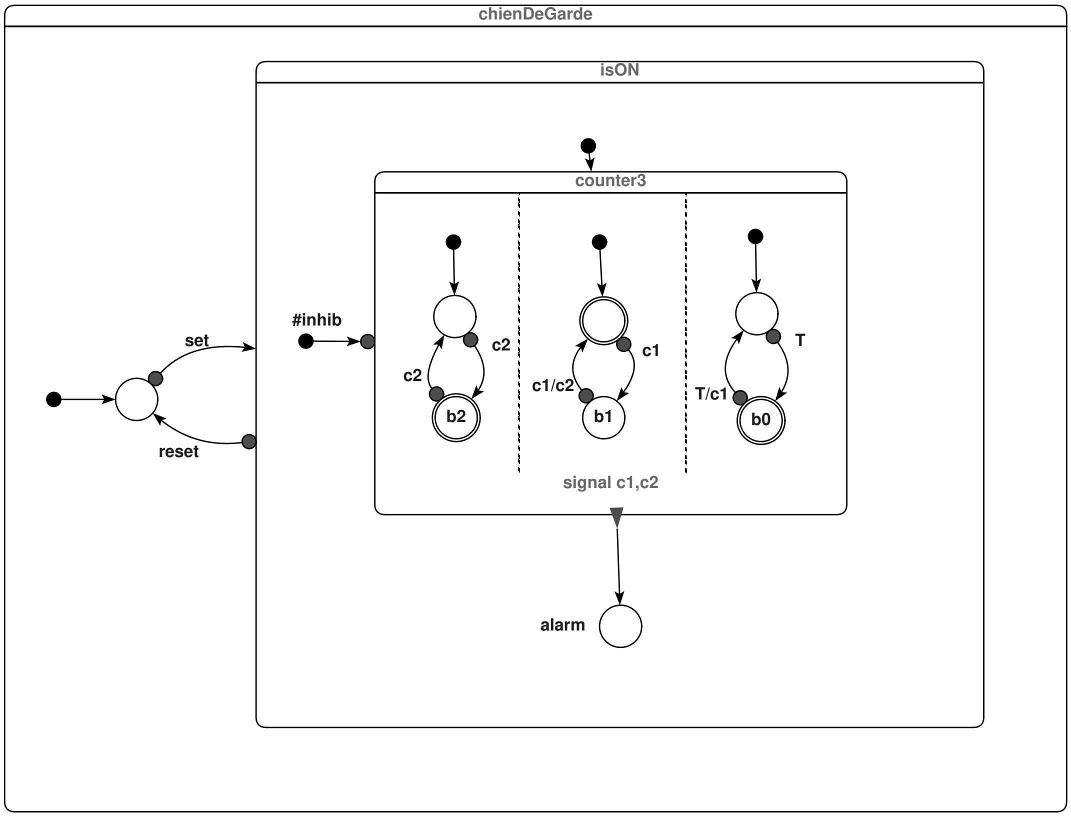


Fig. 3.1 - SyncCharts *de l'application "Chien de garde"*

Le modèle est basé sur la notion d'états et de transitions. Il possède à la fois les caractéristiques des machines de Moore et de Mealy. Ainsi un signal de sortie (ou action) peut être associé soit à un état (cas de b_0), soit à une transition (cas de c_1). Le signal T est un signal d'entrée représentant l'unité de temps. Les états initiaux se distinguent par un arc dont l'origine est un *connecteur*. Les états finals sont quant à eux représentés par un double cercle.

Un syncChart peut contenir plusieurs graphes d'états qui évoluent en parallèle. Les différentes composantes sont séparées par une ligne pointillée. L'état `counter3` est un compteur 3 bits formé de 3 compteurs 1 bit parallèles.

Les signaux c_1 et c_2 représentant les retenues des compteurs un bit sont des signaux locaux utilisés pour synchroniser les 3 compteurs. Étant donné le caractère synchrone du modèle, toutes les composantes perçoivent simultanément les mêmes informations. Ainsi, sur une occurrence de `T`, le compteur passera directement de l'état 3 à l'état 4.

L'aspect hiérarchique de la description apparaît clairement sur la figure. Les états non raffinés sont représentés par des ellipses. Les états raffinés, appelés aussi macro-états, sont des rectangles à coins arrondis.

Les différents types de préemption d'ESTEREL sont directement exprimables en SYNCCHARTS et correspondent chacun à différents arcs de transitions. Sur la figure, le macro-état `counter3` est suspendu chaque fois que le signal `inhib` est présent. Le macro-état `isON` est quant à lui définitivement préempté par le signal `reset`. La préemption forte est indiquée par le rond grisé à l'origine de l'arc.

Une des caractéristiques des SYNCCHARTS est la notion de terminaison normale. Un macro-état termine spontanément ou *normalement* quand toutes ses composantes parallèles sont dans un état final. L'arc de terminaison normale désigné par un triangle à l'origine est alors franchi. Ainsi quand le compteur passe à l'état 5, l'alarme est déclenchée.

Comme nous utilisons les SYNCCHARTS dans nos applications, nous donnons une description concise de la syntaxe en annexe A.

3.2.3 Environnement de programmation

L'ensemble ESTEREL, SYNCCHARTS bénéficie d'un environnement de développement constitué d'outils facilitant la conception, la simulation et la vérification des programmes. Cet environnement est actuellement à l'état de prototype mais une version industrielle est en cours de développement [ABD97]. Les outils disponibles sont :

- Le compilateur de SYNCCHARTS permet de traduire une description graphique en programme ESTEREL.
- Le compilateur ESTEREL est l'outil central. Les programmes ESTEREL peuvent être compilés soit vers un format d'automates déterministes, soit vers

des systèmes d'équations [1].

- Les simulateurs XES et CSIMUL sont utilisés pour mettre au point les programmes.

- L'outil de preuves XEVE est utilisé pour vérifier des propriétés de sûreté.

Le format DC [2], commun aux formalismes synchrones, contribue à unifier les outils au sein d'une plate-forme synchrone commune. D'autres outils sont également au centre de notre approche. Ces outils sont relatifs aux objets synchrones et sont présentés dans la section suivante.

3.3 Les objets synchrones

3.3.1 Introduction

Nous avons vu l'intérêt des langages synchrones pour la programmation des systèmes réactifs. Dans la réalité, il est rare qu'une application soit purement réactive et on a le plus souvent affaire à des applications comportant des parties réactives, des parties transformationnelles et des parties interactives. Ces applications entrent donc dans le cadre plus général du génie logiciel. Or une approche a fait ses preuves dans ce domaine, il s'agit de l'approche orientée objets : le concept d'objet améliore nettement la maintenabilité et la réutilisabilité des logiciels. La combinaison de l'approche objet et de modèles de systèmes réactifs fait déjà l'objet de méthodes de conception comme ROOM [SGW94] ou plus récemment UML temps réel [Dou98]. Ces méthodes ne sont toutefois pas basées sur le paradigme synchrone. L'approche de F. Boulanger [Bou93] permet, elle, de concilier programmation synchrone et programmation par objets. F. Boulanger propose un environnement de développement comprenant les bibliothèques et outils permettant d'intégrer des modules synchrones dans le langage `C++`.

3.3.2 Les langages à objets

Les langages à objets doivent leur intérêt à deux concepts fondamentaux :

- L'**Encapsulation** qui consiste à regrouper des données et les opérations sur ces données dans une même entité appelée *objet*. Les données constituent l'*état* de l'objet et les opérations sont les *méthodes* de l'objet. L'ensemble des méthodes constitue le *protocole* de l'objet. Avec cette approche, il suffit

1. Les progrès récents en matière de compilation ESTEREL, notamment l'utilisation de la sémantique dite de “circuits” [Ber98], permettent de bénéficier des outils d'optimisation qui existent pour les circuits.

2. Declarative Code.

de connaître le protocole d'un objet pour le manipuler. Cette propriété, appelée "abstraction des données" est remarquable car elle permet de séparer clairement spécification et implémentation.
La structure et les méthodes des objets sont décrites dans une *classe* qui est une sorte de moule. Les objets construits à l'aide de ce moule sont appelés *instances* de la classe.

- L'**Héritage** : quand deux classes sont proches, soit par leur structure, soit par leur protocole, l'héritage consiste à décrire l'une à partir de l'autre, en ne spécifiant que les différences. Ainsi une sous-classe hérite de la structure et du protocole de sa super-classe qu'elle aura enrichi et/ou surchargé. L'héritage permet de structurer l'implémentation par une *hiérarchisation* des objets et des concepts associés. Cette structure hiérarchique favorise la réutilisabilité des objets et l'incrémentalité du développement.

À l'époque de ses travaux, F. Boulanger avait essentiellement le choix entre SMALLTALK-80 et `C++` pour implémenter les objets synchrones. Son choix du langage `C++` semble avoir été guidé par plusieurs facteurs. Premièrement, `C++` s'est largement répandu et est devenu une norme de fait dans l'industrie. Ensuite, `C++` est un langage compilé et fortement typé. Le compilateur permet de détecter bon nombre d'erreurs et il produit un code efficace. Les mécanismes tels que la vérification des types, le contrôle d'accès, la résolution des méthodes, se font systématiquement à la compilation à chaque fois que cela est possible. `C++` offre en plus un mécanisme de généricité (*templates*) qui a été largement utilisé pour les objets synchrones. Notons toutefois que `C++` offre les possibilités du langage `C` ce qui n'est pas une garantie de sécurité pour les programmes développés.

3.3.3 Les Objets Synchrones

L'objectif est d'intégrer des programmes synchrones dans la programmation par objets. Dans l'approche synchrone, l'unité de compilation est le module. Ce dernier est caractérisé par ses entrées, ses sorties et son comportement, il est donc déjà très proche de la notion d'objet. L'idée naturelle est de traduire un programme synchrone en un objet qu'on appellera objet synchrone.
Le processeur OCC++ développé par F. Boulanger traduit automatiquement un programme synchrone en une classe synchrone. Cette traduction se base sur le format intermédiaire OC commun aux langages ESTEREL et LUSTRE. La classe générée implémente le comportement réactif spécifié dans le formalisme synchrone source. Elle possède en plus une interface standard élaborée comprenant essentiellement les méthodes suivantes :

- Une méthode d'accès pour chaque signal d'entrée,
- Une méthode d'accès pour chaque signal de sortie,

– Une méthode de transition.

Le premier avantage de cette approche est la *dynamicité* : contrairement à l'implémentation procédurale en C standard qui se prête mal à la duplication, on peut maintenant créer dynamiquement de nouveaux objets synchrones par simple instanciation.
Le deuxième avantage est dû à la séparation entre interface et implémentation : l'utilisateur d'un objet synchrone s'adresse de la même manière à celui-ci qu'il soit implémenté par un automate ou par un système d'équations. On peut aussi implémenter un objet synchrone directement en `C++` du moment que l'interface est respectée. Cette interface standard agit comme un "ciment" favorisant :

– la coopération entre programmes traditionnels et programmes synchrones au sein d'une même application,

– mais aussi, la coopération entre les différents formalismes synchrones facilitant ainsi une approche multiformalisme des systèmes réactifs.

3.3.4 Communication synchrone

Les objets synchrones ne communiquent avec l'extérieur que par l'intermédiaire de leurs signaux d'interface. La bibliothèque `libSync` propose un mécanisme de communication point-à-point unidirectionnel qui permet de *connecter* des objets synchrones entre eux. On peut ainsi former des *réseaux* d'objets synchrones interconnectés. Un des objectifs est de faire en sorte que le comportement *global* de ces réseaux soit synchrone. Il faut pour cela constituer une base de temps globale de manière à ce que chaque signal ait une valeur unique durant un instant. Il suffit pour cela que le graphe de dépendance entre objets soit acyclique. Dans ce cas, on peut déterminer un ordre de réaction assurant une vision unique de tous les signaux par l'ensemble des objets. La figure 3.2 (extraite de [Bou93]) montre que si l'on fait réagir l'objet A puis l'objet B, le comportement de l'ensemble (A,B) reste synchrone.

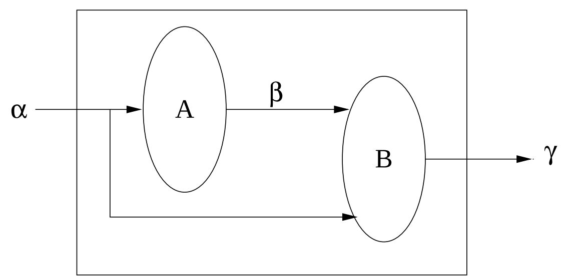


FIG. 3.2 - *Interconnexion d'objets synchrones*

À chaque demande de connexion, le graphe est recalculé et la connexion peut être refusée si elle engendre un cycle de dépendance. On appellera cette méthode de composition le "séquencement synchrone".

Un point important est que les connexions entre objets ainsi que le nombre d'objets synchrones peuvent varier dynamiquement. à chaque instant, les demandes de modification de la topologie sont enregistrées, et un nouvel ordonnancement est calculé pour l'instant suivant. Cette possibilité est très intéressante car elle pallie le manque de dynamicité des langages synchrones. On trouve cet aspect dynamique dans de nombreux systèmes réactifs. Dans les systèmes de gestion du trafic aérien, par exemple, des contrôleurs sont créés et détruits à la demande pour chaque avion qui pénètre ou quitte l'espace aérien.
Un autre avantage de cette composition existait pour les versions 3 et antérieures du compilateur ESTEREL. Ces versions ne pouvaient produire que des automates et quand une application comportait plusieurs modules en parallèle, on arrivait rapidement à une explosion de l'espace d'états. Un mode de compilation dit mode "cascade" consistait à compiler les modules séparément et à les faire réagir séquentiellement ce qui évitait la construction de l'espace produit. L'approche de F. Boulanger permet d'obtenir un résultat équivalent en traduisant chaque module en objet puis en les composant par un séquencement synchrone. On notera toutefois que le problème de taille des automates ne se pose plus depuis la version 5 du compilateur ESTEREL.

Dans un réseau d'objets synchrones, une relation de dépendance entre deux objets (définie par une connexion) ne correspond pas nécessairement à une relation de causalité. Certains réseaux peuvent ainsi être rejetés bien que corrects. Un autre inconvénient vient du fait qu'on ne peut pas établir de preuves formelles sur le comportement global d'un ensemble d'objets synchrones à l'aide des outils dédiés au synchrone.

Modèle d'exécution Deux manières de gérer les réactions des objets ont été implémentées :

- Mode multi-flots : dans la première version de `libSync`, un flot de contrôle (*thread*) est associé à chaque objet synchrone. En début d'instant, l'ordonnanceur fait réagir tous les objets simultanément et chaque objet commence par lire ses signaux d'entrée. Un mécanisme à base de sémaphores de signalisation rend cette opération bloquante si le signal n'a pas été mis à jour. Ainsi un objet qui tente de lire un signal reste bloqué tant que l'objet responsable de ce signal n'a pas réagit. L'ordonnancement se fait ainsi par contrainte sur l'accès aux signaux d'entrée. L'inconvénient de ce mode d'exécution est la nécessité de recourir à un système multi-tâches.

- Mode séquentiel : dans la version actuelle de `libSync`, l'ordonnanceur fait réagir les objets séquentiellement dans un ordre qu'il a déterminé en analysant le graphe de connexion.

3.3.5 Composition statique d'objets synchrones

La topologie des réseaux d'objets synchrones pouvant varier dynamiquement, l'ordonnancement est recalculé à chaque instant. Ceci est un inconvénient majeur, surtout pour des systèmes temps réel. Quand la topologie est fixe et connue a priori, l'ordonnancement peut être déterminé une fois pour toutes. Le langage MDL (**M**odule **D**escription **L**angage) et l'outil MDLC (MDL to C compiler) ont été introduits par F. Boulanger pour répondre à ces besoins.
Le langage MDL permet de :

- Décrire l'interface (la sorte) d'un objet synchrone. Une telle description peut être générée automatiquement pour chaque classe produite par OCC++. Ce sont les fichiers `.mdf` (**m**odule **d**escription **f**ile).
- Construire de nouveaux objets synchrones par composition statique d'autres objets synchrones. La description comprend essentiellement l'interface de l'objet composite, les déclarations des objets composants ainsi qu'une liste de connexions. Ce sont les fichiers `.mdl` (**m**odule **d**escription **l**angage).

Le processeur MDLC utilise les fichiers de description de l'objet composite et les fichiers de description de l'interface des composants pour vérifier la validité de la composition : vérification des types des signaux et de l'absence de cycle de dépendance. Quand la description est valide, une classe `C++` implémentant l'objet synchrone composite est produite. Deux types de composition sont proposés :

Agrégation : L'objet composé est décrit en MDL comme une agrégation d'objets synchrones interconnectés. Cette agrégation se retrouve dans la structure interne de la classe `C++` générée. Vue de l'extérieur, une instance de cette classe a le comportement d'un objet synchrone ordinaire.

Dérivation : Dans l'approche orientée objet, la dérivation correspond à une spécialisation de la structure et du protocole d'une classe. Ce concept s'adapte facilement aux modules synchrones :

- On enrichit la structure et le protocole d'une classe synchrone en ajoutant de nouveaux modules synchrones et de nouveaux signaux d'interface.
- On surcharge le protocole d'une classe synchrone en traitant différemment les signaux d'interface.

Pour garder la sémantique de l'héritage, il faut en plus imposer des contraintes sur le comportement réactif des classes dérivées. Le principe retenu est le suivant : l'altération du comportement dans une classe dérivée ne pourra se faire que par ajout de modules de filtrage en entrée et/ou en sortie[3].

3. Cette contrainte ne résout toutefois pas le problème puisqu'un module de filtrage suffit à modifier radicalement le comportement réactif de la super-classe.

On peut ainsi construire en MDL des classes synchrones dérivées à partir d'une classe de base en spécifiant les objets de filtrage, les nouveaux signaux d'interface et les nouvelles connexions.

3.3.6 Communication avec l'environnement

La communication entre un système synchrone (qu'il soit composé d'un ou plusieurs objets synchrones interconnectés) et l'environnement asynchrone se fait par l'intermédiaire d'*objets d'interface*. Un objet d'interface doit être associé à chaque signal d'entrée/sortie du système synchrone de manière à établir le lien entre ce signal et son équivalent dans l'environnement.
Vis-à-vis du monde synchrone, un objet d'interface possède des entrées, des sorties et peut être activé. Ce ne sont donc que des objets synchrones spécialisés :

- Un objet d'entrée est un objet synchrone qui n'a pas d'entrées (synchrones) et qui n'a qu'un signal de sortie. Sa méthode de transition provoque l'échantillonnage d'une donnée de l'environnement et la mise à jour du signal synchrone associé. Ne dépendant d'aucun signal synchrone, ces objets sont les premiers à être activés dans un réseau d'objets synchrones.

- Un objet de sortie est un objet synchrone qui n'a pas de sorties (synchrones) et qui n'a qu'un signal d'entrée. Sa méthode de transition met à jour une donnée de l'environnement en fonction de son signal d'entrée. Ces objets sont activés quand tous les autres objets du réseau ont réagit.

Le principe est que l'utilisateur doit développer ses propres objets d'interface pour les différents types de couplages synchrone/asynchrone qu'il envisage.

3.3.7 La bibliothèque synchrone (`libSync`)

La bibliothèque synchrone joue un rôle central dans l'environnement de développement. Elle contient l'ensemble des classes sur lesquelles reposent les processeurs `C++` et MDLC. Le recours intensif à des techniques telles que l'héritage (simple ou multiple), la généricité (classes paramétrées), ou les classes abstraites permet de minimiser le code généré. On donne dans ce qui suit un aperçu des principales classes de la bibliothèque :

- Classe `Synchronous` : c'est la classe des objets synchrones. Elle contient la structure et les méthodes communes à tous les objets synchrones. à ce niveau d'abstraction, on considère qu'un objet synchrone peut lire ses entrées, réagir et produire ses sorties. On ne connaît a priori ni le nombre, ni le type de ces entrées/sorties. C'est pourquoi on ne peut pas créer directement d' instances de cette classe : la classe `Synchronous` est *abstraite*. `C++` produit des classes *concrètes* dérivées de `Synchronous` et enrichies par des signaux d'entrées/sortie et un comportement réactif particuliers.

- Classes de signaux : le signal est le support de communication entre un objet synchrone et son environnement. Son rôle est donc essentiel. Dans `libSync`, un signal peut être vu comme une suite d'échantillons (couples *(présence,valeur)*) ou comme un canal de communication unidirectionnel entre 2 objets. Le premier aspect donne lieu aux classes `Signal<T>` (générique par rapport au type des valeurs) et `PureSignal` pour les signaux purs. Le deuxième aspect correspond aux classes `InputSignal` et `OutputSignal`. Ces classes sont abstraites. Les signaux réels héritent des 2 aspects à travers les classes concrètes suivantes : `InSignal<T>`, `OutSignal<T>`, `PureInSignal` et `PureOutSignal`.
  Les méthodes de connexion ne sont définies qu'entre un signal d'entrée et un signal de sortie évitant ainsi certaines incohérences. De plus le typage de `C++` interdit les connexions entre signaux de types différents.

- Classe `Scheduler` : c'est l'objet ordonnanceur. Il a une vue globale sur les objets dont il est en charge. Il enregistre les changements topologiques (connexions, déconnexions, création ou destruction d'objets) et vérifie l'absence de cycles. Sa méthode `tick()` est appelée pour déclencher un nouvel instant : elle calcule le nouvel ordre de réaction puis fait réagir les objets un à un.

- Classes `InputModule` et `OutputModule` : il s'agit des objets d'interface avec l'environnement. Ces classes dérivent de la classe `Synchronous`. Trois variantes existent permettant de positionner les entrées et visualiser les sorties à travers des interfaces textuelle ou graphiques pour X-Windows et Tcl/Tk.

3.3.8 Conclusion sur les objets synchrones

Les objets synchrones ont élargi le domaine de la programmation synchrone :

- Ils permettent d'intégrer des traitements synchrones dans des environnements d'objets. Les applications conçues par cette approche bénéficient à la fois du confort de la programmation par objets et de la rigueur du modèle synchrone.

- Par la même occasion, l'existence d'un protocole standard pour les objets synchrones favorise l'approche multiformalisme synchrone.

- La possibilité de créer et de détruire des objets synchrones en cours d'exécution introduit un caractère dynamique qu'on ne trouve pas dans les distributions actuelles des langages synchrones. Notons toutefois la proposition faite par S. Dissoubray dans sa thèse [Dis96].

Cette approche est concrétisée par un environnement de développement parfaitement opérationnel et remarquablement bien conçu comprenant les processeurs

OCC++, MDLC et la bibliothèque `libSync`.
Nous considérons les objets synchrones comme un acquis : l'approche de F. Boulanger est à la base de nos travaux aussi bien du point de vue des concepts que des réalisations.

3.4 Problème de la mise en œuvre

Si l'hypothèse synchrone est d'un grand intérêt pour les phases de conception et de vérification des systèmes, elle pose des problèmes lors de la phase de réalisation où l'on est confronté à la nature asynchrone du monde réel.

Ces problèmes seraient résolus si l'on disposait de machines infiniment rapides. Malheureusement ce n'est pas le cas.
Les implantations qui se rapprochent le plus de l'hypothèse d'instantanéité des réactions sont sans doute les implantations sous forme de circuits [Ber91] pour lesquelles communications et actions internes se font en un cycle d'horloge. La seule différence avec le modèle idéal est que les sorties ne sont pas simultanées avec les entrées mais sont produites en fin de cycle. Ce type de réalisation montre bien que, lors de la mise en œuvre, on peut approcher l'hypothèse d'instantanéité des réactions par le principe d'**atomicité des réactions** :

- La vision de l'environnement doit rester la figée durant les réactions,
- Toute réaction entamée doit être achevée (ou bien complètement annulée).

Ce principe d'atomicité, associé au respect des contraintes temporelles liées à la dynamique de l'environnement, sont à la base des implémentations. Le principe est le même pour les mises en œuvre logicielles : il s'agit de garantir des réactions atomiques puis de vérifier a postériori que le système est "assez rapide" vis-à-vis des évolutions de son environnement.

Toujours du fait des hypothèses synchrones, les programmes issus de l'approche synchrone ont une vision abstraite de leur environnement :

- Le temps considéré est un temps logique constitué d'une suite discrète d'instants.
- À chaque instant, les événements externes sont perçus sous la forme d'un vecteur de signaux appelé *événement logique*,

Nous appelerons *"processus synchrone"* le résultat de la compilation d'un programme synchrone. Pour ESTEREL, ces processus peuvent être soit des automates, soit des systèmes d'équations. Ils sont caractérisés par des signaux d'entrée/sortie et une fonction d'activation qui permet de déclencher les instants logiques.

Les processus synchrones ne sont pas directement exécutables. Pour obtenir un système opérationnel, il est nécessaire d'**enfouir** le processus synchrone dans son environnement. Cet enfouissement doit se faire en assurant une approximation raisonnable de l'hypothèse synchrone. Il s'agit en particulier de :

- Faire l'interface entre le processus synchrone et la cible d'exécution,
- Faire le lien entre les signaux logiques du processus synchrone et les événements réels de l'environnement,
- Faire le lien entre le temps logique et le temps réel.
- Se charger de l'exécution des traitements asynchrones (tâches ESTEREL).

L'entité qui se charge de cet enfouissement est appelée **machine d'exécution**.

3.5 Conclusion

Nous avons vu dans ce chapitre l'intérêt de l'approche synchrone pour la conception des systèmes réactifs. Les hypothèses synchrones permettent d'abstraire des comportements complexes et donnent aux formalismes synchrones des sémantiques rigoureuses et puissantes.

Le langage ESTEREL et les SYNCCHARTS sont spécialement dédiés à la conception des systèmes réactifs dans lesquels le contrôle est dominant. Les compilateurs produisent un code sûr et efficace. Les comportements programmés peuvent faire l'objet de preuves formelles.

Le concept d'**objets synchrones** et les outils associés permettent d'intégrer des programmes synchrones dans des environnements de développement par objet en les dotant d'une interface standard. Nous avons porté un intérêt particulier à la présentation des objets synchrones car ils constituent un apport important dont ont bénéficié nos implémentations.

Tous ces aspects représentent les points forts de l'approche. En revanche, nous avons vu dans la dernière partie que la mise en œuvre nécessitait une **machine d'exécution** dont le rôle est d'enfouir le processus synchrone dans son environnement asynchrone. Nous présentons dans le chapitre qui suit les machines d'exécution existantes qui ont retenu notre intérêt.

Chapitre 4

Précédentes implantations de processus synchrones

Objectifs

- Étudier l'existant en matière d'implantation de processus synchrone.
- Dégager les points que nous retenons de chacune des réalisations présentées.

4.1 Introduction

On peut classer les implantations de processus synchrones selon le niveau d'abstraction auquel elles se placent :

- Implantation du code haut niveau généré par les compilateurs de langages synchrones sur un exécutif temps réel.
- Génération de code cible pour un processeur donné.
- Câblage de l'application en utilisant de la logique reconfigurable.
- Développement d'une architecture matérielle dédiée à l'application.

Dans ce chapitre, nous présentons l'existant en matière d'implantation de processus synchrones. De nombreuses mises en œuvre ont sans doute été faites pour des applications données et sur des cibles données. Nous présentons ici celles qui nous semblent les plus pertinentes étant donnés les objectifs suivants :

- Prise en compte de l'asynchronisme machine/environnement,
- Possibilité de différents types de connexion entre signaux d'E/S et événements extérieurs,

- Intégration des tâches Esterel,
- Fonctionnement paramétrable,
- Indépendance architecture de la machine/cible d'exécution,
- Génération automatique.

4.2 Simulateurs

Un simulateur de processus synchrone représente une première implémentation de machine d'exécution mais il ne constitue pas une réelle confrontation entre le programme synchrone et le monde asynchrone. En effet des simulateurs comme Csimul ou Xes sont une interface, non pas entre le processus synchrone et le monde réel, mais entre le processus synchrone et un utilisateur. Ce type de simulateur permet de former les *événements logiques* et de déclencher les réactions du processus synchrone. La pertinence des événements formés ainsi que le choix des instants de réaction reposent entièrement sur l'utilisateur. Ce type de simulation permet donc de **vérifier le comportement logique** d'un système mais en aucun cas son comportement temporel.

Les simulateurs sont néanmoins indispensables en tant qu'**outil de mise au point**. Ils font de plus en plus souvent partie des environnements standard de développement des langages synchrones.
Dans l'environnement des premières versions d'Esterel, l'outil de simulation Csimul permettait à travers une interface textuelle de positionner les entrées, de déclencher les réactions du programme et de visualiser les sorties. Dans les versions suivantes, le simulateur Xes[1] a pris la place de Csimul. Fondamentalement, le fonctionnement est le même. En revanche la convivialité et les possibilités sont nettement améliorées : l'interface utilisateur d'Xes est un panneau graphique construit automatiquement, représentant les signaux d'entrées/sortie du module. Xes permet aussi de simuler l'exécution des tâches asynchrones : positionnement du signal de retour d'un `exec` et des arguments passés par référence.
Outre un grand nombre d'options accessibles par des menus déroulants, Xes facilite grandement le travail de mise au point en offrant la possibilité de remontée dans le code source : à chaque instant, on peut visualiser, en plus des signaux d'entrée/sortie, les primitives actives (points d'arrêt) ainsi que des variables et les signaux locaux du programme.

Dans l'environnement Esterel, le code C de la simulation est obtenu par l'option `simul` du compilateur Esterel. Cette option permet de produire, en plus du moteur de transition du processus synchrone, une grande partie des structures de données nécessaires à l'interfaçage des entrées/sorties du module avec le panneau de simulation graphique ainsi que des structures permettant la

1. X Esterel simulator

remontée dans le code source. L'édition des liens avec la bibliothèque XES produit une simulation exécutable.

Bien qu'indispensables pour la mise au point, les simulateurs ne constituent pas une implémentation réaliste de machine d'exécution car ils ne prennent pas en compte le parallélisme machine/environnement : les évolutions se font en alternance. D'autre part, ils ne permettent pas de relier l'application à des signaux réels.

Nous retiendrons le point suivant : les simulateurs sont des machines d'exécution particulières et à ce titre, on devrait pouvoir configurer une machine afin qu'elle se comporte comme un simulateur. Dans un but de prototypage rapide, on devrait même pouvoir réaliser des machines mixtes simulation/exécution dans lesquelles une partie des signaux d'E/S sont reliés aux événements réels tandis que les autres sont simulés par l'utilisateur.

4.3 Prototype de Raphaël Bernhard

Ces travaux [ABM+93] ont été réalisés dans le cadre d'une collaboration entre le CMA, l'I3S et le CNET. L'objectif était de définir, aussi bien du point de vue fonctionnel que structurel, la couche logicielle nécessaire à l'exécution d'un programme ESTEREL, indépendamment de la cible.

Le principe adopté est le suivant : une machine d'exécution présente toutes les caractéristiques d'un système réactif, il semble donc naturel de programmer son comportement en ESTEREL. Dans ces conditions, vis-à-vis de la machine, les réactions du module de l'application ne sont que des traitements transformationnels. La machine proposée est donc constituée d'une partie réactive écrite en ESTEREL qui contrôle une tâche exécutant les réactions du module synchrone. Le parallélisme entre machine et module synchrone permet de :

- Traiter l'arrivée de nouveaux signaux et construire de nouveaux événements d'entrée *pendant* les réactions de l'automate,
- Contrôler et surveiller les réactions de l'automate (transitions qui durent trop longtemps, par exemple).

La machine est composée de 4 modules ESTEREL s'exécutant en parallèle : mémorisation des entrées, construction de l'événement d'entrée, gestion de l'automate et émission des sorties. Ces modules font appel à des *procédures de stratégie* écrites en `C++` pour la prise de décision. Ces procédures définissent :

- la politique de construction des événements,

– la politique de déclenchement des transitions,

– la politique de traitement des anomalies (transition trop longue, par exemple).

Ce recours systématique à des procédures de stratégie permet de *paramétrer* le fonctionnement de la machine selon les besoins de l'application.

Un prototype a été réalisé sous UNIX. La tâche de transition de l'automate y est implémentée par un *thread*. Le support de threads sous-jacent est un simulateur de noyau CHORUS sous UNIX. Les tâches ESTEREL n'y sont pas traitées, ni le problème de la connexion des signaux d'entrée/sortie à l'environnement. Il semble d'ailleurs que pour l'acquisition des signaux d'entrée, seul le mécanisme d'interruption est envisagé. La génération automatique n'est pas abordée.

Concevoir une machine comme un squelette réactif qui pilote les réactions d'un automate correspond bien à la vision conceptuelle que l'on a d'une machine d'exécution. De plus, avec cette approche, on dispose de la puissance du langage ESTEREL pour concevoir et prouver des comportements sophistiqués.
En revanche, d'un point de vue pratique, cette approche pose le problème de la *machine d'exécution de la machine d'exécution* : le prototype réalisé utilise le simulateur CSIMUL comme machine d'exécution.

Les idées que nous retenons de ces travaux sont :

– Concevoir la machine d'exécution comme un système réactif,

– La nécessité de *paramétrer* le fonctionnement de la machine.

4.4 Travaux de M.A. Péraldi

4.4.1 Machine à évaluation par nécessité

4.4.1.1 Machine abstraite

Une partie des travaux de thèse de M.A. Péraldi [Per93] visait à définir une machine d'exécution *abstraite*.

Le passage du monde synchrone au monde asynchrone a été étudié. En particulier la correspondance entre le temps discret vu par le programme synchrone et le temps réel continu est formalisée. Cette formalisation a permis de dégager les notions de fonctions d'évaluation et de présence d'un signal d'entrée : ces fonctions représentent les points d'interaction entre l'automate et son environnement et sont à la base de la technique d'évaluation par nécessité.

Une des zones critiques de la machine d'exécution concerne précisément le partage des signaux d'entrée entre la machine et l'environnement. Cette partie critique a été étudiée pour 2 modes de fonctionnement possibles :

- Le mode séquentiel qui est celui mis en œuvre dans les simulateurs et pour lequel l'environnement et la machine s'exécutent en alternance.

- Le mode parallèle qui est mis en œuvre pour des implantations effectives.

Les 2 modes de fonctionnement ont été modélisés par des réseaux de Pétri. Pour le mode parallèle, cette modélisation a permis de mettre en évidence la nécessité d'une *double bufferisation* des signaux d'entrée de la machine. Les 2 modes de fonctionnement ont ensuite été décrits en ESTEREL en tant que moyen d'expression formel de comportements réactifs.

4.4.1.2 Machine à évaluation par nécessité

La technique originale d'évaluation par nécessité a pour objectif de minimiser les durées des réactions d'un processus synchrone quand celui-ci est codé sous la forme d'un automate. La technique classique d'exécution d'automates peut être qualifiée de *control driven* car tous les signaux d'entrée sont *systématiquement* transférés de l'environnement vers l'automate en début de transition puis remis à absent en fin de transition. L'évaluation par nécessité est quant à elle *demand driven* : cette technique est souvent utilisée dans les langages fonctionnels. Le principe est que l'argument d'une fonction n'est évalué qu'au moment où il y a accès à cet argument. Pour les automates, le passage d'un état à un autre nécessite rarement l'accès à tous les signaux d'entrée. L'évaluation par nécessité permet donc de gagner en temps de transition. Ce gain est d'autant plus important que nombreux sont les signaux d'entrée de l'application.

Pour mettre en œuvre cette technique, un algorithme d'**estampillage des signaux d'entrée** a été mis au point. Cet algorithme est nécessaire pour garantir la cohérence temporelle des signaux vus par l'automate. Le transfert d'un signal d'entrée du buffer externe vers le buffer interne peut avoir lieu

- soit à l'initiative de l'automate quand celui-ci a besoin d'accéder au signal et que l'échantillon contenu dans le buffer interne est périmé,

- soit à l'initiative de l'environnement quand une nouvelle occurrence à lieu alors que le buffer externe est déjà plein : ceci permet de vider le buffer externe.

La machine d'exécution par nécessité a été formalisée en ESTEREL. Le comportement a ensuite fait l'objet de vérifications formelles.

Un prototype a été implémenté sur PC. Dans un soucis de réutilisabilité des composants et de génération automatique, une approche par objets a été adoptée.

Les objets de la machine sont :

- Des gestionnaires d'événements, dont le rôle est de collecter les signaux d'entrée. Le mode d'acquisition mis en œuvre dans ce prototype est le mode par interruption. Il s'agit donc de gestionnaires d'interruption ("*handlers*" en anglais).
- L'automate, dont le comportement réactif est issu de la compilation d'un programme ESTEREL.
- Les signaux d'entrée : ce sont des objets qui encapsulent à la fois des informations concernant le signal logique vu par l'automate : présence et valeur, mais aussi des informations concernant le signal physique associé : date d'occurrence et date de dernier transfert.
- L'horloge, qui se charge de cadencer les évolutions de l'automate et d'estampiller les signaux.

Bilan : En ce qui concerne la technique d'évaluation par nécessité nous ne l'avons pas retenue pour 2 raisons :

- À l'époque de cette étude le codage sous forme d'automates était la seule possibilité offerte par le compilateur ESTEREL. Ce type de codage est très coûteux en taille mémoire et devient vite impossible dès que les systèmes atteignent une certaine taille. À l'heure actuelle la version 5 du compilateur ESTEREL peut produire des processus synchrones sous forme équationnelle ce qui résout le problème de taille. En revanche, pour la forme équationnelle, toutes les entrées doivent être réévaluées à chaque réaction. La technique d'évaluation par nécessité perd donc de son intérêt pour ce type de processus synchrones.
- La technique est intéressante pour des applications de type contrôleur embarqué où les contraintes de temps sont très fortes. Les applications auxquelles nous nous intéressons sont plutôt du type *soft real-time* et sont moins exigentes en termes de performances.

Nous retenons en revanche le principe de la *double bufferisation* des signaux d'entrée qui permet de prendre en compte le parallélisme entre machine et environnement.
Pour ce qui est de l'implémentation, le choix de l'approche à objets nous paraît très intéressant. Nous adopterons la même approche dans nos réalisations.

4.4.2 Implantation sur RTX2000

M.A. Péraldi s'est également penchée sur le problème de l'implantation de programmes ESTEREL sur microcontrôleurs. Ces travaux [AP96] concernent plus particulièrement les applications de type **contrôleur embarqué**. Les contraintes de ces applications sont multiples :

- des contraintes matérielles (taille mémoire réduite, ...)
- des contraintes temporelles fortes et **strictes**,
- des contraintes de sûreté de fonctionnement

Ce type de système est qualifié de *temps réel à contraintes strictes* (*hard real-time system*) car le non respect, même occasionnel, d'une contrainte de temps peut avoir des conséquences dramatiques. Le choix de la cible matérielle est lui aussi critique : l'architecture doit être performante et parfaitement *déterministe*.

L'objectif de ces travaux était de proposer des outils permettant de :

- passer automatiquement d'une spécification de haut niveau d'un système en ESTEREL à son implémentation sur une architecture matérielle donnée.
- évaluer les performances temporelles de cette implémentation.

La cible choisie est un micro-contrôleur RTX2000. Son architecture de type RISC, sans mémoire cache ni pipeline présente à la fois performance et déterminisme. La prise en compte rapide des interruptions en font un bon support pour les applications réactives temps réel. Deux outils ont été développés :

- Un outil de traduction d'automates du format OC vers le langage de programmation du micro-contrôleur : FORTH. Cette traduction est faite selon 2 critères : l'optimisation des temps d'exécution ou l'optimisation de la mémoire.
  Le code de l'automate est généré automatiquement pour chaque application. Une partie fixe de code constitue la machine d'exécution : elle contient les structures de données et les fonctions qui permettent de gérer l'automate.
- Un outil d'évaluation de performance permet de donner les bornes des temps de réaction de l'automate pour chaque transition. Cet outil se base sur une formalisation du comportement de l'ensemble des instructions du micro-contrôleur.

Les techniques utilisées pour l'optimisation du code et l'évaluation des performances reposent en grande partie sur les spécificités de l'architecture cible et de son langage de programmation.

Nous retenons de cette expérience l'intérêt d'une génération automatique de machines d'exécution.

4.5 Automate programmable sous VxWORKS

Ces travaux effectués dans l'équipe LIMI de l'Université de Bretagne Occidentale [SBM96] sont à rapprocher de ceux de M.A. Péraldi dans le sens où le domaine d'applications visé est le même : celui des systèmes temps réel fortement contraints. Cette étude vise en priorité les systèmes de contrôle/commande de processus. C'est pourquoi le formalisme de description préconisé est le GRAFCET. L'objectif est de réaliser un *automate programmable* : le système est spécifié en GRAFCET et le code exécutable doit être généré de manière automatique. Cette génération se fait en 3 étapes :

- traduction du GRAFCET vers le langage synchrone SIGNAL,
- compilation du programme SIGNAL et génération d'un processus synchrone codé en C,
- implantation du processus synchrone sur le système cible : une carte 68040 gérée par le noyau temps réel VxWORKS

De Grafcet vers Signal SIGNAL est un langage synchrone déclaratif flot de données utilisé le plus souvent dans des applications de traitement du signal. Le point commun avec le GRAFCET est l'aspect synchrone. La modélisation de l'évolution d'un grafcet en SIGNAL sort du cadre de cette thèse, pour plus de précisions, on pourra se reporter à [SBM96]. Notons simplement le point suivant : On retrouve le problème de l'instantanéité des réactions au niveau du modèle d'évolution du GRAFCET avec recherche de stabilité. En effet, pour effectuer un pas du grafcet modélisé, un certain nombre de réactions du programme SIGNAL doivent être effectuées. Durant ces réactions, la vision de l'environnement ne doit pas évoluer. Des signaux locaux jouent alors le rôle de *buffers* garantissant une vison figée de l'environnement durant chaque pas du grafcet.

De Signal vers C Le compilateur SIGNAL produit un processus synchrone codé en C. Le caractère flot de données de SIGNAL apparaît clairement au niveau du code généré : la procédure d'évolution fait des demandes explicites de lecture des signaux d'entrée (*demand driven*). Pour ESTEREL, au contraire, c'est l'environnement qui positionne les signaux par des appels de fonctions C (caractère *interrupt driven*). La procédure d'évolution diffère suivant l'interprétation qui est faite du grafcet. Pour une interprétation sans recherche de stabilité, le comportement cyclique est le suivant :

lire les entrées
calculer la nouvelle étape du grafcet
émettre les sorties (actions et temporisations)

Pour une interprétation avec recherche de stabilité, la procédure est modifiée de manière à figer l'image de l'environnement durant les réactions synchrones du GRAFCET :

> *lire les entrées*
> *tantque la situation n'est pas stable*
>   *calculer la nouvelle étape du grafcet*
> *fin tantque*
> *émettre les sorties*

De C vers VxWORKS Les possibilités d'implantation vont de la réalisation d'un processeur dédié au GRAFCET à l'exécution du code C sur un noyau temps réel. C'est ce dernier choix qui est envisagé dans un premier temps.
L'environnement d'exécution est constitué d'une carte 68040 exécutant le noyau VxWORKS. Une carte d'entrée/sortie parallèle comprenant des circuits de type PI/T permet d'accéder à des capteurs et des actionneurs *tout ou rien* et de générer des interruptions.
Deux stratégies d'exécution ont été implémentées :

- *Activation périodique sans mémorisation* : dans ce mode de fonctionnement, l'acquisition des entrées, la réaction du GRAFCET et l'émission des sorties ont lieu périodiquement. Cette méthode est satisfaisante si on a la garantie qu'un même signal d'entrée varie au plus une fois par période. Dans le cas contraire, des occurrences de signaux peuvent être perdues.

- *Activation périodique avec mémorisation* : à chaque signal d'entrée est associée une file d'attente. La variation d'un signal d'entrée provoque une interruption et la routine correspondante place le signal dans la file d'attente. La procédure d'évolution est toujours appelée périodiquement, les entrées ne sont plus lues sur les capteurs mais dans les files d'attente. Se pose alors le problème de la construction des événements : la stratégie choisie consiste à traiter *toutes* les occurrences de signaux : s'il y a n occurrences d'un signal donné, elles apparaitront dans n événements distincts. Ainsi, à chaque période on effectue autant de réactions que nécessaire pour vider les buffers.
  Plusieurs remarques peuvent être faites sur ce mode d'exécution :

  - Le mécanisme de mémorisation de l'histoire d'entrée est intéressant pour les applications du type *soft real-time*. Pour ce type d'applications, l'hypothèse synchrone est vérifiée en moyenne mais il peut y avoir violation occasionnelle lors de *rafales* de signaux d'entrée. En revanche, il n'existe pas de stratégie universelle pour la construction des événements. Une stratégie ne peut être choisie qu'en connaissance de l'application.

- Pour la stratégie proposée, le nombre de transitions par période est variable et dépend de l'histoire d'entrée contenue dans les files d'attente. Il faut donc veiller à ce que toutes ces transitions puissent avoir lieu dans une période. La taille maximale des files d'attente détermine la période minimale d'activation. Le paramétrage de l'exécution sera donc un compromis entre temps de réaction du système et résistance aux rafales.

Nous retenons de cette étude l'idée de paramétrage de l'exécution par le choix de différentes stratégies pour l'acquisition et l'activation.

4.6 Travaux de O. Potoniée

Les travaux d'O. Potoniée [Pot96] portent sur la gestion des processus dans les micro-noyaux de systèmes d'exploitation. Un intérêt particulier a été porté au système Chorus car il est particulièrement bien adapté au développement d'applications réparties. Le comportement de ces micro-noyaux est hautement réactif, le langage Esterel se prête bien à leur programmation. C'est ce qui a amené O. Potoniée à modéliser en Esterel la gestion des processus dans le micro-noyau Chorus. Nous nous intéressons ici à la manière dont ce modèle a été implanté, d'abord dans un cadre centralisé puis dans un cadre réparti.

4.6.1 Exécution d'un modèle synchrone de gestion de processus d'un micro-noyau

4.6.1.1 Composition des modules

Le modèle de gestion des processus comprend plusieurs modules Esterel. Le comportement global de l'application résulte de la *composition* de ces différents modules. Plusieurs possibilités de composition sont envisageables :

Composition synchrone : Il s'agit de composer les modules par l'opérateur `||` du langage Esterel. C'est la solution idéale car les synchronisations et communications entre modules sont résolues à la compilation. Les communications bidirectionnelles entre modules sont autorisées pourvu qu'il n'y ait pas de cycles de causalité. Le résultat est un processus synchrone qu'il reste à interfacer à l'environnement. Cette composition présente néanmoins 2 inconvénients :

- La création et la destruction *dynamique* de modules est impossible. Dans le modèle de gestionnaire de processus, un module de contrôle est associé à chaque processus : il doit être créé et détruit en même temps que le processus.

– Pour des systèmes complexes comportant un grand nombre de modules, la composition synchrone peut ne plus être possible si on atteint les limites du compilateur (soit en temps, soit en mémoire).

Séquencement synchrone : Il s'agit de la composition proposée par F. Boulanger et mise en œuvre dans la bibliothèque synchrone (section 3.3). Le séquencement synchrone permet de composer des modules qui n'ont pas pu l'être par composition synchrone. De plus, il autorise la création et la destruction dynamique d'objets synchrones. En revanche, les outils de preuves formelles pour langages synchrones ne peuvent pas être utilisés pour vérifier le comportement *global* d'un tel système.

Parallèle asynchrone : Dans ce mode de composition, chaque module est compilé séparément et encapsulé dans un objet. Les objets évoluent chacun à leur propre rythme et communiquent entre eux de manière asynchrone. Il n'y a pas de base de temps commune et on ne peut pas parler de réaction *globale* du système. Cette composition autorise la reconfiguration dynamique : on peut créer, supprimer des objets sans même avoir à recalculer d'ordonnancement. Les communications bidirectionnelles sont autorisées puisqu'elles sont asynchrones. En revanche, l'absence d'ordonnancement global rend le système *indéterministe*.

4.6.1.2 Machine d'exécution centralisée

Étant donné le nombre de modules du modèle de gestion des tâches et les contraintes de dynamicité et de déterminisme, O. Potoniée a choisi pour son implémentation, le **séquencement synchrone**. Ce choix ne modifie pas la problématique de la machine d'exécution : vue de l'extérieur, l'ensemble des objets synchrones interconnectés est un processus synchrone qui doit être interfacé à l'environnement.

L'objectif de la machine est d'assurer la communication et la synchronisation entre le processus synchrone et les *threads* d'une part et entre le processus synchrone et les sources d'interruptions matérielles d'autre part, tout en préservant la sémantique du synchronisme, à savoir l'atomicité des réactions.
Le critère principal retenu pour sa conception est la **simplicité** : en effet, le principe est que tout comportement complexe doit être décrit dans un formalisme synchrone prouvable.

La machine proposée a un comportement purement *séquentiel*: les appels au noyau effectués par les threads sont des appels de fonction. C'est au cours de ces appels que le gestionnaire de threads réagit. L'interaction entre la machine et les threads se fait donc de manière synchrone. Les seuls événements qui peuvent se produire au cours d'une réaction de la machine sont les interruptions matérielles : ce sont les seules sources d'asynchronisme.

La machine est structurée en 5 modules activés séquentiellement :

- Collecteur d'entrée. Ce module est chargé de *détecter les modifications de l'environnement.* Il est composé d'un ensemble de modules d'entrée associés chacun à un signal d'entrée du modèle de gestion des tâches. Lorsqu'il y a occurrence d'un signal, le collecteur en informe le module *constructeur d'événements.*

- Constructeur d'événements. Son fonctionnement est basé sur la notion de signal *informatif* ou *stimulant* : un signal informatif est un signal qui modifie l'état du processus synchrone sans provoquer l'émission de signaux de sortie alors qu'un signal stimulant provoque l'émission instantanée de signaux de sortie. Le principe retenu est que l'occurrence d'un signal stimulant doit déclencher une réaction. L'occurrence d'un signal informatif, au contraire, est mémorisée mais ne provoque pas de réaction. Elle sera prise en compte à la prochaine occurrence d'un signal stimulant. Ce mécanisme permet d'éliminer des réactions inutiles du système. Une question qui vient alors à l'esprit est celle du lien existant entre cette stratégie et la stratégie classique. La classe des systèmes pour lesquels toute séquence du type :

  $$informatif_1 : \ \ldots \ : informatif_n : stimulant$$

  est équivalente à l'événement :

  $$(informatif_1 \ and \ \ldots \ and \ informatif_n \ and \ stimulant).$$

  n'a pas été précisée.

- Exécution de la réaction. Pour préserver l'atomicité des réactions, l'occurrence d'une interruption matérielle (seul événement asynchrone du système) pendant une réaction est mémorisée dans une file d'attente et prise en compte à la fin de la réaction.

- Collecteur de sortie. Il a un rôle intermédiaire : il récolte les signaux de sortie émis par le processus synchrone suite à une réaction et forme l'événement de sortie.

- Émetteur de sortie : il transforme l'événement synchrone logique en événements externes. C'est le symétrique du collecteur d'événements d'entrée.

Ce modèle de gestion des threads a été implanté au niveau utilisateur au dessus d'un thread CHORUS. Ce thread CHORUS peut être vu comme un processeur virtuel qui exécute le micro-noyau synchrone.

O. Potoniée a été amené à connecter les signaux ESTEREL à ceux du monde extérieur par différents moyens (messages, mémoire partagée...) et selon différentes stratégies (signaux informatifs ou stimulants).

Ces possibilités de connection entre signaux physiques et signaux logiques rejoignent un des objectifs que nous nous sommes fixé. Il s'agit de faciliter cette tâche à l'utilisateur :

- en proposant des modules permettant différents types de connexion,
- en laissant à l'utilisateur la possibilité de définir des modules de connexion répondant à des besoins particuliers.

De plus, les particularités du gestionnaire de threads font que seuls certains signaux (les interruptions matérielles) présentent un caractère asynchrone. Dans le cas général, une machine devrait systématiquement prendre en compte cet asynchronisme avec l'environnement.

4.6.2 Exécution synchrone répartie

Le problème posé ici est celui de la *répartition d'applications synchrones*. Ce sujet ne relève pas directement de la problématique des machines d'exécution, nous n'en parlons que très brièvement.
L'objectif était de synchroniser un ensemble de sites interconnectés par un réseau de communication et exécutant chacun le noyau de gestion synchrone des tâches. O. Potoniée a proposé pour cela une solution basée sur une forme de synchronisme faible [Mil83].
Dans le domaine de la distribution d'applications synchrones, les approches flot de données ont déjà donné des résultats significatifs. Pour l'approche impérative, le problème paraît bien plus complexe.

4.7 Conclusion

Nous avons vu dans ce chapitre des réalisations souvent très différentes les unes des autres mais qui portent toutes le qualificatif de machine d'exécution.

Les solutions proposées sont très variées et sont le plus souvent conditionnées par des besoins spécifiques :

- Soit en termes d'applications visées comme c'est le cas pour l'exécution du gestionnaire de threads d'O. Potoniée,
- Soit en termes de mode de fonctionnement comme c'est le cas pour les simulateurs.

Dans d'autres machines, la mise en œuvre repose sur les spécificités d'une cible d'exécution particulière comme c'est le cas pour l'implantation sur RTX2000. Dans la plupart des cas, le problème de l'exécution sur une même cible, du processus synchrone et de tâches asynchrones n'est pas considéré.

Il ressort de cette analyse la nécessité de définir une **architecture générique de machines d'exécution** qui permette d'avoir une approche unifiée pour l'implantation des processus synchrones. Cette architecture doit être :

- suffisament *souple* pour s'adapter à des besoins spécifiques en termes de connexion à l'environnement, de stratégie d'activation et de mode de fonctionnement.
- suffisament *abstraite* pour s'adapter à différentes cibles d'implémentation.

Nous tenons aussi à ce que le problème des tâches asynchrones soit pris en compte. Une telle structuration de l'implémentation est indispensable pour la mise au point d'outils de génération automatique.

Chapitre 5

Machine d'exécution

Objectifs

- Présenter sur un exemple simple les problèmes posés par la mise en œuvre.
- Proposer une architecture générique de machine d'exécution.
- Introduire un modèle permettant d'avoir une vision unifiée des différentes stratégies d'exécution.
- Comparer le modèle proposé à celui des objets synchrones.
- Mettre en évidence les problèmes liés à l'exécution des tâches asynchrones.
- Intégrer les tâches dans l'architecture.

Contributions

- Architecture générique de machine d'exécution.
- Notions de contrôleur d'E/S, de séquenceur, d'observateur.
- Modèle des processus.
- Modèle des boîtes réactives.
- Construction de machines par composition de boîtes réactives.
- Gestion des modes de fonctionnement d'une application.
- Intégration des tâches dans l'architecture.
- Différentes sémantiques pour l'exécution des tâches.

5.1 Introduction

Comme nous avons pu le constater dans le chapitre précédent, il y a de nombreuses manières de réaliser une machine d'exécution. Pour passer d'un processus synchrone à une implantation effective, plusieurs facteurs interviennent :

1. Les **caractéristiques de l'application** : il s'agit des caractéristiques du processus synchrone mais aussi de celles de son environnement :

   – La sorte du processus synchrone : le nombre et le type de ses entrées/sorties et éventuellement de ses tâches.

   – Les "entrées/sorties" de l'environnement : les moyens d'agir sur lui (actionneurs) et les moyens de connaître son état (capteurs).

2. Des **paramètres d'exécution** : suivant le type d'application auquel on a affaire, les mécanismes conduisant à une exécution cohérente peuvent varier. Le choix conjoint d'un certain nombre de paramètres devra assurer une approximation satisfaisante de l'hypothèse synchrone. Ces paramètres sont :

   – La stratégie d'activation du processus : activation périodique, activation sur événements déclencheurs, etc. ...

   – La stratégie de construction d'une image de l'environnement pour le processus.

   – Les mécanismes de communication avec l'environnement : scrutation, interruption, avec/sans mémorisation, etc. ...

3. La **stratégie de traitement des anomalies** : pour réaliser des systèmes sûrs, les anomalies de fonctionnement (non respect des contraintes de temps, perte d'informations en entrée, etc...) doivent être détectées et traitées. La manière de traiter ces anomalies dépend fortement de l'application.

4. Des **choix d'implémentation** : il s'agit du niveau d'intégration de la machine. Celui-ci peut aller du développement d'un processeur dédié à l'implantation dans un langage de haut-niveau.

Quelle que soit la cible d'implantation, l'exécution de processus synchrones pose le problème fondamental de l'interaction entre monde synchrone et monde asynchrone. Le modèle synchrone est basé sur une représentation discrète du temps sous la forme d'une suite d'instants élémentaires de durée nulle. Dans la réalité, les réactions ne sont pas instantanées : elles sont composées d'une succession d'opérations et durent un certain temps (figure 5.1).

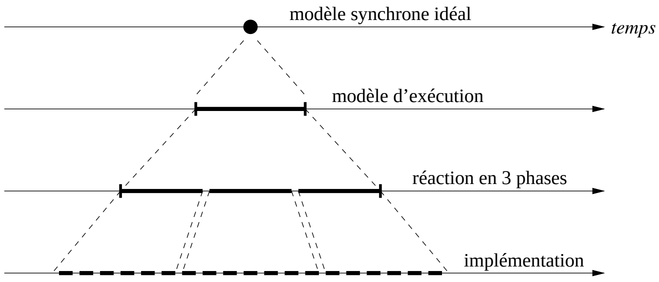


FIG. 5.1 - *Du modèle synchrone vers son implémentation*

L'exécution **atomique** de ces réactions est alors un préalable nécessaire à toute approximation de l'hypothèse synchrone. Le principe de mise en œuvre de l'atomicité est simple ; une réaction se fait en 3 phases :

- Formation d'une image de l'environnement à un instant donné,

- Réaction du processus synchrone à partir de cette image,

- Émission des signaux de sortie vers l'environnement.

Lors de l'implémentation, ces 3 phases ne peuvent plus être considérées comme élémentaires et doivent être détaillées.

Nous proposons un **modèle de boîtes réactives** basé sur le principe de raffinements successifs. Les actions considérées comme élémentaires à un niveau donné sont détaillées au niveau inférieur jusqu'à atteindre le niveau de finesse désiré. Ce modèle a pour but de **structurer l'implémentation** mais il permet aussi de prendre en compte des aspects imposés par la réalité (contraintes d'implémentation). Grossièrement, une boîte réactive est une entité qui a des entrées et des sorties, que l'on peut commander par des signaux de contrôle et qui peut émettre des signaux d'exception. Les boîtes réactives ont une interface et une structure générique commune mais elles possèdent chacunes des spécificités propres. Ce modèle permet de décrire les mécanismes de base de l'interaction synchrone/asynchrone et permet donc d'avoir une vision unifiée des différentes stratégies d'exécution. La mise en œuvre de stratégies particulières d'exécution et de traitement des anomalies se fait par composition de boîtes réactives. L'architecture est modulaire : elle peut s'adapter à une application donnée par ajoût/suppression de *contrôleurs d'entrée/sortie*. Une autre caractéristique est la généricité : aucune référence n'est faite à la cible d'implémentation. Au contraire, l'élaboration de ce modèle permet de définir un support d'exécution *minimal* pour machines d'exécution. Une caractéristique importante est que le modèle intègre la notion de traitements asynchrones (tâches ESTEREL).

Dans un premier temps nous verrons sur un exemple concret les problèmes posés par l'exécution de processus synchrones. Cet exemple nous mènera à une première architecture de machine. Nous introduirons ensuite la notion de boîte réactive pour répondre au problème dans un cadre plus général. Nous nous intéresserons enfin à l'intégration des traitements asynchrones dans cette architecture.

5.2 Machine d'exécution

5.2.1 Exemple d'application réactive

Nous proposons ici un exemple que nous utiliserons tout au long de cette section et qui servira de support pour la présentation du modèle de boîtes réactives. Il s'agit d'une application réactive simplifiée mais dont l'exécution met déjà en évidence un certain nombre de problèmes. Le système est constitué d'un moteur et d'un contrôleur (figure 5.2), l'objectif étant d'asservir le moteur en vitesse[1].

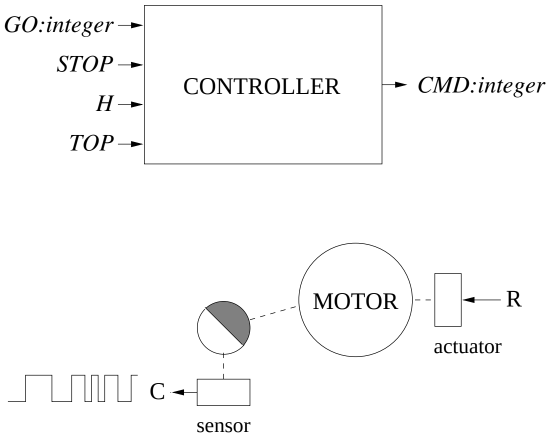


FIG. 5.2 - *Commande simplifiée d'un moteur*

Le dispositif physique La vitesse de rotation du moteur est contrôlée par l'intermédiaire d'un registre de contrôle `R`. La commande comporte 5 niveaux :

- `R=000` ⇒ arrêt du moteur.
- `R=001` ⇒ augmentation du régime.
- `R=010` ⇒ baisse du régime.
- `R=011` ⇒ régime inchangé.

1. Il ne s'agit pas d'un asservissement classique continu mais d'un asservissement destiné à un moteur commandé par paliers grossiers. Le but de cet exemple est avant tout d'illustrer nos propos.

- `R=100` ⇒ vitesse V_m (constante quelconque).

Le moteur entraîne la rotation d'un disque composé de 2 demi-disques de natures différentes. Un capteur *tout ou rien* en contact permanent avec ce disque délivre un signal `C` qui oscille entre "0" et "1" avec une fréquence proportionnelle à la vitesse de rotation du moteur.

Le contrôle Il s'agit d'un asservissement de vitesse : le contrôleur doit émettre la commande adéquate vers le moteur pour obtenir la vitesse de rotation désirée. L'opérateur peut émettre deux consignes :

- Une demande d'arrêt du moteur,
- Une consigne portant la vitesse de rotation désirée.

Ces 2 consignes sont positionnées grâce à une interface graphique.

Pour connaître la vitesse de rotation et calculer la commande, le concepteur du contrôle dispose de 2 informations :

- Le signal `C` émis par le capteur tout où rien,
- Une horloge délivrant un *top* toutes les unités de temps.

La conception du contrôleur dans un langage synchrone (dans notre cas, ESTEREL) est une activité de haut niveau. Dans cette approche, on doit distinguer deux mondes :

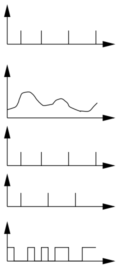
- Le contrôleur qui constitue le système synchrone,
- Le reste du monde qui constitue l'environnement asynchrone.

Pour concevoir le système synchrone, le concepteur se base sur une représentation **abstraite** de l'environnement par l'intermédiaire de **signaux d'interface**. La première étape consiste donc à définir ces signaux d'entrée/sortie.
Pour le contrôleur, la commande du moteur sera modélisée par un signal de sortie valué : `CMD(integer)`. Les événements d'entrée seront représentés par les signaux suivants :

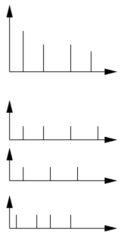
- `GO(integer)` : vitesse de rotation désirée en nombre de tours par unité de temps.
- `STOP` : demande d'arrêt du moteur.
- `H` : horloge (unité de temps).
- `TOP` : signal émis à chaque tour effectué par le disque.

Bien que le signal `C` émis par le capteur soit un signal à 2 niveaux, il est plus commode pour la spécification du contrôleur de représenter la rotation du disque par un signal pur, `TOP`, émis à chaque tour du disque. Le choix des signaux d'interface est l'occasion de faire abstraction de certains détails dont la prise en compte à ce niveau alourdirait inutilement le travail de conception. La figure 5.3 représente les signaux d'entrée/sortie du système synchrone ainsi que les grandeurs et/ou événements physiques correspondant.

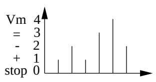
ENTREES



événement ou grandeur physique				*abstraction logique*		
source	*événement*	*domaine de valeurs*	*signification*	*signal Esterel*	*domaine de valeurs*	*signification*
souris	click		validation consigne de vitesse	GO	integer	consigne de vitesse
panneau graphique		chaine de caracteres	consigne de vitesse			
souris	click		consigne d'arret	STOP		demande d'arret
horloge	top d'horloge		unité de temps	H		unité de temps
registre C du moteur		booléen	position du moteur	TOP		1 tour du moteur



SORTIES



abstraction logique			*événement ou grandeur physique*			
signal Esterel	*domaine de valeurs*	*signification*	*destination*	*action*	*domaine de valeurs*	*signification*
CMD	integer	commande du moteur	registre R du moteur		champ de bits	commande du moteur

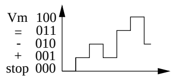


FIG. 5.3 - *Correspondance signaux physiques/signaux logiques*

5.2.2 Exécution *logique* ou simulation

Le but poursuivi par le concepteur est de contraindre l'environnement, ici le moteur, de manière à obtenir le comportement désiré. La description du comportement du contrôleur dans un langage synchrone revient à donner une spécification des signaux de sortie comme une fonction des signaux d'entrée et d'un état interne. L'hypothèse synchrone fait abstraction des aspects temporels en considérant que cette contrainte entre entrées et sorties est vérifiée en permanence. Ceci est évidemment impossible à réaliser.

Dans la réalité, le résultat de la compilation d'un programme ESTEREL est un **processus synchrone** caractérisé par des entrées, des sorties et un comportement réactif. L'interaction entrées/sorties ne peut pas être *permanente* : elle ne peut se faire que par activations ponctuelles du comportement réactif. Ces activations, qui correspondent aux *instants* du modèle synchrone consistent à réévaluer les signaux de sortie et l'état interne du processus.

Une première implémentation du processus nécessite un moyen de *contrôler* ces activations. Si les instants d'activation sont déterminés par l'environnement, leur durées, elles, sont imposées par le processus. Il nous faut donc un moyen de déclencher une réaction et un moyen de savoir quand celle-ci a terminé. On peut représenter la machine d'exécution minimale par un processus synchrone doté de 2 signaux de contrôle (voir figure 5.4) :

- *BoI* (*Beginning Of Instant*) : signal émis *vers* le processus pour déclencher une réaction.

- *EoI* (*End Of Instant*) : signal émis *par* le processus pour indiquer la fin d'une réaction.

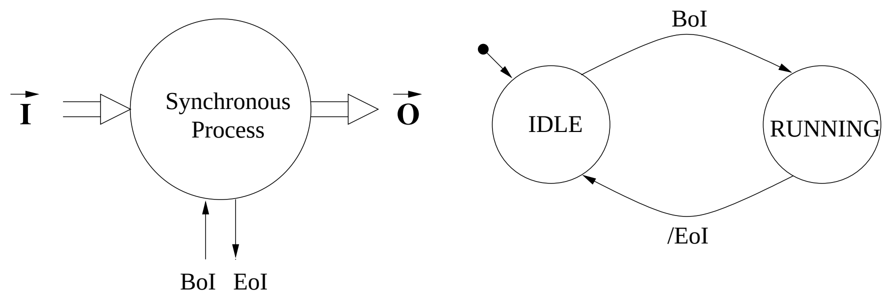


FIG. 5.4 - *Processus synchrone*

Les vecteurs $\vec{I}$ et $\vec{O}$ représentent l'ensemble des signaux d'entrée et de sortie à un instant donné. On les appelle aussi événements d'entrée et de sortie.
La machine possède 2 états : elle est soit au repos (`IDLE`), soit en cours de réaction (`RUNNING`) (figure 5.4).

Si l'on considère un mode d'exécution dans lequel processus synchrone et environnement évoluent en alternance, cette structure de machine rudimentaire est suffisante pour vérifier le comportement *logique* du processus. C'est ainsi que fonctionnent la plupart des simulateurs de langages synchrones comme XES. Les fonctions essentielles de ces simulateurs sont le positionnement des entrées ($\vec{I}$), l'activation du processus (*BoI* et *EoI*) et la visualisation des sorties ($\vec{O}$). La plupart permettent aussi de remettre le processus dans son état initial. On peut modéliser cette fonction par un signal de contrôle supplémentaire : *reset*.

Ce type d'exécution permet de mettre au point le comportement logique du contrôleur mais ne constitue pas une réelle confrontation du processus à l'environnement asynchrone :

- L'hypothèse d'exclusion entre les évolutions de l'environnement et de la machine doit être levée : il faut prendre en compte l'asynchronisme qui existe entre l'environnement et le processus synchrone.

- Il faut établir un lien réel entre les signaux d'E/S du processus et les événements asynchrones correspondants.

5.2.3 Exécution dans un environnement asynchrone

Prise en compte du parallélisme machine/environnement Le modèle d'exécution que nous venons de proposer n'est pas satisfaisant car il fait abstraction des événements qui peuvent se produire pendant une réaction. Dans la réalité, des événements peuvent survenir à tout moment et on doit trouver un moyen de les prendre en compte.
Une première possibilité est d'interrompre la réaction en cours et d'en démarrer une nouvelle avec l'événement en question. En cas d'interruption de réaction, suivant l'état d'avancement de celle-ci, les signaux de sortie peuvent soit ne pas avoir été émis, soit avoir été émis partiellement. Le risque de mise à jour partielle des signaux de sortie est tout à fait inacceptable car on ne respecte plus le principe d'**atomicité des réactions** : toute réaction entamée doit être soit achevée, soit complètement annulée. L'annulation d'une réaction pose non seulement des problèmes de mise en œuvre, mais aussi des problèmes sémantiques dûs au fait qu'on ne peut pas "remonter le temps". Nous rejetons donc dans un premier temps la possibilité d'interrompre les réactions.
Pour les raisons que nous venons de voir, la solution classique est basée sur le principe que toute réaction entamée doit être achevée. Les événements qui se produisent pendant une réaction du processus sont mémorisés et pris en compte à la réaction suivante. La figure 5.5 représente une exécution basée sur ce principe.

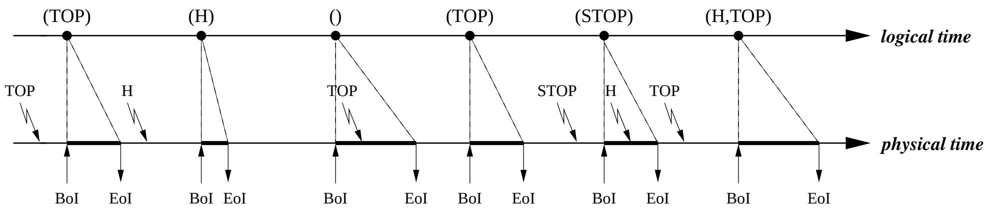


FIG. 5.5 - *Exécution selon le principe d'atomicité des réactions*

Les réactions des processus synchrones n'ont pas de structures bien définies permettant de localiser les points d'interaction entre monde synchrone et monde asynchrone : au cours d'une réaction, on ne sait pas avec précision quand les entrées sont évaluées ni quand les sorties sont émises. On pourait localiser ces points d'interaction pour une cible donnée en analysant le code exécutable. Toutefois ce travail d'analyse est très délicat sur des exécutifs temps réel en raison des possibilités de préemption par le système (interruptions, gestion de l'horloge,...). Pour assurer l'atomicité des réactions, il faut isoler les points d'interaction avec l'environnement de manière à les maîtriser. La notion de contrôleur d'E/S et de module d'E/S que nous allons voir répond à cette exigence.

Connexion à l'environnement L'exécution d'un processus synchrone passe par la *connexion* des signaux d'E/S logiques aux événements du monde extérieur.

Pour notre contrôleur par exemple, on doit mettre en relation le signal d'entrée `GO(integer)` avec sa *source* dans l'environnement. Cette source peut aussi bien être logicielle (interface graphique) que matérielle (dispositif physique du type potentiomètre+bouton poussoir). De même, en sortie, le signal `CMD(integer)` doit être connecté au registre de commande du moteur.
Pour une application donnée, les sources d'événements asynchrones à prendre en compte sont multiples et variées. Pour cette raison, nous avons choisi d'associer à chaque signal d'interface du processus synchrone un **contrôleur d'entrée/sortie** individuel. D'une manière générale, un contrôleur d'entrée/sortie est chargé de faire le couplage entre une grandeur et/ou un événement (physiques ou logiques) de l'environnement et son abstraction logique : le signal, c'est à dire un couple *(présence,valeur)*. Ces contrôleurs constituent l'**interface entre le monde synchrone et le monde asynchrone**.

Contrôleurs d'entrée : On peut représenter un contrôleur d'entrée par le schéma de la figure 5.6.

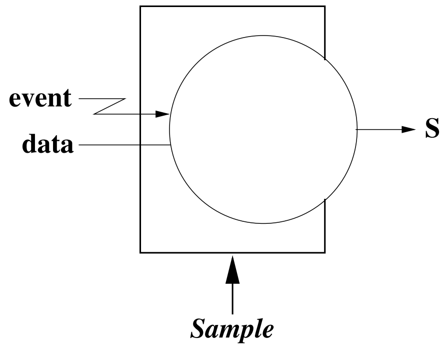


FIG. 5.6 - *Contrôleur d'entrée*

Les contrôleurs d'entrée sont les organes de prise en compte du parallélisme processus/environnement, ils possèdent une double interface :

– Interface avec l'environnement : ils doivent "surveiller" l'environnement, détecter les occurrences d'événements et les mémoriser jusqu'à leur prise en compte.

– Interface avec le processus synchrone : c'est le signal de sortie `S`.

Les contrôleurs d'entrée sont activés par la commande d'échantillonnage `Sample` qui provoque l'élaboration d'un nouvel échantillon et la mise à jour du signal synchrone `S`.
La sémantique de cette activation, c'est à dire la stratégie de construction d'un nouvel échantillon, peut être plus ou moins sophistiquée. Revenons à notre

exemple :

- L'utilisateur émet la consigne `GO(integer)` par l'intermédiaire d'une interface graphique. La valeur de la consigne est saisie sous la forme d'une chaîne de caractères dans une zone de texte et la validation se fait par un *"click"* sur un bouton graphique (voir figure 5.7). L'activation du contrôleur associé devra produire un échantillon présent si et seulement si l'utilisateur a cliqué sur le bouton depuis la précédente activation du contrôleur. La valeur portée par l'échantillon résultera d'une conversion de la chaîne de caractères vers le type entier.

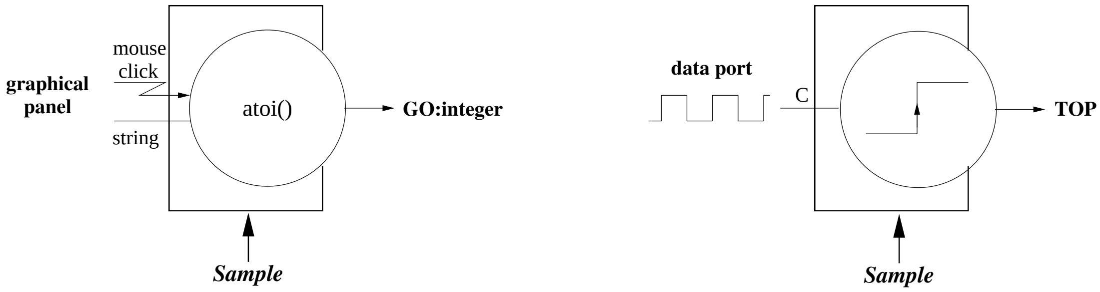


FIG. 5.7 - *Exemples de contrôleurs d'entrée*

- Le signal `STOP` est traité de la même manière : le contrôleur associé fait correspondre à l'événement *"click"* de l'environnement une occurrence (un échantillon présent) du signal `STOP`. Ce signal étant pur, aucun traitement de type conversion n'est nécessaire.

- Le signal `H` qui représente l'unité de temps du contrôleur sera connecté à une horloge logicielle (service d'un système temps réel) ou matérielle (composant timer). Le contrôleur associé devra produire un échantillon présent si un top d'horloge a eu lieu depuis la précédente activation du contrôleur.

- Nous avons choisi à la conception de nous baser sur le signal pur `TOP` pour représenter la rotation du disque. Or nous avons vu que le signal `C` émis par le moteur est un booléen. Si ce signal correspond à un bit d'un registre de sortie du moteur, le contrôleur d'entrée du signal `TOP` devra surveiller l'état de ce registre et détecter ses fronts montants (ou descendants). L'activation du contrôleur devra fournir un échantillon présent s'il y a eu un front montant de `C` depuis la dernière activation (voir figure 5.7).

Contrôleurs de sortie : Comme pour les signaux d'entrée, on associe à chaque signal de sortie un contrôleur individuel qui le relie à une grandeur physique. La structure des contrôleurs de sortie est symétrique de celle des contrôleurs d'entrée. Sur activation, ils transforment le signal logique qu'ils ont en entrée en une grandeur externe. Leur comportement est plus ou moins sophistiqué. Pour le signal $CMD(integer)$, l'activation du contrôleur avec un échantillon présent devra mettre à jour le registre de contrôle du moteur avec la nouvelle valeur.

Modules d'E/S : Nous disposons maintenant de contrôleurs permettant d'établir le lien entre les signaux d'interface du processus synchrone et l'environnement. En particulier, les contrôleurs d'entrée permettent, dans une certaine mesure, de prendre en compte le parallélisme environnement/processus. Il s'agit maintenant d'assembler ces composants pour exécuter le processus.

Les contrôleurs d'entrée servent à former une *image de l'environnement* pour le processus à un instant donné. Idéalement, on devrait pouvoir activer simultanément l'ensemble des contrôleurs d'entrée de manière à obtenir un effet *photographie instantanée*. Les contrôleurs d'entrée doivent donc être couplés pour former l'événement d'entrée $\vec{I}$ du processus. Nous les encapsulons dans un module d'entrée appelé **inputModule** (figure 5.8).

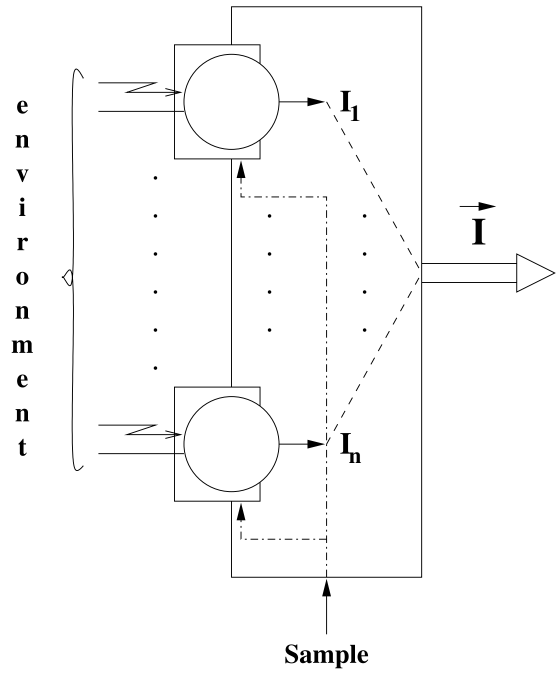


FIG. 5.8 - *Module d'entrée*

L'activation du module d'entrée se traduit par l'activation simultanée de tous les contrôleurs et la formation d'un événement d'entrée $\vec{I}$ à partir des signaux de sortie des contrôleurs $(I_1, ..., I_n)$. Bien évidement, en réalité, sur un système monoprocesseur du moins, l'échantillonnage logiciel ne peut pas être simultané et il y aura toujours un biais sur l'acquisition. On devra vérifier lors de l'implémentation de l'InputModule que ce biais est acceptable pour l'application concernée.

De manière symétrique, les événements à produire dans l'environnement suite à une réaction du processus sont pris en charge par un module de sortie : **outputModule**. Ce module transforme l'événement de sortie $\vec{O}$ du processus en actions sur l'environnement. Son activation provoque la réaction considérée comme simultanée de tous les contrôleurs de sortie.

Réaction en 3 phases : Les modules d'entrée et de sortie concentrent les points d'interaction entre monde synchrone et monde asynchrone et permettent de mieux maîtriser ces interactions. L'application de ce principe nous mène à la définition d'une première structure de machine d'exécution représentée figure 5.9.

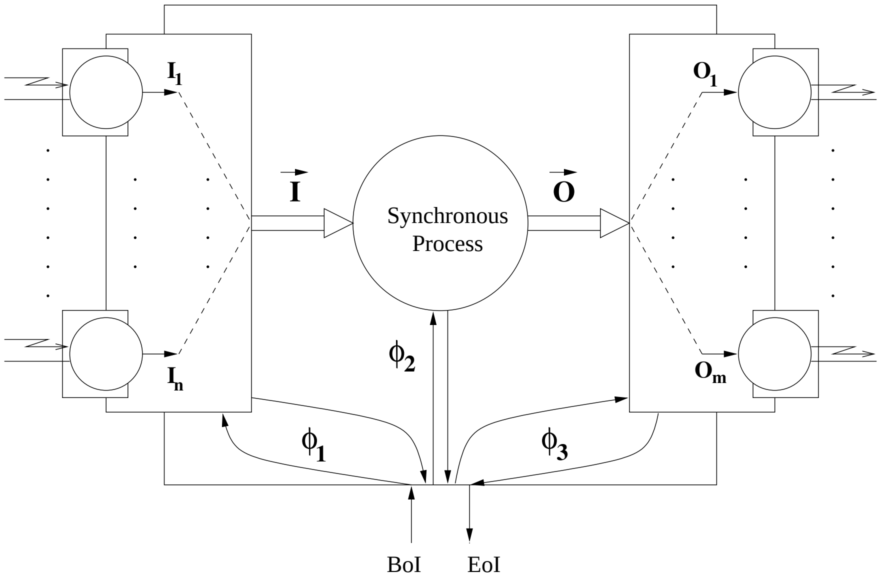


FIG. 5.9 - *Première structure de machine d'exécution*

Une réaction de la machine se déroule en 3 phases :

- ϕ_1 : Activation de l'inputModule : échantillonnage des signaux d'entrée et construction de l'événement $\vec{I}$,
- ϕ_2 : Activation du processus synchrone : élaboration de l'événement de sortie $\vec{O}$,
- ϕ_3 : Activation de l'outputModule : émission des événements vers l'environnement.

Une réaction comprenant ces 3 phases est déclenchée par le signal BoI et se termine par l'émission du signal EoI. On peut représenter le comportement de la machine par le syncChart de la figure 5.10. La figure 5.11 représente un scénario d'exécution.

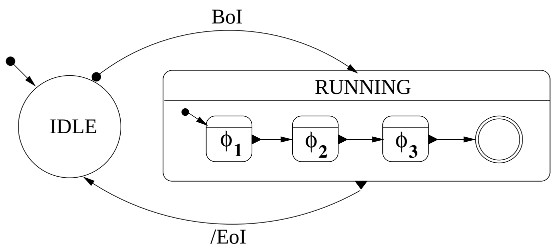


FIG. 5.10 - *Comportement de la machine*

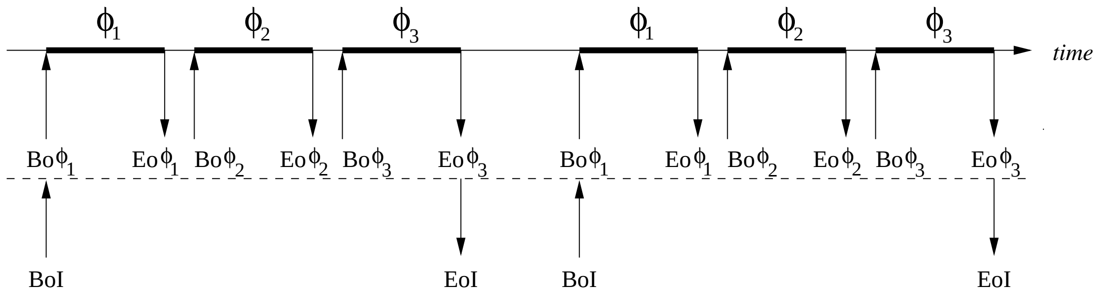


FIG. 5.11 - *Scénario d'exécution*

Application à l'exemple On peut réaliser un couplage entre le contrôleur et son environnement (moteur et opérateur) en appliquant la structure que nous venons de voir. Nous obtenons ainsi une machine d'exécution que l'on peut contrôler par le signal BoI.
Une condition nécessaire pour que le couplage soit fiable (pas de perte d'événements d'entrée) est l'existence d'un temps minimum δ_{min} entre 2 occurrences d'un même événement d'entrée. Cette condition étant vérifiée, il reste à définir une stratégie d'activation de la machine. On distingue 2 stratégies de base :

- Activation sur occurrence d'événements : une réaction est déclenchée dès qu'un événement d'entrée a lieu. Cette stratégie est bien adaptée pour les applications dans lesquelles les événements ont lieu de manière sporadique : la machine est le plus souvent au repos, elle n'est activée que quand cela est nécessaire. Si τ est la durée maximale d'une réaction, le couplage sera opérationnel si $\tau < \delta_{min}$.

- Activation périodique : les réactions sont déclenchées périodiquement, indépendamment des événements d'entrée. Cette stratégie est utilisée dans les applications où les entrées ont un caractère répétitif. Si T est la période d'activation, on doit avoir $T < \delta_{min}/2$ (théorème d'échantillonnage).

Dans notre exemple, certains événements sont sporadiques (les consignes `GO` et `START`), d'autres sont répétitifs (`H` et `C`) et δ_{min} est déterminé par la vitesse maximale du moteur (fréquence maximale du signal `C`). On peut raisonnablement appliquer l'une ou l'autre des stratégies :

- Pour une activation sur événements, les événements déclencheurs sont les "click" de la souris pour la validation des consignes `GO` et `STOP`, les tops d'horloge (`H`) et les fronts du signal `C`. On devra en outre vérifier que $\tau < \delta_{C_{min}}$.
- Pour une activation périodique, il faudra vérifier que $T < \delta_{C_{min}}/2$.

5.2.4 Dysfonctionnements de la machine

Les défaillances susceptibles de survenir au cours de l'exécution d'un processus synchrone peuvent être classées en deux catégories :

- les défaillances *"logiques"*,
- les défaillances propres à la machine d'exécution.

5.2.4.1 Défaillances *"logiques"*

Ce sont des défaillances qui restent dans le cadre du code réactif de l'application. Elles peuvent être prises en compte et traitées directement par le concepteur de l'application. Le cas des violations de relation Esterel est classé dans cette catégorie. Prenons l'exemple de la relation d'exclusion entre signaux d'entrée d'un programme Esterel. Supposons que le concepteur ait fait l'hypothèse que 2 signaux d'entrée sont exclusifs. S'il apparaît au cours de l'exécution que cette hypothèse est fausse, la machine d'exécution construira un événement d'entrée contenant les 2 signaux supposés exclusifs, ce qui mènera à une situation imprévue donc dangereuse. Cette situation peut très bien être *détectée* par la machine d'exécution, en revanche, elle peut difficilement être *traitée* à ce niveau : le traitement à effectuer dépend fortement de l'application. Il serait quasiment impossible, au niveau de la machine, de proposer un traitement efficace adapté à chaque application. Il semble plus naturel de prévoir et traiter ces situations directement au niveau de description de l'application. En somme, plutôt que de faire abstraction de situations exceptionnelles à l'aide de relations Esterel, il est plus judicieux de prévoir ces situations et de les traiter dans le code. Il est bien entendu que les relations Esterel restent d'un grand intérêt notamment pour l'établissement de preuves et la réduction de la taille du code généré.
On voit donc que ce type de défaillance concerne plutôt le domaine de la conception d'applications *robustes* et ne relève pas directement de la problématique des machines d'exécution.

5.2.4.2 Défaillances propres à la machine d'exécution

Ces défaillances ne peuvent être ni traitées, ni même détectées directement au niveau de l'application réactive. Le cas le plus classique de défaillance est la perte d'informations en entrée (*overrun*). Dans l'exemple du contrôleur, la perte d'un échantillon du signal d'entrée `TOP` représentant un tour du moteur (figure 5.12) peut avoir de graves conséquences.

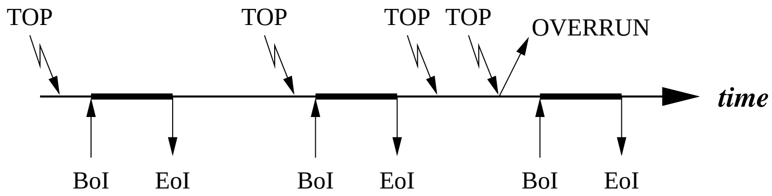


FIG. 5.12 - *Perte d'une occurrence du signal TOP*

Le choix de la période d'activation écarte théoriquement ce risque. Toutefois, si pour une raison quelconque (transition trop longue, périodicité non respectée, etc...) cette situation arrive, il faut la détecter. De même, le non respect de la durée maximum de réaction du processus synchrone doit pouvoir être *détecté*.
Dans certains cas, le dysfonctionnement peut être *traité* par la machine elle-même : certaines applications, notamment dans le domaine du multimédia, font intervenir des signaux dont seule la valeur la plus récente est d'intérêt. Pour ces signaux, en cas de perte d'échantillons, le traitement consistera simplement à écraser l'ancien échantillon au profit du plus récent.
Dans la majorité des cas, toutefois, le traitement ne pourra être effectué par la machine. Celle-ci devra détecter la situation d'exception et se contenter de la signaler pour qu'elle soit prise en compte à un niveau supérieur. C'est le cas dans notre exemple en cas d'*overrun*.

On introduit un module supplémentaire dans la structure de la machine que nous appelons **observateur** (figure 5.13). Son rôle est de *détecter* les dysfonctionnements de la machine, de les *traiter*, ou à défaut, de les *signaler*.

La nouvelle structure de machine fait apparaître un nouveau flux de sortie qui correspond aux signaux d'exceptions qui n'ont pu être traités par l'observateur.
L'observateur a une vue globale sur l'interface et la structure interne de la machine : il "voit" les différents modules (*InputModule, TransitionModule et OutputModule*) ainsi que les signaux qui les contrôlent. Il a aussi connaissance du *temps physique* de manière à armer des temporisations et les surveiller.
La notion d'observateur est un concept abstrait qui permet de modéliser la prise en compte des situations exceptionnelles dans une machine. On ne se préoccupe pas à ce niveau de la manière dont ces observateurs sont implémentés : les implémentations dépendent essentiellement de la situation à observer.

5.2.5 Contrôle et comportement de la machine

Nous avons évoqué plus haut le cas d'une réaction trop longue de la machine. Nous avons introduit des observateurs chargés de détecter de telles situations. Il

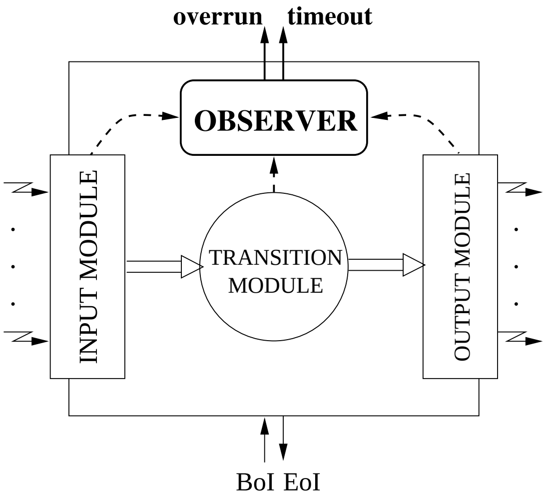


FIG. 5.13 - *Détection des défaillances par un observateur*

semble que pour ce type de défaillance le seul traitement envisageable consiste à **interrompre la réaction en cours**. Comme nous l'avons vu section 5.2.3, l'interruption de réaction pose un problème de cohérence du comportement dû au fait que l'on risque d'interrompre des traitements qui devraient être atomiques, à savoir l'émission des signaux de sortie mais aussi l'acquisition des signaux d'entrée. Une solution consiste à interdire l'interruption des phases ϕ_1 et ϕ_3 de la machine.
Le syncChart de la figure 5.14 représente le comportement d'une machine garantissant l'atomicité des phases d'acquisition et d'émission.

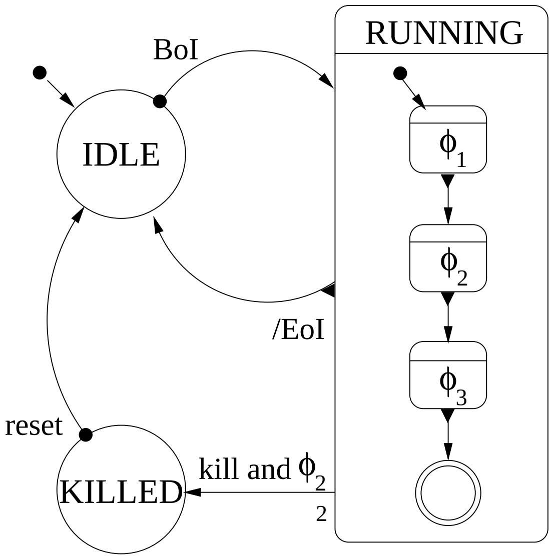


FIG. 5.14 - *Comportement d'une machine avec atomicité des phases ϕ_1 et ϕ_3*

En plus des signaux *BoI*, *EoI* et *reset*, un signal de contrôle supplémentaire *kill* a été introduit pour l'interruption de réaction. Sur le syncChart, la réaction (état `RUNNING`) n'est effectivement interrompue que si l'on est dans la phase de transition du processus synchrone (ϕ_2). Il faut noter que nous utilisons ici les syncCharts comme formalisme de *spécification* de comportements. L'implémentation effective de ce comportement peut se faire par compilation et exécution de ce syncChart mais ce n'est pas la seule façon. On peut notamment avoir recours à des techniques asynchrones pour la mise en œuvre de ce comportement.
Quand une transition (phase ϕ_2 figure 5.14) est interrompue, le processus synchrone se retrouve dans un état indéfini. Cette situation correspond à l'état `KILLED` de la machine. Deux options sont alors envisageables :

- Retour implicite à l'état précédant la réaction avortée : ceci implique une mémorisation de l'état avant chaque réaction, ce qui peut être pénalisant en termes de performances. Il faudrait dans ce cas doter les processus synchrones de signaux *save* et *restore*. Cette solution est fortement déconseillée car si l'on est capable d'annuler un instant logique, on ne peut pas "remonter le temps" : l'environnement et les tâches asynchrones continuent d'évoluer.

- Pas de retour implicite à un état précédent. Dans ce cas, l'émission du signal *reset* remettra le processus dans son état initial. C'est cette option qui est représentée sur le diagramme syncCharts.

On voit sur la figure 5.14 que le comportement d'une machine peut devenir de plus en plus complexe. C'est pourquoi on introduit un nouveau module que nous appelons **séquenceur** (figure 5.15).

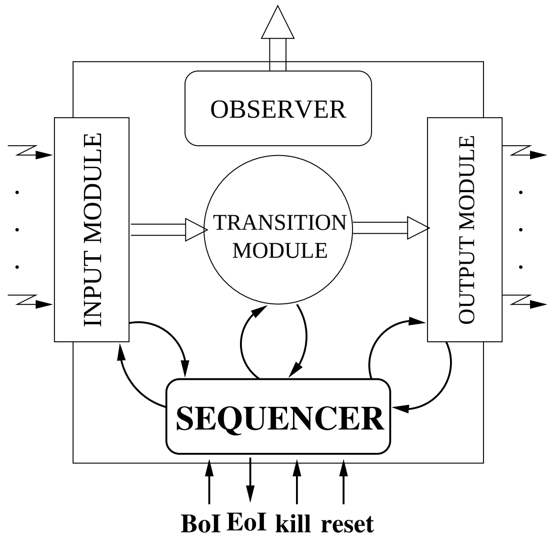


FIG. 5.15 - *Rôle du séquenceur dans la machine*

Le séquenceur est l'entité qui définit la sémantique d'exécution de la machine :

- Il définit le déroulement d'une réaction. La stratégie la plus simple consiste à enchaîner les 3 phases de manière séquentielle.
- Il définit la stratégie de prise en compte d'une interruption de réaction en définissant la sémantique du signal *kill* : interruption inconditionnelle (non souhaitable), interruption *entre* les phases, interruption uniquement en phase de transition etc, ...

5.3 Modèle des boîtes réactives

5.3.1 Introduction

Nous avons vu à l'aide d'un exemple les problèmes que pose la mise en œuvre de processus synchrones. Cet exemple nous a mené progressivement à une architecture de machine d'exécution relativement modulaire. Nous présentons dans cette section deux modèles :

- Le modèle de **processus** qui permet d'avoir une vision unifiée des différents composants d'une machine.
- Le modèle **boîtes réactives** qui reprend les principes de l'architecture définie et les étend afin de répondre au problème de l'exécution dans un cadre plus général.

5.3.2 Processus

5.3.2.1 Définition

Un processus (figure 5.16) est caractérisé par :

- *un vecteur de signaux d'entrée* $\vec{I}$*,*
- *un vecteur de signaux de sortie* $\vec{O}$*,*
- *un vecteur de contrôle* $\vec{\gamma}$*,*
- *un vecteur d'exceptions* $\vec{\epsilon}$*,*
- *et un comportement logique* $\mathcal{M}$ *.*

On peut définir un processus comme une entité qui établit une correspondance entre ses signaux d'entrée $\vec{I}$ et ses signaux de sortie $\vec{O}$. $\mathcal{M}$ (comportement réactif) dénote cette correspondance : il s'agit d'une spécification des signaux de sortie comme une fonction des signaux d'entrée et d'un état interne.

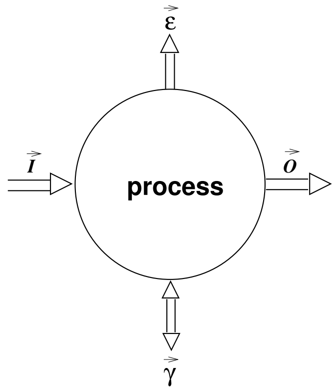


FIG. 5.16 - *processus*

Les réactions d'un processus, c'est à dire l'ensemble des traitements permettant de calculer $\vec{I}$ à partir de $\vec{O}$, ont une **durée non nulle mais bornée**. L'accès à $\vec{I}$ et à $\vec{O}$ par le processus ou l'environnement peut se faire à tout moment durant la réaction.
Le vecteur de contrôle $\vec{\gamma}$ permet de contrôler les réactions et donc de contrôler le comportement temporel du processus.
Le vecteur d'exceptions $\vec{\epsilon}$ permet de signaler d'éventuelles anomalies survenues au cours d'une réaction. Contrairement à $\vec{\gamma}$, sa composition varie selon le processus considéré.
Les processus synchrones sont des cas particuliers de processus pour lesquels, durant une réaction, l'environnement ne doit ni modifier $\vec{I}$ ni solliciter $\vec{O}$.

5.3.2.2 Contrôle

Au cours de son exécution, un processus change d'état interne et transforme ses signaux d'entrée en signaux de sortie, en fonction de son comportement et sous l'action de son vecteur de contrôle. Les composantes du vecteur de contrôle sont

$$\vec{\gamma} =< start, kill, suspend, resume, return, reset >$$

Si ces signaux existent pour tous les processus, ils ne sont pas forcément tous maîtrisés. Le sous-ensemble des ces signaux dont on a la maîtrise exclusive détermine le contrôle que l'on a sur le processus. Nous verrons plus loin que l'existence de signaux non maîtrisés permet de prendre en compte des aspects dus à l'implémentation.
La figure 5.17 représente un scénario d'exécution de processus.

Une réaction est déclenchée par le signal *start* et se termine par le signal *return*[2]. Elle correspond à une activation du comportement $\mathcal{M}$ accompagnée

2. Les signaux *start* et *return* correspondent respectivement aux *BoI* et *EoI* des sections précédentes. Nous n'utilisons plus ces noms car la notion de processus est plus large que celle

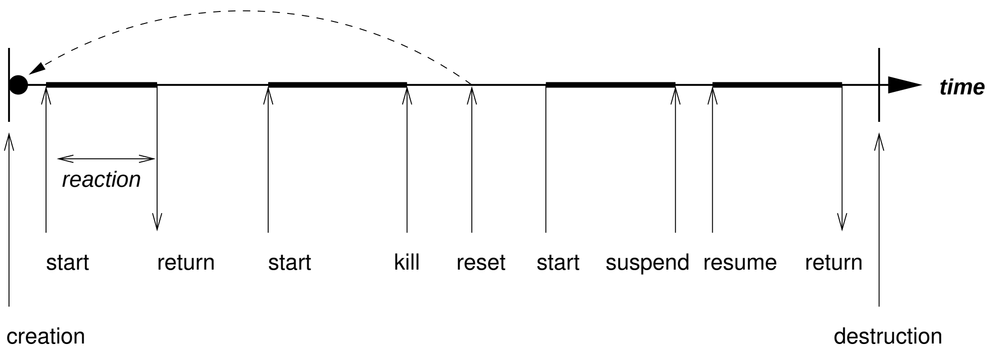


FIG. 5.17 - *Scénario d'exécution d'un processus*

d'une émission des signaux de sortie du processus. Les réactions ne sont pas nécessairement atomiques: elle peuvent être interrompues soit définitivement par le signal *kill*, soit provisoirement par le signal *suspend*. Dans ce dernier cas, le signal *resume* permet de reprendre la réaction. Le signal *reset* est utilisé pour remettre le processus dans son état initial (notamment après un *kill*). L'exécution d'un processus est une séquence du type $(start, return)^*$. La figure 5.18 représente l'état d'un processus en fonction des signaux de contrôle. L'état `RUNNING` correspond aux réactions.

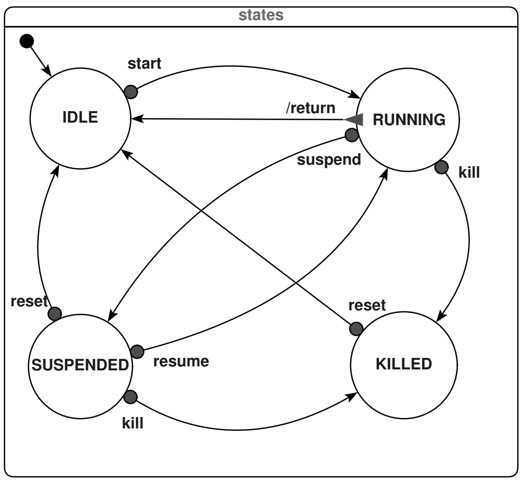


FIG. 5.18 - *États d'un processus*

5.3.2.3 Exemples

La notion de processus est une notion très générale qui permet d'avoir une vision unifiée d'un grand nombre d'objets:

de processus synchrone.

Les processus synchrones entrent évidemment dans ce gabarit :

- $\vec{I}$ est l'événement d'entrée du processus.
- $\vec{O}$ est l'événement de sortie.
- *start* provoque le début de l'évaluation de la fonction de transition.
- *return* est émis à la fin de l'évaluation.
- Le signal *kill* provoque l'interruption de l'évaluation.
- Les signaux *suspend* et *resume* permettent de suspendre et de reprendre l'évaluation.
- *reset* remet le processus dans son état initial.
- Le vecteur d'exception peut contenir des signaux indiquant qu'une erreur a eu lieu pendant la transition (division par zéro, etc...).

Une machine d'exécution peut être vue comme un processus (figure 5.19) :

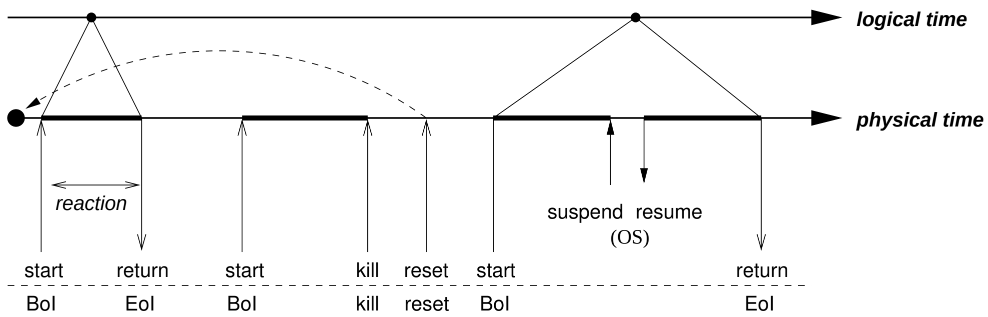


FIG. 5.19 - *Machine d'exécution en tant que processus*

- $\vec{I}$ est constitué des signaux *physiques* d'entrée de la machine,
- $\vec{O}$ contient les signaux *physiques* de sortie.
- *start* déclenche une réaction de la machine qui se termine pas le signal *return*.
- Le signal *kill* permet d'interrompre une réaction de la machine. Comme nous l'avons vu section 5.2.5 la sémantique exacte de ce signal dépend du séquenceur de la machine (voir 5.3.3.4).

- La présence de signaux *suspend* et *resume* peut sembler douteuse pour une machine d'exécution. Pourtant, dans des machines utilisant les services d'un exécutif temps réel, les réactions peuvent être interrompues par les routines de gestion du système (gestion de l'horloge, interruptions matérielles...). Les signaux *suspend* et *resume* permettent de modéliser ces contretemps involontaires dus à l'implémentation.

- *reset* correspond à une réinitialisation de la machine. Cette opération comprend entre autre la remise du processus synchrone dans l'état initial.

- Le vecteur d'exceptions contient des signaux indiquant un dysfonctionnement de la machine. Le signal *overrun*, par exemple, indique qu'une occurrence d'un signal en entrée n'a pas pu être traitée.

Une tâche Esterel peut être modélisée par un processus :

- $\vec{I}$ est constitué de l'ensemble des valeurs des arguments passés à la tâche au moment de son démarrage.

- $\vec{O}$ est constitué de la valeur finale des arguments passés par référence ainsi que de la valeur de retour de la tâche.

- *start* provoque l'évaluation des arguments et le démarrage de la tâche.

- *return* correspond à la mise à jour du signal de retour et à la terminaison de la tâche. Sauf cas particuliers (que nous verrons section 5.4), les arguments passés par référence sont aussi mis à jour.

- Les signaux *kill*, *suspend* et *resume* permettent d'arrêter, de suspendre et de reprendre la tâche.

- Le vecteur d'exceptions ainsi que le signal *reset* n'ont pas d'équivalent pour les tâches dans leur définition actuelle.

Une procédure vue par un programme Esterel est un processus :

- $\vec{I}$ contient les valeurs des arguments au moment de l'appel.

- $\vec{O}$ contient les valeurs finales des arguments passés par référence.

- *start* correspond à l'évaluation des arguments et l'appel de la procédure.

- *return* correspond à la mise à jour des arguments passés par référence et au retour de l'appel.

- Les signaux *reset*, *kill*, *suspend* et *resume* sont absents du vecteur de contrôle.

- Le vecteur d'exceptions est vide.

5.3.2.4 Structure récursive des processus

Les réactions des processus sont rarement indivisibles : elles sont la plupart du temps composées de *sous-réactions* :

- Pour un processus synchrone, la réaction résulte d'une suite d'actions. Ces actions sont elles mêmes divisées en sous-actions, ainsi de suite jusqu'aux instructions atomiques du processeur.
- Pour une tâche ESTEREL, on peut considérer que les sous-réactions correspondent aux quanta de temps alloués à la tâche par l'ordonnanceur sous-jacent.
- Pour une machine d'exécution, les sous-réactions sont les 3 phases ϕ_1, ϕ_2 et ϕ_3.

On peut donc décomposer un processus en sous-processus en partant de la granularité la plus grossière (la réaction du processus en question) à la granularité la plus fine (les instructions atomiques). La figure 5.20 représente la structure récursive des processus. Le flot de données interne du processus n'a pas été représenté car il n'est pas nécessairement séquentiel : les sous-processus M_1, M_2 et M_3 peuvent tous accéder directement à $\vec{I}$.

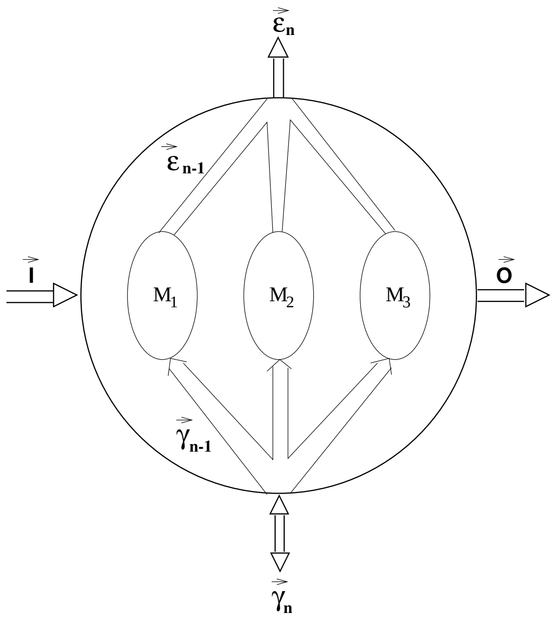


FIG. 5.20 - *Structure récursive d'un processus*

Il existe **plusieurs niveaux d'exécution et de contrôle** d'un processus. À chacun de ces niveaux correspond un sous-vecteur $\vec{\gamma}_i$ de $\vec{\gamma}$. À chaque niveau, les composantes du sous-vecteur de contrôle sont :

$$\vec{\gamma}_i = < start_i, kill_i, suspend_i, resume_i, return_i, reset_i >$$

La figure 5.21 représente un scénario d'exécution de processus à 3 niveaux de granularité :

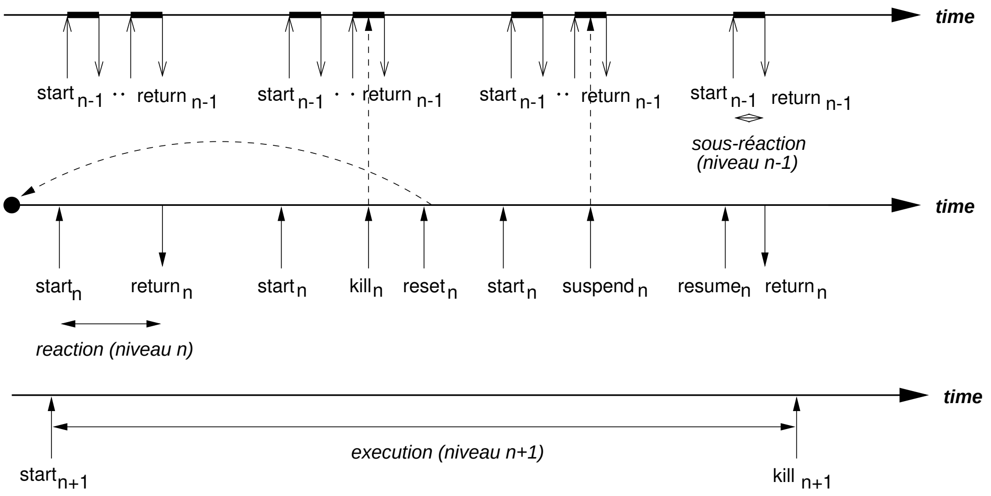


FIG. 5.21 - *Scénario d'exécution d'un processus à 3 niveaux*

- La **réaction** : granularité n. Elle inclut tous les traitements menant de $\vec{I}$ à $\vec{O}$. Elle est constituée d'une ou plusieurs sous-réactions, elle n'est donc pas nécessairement atomique. Une réaction est déclenchée par le signal de contrôle $start_n$, se termine par le signal $return_n$ et peut être interrompue soit définitivement par le signal $kill_n$, soit provisoirement par le signal $suspend_n$.

- La **sous-réaction** : granularité $n-1$. Elle correspond à la réaction d'un processus de niveau $n-1$. Ce processus est contrôlé par les composantes du vecteur $\vec{\gamma}_{n-1}$: le début d'une sous-réaction est déclenché par le signal $start_{n-1}$ et se termine par l'émission du signal $return_{n-1}$. Les signaux $kill_{n-1}$, $suspend_{n-1}$, $resume_{n-1}$ et $reset_{n-1}$ permettent d'interrompre, de reprendre et de réinitialiser le sous-processus.
  Si l'on veut garantir l'atomicité des sous-réactions, aucun signal de contrôle de niveau supérieur (niveau n) ne doit être servi entre $start_{n-1}$ et $return_{n-1}$. La figure 5.21 représente une exécution où les signaux de contrôle de niveau supérieur ne sont pris en compte par un processus qu'"entre" 2 réactions.

- L'**exécution** du processus est une séquence caractérisée par l'expression régulière :

$$(start_n, (start_{n-1}, return_{n-1})^*, kill_n)^*$$

On peut envisager un contrôle de niveau $n+1$ que nous appellerons niveau *exécution*. Ce niveau supplémentaire peut paraître artificiel car il n'y a pas réellement de réaction de niveau $n+1$: celle-ci correspond à la suite infinie de réactions de niveau n qui forment l'*exécution* du processus. Le vecteur $\vec{\gamma}_{n+1}$ associé permet de contrôler cette exécution. Il est réduit à :

$$\vec{\gamma}_{n+1} =< start_{n+1}, kill_{n+1}, suspend_{n+1}, resume_{n+1} >$$

$start_{n+1}$ permet de démarrer l'exécution du processus et $kill_{n+1}$ permet de l'arrêter.
Ce niveau de contrôle peut être utilisé dans les machines d'exécution : dans une machine à fonctionnement périodique, le déclenchement des réactions est géré par une horloge. Le signal $start_n$ sera émis par l'horloge, les signaux $start_{n+1}$ et $kill_{n+1}$ seront alors utilisés par un module de supervision pour démarrer et arrêter la machine (voir 5.3.3.5).
La remarque sur l'atomicité des réactions est aussi valable à ce niveau : si l'on veut garantir l'atomicité des réactions de niveau n, les signaux de contrôle de niveau $n+1$ ne doivent être servis qu'entre les réactions.

Les processus primitifs (de niveau 0) sont des processus dont les réactions sont atomiques. Ceci n'implique pas que leur temps de réaction soit nul : leurs entrées peuvent varier en cours de réaction, on peut s'attendre dans ce cas à un comportement incohérent. On ne peut pas les interrompre. Leur vecteur de contrôle est réduit à $\vec{\gamma}_0 =< start_0, return_0, reset_0 >$.

5.3.2.5 Processus d'interface

Ce sont des processus primitifs, leurs réactions sont donc atomiques. Ils peuvent être contrôlés à 2 niveaux par $\vec{\gamma}_0 =< start_0, return_0, reset_0 >$ et $\vec{\gamma}_1 =< start_1, kill_1, suspend_1, resume_1 >$. La particularité de ces processus réside dans le fait que **leur contrôle provient de 2 sources distinctes**. Ils sont contrôlés à la fois par leur processus père et par leur environnement. Les processus d'interface constituent le point d'interaction entre un processus et son environnement. La notion de processus d'interface permet de modéliser l'émission et l'acquisition des signaux d'entrée/sortie dans les machines d'exécution. Étant donné leur caractère particulier, nous les représenterons par des rectangles grisés.

Processus d'entrée : Pour un processus d'entrée, le vecteur $\vec{I}$ représente le signal externe de l'environnement et $\vec{O}$ le signal synchrone. Dans une machine d'exécution, la stratégie d'acquisition peut varier suivant les besoins de l'application. On en distingue essentiellement 2 correspondant à 2 processus d'interface distincts :

Échantillonnage : Sampler (figure 5.22).

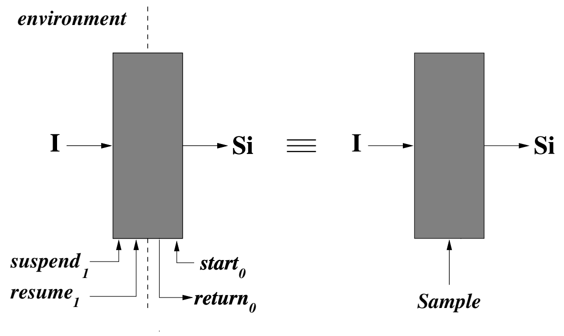


FIG. 5.22 - *Échantillonneur de signal d'entrée (sampler)*

Les réactions du sampler correspondent à un échantillonnage. Le signal $start_0$ produit un nouvel échantillon du signal synchrone S_i à partir des données présentes à l'interface avec l'environnement. Le signal $start_0$ est destiné à être émis par la machine d'exécution en début d'instant. Coté environnement, le contrôle du sampler se fait par les signaux $suspend_1$ et $resume_1$: toute variation de I est gardée par une suspension ($suspend_1$) et une reprise ($resume_1$) des exécutions[3] du sampler. Ce couplage à l'environnement permet d'assurer *l'atomicité des réactions du sampler vis-à-vis des variations du signal d'entrée*. La figure 5.23 représente un scénario d'interaction entre sampler et environnement.

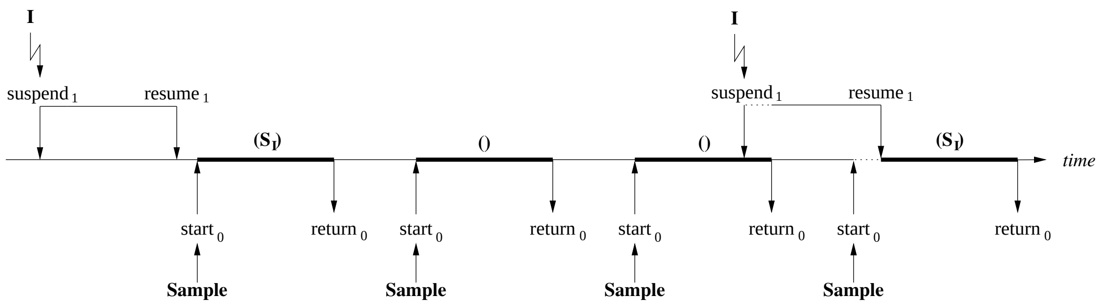


FIG. 5.23 - *Scénario d'interaction sampler/environnement*

Il est à noter que l'implémentation des processus d'interface fait le plus souvent appel à des mécanismes d'*exclusion mutuelle* de manière à garantir l'exclusivité des sections ($start_0, return_0$) et ($suspend_1, resume_1$). Pour simplifier, nous représenterons dans la suite les samplers par des boîtes dotées de la commande `Sample` (figure 5.22).

Cette stratégie de scrutation (*polling*) est particulièrement bien adaptée aux signaux d'entrée valués non déclencheurs (`sensor` ESTEREL). En revanche, les samplers sont moins bien adaptés aux entrées événementielles. En particulier, le

3. Pas ses réactions

déclenchement se faisant à l'initiative de la machine, si le signal I varie 2 fois entre 2 demandes d'échantillonnage, il n'y a aucun moyen de s'en rendre compte.

Interruption : Handler (figure 5.24).

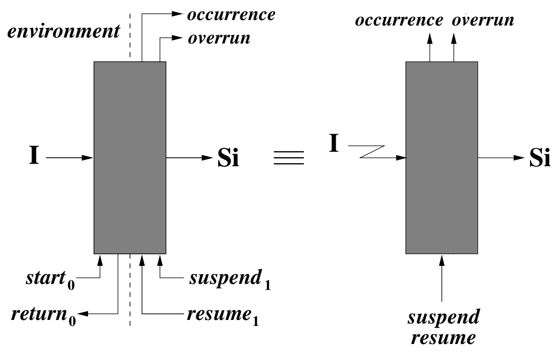


FIG. 5.24 - *Handler de signal d'entrée*

Les handlers sont les symétriques des samplers. Les réactions du handler sont contrôlées par l'environnement ($start_0, return_0$) de manière à ce que *toute variation de I soit suivie d'une réaction du handler* et donc d'une mise à jour de S_i. Le signal S_i suit ainsi les évolutions de l'environnement. Côté synchrone, le contrôle du handler n'est que partiel : on ne peut que suspendre et réactiver son exécution[4] grâce aux signaux $suspend_1$ et $resume_1$, ce qui revient à masquer et démasquer le signal $start_0$. Les handlers, permettent de modéliser l'acquisition faite par des gestionnaires d'interruptions. Ils peuvent toutefois être implémentés par d'autres mécanismes, notamment par des threads. Dans la suite, nous représenterons les handlers par des boîtes dotées d'un signal d'entrée en *éclair* pour indiquer le caractère déclencheur. Deux signaux d'exceptions peuvent être émis par un handler :

- *occurrence* indique une nouvelle occurrence de signal d'entrée,
- *overrun* indique la perte d'un échantillon : une nouvelle occurrence a eu lieu (I) alors que la précédente (S_i) n'a pas été traitée.

Ces signaux peuvent être utilisés par la machine pour modifier son comportement : le signal *occurrence* peut être utilisé pour provoquer la réaction du processus de niveau supérieur, réalisant ainsi un fonctionnement *event driven*.

Dans notre application, ce sont des handlers qui sont utilisés pour tous les signaux d'entrée. D'une manière générale, pour des entrées événementielles, il est plus judicieux de choisir des handlers car ils permettent de détecter les pertes d'événements (*overrun*).

4. Pas ses réactions

Processus de sortie : Pour un processus de sortie, le vecteur $\vec{I}$ représente le signal synchrone et $\vec{O}$ l'événement externe à émettre. On retrouve en sortie 2 processus symétriques, l'un correspondant à des mises à jour passives, l'autre à des mises à jour avec interruptions.

5.3.2.6 Conclusion

Nous avons présenté un modèle récursif de processus en insistant sur l'aspect contrôle. Nous avons vu que les processus synchrones entraient dans le cadre de ce modèle. Toutefois ce modèle général ne fait aucun état des interactions avec l'environnement. Des processus particuliers ont donc été introduits pour modéliser ces interactions entre monde synchrone et environnement. Nous allons maintenant voir comment construire des machines d'exécution en composant processus classiques et processus d'interface.

5.3.3 Boîtes réactives

5.3.3.1 Boîte réactive de base

La notion de boîte réactive est un modèle permettant de décrire les machines d'exécution. Le rôle d'une boîte réactive est de gérer de manière cohérente l'exécution d'un processus synchrone dans un environnement asynchrone. La boîte réactive de base est représentée figure 5.25.

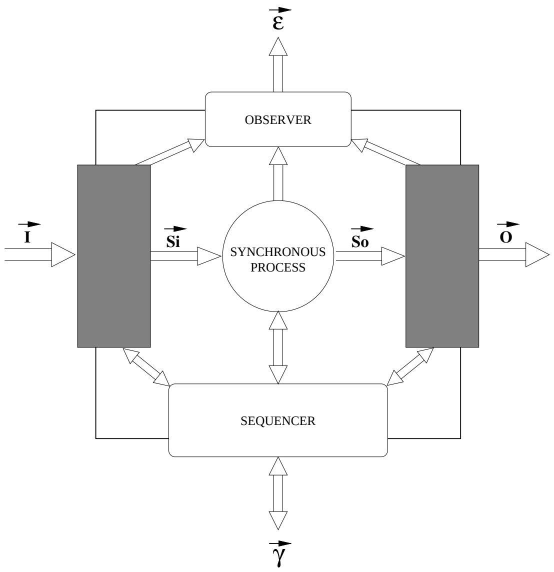


FIG. 5.25 - *Boîte réactive de base*

Les boîtes réactives ont toutes les caractéristiques des processus tels que nous les avons décrits dans la section précédente. Leur particularité vient du fait qu'elles sont composées de 3 sous-processus :

- Un processus d'interface d'entrée,
- Un processus synchrone central,
- Un processus d'interface de sortie.

Contrairement aux processus, dans une boîte réactive, le flot de données est séquentiel : le processus synchrone n'interagit avec l'environnement qu'à travers les modules d'interface.
Le séquenceur est chargé de gérer les 3 phases des réactions en fonction du vecteur de contrôle $\vec{\gamma}$.
L'observateur est chargé de détecter et de traiter ou signaler les anomalies de fonctionnement de la boîte.

5.3.3.2 Boîtes réactives composées

La boîte réactive de base représentée figure 5.25 est une boîte primitive. Nous allons voir maintenant qu'on peut construire des boîtes réactives plus élaborées par composition de processus. Pour que le résultat de cette composition soit une boîte réactive, la contrainte à respecter est la présence de processus d'interface en entrée et en sortie. Nous présentons à titre d'exemple les boîtes réactives associées aux :

- Contrôleurs d'entrée,
- Contrôleurs d'entrée avec mémorisation,
- Modules d'entrée,
- Machines d'exécution.

Contrôleur d'entrée

Nous avons vu section 5.2.3 que les contrôleurs d'entrée incluaient souvent un traitement de type conversion ou même plus élaboré (détection de fronts, etc...). On peut modéliser ces contrôleurs par une composition de 2 processus :

- Un processus d'interface d'entrée pour l'acquisition proprement dite,
- Un processus de traitement du signal acquis.

On peut voir un contrôleur d'entrée comme une boîte réactive dégénérée sans module de sortie. La figure 5.26 représente deux contrôleurs d'entrée issus l'un d'un sampler et l'autre d'un handler.

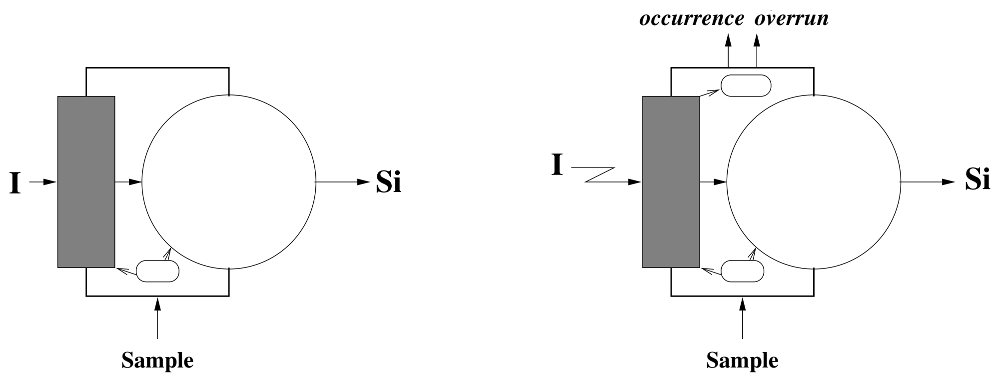


FIG. 5.26 - *Contrôleurs d'entrée*

Le séquenceur du premier ne fait qu'activer séquentiellement le processus d'interface puis le processus de traitement. Dans le deuxième cas, le séquenceur suspend l'exécution du handler avant d'activer le processus de traitement. On notera que malgré sa commande `Sample` le deuxième contrôleur n'a ni la sémantique ni l'interface d'un sampler : le signe *éclair* indique que l'environnement est déclencheur.

Contrôleur d'entrée avec mémorisation

On peut utiliser les handlers et samplers pour modéliser des contrôleurs d'entrée plus sophistiqués. La figure 5.27 représente la structure interne d'un contrôleur permettant de mémoriser les événements asynchrones à mesure de leur arrivée et de les resynchroniser sur les réactions de la machine.

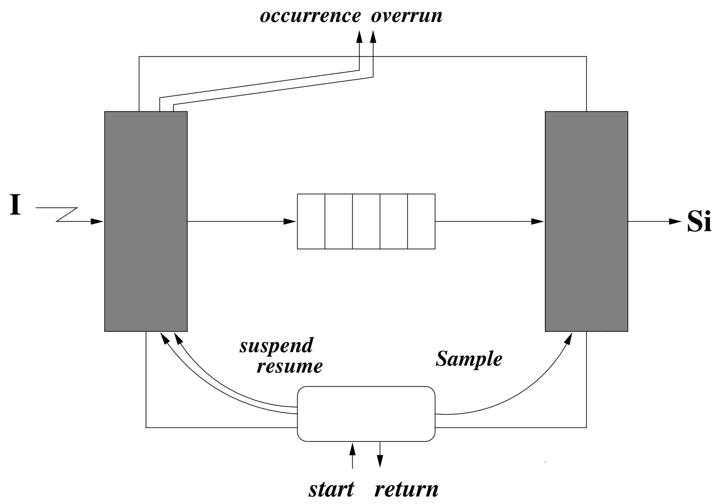


FIG. 5.27 - *Contrôleur avec mémorisation des occurrences dans une file d'attente*

Le handler est couplé à l'environnement et place les signaux dans une file

d'attente. Le signal *overrun* indique ici, que la taille de la file est dépassée. L'accès à la file d'attente est critique : l'activation du contrôleur se traduit par la suspension de l'exécution du handler (*handler.suspend*), l'activation du sampler (*sampler.sample*) puis la réactivation du handler (*handler.resume*).
Ce type de contrôleur d'entrée est utilisé dans des applications où l'hypothèse synchrone est vérifiée *en moyenne* et permet de faire face à des *rafales* occasionnelles de signaux d'entrée.

Module d'entrée

Le module d'entrée d'une machine d'exécution tel qu'il a été présenté section 5.2.3 peut être modélisé par une composition de processus (figure 5.28).

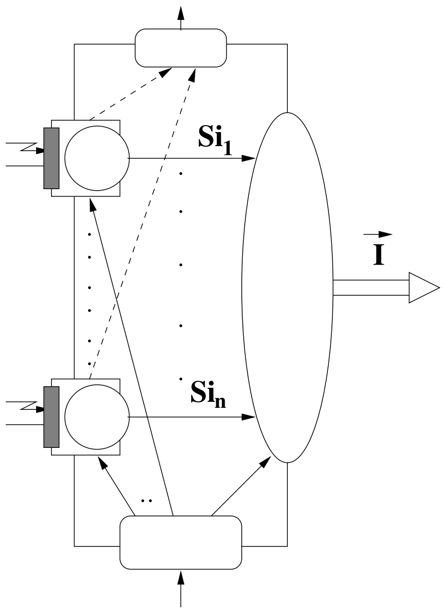


FIG. 5.28 - *Module d'entrée*

Les composants du module d'entrée sont :

- Un contrôleur pour chacun des signaux d'entrée,
- Un module de construction de l'événement $\vec{I}$.

Le séquenceur active *simultanément* tous les contrôleurs d'entrée puis le module de construction de l'événement. Dans le cas général les signaux S_{I_i} de sortie des contrôleurs ne sont pas directement les composantes de $\vec{I}$. Le constructeur d'événement est un processus classique qui construit $\vec{I} = (I_1, ...I_m)$ à partir des signaux $S_{I_1}, ..., S_{I_n}$.
Les modules d'entrée permettent de constituer une image de l'environnement à

un instant donné pour le processus synchrone.

Les contrôleurs de sortie et le module de sortie étant les symétriques de ceux d'entrée, nous n'entrerons pas dans leur détail.

Machine d'exécution

Nous pouvons décrire la machine d'exécution de l'application d'asservissement comme une composition de processus formant une boîte réactive. Le résultat est représenté figure 5.29.

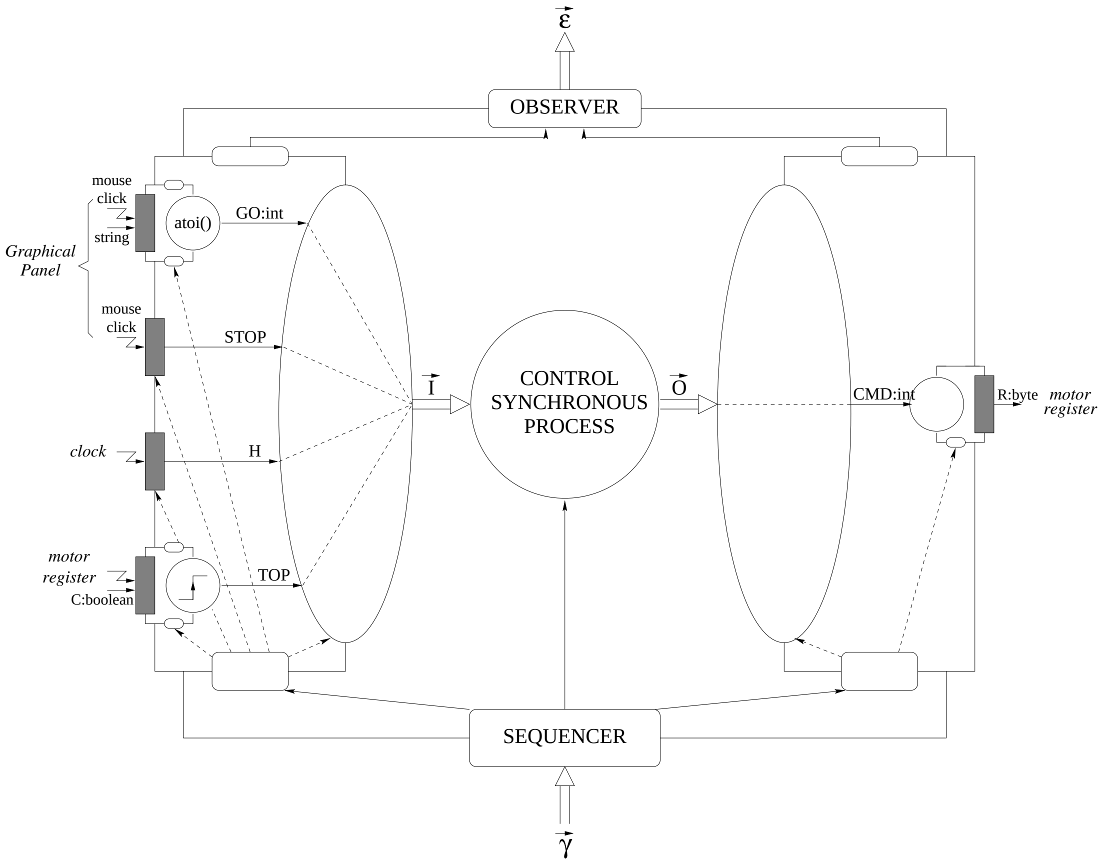


FIG. 5.29 - *Machine d'exécution de l'application*

5.3.3.3 Observateurs

Nous avons vu qu'à chaque boîte réactive est associé un observateur chargé de détecter les anomalies d'exécution. Ces anomalies peuvent être traitées à différents niveaux :

- Localement : l'observateur d'un contrôleur d'entrée peut, par exemple, décider en cas d'"overrun" d'écraser la donnée précédente au profit de la plus récente.

- À un niveau supérieur : si l'anomalie ne peut pas être traitée par l'observateur local, il y a propagation d'un signal d'exception à la manière des exceptions dans les langages de programmation comme ADA ou `C++`. Nous avons vu par exemple que l'occurrence d'un signal d'entrée pouvait provoquer l'émission d'un signal d'exception par l'observateur du contrôleur. Si l'occurrence du signal d'entrée doit déclencher une réaction de la machine, le signal d'exception sera propagé au niveau supérieur puis capturé et traité à un niveau suffisamment haut pour permettre le déclenchement d'une réaction. D'une manière générale, les exceptions propagées peuvent être traitées :
  - soit par un observateur de niveau supérieur dans le cas de boîtes réactives composées récursivement,
  - Si aucun observateur n'a traité l'exception, celle-ci sera reportée dans le vecteur d'exceptions de la boîte de niveau le plus haut. Dans ce cas un traitement adéquat devra être mis en œuvre par l'utilisateur. Cette exception peut par exemple être traitée par une autre boîte réactive dans le cas d'une composition hiérarchique (voir 5.3.3.5).

5.3.3.4 Séquenceur

Le séquenceur est le "chef d'orchestre" de la boîte réactive. Il reçoit les signaux du vecteur de contrôle $\vec{\gamma}$ en fonction desquels il doit commander de manière cohérente les sous-processus qui la composent. Il a 2 rôles :

- Définir la sémantique des réactions : il s'agit de définir l'ordonnancement des sous-processus. Cet ordonnancement est spécifique à chaque boîte : il n'est pas le même dans un InputModule que dans une machine d'exécution par exemple. Pour cette dernière, l'ordonnancement le plus simple et souvent le plus efficace consiste à enchaîner les réactions des 3 sous-modules d'acquisition de transition et d'émission.

- Définir la sémantique exacte des signaux de contrôle $kill$, $suspend$ et $resume$. Pour éviter des problèmes d'incohérence, nous retenons le principe suivant : un signal de contrôle de niveau n ne doit pas interrompre une réaction de niveau $n-1$. La figure 5.30 représente le comportement d'un séquenceur basé sur ce principe. On y retrouve la hiérarchie d'états correspondant aux niveaux n et $n-1$. Les signaux $kill_n$, $suspend_n$ et $reset_n$ ne sont servis que si le processus de niveau inférieur est au repos (état `idleN-1`). On notera que les signaux reçus en cours de sous-réaction ne sont pas mémorisés. Il appartient à l'entité qui les émet de les maintenir jusqu'à ce qu'ils soient servis. On peut modéliser ce comportement en ESTEREL de la manière suivante :

```
weak abort
  sustain killN
when killedN
```

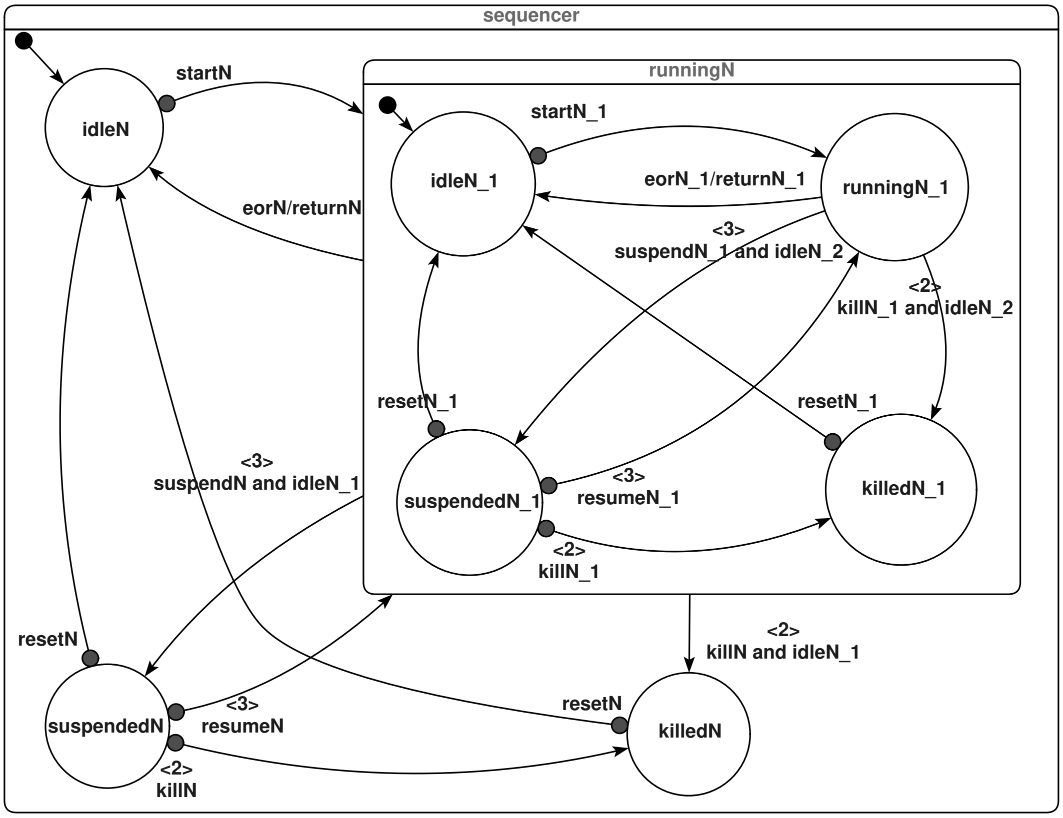


FIG. 5.30 - *Comportement du séquenceur*

5.3.3.5 Composition hiérarchique de boîtes réactives

On peut construire des machines d'exécution sophistiquées en composant des boîtes réactives de manière récursive. Dans cette composition, le flux de données (vecteurs d'entrée et de sortie) est traité séquentiellement par plusieurs sous-boîtes. Il y a d'autres manières de composer les boîtes, notamment la composition hiérarchique. Une boîte réactive A peut *contrôler* une boîte B par l'intermédiaire de son vecteur de contrôle. On obtient ainsi une hiérarchie de boîtes réactives dans laquelle, le flux de données du niveau n est utilisé partiellement ou totalement comme flux de contrôle pour le niveau inférieur. La composition hiérarchique permet, dans les applications de contrôle par exemple, de gérer la commutation entre les différents modes de fonctionnement.

Application à l'asservissement. La perte d'un front montant du signal d'entrée `C` de la machine peut induire des erreurs sur le calcul de la commande du moteur. Les contraintes de sûreté de fonctionnement peuvent imposer que si cette anomalie se reproduit trop souvent, le contrôle passe dans un mode dégradé. Dans ce mode, il n'y a plus de boucle d'asservissement, l'opérateur a le choix entre l'arrêt du moteur (`STOP`) ou une vitesse de repli prédéfinie.
Si l'anomalie qui a causé le passage en mode dégradé disparaît, on peut décider de repasser en mode normal. Si au contraire elle persiste, on peut considérer que le dysfonctionnement est irrémédiable et arrêter complètement le système.
On peut réaliser cette commutation en introduisant un module de supervision. Ce module est lui-même un système réactif dont les entrées sont les signaux d'exception des machines normale et dégradée et dont les sorties sont les signaux de contrôle de ces 2 machines. La figure 5.31 représente l'ensemble de l'application avec gestion des modes de fonctionnement.

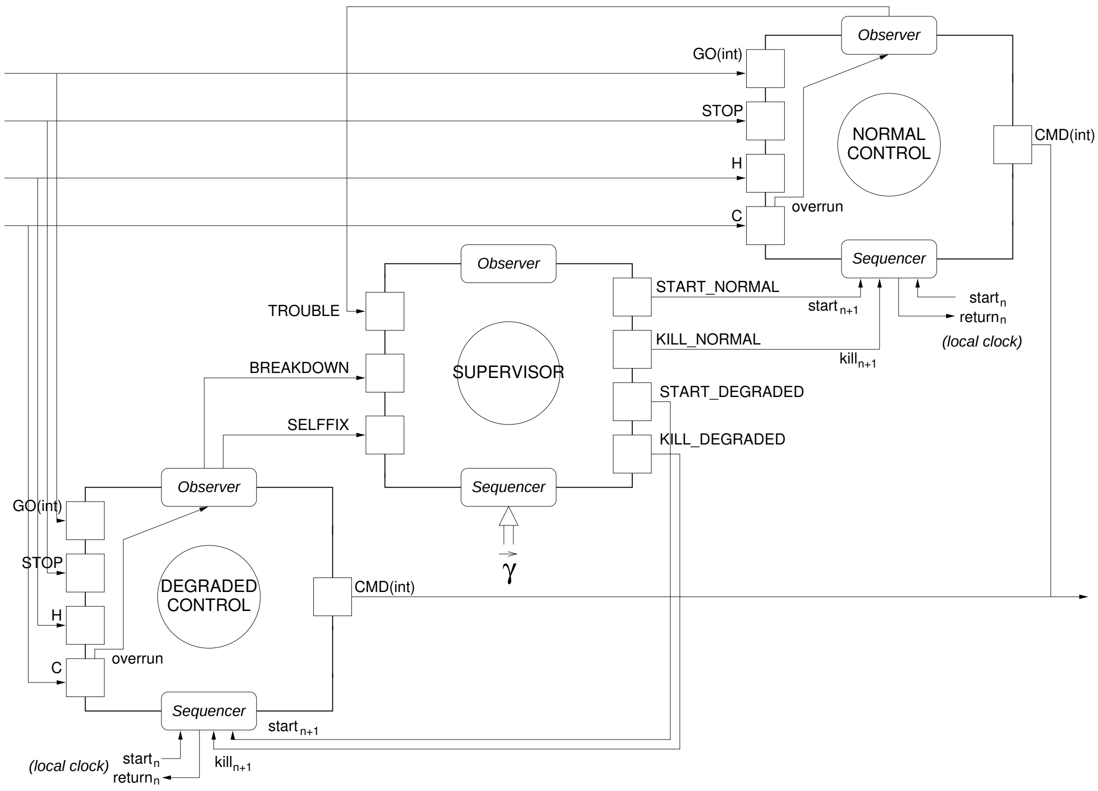


FIG. 5.31 - *Architecture complète de l'application*

On y retrouve les 2 machines d'exécution des 2 modes ainsi que la machine du module de supervision. Ces 3 machines s'exécutent indépendamment l'une de l'autre : elles ont chacune une horloge locale qui leur délivre le signal $start_n$. La machine du superviseur gère l'*exécution* des 2 autres par l'intermédiaire de leurs

signaux $start_{n+1}$ et $kill_{n+1}$[5].
Les observateurs des machines en mode normal et dégradé peuvent être des systèmes plus ou moins complexes. On donne les comportements suivants à titre d'exemple :

- En mode normal, le signal d'exception `TROUBLE` est émis s'il y a eu plus de N *overrun* consécutifs.
- En mode dégradé, `SELFFIX` est émis s'il n'y a pas eu d'*overrun* depuis plus de N_1 instants de la machine. `BREAKDOWN` est émis s'il y a eu plus de N_{max} *overrun* consécutifs.

5.3.4 Comparaison avec le modèle des Objets Synchrones

Les objets réactifs et les boîtes réactives ont un certain nombre de points communs : les deux sont des boîtes noires qui ont des signaux d'entrée, des signaux de sortie et un comportement que l'on peut activer. La différence fondamentale est que contrairement aux objets synchrones, le modèle de boîtes réactives ne fait aucune supposition sur l'environnement d'exécution.

Les objets synchrones n'ont pas été conçus pour s'exécuter dans un environnement asynchrone. Ils ne peuvent s'exécuter de manière cohérente que s'ils évoluent en alternance avec leur environnement. Il est donc implicitement supposé que l'environnement n'évolue pas au cours des réactions des objets synchrones. Considérer que l'environnement est figé durant les réactions revient à considérer le temps de réaction des objets comme nul. Le problème de l'atomicité des réactions n'est donc pas posé. Ceci explique par exemple que les objets synchrones ont une méthode d'activation mais que contrairement aux boîtes réactives ils n'ont pas de méthode permettant de savoir si la réaction est terminée.
Pour résoudre le problème, F. Boulanger propose d'introduire des modules d'interface. Les modules proposés dans `libSync` ont tous des comportements d'échantillonneurs : ils permettent de faire de la simulation mais ne prennent pas réellement en compte le parallélisme entre monde synchrone et environnement. Le principe des modules d'interface a été adopté et devient un concept de base dans les boîtes réactives. On peut dire que F. Boulanger s'est concentré sur la composition synchrone d'objets synchrones tandis que notre préoccupation centrale porte sur la frontière synchrone/asynchrone.

Contrairement à un objet synchrone, une boîte réactive bien conçue a un comportement cohérent dans un environnement asynchrone et réalise ainsi une machine d'exécution. La prise en compte du parallélisme environnement/machine se traduit d'abord par un temps de réaction non nul : une réaction a un début, une fin, elle peut être suspendue, ... d'où la nécessité d'un vecteur de contrôle.

5. Ces signaux permettent d'arrêter et redémarrer une machine sans pour autant interrompre la réaction en cours

Dans ce contexte, l'atomicité des réactions constitue un impératif à garantir. Ceci est réalisé en concentrant les interactions avec l'environnement dans des phases d'acquisition et de restitution des signaux d'entrée/sortie. La présence de handlers d'entrée et d'observateurs permet de traiter les évolutions de l'environnement en cours de réaction. En résumé, on peut dire que les boîtes réactives sont des machines d'exécution pour objets réactifs.

Par ailleurs, F. Boulanger a proposé une interface standard pour la gestion des tâches asynchrones, laissant à l'utilisateur le soin d'en fournir le support. La gestion des tâches est du ressort de la machine d'exécution et fait l'objet de la section 5.4.

5.3.5 Conclusion

Le modèle de boîtes réactives a été introduit comme modèle de base pour la description des machines d'exécution. Ce modèle permet de décrire les différentes stratégies d'acquisition/émission de signaux. Il permet aussi de définir des stratégies ad-hoc pour l'enchaînement des phases des réactions et le traitement des anomalies. La composition récursive de boîtes réactives permet de concevoir des machines sophistiquées répondant à des besoins particuliers. La composition hiérarchique permet, elle, de gérer la commutation des modes de fonctionnement d'une application. En somme, ce modèle et l'architecture qui en découle semblent pouvoir décrire la plupart des mécanismes d'exécution de processus synchrones. Néanmoins, jusqu'à présent, nous n'avons parlé que d'applications purement réactives. Or un grand nombre d'applications réelles comportent une partie réactive et des traitements transformationnels. Ceux-ci doivent souvent s'exécuter de manière asynchrone sur la même cible que le processus synchrone. La gestion de ces traitements entre entièrement dans le cadre de la machine d'exécution. Il s'agit donc à présent d'intégrer ces traitements asynchrones dans l'architecture de la machine.

5.4 Tâches

5.4.1 Introduction

Tâches Esterel: Tous les traitements qui ont lieu au cours d'une réaction d'un programme synchrone doivent avoir une durée négligeable par rapport à la constante de temps de l'environnement. Les autres traitements, ceux dont on ne peut pas négliger la durée, ne peuvent pas avoir lieu *dans* l'instant. Ces derniers sont caractérisés dans le monde synchrone par un début et une fin, et ne peuvent s'exécuter que dans l'environnement, **parallèlement** au programme synchrone. Ces traitements peuvent correspondre à des activités autonomes de l'environnement comme le déplacement du bras d'un robot par exemple. Il peut aussi s'agir

de traitements transformationnels complexes (inversion d'une matrice).
La notion de tâche (`task`) et la primitive `exec` ont été introduits [Par92] dans le langage Esterel pour permettre d'inclure de tels traitements dans du code réactif synchrone. Chaque instruction `exec` représente l'exécution d'une tâche. On peut ainsi, à partir d'un programme synchrone, décider de démarrer un calcul, de le suspendre, de l'arrêter, ... chacune de ces actions se traduisant par l'émission d'un signal vers l'environnement. L'exécution réelle de la tâche dans l'environnement est considérée comme un détail d'implémentation et est entièrement laissée aux soins de l'utilisateur.

Objectif : On voit donc que l'exécution des tâches entre complètement dans la problématique des machines d'exécution. En particulier, pour les tâches non autonomes (tâches qui nécessitent des ressources informatiques pour s'exécuter), la machine est tenue de fournir le support nécessaire à une exécution en parallèle avec le processus synchrone. On suppose pour cela que l'on dispose d'un exécutif multi-tâche. Trois éléments entrent donc en jeu :

- Le code réactif synchrone avec sa vision abstraite des tâches,
- Un exécutif multi-tâche fournissant les services de gestion d'activités parallèles,
- Le code des tâches fourni par l'utilisateur.

Le rôle de la machine est de faire l'interface entre ces 3 entités et d'exécuter en parallèle le code synchrone et les tâches de manière cohérente.
Nous présenterons dans un premier temps deux concepts de tâche différents : celui du langage Esterel puis celui des exécutifs multi-tâche. Nous définirons ensuite le rôle de la machine et proposerons à travers un contrôleur d'`exec`, une sémantique pour la gestion des tâches. Enfin, nous verrons comment intégrer cette gestion dans l'architecture générale de la machine d'exécution.

5.4.2 Modèle Esterel des tâches

La primitive `call` permet d'exécuter à partir d'un programme Esterel, une procédure externe écrite dans un langage séquentiel classique comme C, à condition que celle-ci puisse être considérée comme instantanée. La notion de tâche a été introduite pour couvrir le cas des traitements non instantanés. Une tâche Esterel est une procédure externe qui s'exécute en concurrence avec le programme synchrone. Au niveau d'abstraction d'Esterel, on ne se préoccupe pas des moyens mis en œuvre pour l'exécution effective de la tâche en parallèle avec le programme. Esterel a une vision abstraite de l'exécution d'une tâche, représentée par les événements suivants :

- La tâche doit être démarrée,

- La tâche doit être arrêtée,
- La tâche doit être suspendue,
- La tâche doit être reprise,
- La tâche a terminé.

5.4.2.1 Déclaration d'une tâche : `task`

Une tâche est déclarée d'une manière analogue à celle d'une procédure :

```
task <task-id> (<type-list>) (<type-list>);
```

La première liste d'arguments est passée par référence, la deuxième par valeurs. Le signal de retour ne fait pas partie de la déclaration de tâche (voir plus loin).

5.4.2.2 Exécution d'une tâche : `exec`

Si l'on fait abstraction des arguments, on peut penser que la macro suivante suffit à gérer l'exécution d'une tâche :

```
emit startTASK;
await TASKreturn
```

Si cette séquence est préemptée (avortement ou suspension), il faut interrompre l'exécution de la tâche dans l'environnement en émettant un signal du type `killTASK` ou `suspendTASK`. J-P. Paris a montré [Par92] que la gestion de ces signaux ne pouvait pas se faire par une simple macro. Ceci a mené à l'introduction de la primitive `exec` :

```
exec <task-id> (<ref-params>) (<val-params>) return <signal-id>;
```

Chaque `exec` correspond à l'exécution d'une tâche. On peut ainsi avoir dans un même programme plusieurs exécutions de la même tâche. Un `exec` peut donc être considéré comme une instance de tâche. À chaque `exec` est associé un signal de retour et 4 signaux que nous qualifierons d'implicites car ils n'apparaissent pas dans la sorte du programme synchrone : `start, kill, suspend, activate`.

- Le signal `start` est émis quand l'`exec` devient actif. Il indique à l'environnement qu'une nouvelle instance de la tâche doit être démarrée. `start` porte la liste des paramètres par valeurs et par référence.
- Le signal `kill` est émis quand un `exec` est avorté (par un `abort...when` par exemple). Il indique à l'environnement que la tâche doit être arrêtée.
- De même, à chaque instant où l'`exec` est suspendu, le signal `suspend` est émis vers l'environnement.

- Le signal `activate` est lui émis à chaque instant où l'`exec` est actif: c'est à dire à chaque instant où l'`exec` a démarré, n'a ni terminé, ni été avorté, ni été suspendu.

- `signal-id` est le signal de retour de l'`exec`. C'est un signal d'entrée du module ESTEREL. Contrairement aux signaux d'entrée classiques, il ne peut pas être émis par le code synchrone. Il est déclaré de la manière suivante:

  ```
  return <signal-id>: <type>;
  ```

  Il peut être valué et il doit être émis par l'environnement pour indiquer que la tâche a terminé. À sa réception, les arguments passés par référence sont mis à jour et la primitive `exec` termine instantanément.

Redémarrage immédiat d'un `exec` Dans certaines situations, un `exec` peut être avorté puis redémarré dans le même instant. C'est le cas par exemple dans:

```
every S do exec T()() return R end every
```

À la réception du signal S, les signaux implicites `start` et `kill` sont émis simultanément indiquant que l'instance courante de la tâche doit être arrêtée et qu'une nouvelle instance doit être démarrée.

Suspension d'un `exec` et terminaison de la tâche Dans un même instant, il peut arriver qu'une tâche termine et que l'`exec` soit suspendu. D'après la sémantique d'ESTEREL, l'`exec` étant suspendu, l'occurrence du signal de retour n'est pas prise en compte et est perdue. Bien que peu probable, cette situation doit être prise en compte: l'environnement doit **maintenir** le signal de retour jusqu'à ce que l'`exec` ne soit plus suspendu.

La figure 5.32 représente les 5 signaux d'interaction entre un module synchrone et une instance de tâche.

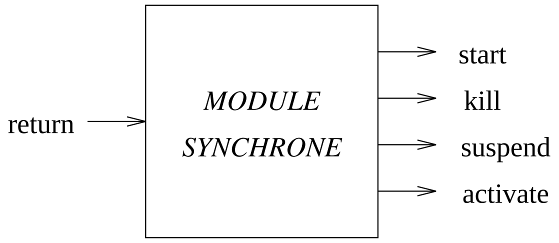


FIG. 5.32 - *Signaux de gestion d'une tâche*

5.4.3 Modèle des tâches d'un exécutif

Nous avons vu l'abstraction d'une tâche faite par ESTEREL. Nous allons maintenant caractériser la notion de tâche telle qu'elle existe dans les exécutifs.

- Une tâche est un flot de contrôle séquentiel exécutant une suite d'instructions qui forme le *corps* de la tâche.

- Dans la plupart des exécutifs, le corps d'une tâche peut prendre des arguments et retourner une valeur. Dans les exécutifs qui n'offrent pas cette possibilité (comme RTC par exemple), on peut mettre à profit le partage de la mémoire pour l'implémenter.

- Dans un système donné, toutes les tâches partagent les ressources d'exécution (processeurs) disponibles: elles s'exécutent en *concurrence*. Un ordonnanceur gère le partage de ces ressources entre les différentes tâches. Dans la plupart des exécutifs, l'ordonnancement est basé sur des *priorités* attribuées à chaque tâche. Ce point est important car il permet de donner la priorité aux réactions du processus synchrone par rapport aux traitements asynchrones.

- 2 tâches peuvent ou non partager le même espace d'adressage et donc les mêmes données. Suivant le cas, on a affaire à des *threads* ou à des *processus*.

Primitives de manipulation des tâches. La plupart des exécutifs présentent les primitives de manipulation de tâches suivantes:

- `START`: création et démarrage d'une tâche. Cette primitive permet de spécifier le corps à exécuter et éventuellement un argument à passer au corps. Dans certains exécutifs, démarrage d'une tâche et création de la structure de donnée associée font l'objet de 2 primitives différentes. On peut alors facilement se ramener à une primitive unique.

- `KILL`: arrêt de la tâche[6].

- `SUSPEND`: suspension momentanée de l'exécution.

- `RESUME`: reprise de l'exécution après une suspension.

D'autre part, on peut implémenter un mécanisme de signalisation de la terminaison d'une tâche: un indicateur `RETURNED` indique que la tâche a terminé son exécution.

6. Cette primitive n'existe pas dans Solaris2.

Asynchronisme des exécutifs. Dans la plupart des exécutifs, ces primitives ont un comportement *asynchrone* : elles ne font qu'enregistrer les demandes. Les opérations sont réellement effectuées dans un délai qui varie selon le caractère plus ou moins *"temps réel"* de l'exécutif. Si par exemple, on invoque la primitive de suspension d'une tâche, au retour de la primitive, on n'a aucune garantie que la tâche en question a bien été suspendue. Dans certaines situations, ces opérations peuvent être critiques et cette absence de garantie devient inacceptable.
Nous proposons une solution qui consiste à mettre au point un mécanisme d'acquittement : à chaque requête, un acquittement est émis une fois que l'opération a été accomplie :

Requête	Acquittement
START	STARTED
KILL	KILLED
SUSPEND	SUSPENDED
RESUME	RESUMED

Ces mécanismes d'acquittement ne sont pas toujours indispensables et ils peuvent être plus où moins difficiles à mettre en place selon l'exécutif. Un exemple d'interface utilisateur d'une tâche dans le cas idéal est représenté figure 5.33.

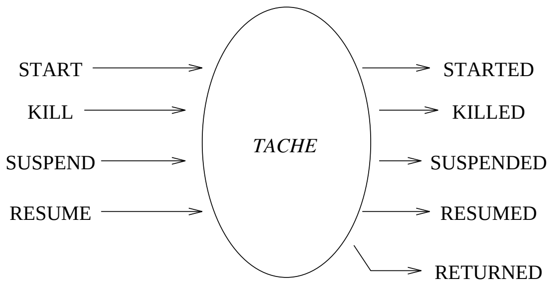


FIG. 5.33 - *Exemple d'interface d'une tâche d'un exécutif*

5.4.4 Rôle de la machine d'exécution

5.4.4.1 Contrôleur d'`exec`

Pour chaque instance de tâche d'un programme ESTEREL, c'est-à-dire pour chaque `exec`, la machine doit :

- Gérer la tâche à l'aide des primitives de manipulation de l'exécutif (`START`,...) et en fonction des signaux émis par le code synchrone (`start`,...).
- Mettre à jour le signal de retour (`return`) de la tâche en accord avec la sémantique de la primitive `exec`.

La partie de la machine qui s'occupe de la gestion d'un `exec` est un système réactif qui fait l'interface entre le code synchrone et la tâche réelle. On l'appellera **contrôleur d'**`exec` (`execController` : figure 5.34).

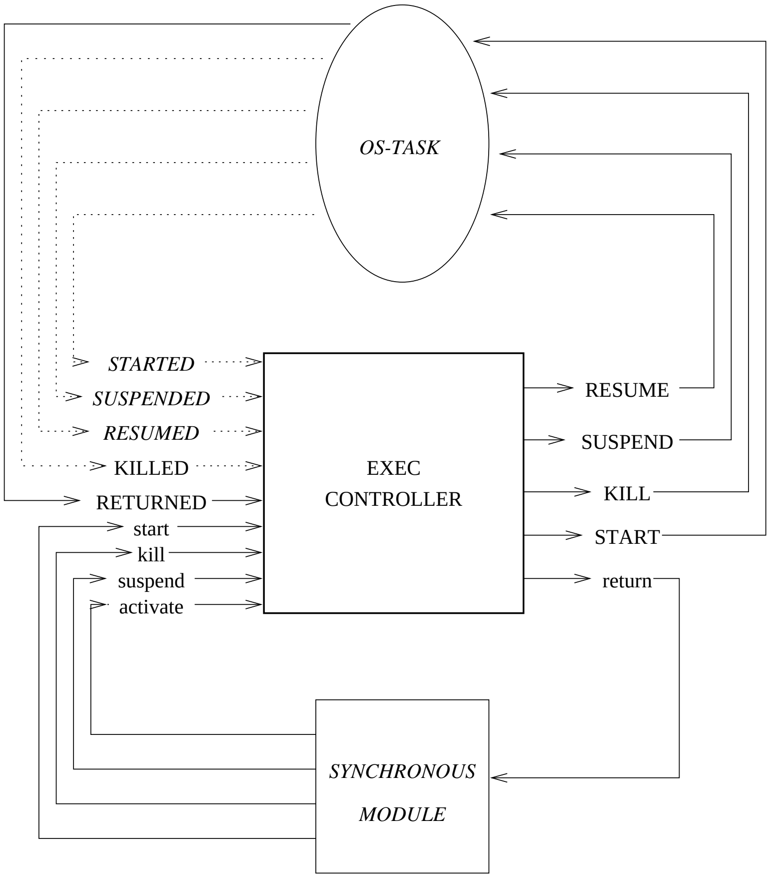


FIG. 5.34 - *Contrôleur d'*`exec`

Ses signaux d'entrée sont d'une part les signaux ESTEREL provenant du programme et d'autre part les acquittements provenant de l'exécutif. Ses signaux de sortie sont des requêtes vers l'exécutif et le signal de retour de tâche vers le code synchrone.
Sur la figure 5.34 les liaisons pointillées en entrée du contrôleur indiquent que les signaux `KILLED, STARTED, SUSPENDED, RESUMED` peuvent ne pas être nécessaires (voir section 5.4.4.2).

Dans la suite, les signaux en minuscule (`start`...) correspondent aux signaux entre le module synchrone et le contrôleur d'`exec` et les signaux en majuscules (`START`...) sont ceux entre le contrôleur et l'exécutif.

5.4.4.2 Sémantique des tâches dans la machine

Le code synchrone ne fait que donner des ordres à la machine quand l'état d'une tâche doit être modifié. Rien ne spécifie la manière dont ces ordres doivent être exécutés ni leur délai d'exécution. Une machine donnée peut donner plus où moins de garanties quant à l'exécution de l'opération. Il y a donc à faire des choix qui sortent du domaine du langage synchrone.
Prenons l'exemple du signal `start` émis par un programme : il signifie qu' *"une tâche doit être démarrée"*. On peut envisager plusieurs sémantiques :

Couplage "maximal" ou "synchrone" La machine d'exécution garantit que le démarrage de la tâche a lieu avant le début de l'instant suivant. Si ce n'est pas le cas, la machine doit le détecter et lever une exception comme sur le chronogramme suivant :

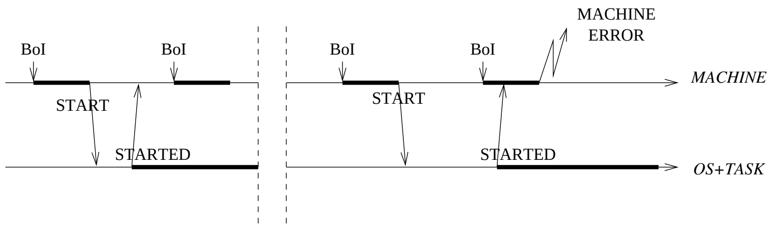


FIG. 5.35 - *Retard dans le démarrage d'une tâche*

On voit sur la figure 5.35 que l'émission de `start` par le processus synchrone se traduit par l'invocation de la primitive `START` par la machine pour demander le démarrage d'une tâche. Si la tâche n'a pas démarré (signal `STARTED`) avant le début de l'instant suivant, comme c'est le cas dans le deuxième scénario, la machine lève une exception. Cette sémantique nécessite, bien entendu, que le mécanisme d'acquittement ait été mis au point sur l'exécutif en question.
On peut envisager une machine qui garantit que toutes les opérations `start, kill, suspend, resume` sont effectuées avant le début de l'instant suivant et qui lève des exceptions en cas de violation. Ces exceptions peuvent être traitées de différentes manières suivant le caractère plus ou moins critique de la situation : on peut aussi bien les ignorer que procéder à un arrêt d'urgence. L'essentiel est que la machine soit en mesure de *détecter* ce genre de situations en cas de besoin. Une telle machine permettrait d'avoir un **couplage synchrone**[7] entre l'`exec` ESTEREL et la tâche réelle.

Couplage "minimal" Le principe de la machine "*minimale*" consiste à laisser l'exécutif faire de son mieux pour gérer les tâches tant qu'il n'y a pas d'"incohérence logique". On considère qu'une situation est "logiquement incohérente" si, pour

7. Ce couplage est synchrone dans le sens où les conséquences de la réaction n sont effectives avant la réaction $n + 1$.

un `exec` donné, le programme synchrone émet un nouvel ordre vers la tâche alors que le précédent n'a toujours pas été acquitté. Comme dans le cas du couplage maximal, la machine se contentera de lever une exception qui pourra éventuellement être ignorée. À priori, on peut appliquer ce principe à toutes les primitives : `START`, `KILL`, `SUSPEND` et `RESUME`.
Nous choisissons, pour illustrer nos propos, le cas de la primitive `KILL` qui est souvent le plus critique. En effet, l'absence de garantie qu'une tâche a été arrêtée peut conduire à des situations critiques comme celle de la figure 5.36. L'émission du signal `KILL` ne garantit pas que la tâche a réellement été tuée. Pourtant la primitive `exec` correspondante est avortée et peut donc être réactivée à l'instant suivant. Le 2^{eme} signal `START` va entraîner le démarrage d'une nouvelle instance de tâche (*TASK2*). On se retrouve donc avec 2 instances de tâches pour un même `exec`, ce qui est incohérent avec la sémantique de la primitive.

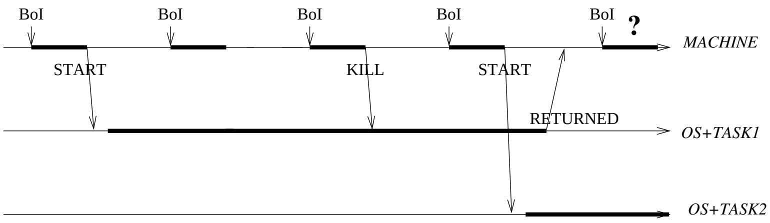


FIG. 5.36 - *Deux tâches actives pour un même* `exec`

Comme le montre la figure 5.37, un mécanisme d'acquittement de la primitive `KILL` permet de détecter une telle situation et de lever une exception.

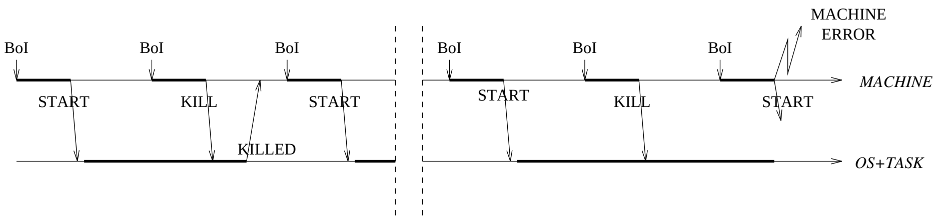


FIG. 5.37 - *Démarrage d'une tâche encore active : levée d'exception*

Contrairement au couplage maximal, dans cet l'exemple, l'exception n'est levée que si la demande de démarrage est émise alors qu'un arrêt n'a toujours pas été acquitté.
Si la machine se doit de détecter une telle situation, seul l'utilisateur est en mesure de définir le traitement adapté. On ne peut donc pas exclure la possibilité d'ignorer purement et simplement l'exception. Dans ce cas, les 2 tâches étant actives, la réception du signal de retour (`RETURNED`) devient ambigüe. Pour lever cette ambigüité, l'`execController` devra gérer un **indice** permettant de distinguer les différentes *incarnations* de la tâche et de ne considérer que la dernière en date (figure 5.38).

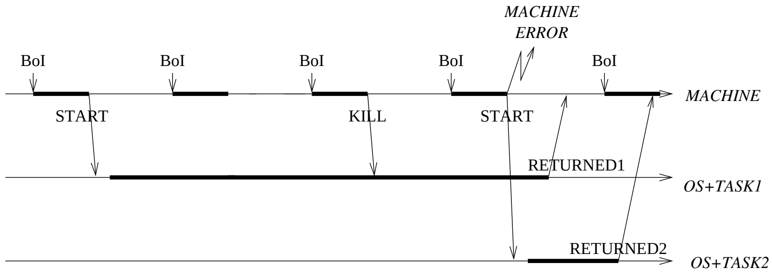


FIG. 5.38 - *Deux tâches actives pour un même `exec` : gestion d'un indice*

Cette solution peut paraître surprenante, mais c'est souvent une alternative préférable à un arrêt complet du système quand la primitive `KILL` ne réagit pas. Évidemment, cette solution ne peut être envisagée que si la continuation de la première tâche n'a pas de conséquences sur l'évolution du système : c'est peut être le cas pour une tâche de calcul, mais sûrement pas pour une tâche de déplacement d'un robot par exemple.

La sémantique "minimale" autorise un couplage plus lâche entre le processus synchrone et les tâches tout en garantissant le respect d'une certaine logique. Pourvu que les primitives de l'exécutif aient un délai "raisonnable", ce couplage est bien adapté aux applications non critiques où le non respect occasionnel des contraintes temporelles a des conséquences acceptables.
Dans un souci de clarté, nous omettrons dans la suite les signaux d'acquittement `STARTED`, `SUSPENDED` et `RESUMED` et ne traiterons que le cas de `KILLED`.

5.4.5 Intégration dans l'architecture de la machine

Vue externe : Conceptuellement, les signaux de gestion de tâches sont des signaux d'entrée/sortie de la machine comme les autres. Ils sont donc dotés de handlers d'E/S. La figure 5.39 représente l'interface d'une machine contenant 2 `exec`.

Comme tous les signaux d'entrée, `E1_RETURNED`, `E2_RETURNED`, `E1_KILLED` et `E2_KILLED` sont mémorisés par leurs handlers puis échantillonnés en début d'instant. Les primitives de manipulation des tâches (`E1_START`, `E2_START`,...) sont, elles, invoquées à la fin de l'instant.

Structure interne : On introduit dans l'architecture de la machine un gestionnaire d'`exec` (`execManager` ou EM) qui contient un contrôleur pour chaque `exec` du programme. Son activation provoque l'activation de tous les EC. Rappelons que le rôle de l'EM est de :

- Mettre à jour les signaux de retour des tâches `Ei.return` avant la réaction du module synchrone.
- Effectuer les opérations sur les tâches après la réaction du code synchrone.

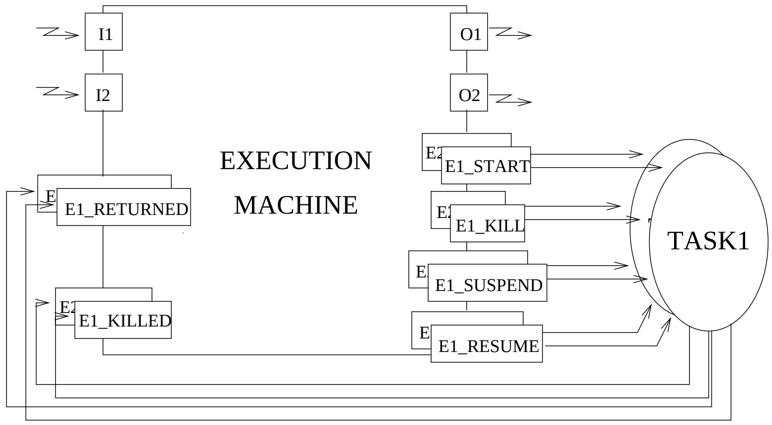


FIG. 5.39 - *Machine d'exécution avec 2 `exec`*

À priori, l'EM devrait donc s'exécuter *avant* et *après* chaque réaction du code synchrone.

5.4.5.1 Machine à activation périodique

On peut, dans le cas d'une machine à activation périodique, se dispenser d'une réaction de l'EM en ne le faisant réagir qu'*après* le module synchrone (figure 5.40). En effet, le seul inconvénient est que la terminaison d'une tâche est perçue par le

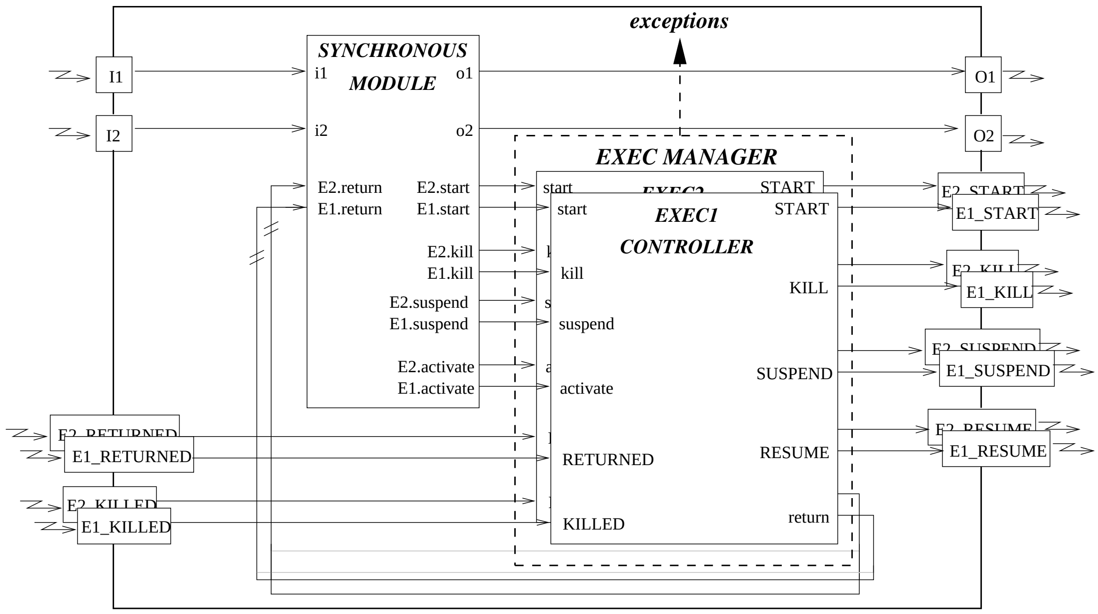


FIG. 5.40 - *Intégration d'un gestionnaire d'`exec` dans la machine*

code synchrone avec un retard d'un instant. D'une part, cette imprécision ne va pas à l'encontre de la sémantique ESTEREL des tâches, d'autre part, elle est tout à fait acceptable dans un contexte d'activation périodique. Les réactions ont lieu de la manière suivante :

1. Échantillonnage des signaux d'entrée, y-compris `RETURNED` et `KILLED`,

2. Réaction du code synchrone avec émission des signaux ESTEREL de gestion des tâches,

3. Réaction de l'EM avec émission des opérations vers les handlers de sortie et positionnement des signaux de retour pour l'instant suivant, si nécessaire.

4. Activation des handlers de sortie et invocation des primitives de l'exécutif.

Dynamique du contrôleur d'`exec` *minimal*

Pour gérer les tâches selon la sémantique choisie, les contrôleurs d'`exec` doivent avoir une double vision : ils doivent connaître à tout moment l'état de la tâche et l'état de l'`exec` associé. La difficulté consiste à maintenir cette double vision et détecter les situations nécessitant une intervention (opération sur la tâche ou levée d'exception).
Le couplage maximal est relativement simple à mettre en œuvre : toute opération doit être acquittée à l'instant suivant. Nous nous intéresserons donc au couplage minimal. Une fois de plus nous ne traiterons que les exceptions liées au retard d'acquittement de la primitive `KILL`. La figure 5.41 représente le comportement du contrôleur d'`exec` (EC).

L'EC peut être dans 5 états différents :

- *T_IDLE* : tâche inactive, `exec` inactif ou suspendu.
  C'est l'état initial. C'est aussi l'état dans lequel l'EC revient une fois qu'une tâche a terminé ou qu'elle a été tuée : si la tâche a terminé, le signal `return` a été positionné, l'`exec` est donc inactif à moins qu'il ne soit suspendu.

- *E_ACTIVE* : `exec` actif, tâche considérée comme active.
  C'est l'état dans lequel la tâche s'exécute. Avant d'entrer dans cet état la primitive `START` est invoquée.

- *E_SUSPENDED* : `exec` suspendu, tâche considérée comme suspendue.
  Avant d'entrer dans cet état la primitive `SUSPEND` est invoquée.

- *E_KILLED* : `exec` avorté.
  Il s'agit d'un état transitoire dans lequel une demande d'arrêt de la tâche a été émise et on en attend l'acquittement.

- *E_RESTARTED* : `exec` avorté et redémarré instantanément.
  Dans cet état transitoire, une demande d'arrêt de la tâche a été émise et on en attend l'acquittement pour démarrer la nouvelle instance.

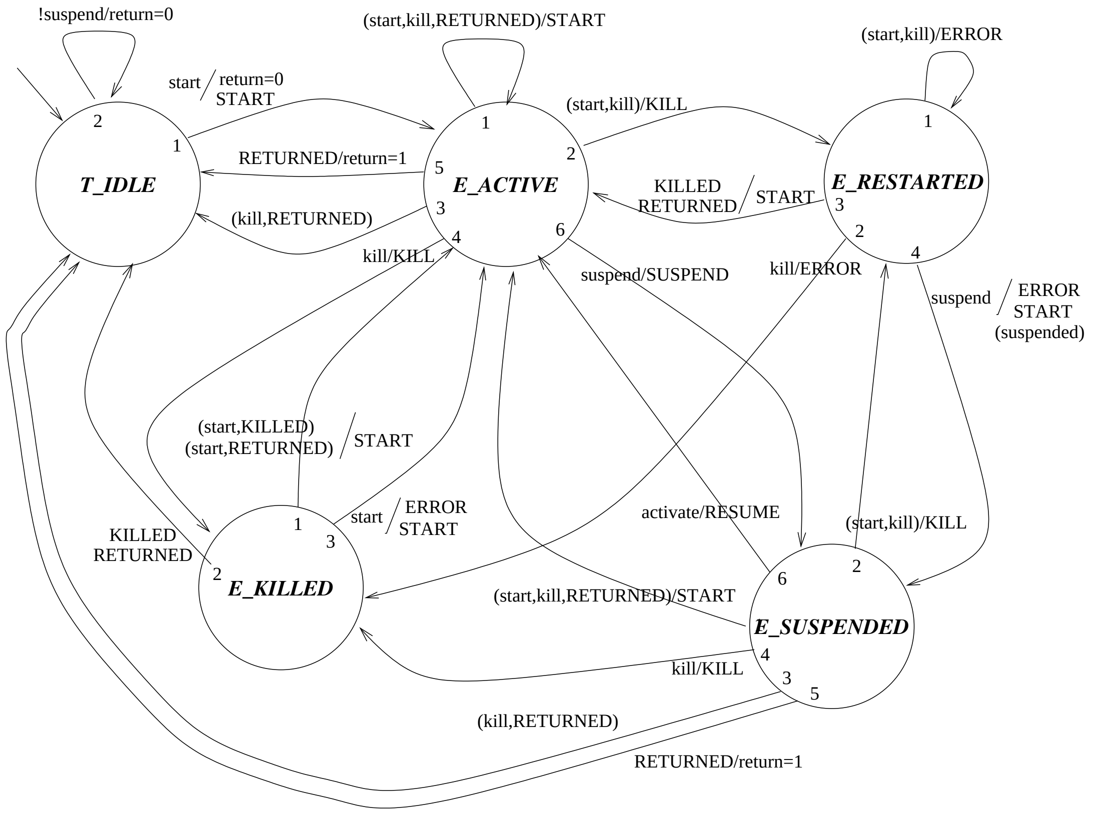


FIG. 5.41 - *Comportement de l'EC*

En plus de ces états, l'EC gère un indice permettant de distinguer entre les différentes incarnations d'un même `exec` en cas de besoin.

Scénario classique : L'EC est initialement dans l'état *T_IDLE*. L'activation de l'`exec` se traduit par le signal `start` qui fait passer l'EC dans l'état *E_ACTIVE* en invoquant la primitive de démarrage de tâche `START`. Quand la tâche termine (signal `RETURNED`), le signal de retour (`return`) est positionné à `présent` pour le module synchrone, et on revient à l'état initial.

Dans les paragraphes qui suivent, on illustre le comportement de la figure 5.41 en traitant les différents scénarios possibles. Ces paragraphes sont assez techniques et pourront être omis dans une première lecture.

Arrêt : Une tâche active (état *E_ACTIVE*) peut être tuée, on passe alors dans l'état transitoire *E_KILLED*, état dans lequel on se met en attente de l'acquittement `KILLED`. Si une demande de démarrage d'une nouvelle tâche (`start`) arrive alors, une exception est levée (`ERROR`) indiquant que la tâche courante n'a pas encore été tuée. Si aucun traitement particulier n'est prévu, le démarrage de la nouvelle tâche a lieu et l'indice de la tâche courante est mis à jour. On repasse alors dans l'état *E_ACTIVE*. Si le signal `start` est reçu dans le même instant que l'acquittement, la tâche peut être redémarrée (`START`) sans levée d'exception.

Redémarrage instantané : Une tâche active (état *E_ACTIVE*) peut faire l'objet d'une demande de redémarrage instantané (`start,kill`). Cette opération ne pouvant être instantanée dans le monde asynchrone, on demande à tuer la tâche (`KILL`), puis on passe dans l'état *E_RESTARTED* et on se met en attente de l'acquittement. Dans cet état, toute demande d'action sur la tâche déclenche une exception (`ERROR`). Si cette dernière est ignorée, le comportement est le suivant :

- S'il s'agit d'une nouvelle demande de redémarrage instantané, on se contentera de servir la précédente.

- S'il s'agit d'une demande d'arrêt simple (`kill`), comme on est déjà en attente d'un acquittement `KILLED`, on passe dans l'état *E_KILLED*.

- S'il s'agit d'une demande de suspension, la tâche précédente n'ayant toujours pas été arrêtée, on démarre une nouvelle tâche en mode suspendu (`START(suspended)`), on met à jour l'indice de la tâche courante et on passe dans l'état *E_SUSPENDED*.

Suspension : Une tâche active peut être suspendue (état *E_SUSPENDED*) puis réactivée par le signal `activate`. Toutefois, dans cet état, elle peut aussi être tuée ou redémarrée instantanément.

Asynchronisme : Dans les états *E_KILLED* et *E_RESTARTED*, une demande d'arrêt de la tâche (signal `KILL`) a été émise, mais non acquittée. Étant donné l'asynchronisme de l'exécutif, la tâche est peut-être encore active et elle peut donc terminer (signal `RETURNED`). Dans ce cas, la terminaison de la tâche (`RETURNED`) tient lieu d'acquittement.
De même, dans l'état *E_SUSPENDED*, une demande de suspension (signal `suspend`) a été émise, mais la tâche peut terminer (signal `RETURNED`) avant que la suspension n'ait été effectuée.

Terminaison et arrêt simultanés : Il se peut qu'une tâche termine à l'instant même où l'`exec` est avorté. Ceci se traduit pour l'EC par la réception de `kill` et `RETURNED` dans le même instant. On fait dans ce cas le choix de ne pas signaler le retour de la tâche au module synchrone . On garantit donc que si un `exec` a été avorté, le module synchrone ne peut plus recevoir le signal `return`.

Terminaison et suspension simultanées : (voir 5.4.2.2) En cas de suspension et de terminaison simultanées (`suspend, RETURNED`), étant données les priorités des transitions, l'EC passe dans l'état *T_IDLE* et `return` est positionné à présent. Dans cet état, le signal `return` demeure présent tant que l'`exec` est suspendu (`!suspend/return=0`).

Sur la figure 5.41, le seul état de l'EC pour lequel on est certain de l'état de la tâche est l'état *T_IDLE* dans lequel la tâche est inactive. Ceci vient du fait qu'on ne traite que l'acquittement du signal `KILL`. En traitant les opérations `START`, `SUSPEND` et `RESUME` de la même manière on obtient 3 états supplémentaires : *T_RUNNING*, *T_SUSPENDED* et *E_RESUMED*.

5.4.5.2 Machine déclenchée sur événements

Dans le cas d'une activation sur événements, la terminaison d'une tâche pouvant déclencher une réaction de la machine, il n'est pas question de différer sa prise en compte. L'EM doit donc réagir avant *et* après le module synchrone comme le montre la figure 5.42.

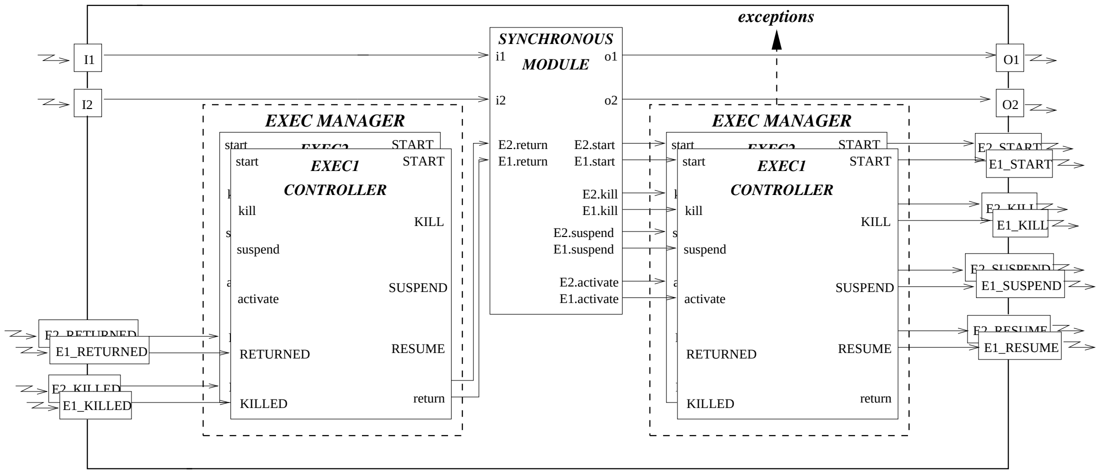


FIG. 5.42 - *Machine déclenchée sur événements*

- L'activation en début d'instant permet la prise en compte des signaux `RETURNED` et `KILLED` et éventuellement le positionnement du signal de retour pour le module synchrone.

- L'activation en fin d'instant permet la prise en compte des signaux du module synchrone et l'émission des actions vers les tâches.

Dynamique du contrôleur d'exec *minimal*

Ce paragraphe traitant de détails techniques, il pourra être omis dans une première lecture.
Supposons dans un premier temps que le comportement du contrôleur d'`exec` soit le même que dans le cas périodique (figure 5.41). Ce contrôleur est maintenant activé en début et en fin d'instant. Distinguons les transitions susceptibles d'être franchies en début d'instant de celles susceptibles d'être franchies en fin d'instant. Les premières sont celles provoquées par les signaux provenant des tâches : `KILLED` et `RETURNED`. Les deuxièmes sont celles provoquées par les signaux provenant du module synchrone : `start, kill, suspend` et `activate`. Quant aux transitions traitant les cas de simultanéité de signaux des 2 types, elles deviennent inutiles. Pour préserver le principe d'atomicité des réactions, il faut s'assurer qu'une tâche ne peut pas être mise à jour suite à une activation de l'EM en début d'instant. Il faut donc vérifier que toutes les transitions franchissables en début d'instant n'ont pas d'influence sur la tâche. C'est le cas pour toutes ces transitions, sauf une : dans l'état *E_RESTARTED*, à la réception de `KILLED` ou `RETURNED` (donc en début d'instant), une tâche est redémarrée (émission de `START`). Ceci est la seule situation où une action sur la tâche ne fait pas suite à une activation du module synchrone et peut donc avoir lieu en début d'instant. La figure 5.43

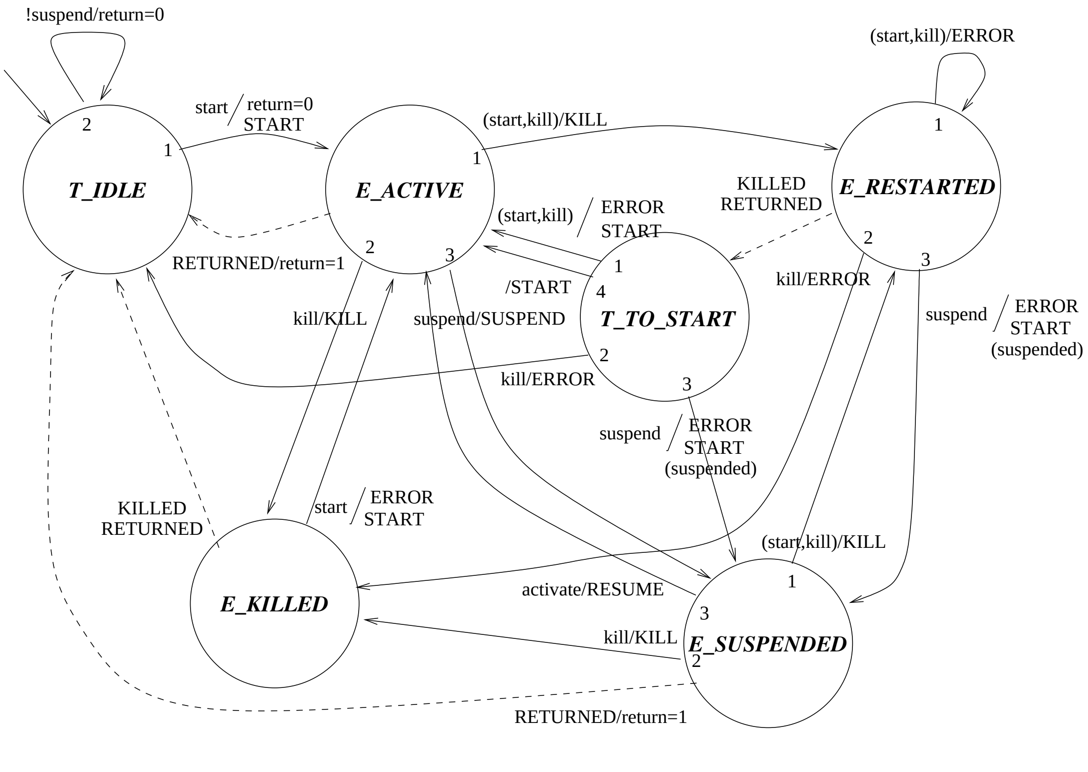


FIG. 5.43 - *Comportement de l'EC*

représente le comportement d'un contrôleur d'`exec` compatible avec un déclenchement sur événements. L'activation en début d'instant se fait selon les transitions pointillées. L'activation en fin d'instant se fait selon les transitions pleines. Le problème soulevé plus haut est résolu par l'introduction d'un état supplémentaire *T_TO_RESTART* qui permet de reporter le redémarrage de la tâche à la fin de l'instant.

On notera que la double activation de l'EM associée au comportement de la figure 5.43 est une solution qui a l'avantage d'être générale : elle s'applique aussi bien pour une activation périodique que pour une activation sur événements. Néanmoins, elle a été beaucoup moins utilisée (et donc moins testée).

5.4.6 Conclusion sur la gestion des tâches

Les tâches ne sont gérées par les programmes ESTEREL que d'un point de vue *logique*. La machine d'exécution fait appel aux services d'un exécutif multi-tâche pour leur exécution effective.
Contrairement à la vision idéale des tâches qu'ont les programmes synchrones, les primitives de manipulation fournies par l'exécutif ne sont pas instantanées, le cas le plus caractéristique étant celui de l'avortement/redémarrage instantané. Ce décalage peut conduire à des situations incohérentes voire dangeureuses.
Nous avons proposé 2 sémantiques qui donnent plus ou moins de garanties à l'utilisateur quant à l'état réel des tâches :

- La sémantique *maximale* pour un couplage maximal entre `exec` et tâches réelles,
- La sémantique *minimale* permet un couplage plus lâche tout en garantissant une certaine cohérence.

Dans tous les cas, la machine d'exécution ne fait que signaler les situations qu'elle juge dangereuses. L'utilisateur est libre de les ignorer s'il veut s'en tenir au strict respect de la sémantique de la primitive `exec`.
Cette gestion est intégrée dans l'architecture de machine d'exécution par l'intermédiaire de modules spécifiques (`execManager`) dont nous avons décrit le comportement.

5.5 Conclusion

Nous avons consacré ce chapitre à l'étude d'un modèle abstrait pour la description des machines d'exécution. Dans un premier temps, nous avons étudié les problèmes posés par l'exécution d'un exemple simple. Cet exemple nous a mené progressivement à une première architecture de machine. Nous avons ensuite introduit le modèle de boîtes réactives qui reprend l'architecture déjà définie en lui ajoutant un caractère récursif. Ce modèle permet de concevoir des machines élaborées, par composition de boîtes réactives de base. La dernière partie a porté sur la gestion des tâches asynchrones. Deux sémantiques ont été proposées pour leur exécution. La gestion de ces tâches a été intégrée dans l'architecture de la machine par l'introduction d'un gestionnaire de tâches.

Les machines d'exécution issues de cette approche ont toutes une architecture générique commune. Mais elles ont chacune des spécificités propres qui leur permettent de répondre aux besoins d'une application particulière. La prise en compte de ces spécificités se fait par le choix conjoint de contrôleurs d'E/S, d'observateurs et de séquenceurs particuliers. Cette approche générique permet de structurer les implémentations, ce qui constitue un préalable à la génération automatique. Des outils de production automatique ont été développés et seront présentés dans le chapitre 6.

Chapitre 6

Exemple de mise en œuvre et vérification de propriétés

Objectifs

- Illustrer sur un exemple concret la méthode de conception de machines d'exécution et présenter les outils qui ont été développés pour la génération automatique.
- Analyser les propriétés de la machine, c'est à dire celles qui portent sur les événements physiques, et les confronter aux propriétés du programme synchrone portant sur les signaux logiques. On verra ainsi dans quelle mesure les propriétés du modèle sont vérifiées par sa réalisation.

Contributions

- Spécification et conception de l'exemple de l'ATM en SyncCharts.
- Exemple d'utilisation des tâches Esterel.
- Vérification de propriétés de sûreté par la technique des observateurs.
- Outils de génération automatique de machines sur le système Solaris 2.
- Mise en évidence des problèmes dûs à la discrétisation du temps.
- Modélisation des distorsions introduites par la machine d'exécution.
- Prise en compte des distorsions et vérification des propriétés "de bout en bout".

6.1 Introduction

L'exemple traité dans ce chapitre est celui de l'ATM présenté en 2.3.2.
Dans un premier temps nous nous attachons à spécifier l'interface et le comportement des modules synchrones de l'application. Le module ATM est conçu en SyncCharts. Nous utilisons ensuite un outil de preuves formelles (Xeve) pour montrer que ce module vérifie certaines propriétés de sûreté. Dans une troisième partie, nous utilisons des outils de configuration et de génération automatique pour produire un code exécutable. Une première machine est générée pour la simulation, une deuxième, pour une exécution répartie avec communication entre banque et ATMs. Les propriétés vérifiées par le module ATM concernent les signaux logiques du module synchrone. Au cours de l'exécution, la correspondance signaux physiques/signaux logiques n'est bien sûr pas parfaite. Dans la dernière partie, on modélise les distorsions introduites par la machine afin de savoir dans quelle mesure les propriétés du modèle synchrone de l'ATM sont vérifiées à l'exécution.

6.2 Spécification et conception des modules `bank` et `ATM`

Nous abordons le problème de la même manière que ce qui a été fait pour les Processus Réactifs Communiquants [BRS93] : le système est conçu comme un ensemble de modules réactifs synchrones qui communiquent par l'intermédiaire d'un canal de communication asynchrone. Nous allons donc déterminer l'interface de ces modules avec leur environnement puis spécifier leur comportement dans un formalisme synchrone.

Que ce soit pour établir l'interface ou le comportement des modules bank et ATM, on procède à l'aide de scénarios : on envisage d'abord les scénarios "normaux" puis les cas exceptionnels. Les modules sont ainsi conçus par raffinement successifs.
L'utilisation de scénarios est préconisée dans des méthodologies comme Room ou Uml temps réel pour définir l'interface et le comportement des objets. Toutefois la comparaison avec notre approche s'arrête là. Dans ces méthodologies, un système réactif, même à forte cohésion interne, peut être décomposé en plusieurs objets communicants selon des protocoles variés. Dans notre approche, à partir du moment où l'hypothèse de diffusion instantanée est raisonnable (c'est le cas pour la plupart les systèmes centralisés), la composition synchrone est la solution préconisée.

6.2.1 Interface de l'ATM

Interface ATM-client : Le scénario suivant permet de recencer les événements échangés entre le client et l'ATM :

- Au repos, un ATM indique qu'il est prêt à recevoir une carte (événement *ATM ready*),
- Une fois qu'une carte a été insérée (*card insertion*), l'ATM demande au client d'entrer son code secret (*enter code*),
- Si le code saisi (*code:string*) est erroné, un message d'erreur est affiché (*erroneous code*),
- Quand le code correct a été saisi, l'ATM demande au client d'entrer la somme désirée (*enter amount*),
- La somme ayant été saisie (*amount:integer*), les billets peuvent être délivrés (*cash delivery*) ou la transaction peut être refusée (*transaction refused*).
- Si la carte n'a pas été confisquée (événement *card kept*, en cas de 3 erreurs de code par exemple), l'ATM demande au client de retirer sa carte (*take card*).
- Le retrait de la carte par le client correspond à l'événement *card taken*.
- Enfin, le client peut décider d'annuler la transaction à tout moment (*cancellation*).

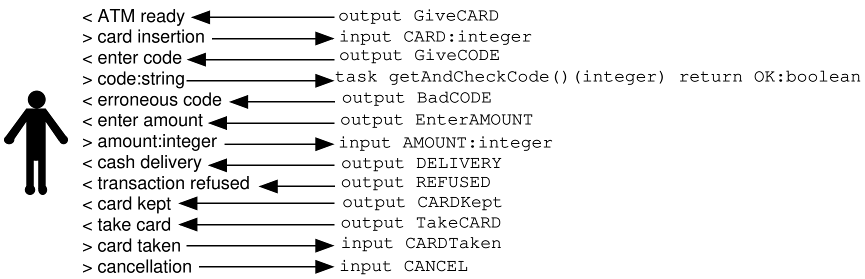


FIG. 6.1 - *Interface ATM-client*

On déduit facilement de ce dialogue l'interface du module `ATM.strl` côté client. La figure 6.1 est une synthèse des événements échangés. Tous ces événements sont portés par des signaux ESTEREL à l'exception du code secret. En effet, la vérification du code se fait en 2 temps : saisie puis cryptage et vérification. Le cryptage est un processus transformationnel complexe, nous en avons donc fait une tâche externe à ESTEREL (il s'agit en l'occurrence d'un script Perl qui applique l'algorithme de cryptage d'Unix). La saisie du code, elle, aurait pu être faite par un signal ESTEREL. Toutefois, pour des raisons de sécurité et afin de minimiser la manipulation du code non crypté, nous avons choisi de la placer dans la tâche de vérification. La tâche `getAndCheckCode` joue donc le rôle d'interface entre le module `ATM` et le client pour ce qui concerne le code. Ce qui est vu par le module `ATM` est le signal booléen de retour de la tâche dont l'occurrence signifie qu'un code a été saisi et dont la valeur indique la validité de ce code.

Interface ATM-Banque : L'ATM et la banque communiquent à travers un canal de communication bidirectionnel. L'interaction entre les deux se fait à 2 niveaux (figure 6.2) :

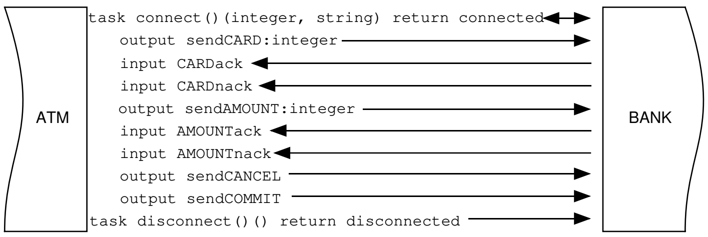


FIG. 6.2 - *Interface ATM-banque*

- Au niveau *application*, il s'agit des messages échangés entre la banque et l'ATM. Le dialogue est assez classique : l'ATM émet le signal `sendCARD` portant un numéro de carte, la banque répond soit par le signal `CARDack`, soit par le signal `CARDnack` indiquant la validité ou la non validité de la carte. L'ATM émet ensuite le signal `sendAMOUNT` portant la somme demandée par le client. Si la somme peut être retirée, la banque répond par `AMOUNTack`, sinon par `AMOUNTnack`. En fin de transaction, l'ATM émet soit une confirmation de transaction (`sendCOMMIT`), soit une demande d'annulation (`sendCANCEL`).

- Le deuxième niveau (le plus bas), concerne la mise en service de la communication entre les deux entités, à savoir, les procédures de connexion et de déconnexion. Nous verrons que ces procédures sont fournies par la machine d'exécution (voir 6.4.2). À ce niveau, il s'agit simplement de s'assurer qu'une connexion a été établie avant de commencer le dialogue et que cette connexion soit fermée à la fin du dialogue. Il est donc naturel de placer ces procédures dans des tâches externes, d'autant plus que la procédure de connexion est généralement de durée indéfinie. Il s'agit des tâches `connect` et `disconnect`.

6.2.2 Interface du module `bank`

La banque doit pouvoir communiquer et servir plusieurs clients simultanément. La figure 6.3 représente l'interface d'une banque pouvant servir 2 clients à la fois.

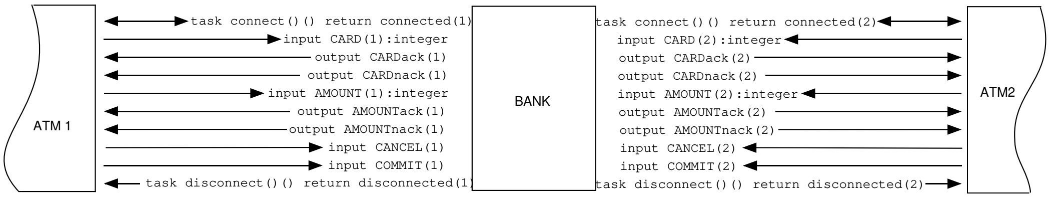


FIG. 6.3 - *Interface Banque-ATMs*

6.2.3 Module `ATM`

6.2.3.1 Comportement réactif

Le comportement de l'ATM est relativement évolué. Une première ébauche a été faite en ESTEREL. Les raffinements successifs ont rapidement mené à un programme difficilement lisible. Il nous semble en particulier, que la prise en compte, différente suivant le contexte, de situations exceptionnelles et leur traitement se prête beaucoup plus naturellement à une description en termes d'états et de transitions. Le module `ATM` a finalement été programmé en SYNCCHARTS.

La figure 6.4 représente une première version du module ATM. L'ATM commence par maintenir le signal `GiveCARD` jusqu'à ce qu'une carte ait été introduite (signal `CARD`). On entre alors dans le macro-état `checkCode` dans lequel la tâche `getAndCheckCode` est exécutée au plus 3 fois de suite. Une fois le code correct saisi, l'ATM entre dans le macro-état `transaction` dans lequel a lieu le dialogue avec la banque. Nous détaillerons ce macro-état dans un deuxième temps. Le dialogue se termine normalement par l'émission du signal `sendCOMMIT` vers la banque et la somme est délivrée dans l'état `delivery`. Enfin, le signal `TakeCARD` est maintenu tant que la carte n'a pas été récupérée par le client (signal `CARDTaken`).
Si le client entre 3 fois consécutives un code erroné, ou si la banque le demande au cours de la transaction, le signal local `keepcard` est émis et la carte est confisquée (macro-état `Keep`). Le client peut lui-même décider d'annuler sa demande (signal `CANCEL`). Si cette annulation a lieu en cours de transaction avec la banque, le signal `sendCANCEL` est émis vers la banque. Enfin, la transaction peut être refusée par la banque (`signal noMoney`).

La figure 6.5 représente le comportement complet de l'ATM. Les modifications apportées à la première version sont les suivantes :

– Connexion à la banque : les états `initConnect`, `connect`, `disconnect` et `disconnect1` ont été ajoutés pour gérer les connexions et les déconnexions.

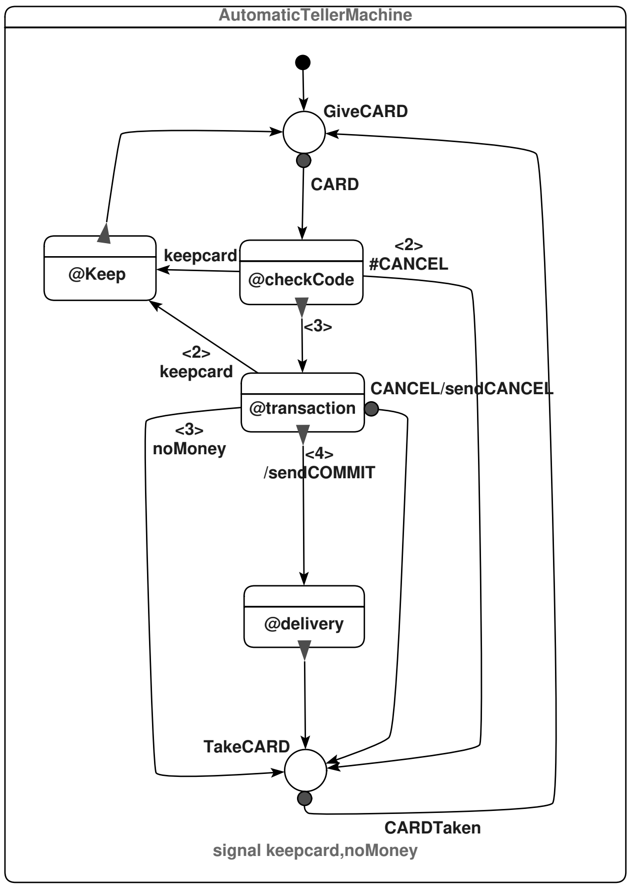


FIG. 6.4 - *Première version du module ATM*

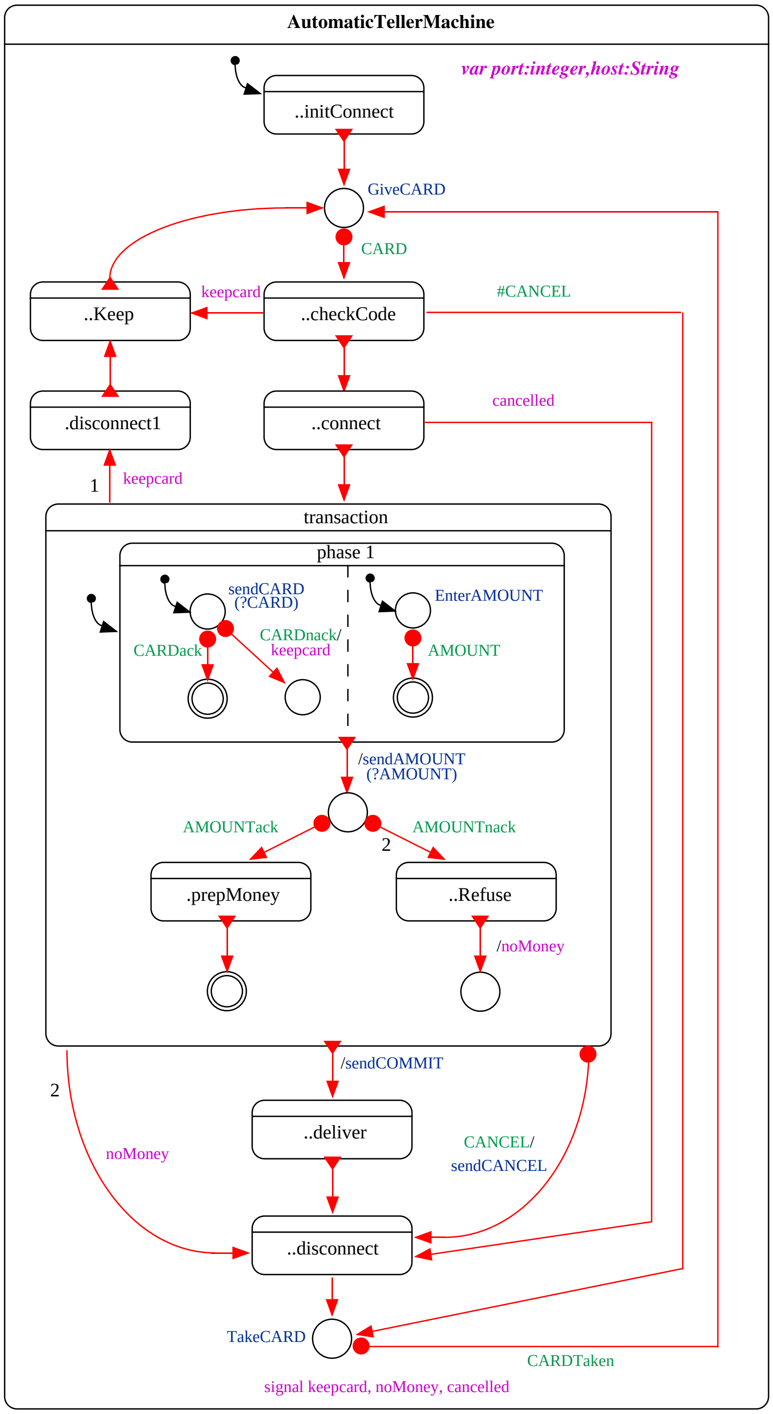


FIG. 6.5 - *SyncChart de l'ATM*

Ces états font appel aux tâches de connexion/déconnexion dont nous avons déjà parlé. Ces tâches ont un caractère critique et ne doivent pas être interrompues. Ainsi, dans l'état `connect`, une composante parallèle à l'exécution de la tâche de connexion se charge de mémoriser l'occurrence de `CANCEL` jusqu'à ce que la tâche ait terminé.

- Le macro-état `transaction` a été raffiné : une transaction se déroule en 2 phases :
  - Dans un premier temps, 2 activités ont lieu en parallèle :
    - Saisie du montant par le client
    - Émission du numéro de carte vers la banque et attente d'un acquittement. Si l'acquittement est négatif, le signal `keepcard` est émis, entrainant la confiscation de la carte.
  - Si ces 2 activités terminent normalement, on sort du macro-état `phase1` en émettant la somme demandée vers la banque (signal `sendAMOUNT`). Si cette dernière répond favorablement (`AMOUNTack`), une tâche simulant le comptage des billets est exécutée (état `prepMoney`), l'état `transaction` termine normalement et la somme est délivrée. Sinon (`AMOUNTnack`), l'ATM indique qu'il refuse la transaction (état `Refuse`) et invoque la procédure de déconnexion (émission du signal local `noMoney`).

6.2.3.2 Tâches asynchrones

Nous rappelons ici brièvement les tâches utilisées par le module `ATM`.

- `initATM(integer, String)()` : initialisation des paramètres réseau permettant la connexion à la banque. Ces paramètres sont ensuite utilisés par la tâche de connexion.
- `getAndCheckCode()(integer)` : saisie, cryptage et vérification du mot de passe.
- `prepareMoney()()` : simulation du comptage des billets qui prend généralement du temps.
- `connect()(integer, String)` et `disconnect()()` : tâches de connexion/-déconnexion fournies par la machine d'exécution. Elles seront traitées en 6.4.2.

6.2.4 Module `bank`

6.2.4.1 Comportement réactif

Le module `bank` a été écrit en Esterel. Il est constitué de la composition parallèle de modules identiques qui jouent le rôle de serveurs : chaque module `server` sert les requêtes d'un ATM. Par rapport à l'interface annoncée en 6.2.2, nous avons ajouté un signal de sortie supplémentaire : le signal `ACCOUNT:String` qui porte une chaîne de caractères représentant l'état du compte en cours de transaction. Ce signal permet à un superviseur de suivre le déroulement des transactions.

6.2.4.2 Tâches asynchrones

Outre les tâches de connexion et de déconnexion, le module `server` fait appel à des tâches de gestion des comptes des clients. Deux critères principaux ont été déterminants pour l'écriture de ces tâches :

- Accès concurrents : plusieurs serveurs communiquant chacun avec un ATM s'exécutent en parallèle. L'accès aux comptes des clients se fait donc de manière concurrente. L'utilisation de sémaphores d'exclusion mutuelle permet de garantir un accès exclusif à un compte donné par chaque tâche[1].

- Annulation de transaction : entre le moment où le client entre la somme demandée et la fourniture des billets, le client doit pouvoir annuler la transaction. Un protocole de transaction en 2 phases (*prepare, commit*) permet de répondre à cette exigence.

Les tâches de gestion des comptes sont les suivantes :

- `checkCard` : vérifie la validité d'une carte. Le signal de retour indique si la la carte est répertoriée dans la base de données.

- `prepareTransaction` : tente de réserver une somme sur un compte. Le signal de retour indique si la somme a pu être réservée.

- `commitTransaction` : rend effective une transaction préalablement préparée par la tâche `prepareTransaction`.

- `giveUpTransaction` : annule une transaction préalablement préparée par `prepareTransaction`.

6.3 Vérification de propriétés du module `ATM`

L'objectif est ici de vérifier que le comportement du module synchrone `ATM` que nous avons conçu obéit à certaines règles de sûreté. Nous utilisons pour celà l'outil Xeve.

1. En effet, certaines banques permettent à des conjoints par exemple, d'accéder au même compte à l'aide de 2 cartes différentes.

6.3.1 L'outil Xeve

XEVE [Bou97] est un environnement de vérification pour programmes ESTEREL qui utilise une représentation sous forme de machines à états finis. Il permet de :

- minimiser la machine à états finis sous-jacente,
- tester le *statut* des signaux de sortie par analyse des états accessibles.

C'est cette dernière fonction qui a été utilisée. On peut vérifier qu'un signal de sortie est :

- soit *potentiellement émis* : il existe un état accessible dans lequel le signal est présent,
- soit *potentiellement non émis* : il existe un état accessible dans lequel le signal est absent.

L'examen du statut d'un signal de sortie peut ainsi être utilisé pour vérifier des **propriétés de sûreté** du type *"une mauvaise chose n'arrive jamais"*[2].
Le principe de vérification consiste à placer en parallèle au programme à vérifier, un **observateur** [Hal93] qui suit son évolution et émet un signal en cas de violation d'une propriété. Après analyse par XEVE, si ce signal n'est pas *"potentiellement émis"*, la propriété est vérifiée. Dans le cas contraire, XEVE fournit l'histoire d'entrée qui a mené à l'émission du signal. Cette histoire d'entrée est d'ailleurs sauvegardée au format `csimul` et peut être "rejouée" par le simulateur XES. Avec la technique des observateurs, les propriétés sont exprimées dans le même formalisme que le programme à vérifier.
Le format d'entrée de XEVE est le format `blif` issu de la chaîne de compilation ESTEREL.

6.3.2 Passage de ATM à pure-ATM

6.3.2.1 Esterel pur

À l'heure actuelle, nous ne disposons pas d'outils assez puissants pour vérifier le comportement de programmes ESTEREL valués : XEVE ne traite que les programmes ESTEREL purs. Nous devons donc dans un premier temps extraire du module ATM la partie purement réactive.
Les valeurs portées par les signaux valués étant considérées comme des variables, les modifications à effectuer pour passer d'un programme valué à un programme pur peuvent être guidées par les règles suivantes :

1. Éliminer les variables qui ne sont pas déterminantes pour le comportement réactif du programme : ce sont celles dont les valeurs n'influent pas de ma-

2. Les propriétés de vivacité sont plus compliquées à vérifier et ne sont pas traitées par XEVE.

nière directe sur le contrôle. Ces variables peuvent être détectées par une analyse syntaxique.

2. Remplacer chaque variable déterminante par des signaux représentant les différents sous-domaines déterminants de valeurs.

3. Remplacer les tests de valeurs par des tests de présence sur ces nouveaux signaux.

4. Éliminer les procédures et les fonctions.

5. Remplacer les tâches par des tâches pures (pas d'arguments et signal de retour pur).

L'automatisation de ces règles sort du cadre de nos travaux.

Le processeur `sscblif` considère qu'un programme contenant une tâche, même pure, n'est pas pur. Ceci est une faiblesse du processeur actuel que nous avons contournée en remplaçant chaque `exec` par une modélisation adéquate. L'instruction `exec` étant une primitive du langage ESTEREL, elle ne peut pas être modélisée de manière systématique : comme le montre l'exemple suivant, un simple `emit start; await return` ne suffit pas en cas de préemption :

```
weak abort                              weak abort
  loop                                    loop
    exec T()() return R;                    emit START_T; await R;
    present STOP then emit EXIT end         present STOP then emit EXIT end
  end loop                                end loop
when EXIT                               when EXIT
```

Si `R` et `STOP` sont présents au même instant, les deux modules terminent, mais dans le deuxième cas le signal `START_T` représentant le démarrage d'une nouvelle tâche est émis. Il faut donc trouver une modélisation adaptée à chaque situation rencontrée.

6.3.2.2 Application à l'ATM

Nous illustrons ici l'application des règles énoncées au paragraphe précédent à l'exemple de l'ATM afin d'en extraire la partie purement réactive (figure 6.5) :

Les variables `port` et `host` utilisées par les tâches de connexion peuvent être ignorées; elles n'ont pas d'incidence sur le contrôle.

Les valeurs portées par les signaux `CARD`, `AMOUNT`, `sendCARD` et `sendAMOUNT` n'ont pas d'influence directe sur le contrôle. Ces signaux sont donc remplacés par des signaux purs.

Les tâches d'initialisation, de connexion, de déconnexion et de préparation des billets, contenues dans les macro-états `initConnect`, `connect`, `disconnect`,

`disconnect1` et `prepareMoney` ont toutes des signaux de retour purs et des arguments non déterminants. D'autre part, elles ne sont jamais préemptées. Elles seront donc modélisées par un macro-état du type de la figure 6.6.

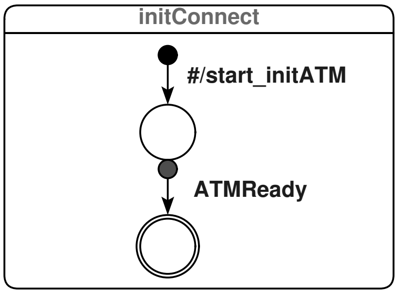


FIG. 6.6 - *Modélisation de la tâche* `initConnect`

La tâche `getAndCheckCode` (macro-état `checkCode`) ne peut pas être modélisée aussi simplement pour 2 raisons :

- le retour de la tâche est un booléen indiquant la validité du code saisi. Sa valeur est déterminante pour la suite,
- la tâche peut être avortée en cas d'annulation par l'utilisateur (signal `CANCEL`).

On introduit 2 signaux `CODEright et CODEwrong` et on modélise l'`exec` par la séquence suivante :

```
emit start_getAndCheckCode; await [CODEright or CODEwrong]
```

On ajoute la contrainte d'incompatibilité `CODEright # CODEwrong` pour indiquer au compilateur que ces 2 signaux ne peuvent être présents au même instant. Comme cette tâche peut être préemptée par le signal `CANCEL`, cette modélisation n'est pas suffisante. En effet, comme on l'a vu, avec cette modélisation, l'émission de `start_getAndCheckCode`, ne signifie pas forcément qu'une nouvelle instance est démarrée. La condition de démarrage est en fait `[start_getAndCheckCode and not CANCEL]`. Les observateurs s'aideront de cette modélisation et du signal `CANCEL` pour déduire l'état réel de la tâche (voir 6.5.3).
Une modification supplémentaire consiste à éliminer les états `Keep, Refuse` et `Deliver`. Leur rôle était de maintenir un signal un nombre suffisant d'instants pour qu'il soit visible à l'exécution. Ces trois états sont remplacés par l'émission respective des signaux `CARDKept, REFUSED` et `DELIVERY`. Le syncChart obtenu suite à ces modifications (pure-ATM) est représenté figure 6.7.

6.3.3 Propriétés vérifiées

Les propriétés que nous allons vérifier, peuvent, par leur expression informelle en français, apparaître comme étant des propriétés de vivacité. Il s'agit en fait

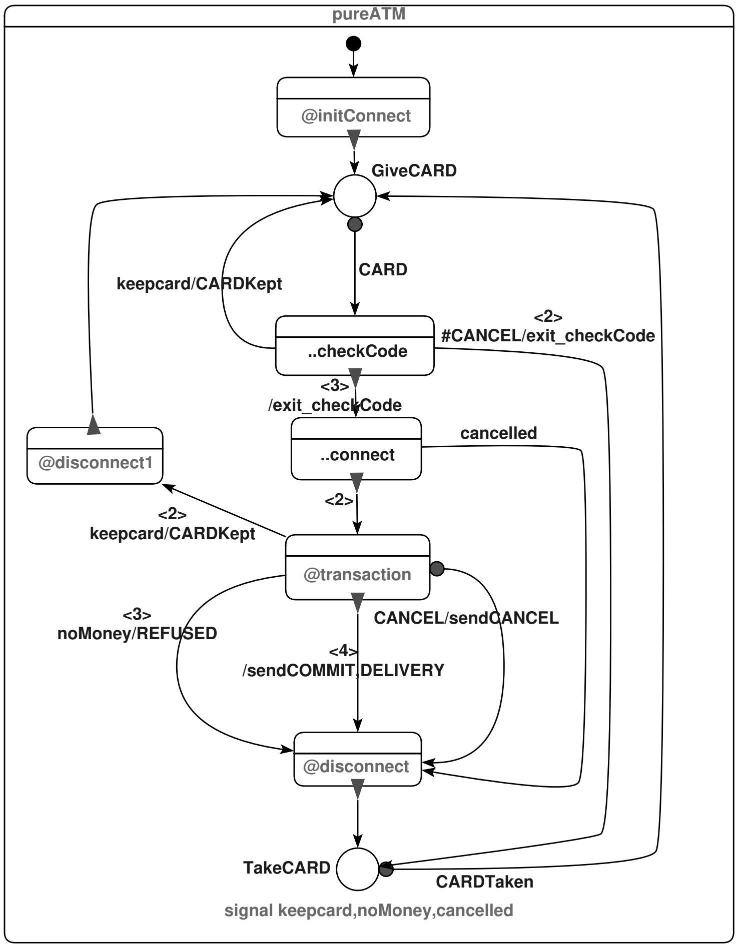


FIG. 6.7 - SYNCCHARTS *de pure-ATM*

de propriétés de “vivacité bornée” pour lesquelles “une bonne chose doit arriver à temps”. Ces propriétés sont équivalentes, du point de vue des techniques de vérification, à des propriétés de sûreté.

6.3.3.1 Saisie du code (P1)

Une première propriété à vérifier est que si le client entre 3 fois un code erroné, la carte est confisquée. Les signaux qui interviennent dans la vérification de cette propriété sont : `GiveCODE` (invitation à la saisie), `CODEright`, `CODEwrong` et `CANCEL` (annulation de la saisie par le client). Informellement, la propriété s'exprime de la manière suivante : “Toute trace commençant par `GiveCODE`, contenant ensuite 3 fois `CODEwrong` sans aucun `CODEright` ou `CANCEL` doit contenir une occurrence de `CARDKept` simultanée avec le dernier `CODEwrong`”. On peut formaliser la propriété à l'aide d'une expression régulière à la Auto. On donne alors l'expression inverse (celle qui mène à la violation de la propriété) :

$$\begin{aligned} \textit{violation-expr} = \textit{true}^* &: (GiveCODE \textit{ and not } CANCEL) \\ &: [(not(CODEright \textit{ or } CANCEL \textit{ or } CODEwrong))^* \\ &\quad : (CODEwrong \textit{ and } not(CODEright \textit{ or } CANCEL))]^2 \\ &: (not(CODEright \textit{ or } CANCEL \textit{ or } CODEwrong))^* \\ &: (CODEwrong \textit{ and not } CARDKept) \end{aligned}$$

L'observateur correspondant émet le signal `three_CODEwrong_without_CARDKept` si la propriété est violée :

```
loop
  await [GiveCODE and not CANCEL];
  weak abort
    await CODEwrong;
    await CODEwrong;
    await CODEwrong;
    present not CARDKept then
      emit three_CODEwrong_without_CARDKept
    end present
  when [CODEright or CANCEL]
end loop
```

L'utilisation d'une préemption faible (`weak abort`) permet de garantir que la carte est confisquée même si la 3^{eme} occurrence de `CODEwrong` est simultanée avec le signal `CANCEL`. L'analyse par Xeve de la machine d'états résultante indique que la propriété est toujours vérifiée.
On notera sur cet exemple, que la formulation d'une propriété à l'aide du langage Esterel peut être plus lisible que celle s'appuyant sur les expressions régulières.

6.3.3.2 Annulation de transaction (P2)

Une autre propriété essentielle est qu'aucune somme ne doit être délivrée si le client décide d'annuler la transaction : on ne peut pas avoir de séquence du type :

$$CANCEL : (not\ DELIVERY)^* : DELIVERY$$

ni du type :

$$(CANCEL\ and\ DELIVERY)$$

qui ne contienne au moins une occurrence du signal `CARD` (ce signal indique qu'une nouvelle carte a été introduite entre temps). L'observateur correspondant est :

```
loop
  weak abort
    await immediate CANCEL;
    await immediate DELIVERY;
    % shouldn't reach this point
    emit DELIVERY_after_CANCEL;
    await tick
  when CARD % a new card
end loop
```

L'instruction `await tick` permet d'éliminer la boucle instantanée qui empêche la compilation. XEVE indique que la propriété est toujours vérifiée.

6.3.3.3 Cohérence ATM-banque (P3, P4)

L'ATM et la banque étant distants, la cohérence entre les 2 sites est une propriété fondamentale. Une transaction commencée par l'envoi vers la banque du signal `sendCARD` (figure 6.5) peut se terminer

- soit à l'initiative de la banque : signal `CARDnack` ou `AMOUNTnack`,
- soit à l'initiative de l'ATM. Dans ce cas, il y a 2 possibilités :
  1. L'ATM délivre au client la somme demandée (signal `DELIVERY`). Il faut alors vérifier qu'il y a bien émission du signal `sendCOMMIT` vers la banque.
  2. Le client annule la transaction (signal `CANCEL`), il faut alors vérifier que le signal `sendCANCEL` notifiant l'annulation est bien émis vers la banque.

Pour la première propriété (P3), il suffit de vérifier que les signaux `DELIVERY` et `sendCOMMIT` sont synonymes. Pour la deuxième propriété (P4), il faut vérifier que si une annulation survient en cours de transaction, le signal `sendCANCEL` est émis. Autrement dit, toute séquence du type :

$$sendCARD : (not\ start\_disconnect\ and\ not\ CANCEL)^* : CANCEL$$

ou du type :

$$(sendCARD \text{ and not } start\_disconnect \text{ and } CANCEL)$$

doit contenir une occurrence de `sendCANCEL` au dernier instant. C'est ce qu'exprime l'observateur suivant :

```
loop
  weak abort
    await immediate sendCARD; % start of transaction
    loop
      present [CANCEL and not sendCANCEL] then
        emit CANCEL_without_sendCANCEL
      end present
    each tick
  when start_disconnect % transaction completed
end loop
```

Ces deux propriétés sont vérifiées par Xeve.

6.4 Machine d'exécution

Dans un premier temps, nous utilisons l'environnement de machines d'exécution pour simuler individuellement chacun des modules `ATM` et `bank`. Nous allons reconfigurer ensuite les 2 machines pour obtenir une exécution répartie avec communication réelle entre les 2 modules.

6.4.1 Simulation individuelle et mise au point des 2 modules

6.4.1.1 Environnement de machines d'exécution

L'environnement de machines d'exécution fournit :

- Une bibliothèque de machines d'exécution `libmex` qui contient l'architecture générique des machines sous la forme d'un ensemble de classes `C++` abstraites. Cette bibliothèque repose sur la bibliothèque synchrone `libSync` de F. Boulanger.
- Une bibliothèque d'entrée/sortie fournissant des classes de contrôleurs permettant divers types de connexion entre signaux Esterel et environnement.
- Un générateur `xmdf2c++` permettant de générer le code exécutable d'une machine à partir d'une description textuelle au format `xmdl` (eXecution Machine Description Language).

Les bibliothèques développées font appel à un *exécutif virtuel* : il s'agit d'une couche logicielle regroupant l'ensemble des services que doit fournir l'exécutif sous-jacent (threads, synchronisations, entrées/sortie…). L'environnement actuel a été développé pour le système Solaris2.

Nous avons choisi `C++` comme langage de développement pour plusieurs raisons :

- `C++` est un langage à objets *multiparadigme*. Classes, héritage, polymorphisme, surcharge de fonctions, modules, programmation procédurale et généricité sont autant de possibilités qui facilitent le travail de programmation. De plus, son style impératif est bien adapté aux systèmes réactifs, en l'occurrence la machine d'exécution.

- Comme `C`, `C++` permet un accès aisé aux services du système. Ceci en fait un langage de choix pour programmer les couches basses de la machine d'exécution (exécutif virtuel).

- La bibliothèque d'objets synchrones de F. Boulanger est elle même écrite en `C++`.

- `C++` est un langage compilé donc efficace à l'exécution.

- `C++` est fortement typé ce qui permet de détecter bon nombre d'erreurs à la compilation.

Toutefois les garanties apportées par le langage `C++` ne sont pas du même ordre que celles apportées par les langages synchrones : les parties écrites en `C++` n'ont pas fait l'objet de preuves formelles. En ce sens, seul le *corps* des objets synchrones est sûr.

6.4.1.2 Compilation des modules Esterel

Le code des modules Esterel doit être compilé en objets synchrones. La première étape consiste donc à générer les fichiers `ATM.H`, `ATM.C`, `bank.H` et `bank.C` qui contiennent le code des classes `ATM` et `bank`. Cette génération se fait en invoquant le processeur `occ++` en option du compilateur Esterel :

```
esterel -Sc++ bank.strl; esterel -Sc++ ATM.strl.
```

6.4.1.3 Configuration de la machine

La configuration de la machine consiste à :

- Connecter les signaux d'interface des modules Esterel à des contrôleurs d'E/S,

– Faire correspondre à chaque tâche ESTEREL son corps sous la forme d'une fonction C,

– Déterminer une stratégie d'activation.

Une classe de contrôleurs d'E/S pour la simulation a été développée : la classe `TKHandler`. La connexion d'un signal d'entrée à un `TKHandler` d'entrée permet de mettre à jour ce signal en cours d'exécution par l'intermédiaire d'une interface graphique. Pour un signal de sortie, le `TKHandler` permet de visualiser la valeur du signal en cours d'exécution.
L'exécution des tâches en parallèle avec la machine est gérée par un objet `execManager`.
La stratégie d'activation par défaut est périodique.
La figure 6.8 représente la structure de la machine d'exécution de l'ATM pour la simulation. Elle est composée d'un module d'entrée (`ATMinModule`), d'un module de sortie (`ATMoutModule`), d'un contrôleur d'`exec` (`execManager`) et du module de l'application `ATM`. La description de cette machine se fait par l'écriture du fichier `ATM.xmdf`.

6.4.1.4 Code généré

Le générateur `xmdf2c++` engendre 5 fichiers `C++` :

– `ATMExec.H, ATMExec.C` : ces fichiers contiennent la spécification et l'implémentation de la structure de la machine sous la forme d'une classe `ATMExecMachine`.

– `ATMTasks.H, ATMTasks.C` : ces fichiers contiennent des classes des tâches. Ces classes font l'interface entre le code `C` ou `C++` fourni par l'utilisateur, la tâche ESTEREL correspondante et les services de l'exécutif. Ces classes se chargent de :

  – L'exécution de la tâche en parallèle avec le module synchrone,

  – Le passage des paramètres par valeur et par référence,

  – La gestion du signal de retour.

– `ATMX.C` : programme principal.

6.4.1.5 Code fourni par l'utilisateur : fichiers `ATMDef.H, ATMDef.C`

Ces fichiers doivent contenir :

– La définition des types externes à ESTEREL sous forme de classes,

– Le corps des tâches sous formes de fonctions `C` ou `C++`.

Dans un premier temps, pour une simulation individuelle des 2 modules, le corps des tâches de connexion déconnexion resteront vides.

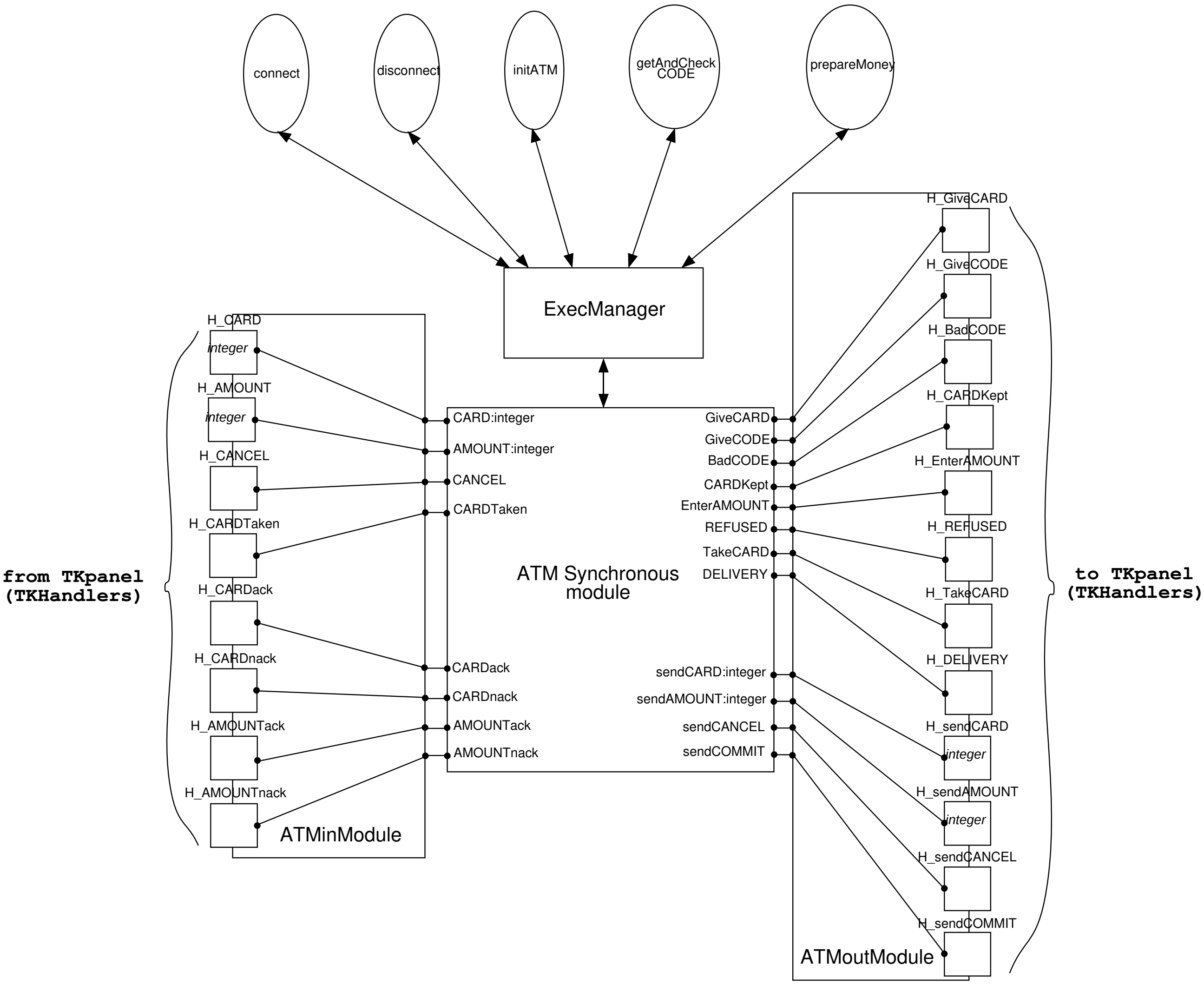


FIG. 6.8 - *Machine d'exécution en simulation du module ATM*

6.4.2 Connexion des 2 modules et simulation répartie

L'objectif est ici de réaliser des machines d'exécution pour les modules `bank` et `ATM` qui permettent une communication réelle entre la banque et plusieurs ATMs. Nous proposons d'utiliser les **sockets** d'Unix pour implémenter la communication.

6.4.2.1 Contrôleurs d'E/S sur Sockets

Des classes ont été développées pour les besoins de l'application. Elles sont maintenant disponibles en bibliothèque et peuvent être utilisées pour d'autres applications.

Classe `SKHandler`: cette classe permet la "connexion" d'un signal Esterel valué à un socket Unix. Le fonctionnement de ces contrôleurs est le suivant :

- En sortie d'un module Esterel : l'émission du signal, provoque l'écriture sur le socket de la valeur du signal dans un format textuel. Comme pour les `TKHandlers`, l'opérateur `C++` d'insertion dans un flux de sortie (<<) doit être défini pour ce type.
- En entrée du module Esterel : la réception par un contrôleur d'entrée d'une donnée sur le socket, se traduit par la mise-à-jour du signal Esterel auquel il est connecté. L'opérateur `C++` d'extraction d'un flot de sortie (>>) doit être défini pour le type de ce signal.

Classes de sockets Les contrôleurs possèdent toutes les méthodes nécessaires à la lecture ou à l'écriture sur un socket, mais un socket est une entité indépendante qui doit être créée explicitement.
La bibliothèque fournit les classes de sockets suivantes :

- `serverSocket` : le constructeur prend comme argument le numéro de port du serveur créé. La méthode `connect` prend comme argument un `SKHandler`, se met en attente d'une connexion côté client et l'associe au handler spécifié. La méthode `disconnect` ferme la connexion avec le client et déconnecte le handler.
- `clientSocket` : cet objet implémente les sockets côté client et possède les mêmes méthodes que pour le serveur.

6.4.2.2 Reconfiguration de la machine

La reconfiguration consiste pour le module `ATM` à :

- Associer aux signaux de communication avec la banque, non plus des `TKHandler` mais des `SKHandler`,

- Écrire le corps des tâches de connexion et de déconnexion en utilisant les objets `clientSocket` et `serverSocket` de la bibliothèque.

Les signaux échangés entre la banque et l'ATM sont : `sendCARD, sendAMOUNT, sendCANCEL`, `sendCOMMIT` et `CARDack, CARDnack, AMOUNTack, AMOUNTnack`. Dans le cas général, on peut envisager de créer un socket par paire de signaux d'entrée/sortie ou même un socket par signal. Dans notre cas, les signaux proviennent et sont émis vers une même entité, à savoir, la machine d'exécution de la banque. Le multiplexage de l'ensemble des signaux de communication sur un socket unique est donc préférable. En contrepartie, l'utilisateur doit fournir les procédures de multiplexage et de démutiplexage. La génération automatique de ces procédures est envisageable mais n'a pas été notre priorité.
La figure 6.9 représente la machine d'exécution de l'ATM après reconfiguration.

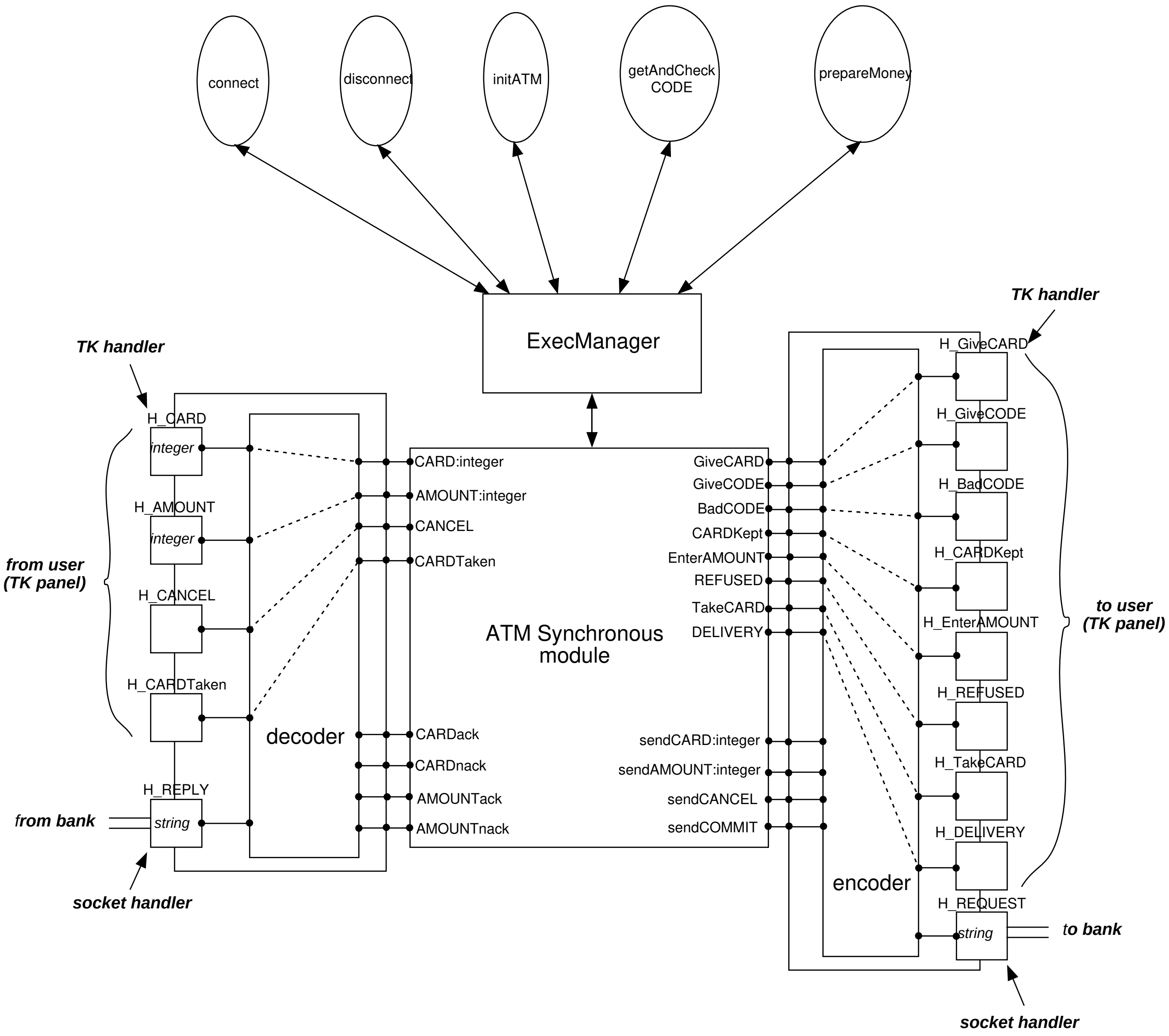


FIG. 6.9 - *Machine d'exécution du module ATM*

6.5 Analyse des propriétés de la machine d'exécution

6.5.1 Problèmes d'asynchronisme

Les propriétés qui ont été vérifiées en section 6.3 portent sur les signaux logiques du module ATM. À l'exécution, la correspondance entre les signaux logiques du module et les signaux réels ne peut pas être parfaite. La figure 6.10

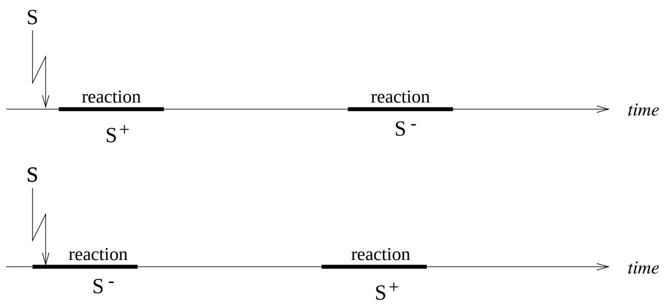


FIG. 6.10 - *Influence des instants d'activation sur la prise en compte des entrées*

représente les réactions d'un module synchrone : ces réactions se font de manière **atomique** et les occurrences de signaux au cours d'une réaction sont reportées à la réaction suivante[3].

La conséquence immédiate est un retard sur la prise en compte des signaux. Définissons l'horloge H comme étant l'ensemble discret des dates d'activation du module : $H = \{t_0, t_1, \ldots, t_n, \ldots\},\ t_i \in \mathbb{R}$.
Soit A l'ensemble des instants d'activation du module : $A = \{n \in \mathbb{N} \mid t_n \in H\}$.
Le retard sur la prise en compte d'une occurrence S d'un signal est :

$$R(S) = t_i - t(S) \quad \text{avec} \quad t_i = min\{t_k \in H \mid t_k \underset{\mathbb{R}}{>} t(S)\}$$
$$\text{et} \quad t(S) \text{ la date de l'occurrence du signal S}$$

R dépend donc de l'horloge H du module. Si le temps entre 2 activations consécutives du module est borné alors le retard est lui aussi borné :

$$\forall i \in A,\ t_{i+1} - t_i \underset{\mathbb{R}}{<} M \Rightarrow \forall S\ R(S) \underset{\mathbb{R}}{<} M$$

En pratique, si cette borne est négligeable par rapport à la constante de temps de l'environnement à contrôler, le retard est acceptable.
En revanche, on voit sur la figure 6.10 que le retard dépend du choix des instants d'activation de la machine : quand une occurrence de S a lieu au voisinage d'un début de transition, selon qu'elle a lieu avant ou après le début de la transition,

3. S^+ (S^-, resp.) indique que le signal est présent (absent, resp.) dans l'instant.

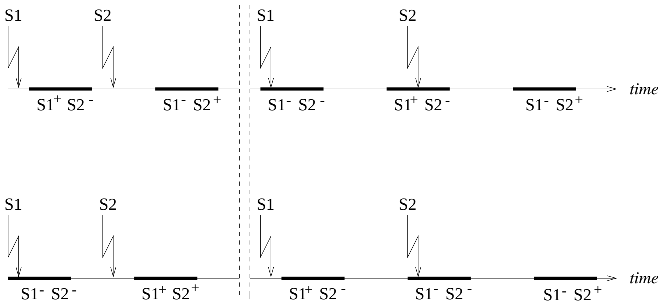


FIG. 6.11 - *Un même scénario peut engendrer 2 histoires d'entrée différentes*

elle peut être prise en compte à 2 instants différents.
Si l'on considère 2 signaux S_1 et S_2, on s'aperçoit que les conséquences peuvent être d'ordre **logique** : la figure 6.11 illustre le fait qu'un même scénario de signaux réels peut engendrer 2 histoires d'entrée différentes. La figure 6.12 montre en

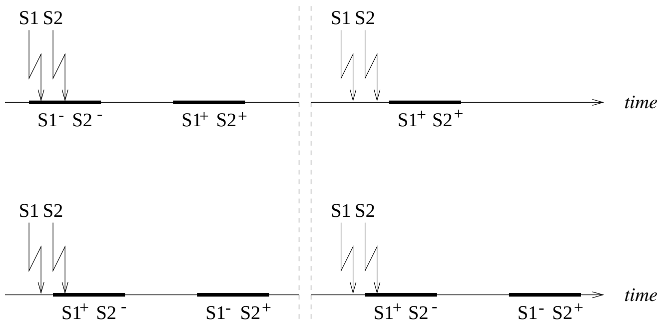


FIG. 6.12 - *Cas de 2 occurrences arbitrairement proches*

particulier que 2 signaux dont les occurrences sont arbitrairement proches mais voisines de l'instant d'activation peuvent apparaître dans 2 événements distincts. On voit donc que l'exécution introduit une source d'indéterminisme qui peut modifier l'évolution du système.
Plaçons nous dans le cas où le temps entre 2 activations consécutives possède une borne supérieure :

$$max\{t_{i+1} - t_i, i \in A\} = M$$

Le temps entre 2 réactions consécutives possède aussi une borne inférieure qui est le temps minimum τ de la réaction atomique du module :

$$min\{t_{i+1} - t_i, i \in A\} = \tau$$

Soit $I(S)$ l'instant de prise en compte de l'occurrence d'un signal S.
Si l'on considère 2 occurrences S_1 et S_2 de 2 signaux distincts, 2 cas se présentent :

$$t(S_1) \underset{\mathbb{R}}{\leq} t(S_2) \underset{\mathbb{R}}{\leq} t(S_1) + \tau \quad \Rightarrow \quad I(S_1) \leq I(S_2) \leq I(S_1) + 1$$
$$t(S_1) \underset{\mathbb{R}}{\leq} t(S_1) + M \underset{\mathbb{R}}{\leq} t(S_2) \quad \Rightarrow \quad I(S_1) < I(S_1) + 1 \leq I(S_2)$$

Notons que le cas particulier d'une exécution périodique se traduit par $M = \tau = T$. La première implication correspond au scénario de gauche de la figure 6.11 et aux scénarios de la figure 6.12. La deuxième implication correspond au scénario de droite de la figure 6.11. Un ordre stricte entre 2 occurrences de signaux se traduit par un ordre faible entre leurs instants de prise en compte. Ceci est dû au fait que les dates d'occurrence des signaux sont arbitraires par rapports aux instants d'activation. Dans la section qui suit, nous modélisons ce biais introduit par la machine.

6.5.2 Modélisation du biais introduit par la machine

Pour un signal d'entrée S, le biais est modélisé par le module `bias.strl`. La machine de Mealy correspondante est représentée figure 6.13.

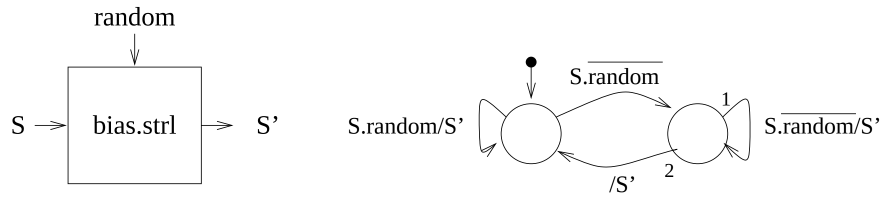


FIG. 6.13 - *Module `bias.strl` pour un signal d'entrée*

`S` est le signal d'entrée de la machine, `S'` est le signal correspondant du module synchrone. Le signal `random` a été rajouté au modèle pour prendre en compte l'indéterminisme introduit. Bien sûr, ce signal n'a pas de réalité physique, il n'est qu'un intermédiaire utile pour établir les propriétés. En présence de `S`, le signal `S'` est émis soit dans le même instant, soit à l'instant suivant en fonction du signal `random`.

On pourrait modéliser le biais introduit sur un signal de retour de tâche de la même manière. Toutefois, l'objectif étant de faire des preuves, il est nécessaire de prendre en compte la particularité de ces signaux pour avoir une modélisation suffisamment fine. Le module de biais pour un signal de retour de tâche est représenté figure 6.14.

Par rapport au module `bias.strl`, deux contraintes supplémentaires sont intégrées :

- Il ne peut pas y avoir d'occurrence de `R` avant que la tâche n'ait démarré (`start`).
- Pour une instance d'`exec`, il ne peut y avoir qu'une occurrence du signal de retour. Autrement dit, il ne peut pas y avoir de nouvelle occurrence de `R` si la précédente n'a pas été prise en compte par le programme et qu'une nouvelle instance n'a pas été démarrée.

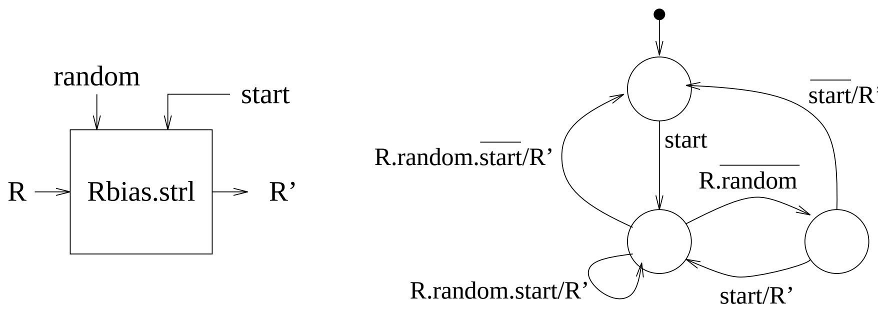


Fig. 6.14 - *Module* `Rbias.strl` *pour un signal de retour de tâche*

On peut prouver, toujours en utilisant la technique des observateurs, que le module `Rbias.strl` est celui obtenu par composition de `bias.strl` et d'un module `constraint.strl` exprimant les contraintes spécifiques aux signaux de retour de tâches (figure 6.15).

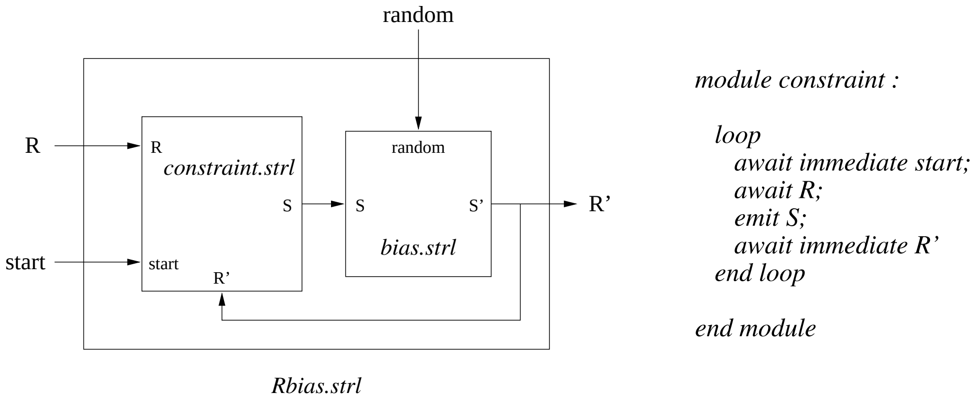


Fig. 6.15 - *Module* `Rbias.strl` *: une version contrainte de* `bias.strl`

Le module `Rbias.strl` ne prend pas en compte la possibilité de préemption d'un `exec`. Il convient pour les tâches `initATM, connect, disconnect` et `prepareMoney`.

Pour la tâche `getAndCheckCode` qui peut être avortée, le module de biais est plus complexe. Cette tâche est exécutée dans le macro-état `checkCode` (figure 6.7). L'introduction du signal `exit_checkCode`, émis (entre autres) en cas de préemption par `CANCEL` permet de détecter un éventuel avortement de la tâche. Ce signal est utilisé dans le module de biais `CODEbias.strl` représenté en SyncCharts[4] figure 6.16. Conformément à la remarque de la section 6.3.2.2, la condition de démarrage d'une nouvelle instance de tâche est ici : `[start and not exit_checkCode]`.

4. La machine de Mealy correspondante est plus difficilement lisible.

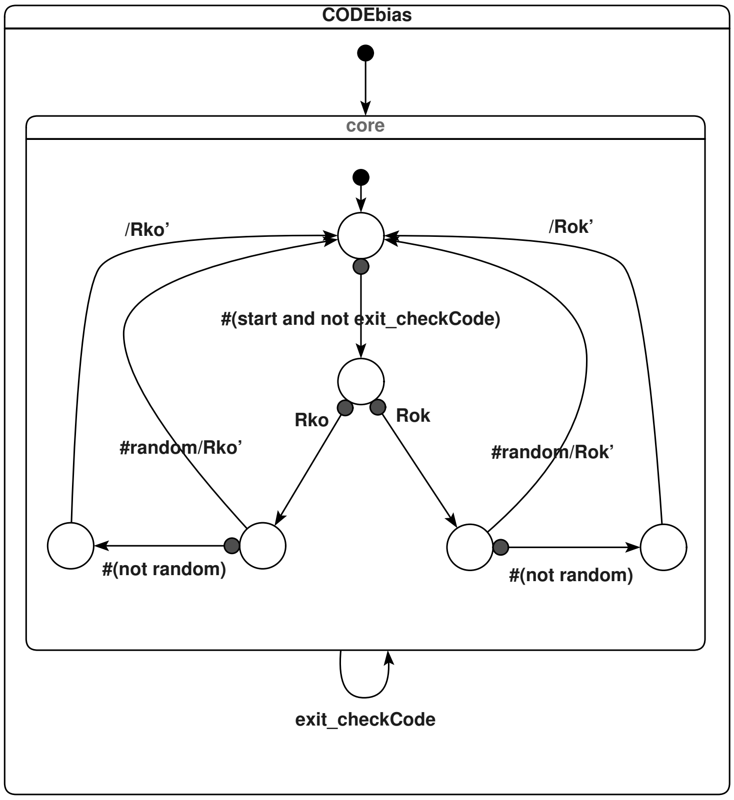


FIG. 6.16 - *Module de biais pour la tâche* `getAndCheckCode`

6.5.3 Analyse modifiée

Les modules de biais proposés dans la section précédente sont un modèle représentant la correspondance entre les signaux externes (ceux vus par la machine d'exécution) et les signaux synchrones du module. En plaçant un module de biais pour chaque entrée de pure-ATM, nous obtenons un système (pure-ATM, modules de biais) (figure 6.17) qui modélise la machine d'exécution de pure-ATM. Sur cette figure, les entrées "non connectées" des modules de biais correspondent

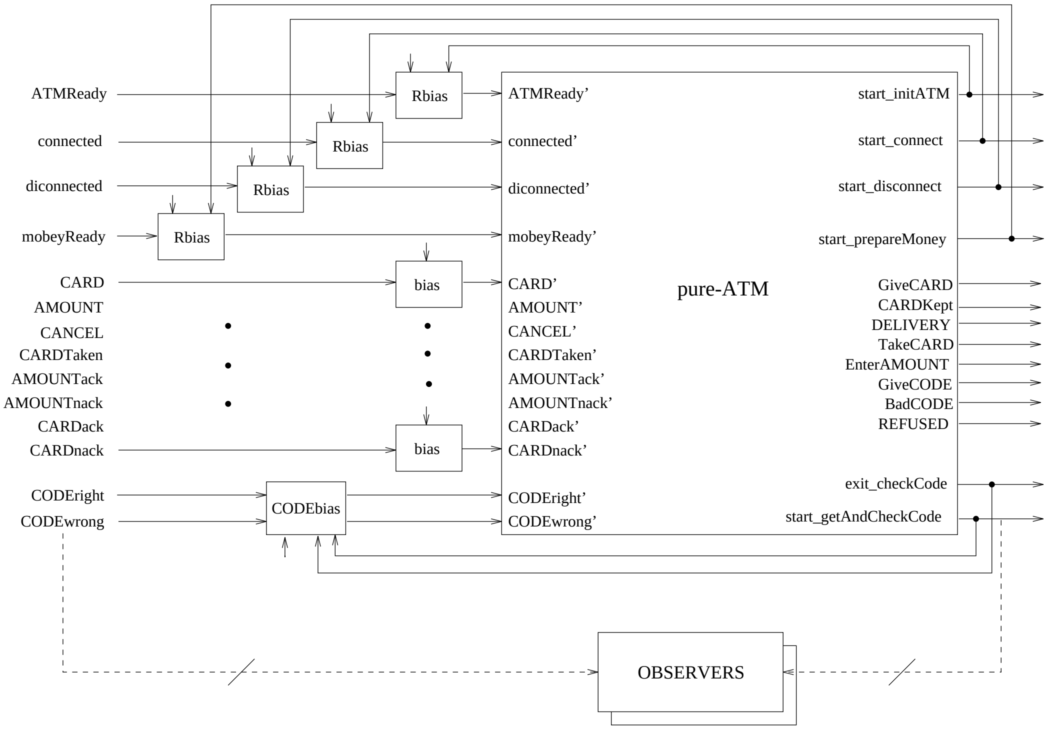


FIG. 6.17 - *Ensemble (pureATM,modules de biais)*

chacune à une entrée `random`. Toujours en utilisant la technique des observateurs, nous pouvons étudier les propriétés de ce modèle. Il s'agit en particulier de savoir si les propriétés vérifiées par le module pure-ATM restent valables pour sa machine d'exécution.

En plaçant les observateurs de la section 6.3.3, il s'avère que seule la propriété P3 reste vérifiée. En effet, le biais sur les signaux d'entrée n'a pas d'influence sur cette propriété car elle consiste en la synonymie de 2 signaux de *sortie*.

Pour la propriété P2 (*"aucune somme n'est délivrée si le client annule la transaction"*), l'histoire d'entrée sauvegardée par XEVE montre qu'il y a violation dans 2 cas :

- On a simultanément les signaux `CARD` et `CANCEL`. Cet événement d'entrée du modèle correspond à des occurrences très proches dans le temps des 2 signaux de la machine (figure 6.12). On a vu que ces 2 occurrences, aussi proches soient-elles, peuvent être reportées dans 2 événements d'entrée distincts du module `pure-ATM`. En particulier, si `CANCEL` est pris en compte avant `CARD`, son occurrence est perdue car elle arrive un instant trop tôt (voir figure 6.7) et par conséquent le signal `DELIVERY` peut être émis.

– On a simultanément `CANCEL` et `AMOUNTack` avec cette fois `CANCEL` pris en compte un instant après `AMOUNTnack`. Tout se passe alors comme si le client avait appuyé trop tard sur la touche d'annulation, celle-ci n'est pas prise en compte et le signal `DELIVERY` peut être émis.

Si on modifie l'observateur initial pour exclure ces 2 cas de simultanéité, la propriété est vérifiée. Voici l'observateur modifié :

```
loop
  weak abort
    await CANCEL; % was immediate
    await DELIVERY; % was immediate
    emit DELIVERY_after_CANCEL;
    await tick % shouldn't reach this point
  when CARD % a new card
end loop
```

La propriété P4 garantit que si une transaction est annulée par le client (`CANCEL`), le signal `sendCANCEL` est émis *dans le même instant*. À cause du biais, cette propriété n'est plus strictement vérifiée : il peut y avoir un décalage d'un instant entre `CANCEL` et l'émission de `sendCANCEL`. De plus, contrairement à la version sans biais, s'il y a simultanéité entre `CANCEL` et `AMOUNTack` (figure 6.5), il n'est pas garanti que l'annulation soit prise en compte. C'est ce qu'exprime l'observateur suivant :

```
loop
  weak abort
    await immediate sendCARD;
    loop
      await immediate CANCEL;
      present not sendCANCEL then
        await tick;
        present not sendCANCEL then
          emit CANCEL_without_sendCANCEL
        end present
      else
        await tick
      end present
    end loop
```

```
    when start_disconnect
  end loop
```

Cette nouvelle propriété est vérifiée par XEVE.

La propriété P1 concernant la confiscation de la carte subit la même altération que précédemment :

- Le signal `CARDKept` n'est pas forcément synchrone avec le troisième `CODEwrong`, il peut être retardé d'un instant.
- En cas de simultanéité entre la 3^{eme} occurrence de `CODEwrong` et `CANCEL`, il se peut que la carte ne soit pas confisquée.

L'observateur est plus complexe que celui de la section 6.3.3 :

```
loop
  await start_getAndCheckCode;
  weak abort
    await CODEwrong;
    await immediate start_getAndCheckCode;
    await CODEwrong;
    await immediate start_getAndCheckCode;
    await CODEwrong;
    present not CARDKept then
      await tick;
      present not CARDKept then
        emit three_CODEwrong_without_CARDKept
      end present
    end present
  when immediate exit_checkCode
end loop
```

Plusieurs attentes consécutives d'un signal d'entrée entraînent, à cause des modules de biais, une désynchronisation entre l'observateur et le module `pure-ATM`. L'utilisation du signal `start_getAndCheckCode`, qui indique que la saisie du code peut commencer, permet d'effectuer une resynchronisation. Le signal `exit_checkCode` quant à lui, est utilisé pour détecter un éventuel avortement de la tâche. La propriété est bien vérifiée par `Xeve`.

On notera que l'automate résultant de la mise en parallèle des observateurs, des modules de biais et du module pure-ATM possède environ 500 000 états[5] (contre une quarantaine pour la version non biaisée). Le temps mis par XEVE pour les parcourir est d'environ 10 minutes (contre 1 minute pour la version non biaisée).

5. Au cours de l'expérimentation, le chiffre de 8 millions d'états a été atteint (avec des modules de biais insuffisamment contraignants).

6.5.4 Conclusion

En première analyse, cette étude montre que les propriétés du programme synchrone ne sont pas strictement vérifiées à l'exécution. Il s'avère en fait que les seuls cas de violation des propriétés sont ceux qui font intervenir la simultanéité de 2 signaux d'entrée de la machine d'exécution. En effet, il y a une probabilité non nulle pour que 2 occurrences de signaux, aussi proches soient-elles, se retrouvent dans 2 événements d'entrée distincts. En reprenant les notations de la section 6.5, on a :

$$\begin{aligned} \Delta I = 0 &\Rightarrow \Delta S \underset{\mathbb{R}}{<} M \\ \Delta S \underset{\mathbb{R}}{<} M &\not\Rightarrow \Delta I = 0 \end{aligned}$$

Il faut donc être très prudent quant à l'interprétation qui est faite de la simultanéité de 2 signaux d'entrée dans un programme synchrone.

6.6 Conclusion

L'objectif de ce chapitre était :

- d'illustrer sur un exemple concret la méthode de conception de machines d'exécution,
- voir dans quelle mesure les propriétés du programme synchrone restent vérifiées par sa machine d'exécution.

Pour cela, nous avons traité entièrement l'exemple de l'ATM :

- Le contrôleur a été conçu en SyncCharts.
- Des propriétés de sûreté essentielles pour l'application ont été vérifiées formellement en utilisant la technique des observateurs. Nous avons été amenés pour cela à extraire du contrôleur la partie purement réactive.
- Nous avons ensuite appliqué l'architecture de machines d'exécution à l'ATM en utilisant des outils de génération automatique.
- Nous avons enfin analysé les propriétés de la machine d'exécution, c'est-à-dire celles portant sur les signaux physiques. Il a fallu pour cela modéliser le biais introduit à l'acquisition des signaux d'entrée.

L'expérimentation a d'abord mis en évidence le problème du passage d'un programme Esterel valué à un programme pur. Ce traitement n'est pas automatisable dans le cas général. Il demande une connaissance fine du contrôle à réaliser. C'est une phase *critique* dans notre approche.

Un autre point important est la nécessité d'une modélisation *réaliste* des signaux physiques, en particulier pour le retour des tâches. Des modules de biais insuffisament contraints conduisent rapidement à une explosion de l'espace des états accessibles.

Les résultats des tests montrent que certaines propriétés du programme initial ne sont plus strictement vérifiées à l'exécution. En fait, il a fallu modifier les observateurs pour tenir compte des incertitudes introduites par la discrétisation du temps. Le non respect de propriétés peut conduire, dans certains cas, à modifier le programme initial pour le rendre moins sensible aux petites incertitudes temporelles sur les signaux d'entrées.

Chapitre 7

Conclusion

Nous avons adopté dans cette thèse une approche synchrone pour le développement des applications réactives temps réel. Cette approche est d'un grand intérêt pour la conception de ces systèmes. Les hypothèses synchrones permettent d'exprimer des comportements complexes et donnent aux différents formalismes des sémantiques rigoureuses et puissantes. Nous nous sommes intéressés plus particulièrement au langage Esterel et au formalisme graphique SyncCharts qui sont dédiés aux systèmes réactifs à contrôle dominant. Leurs compilateurs produisent un code sûr et efficace qui peut faire l'objet de preuves formelles.

En revanche les programmes issus de l'approche synchrone ne sont pas directement exécutables. Le passage à un système opérationnel nécessite une **machine d'exécution** dont le rôle est d'enfouir le processus synchrone dans l'environnement asynchrone en assurant une approximation satisfaisante des hypothèses synchrones. Développer une machine d'exécution générique pour programmes synchrones a été notre objectif. La définition et la réalisation d'une telle machine est notre principale contribution. Ces travaux entrent dans le cadre plus large de l'approche synchrone. Ils ont été partiellement soutenus par un contrat Cnet intitulé “Réalisation de machines d'exécution pour langages parallèles synchrones” [1] auquel nous avons participé en collaboration avec le Centre de Mathématiques Appliquées de l'École des Mines de Paris [ABM+97].

Apports de la thèse

Architecture des machines d'exécution

Une première architecture modulaire de machines a été définie à partir d'un exemple simple. En ce qui concerne la connexion à l'environnement, des **contrôleurs d'entrée/sortie** sont utilisés pour connecter les signaux logiques du pro-

1. Marché Cnet 94 1B 111.

cessus aux événements réels. Le parallélisme entre machine et environnement est pris en compte au niveau des contrôleurs d'entrée qui sont chargés de surveiller et de mémoriser les évolutions de l'environnement en cours de réaction. La décomposition des réactions de la machine en 3 phases : acquisition, réaction du processus et émission, permet d'assurer une vision figée de l'environnement pendant les réactions. Pour répondre aux exigences de sûreté de fonctionnement, un **observateur** chargé de détecter et éventuellement de traiter les anomalies de fonctionnement de la machine a été intégré à l'architecture. Pour paramétrer le fonctionnement de la machine et afin de répondre à des besoins spécifiques, un module appelé "**séquenceur**" est introduit. Vue de l'extérieur cette architecture est caractérisée par des signaux d'entrée/sortie, un vecteur de contrôle qui permet de contrôler l'exécution et un vecteur d'exceptions.

Afin d'avoir une vision unifiée des divers composants de l'architecture, le modèle général de **processus** a été introduit. Il reprend le principe d'une entité possédant des entrées, des sorties, un vecteur de contrôle et un vecteur d'exceptions en lui ajoutant un caractère récursif. Le concept de processus permet de modéliser aussi bien les processus synchrones que les machines d'exécution. Les contrôleurs d'entrée/sortie sont des composants essentiels de la machine car ils font l'interface entre monde synchrone et monde asynchrone. Ils sont modélisés par des processus dont le contrôle est partagé entre l'environnement et la machine. Deux types de processus d'interface permettent de modéliser les mécanismes d'acquisition par échantillonnage et par interruption. Ces processus peuvent émettre des signaux d'exception indiquant la perte d'échantillons en entrée.

Les **boîtes réactives** sont un modèle permettant de décrire les machines d'exécution. Une boîte réactive est obtenue par composition d'un processus d'entrée, d'un processus synchrone à exécuter et d'un processus de sortie, ainsi que d'un séquenceur et d'un observateur. Toutes les boîtes réactives respectent cette architecture. De plus leur fonctionnement peut être paramétré :

- Par composition récursive : les processus qui forment la boîte réactive peuvent eux-mêmes être composés.
- Par le choix de séquenceurs et d'observateurs répondant à des besoins particuliers.

Par ailleur, une forme de composition hiérarchique de boîtes réactives permet de gérer la commutation entre les différents modes de fonctionnement d'une application.

Intégration des traitements asynchrones

Les tâches ne sont gérées par les programmes ESTEREL que d'un point de vue logique. Nous nous sommes intéressés à leur exécution effective. Pour cela, nous avons eu recours aux services d'un exécutif multitâches. L'exécution des tâches, a d'abord posé des problèmes délicats dûs à la sémantique de la primitive `exec` (terminaison et suspension simultanés, par exemple). Par ailleur, d'autres problèmes sont apparus concernant la cohérence entre l'état réel de la tâche et la vision idéale qu'en a le programme. Le cas le plus typique est celui de l'avortement/redémarrage, considéré comme instantané. Face aux risques de désynchronisation entre tâches et processus synchrone, nous avons eu à faire des choix qui sortent du cadre de la sémantique de la primitive `exec`. Nous avons proposé deux sémantiques pour la gestion des tâches :

- La sémantique dite "maximale" pour un couplage fort entre la tâche réelle et l'`exec`.
- La sémantique dite "minimale" pour un couplage plus lâche entre le programme et les tâches.

En cas d'anomalie, le principe retenu est que la machine d'exécution ne fait que *signaler* les situations qu'elle juge incohérentes. L'utilisateur est alors libre de les ignorer et de s'en tenir strictement à la sémantique de la primitive `exec`.
L'intégration de cette gestion dans l'architecture de machines d'exécution s'est faite par l'introduction de contrôleurs d'`exec` dont nous avons décrit le comportement.

Réalisations

Un de nos objectifs était de fournir à l'utilisateur des **outils pour la génération automatique de machines d'exécution**. La structuration de l'implémentation était un préalable à toute automatisation des traitements. Le développement et la mise au point d'un environnement de machines d'exécution sous le système Solaris2 a nécessité un investissement important en temps. Cet environnement est opérationnel. Il permet de configurer et de produire des machines exécutables. Il est composé de :

- Une bibliothèque d'exécution `libmex` contenant l'architecture générique de machines sous la forme d'un ensemble de classes `C++` abstraites.
- Une bibliothèque d'entrée/sortie fournissant des classes de contrôleurs pour la connexion des signaux Esterel à l'environnement.
- Un générateur de code `xmdf2c++` permettant de produire une machine exécutable à partir d'une description textuelle.

Les bibliothèques développées reposent d'une part sur la bibliothèque d'objets synchrones `libSync` et d'autre part sur une bibliothèque d'*exécutif virtuel*. Cette dernière regroupe l'ensemble des services que doit fournir l'exécutif sous-jacent, en particulier la gestion des tâches et des synchronisations. Ainsi le portage de la machine sur d'autres exécutifs devrait se limiter à la réécriture de cette couche.

Vérification de propriétés de la machine d'exécution

Outre l'implantation proprement dite, il nous a paru intéressant d'analyser la qualité des exécutions produites. La communauté synchrone a mis à disposition des outils efficaces pour vérifier les propriétés de sûreté des programmes synchrones. Toutefois ces propriétés portent sur les signaux logiques du programme. Au cours de l'exécution, la correspondance signaux logiques/signaux physiques n'est bien sûr pas parfaite. Nous avons donc analysé les mêmes propriétés, mais cette fois à partir des signaux physiques. Nous avons été pour cela amenés à modéliser les distorsions introduites par l'acquisition des signaux dans la machine d'exécution. Cette modélisation doit être suffisamment réaliste pour éviter de dépasser les possibilités de traitement des outils disponibles. Les résultats de l'analyse montrent que certaines propriétés ne sont plus strictement vérifiées à l'exécution. Les cas de violation sont ceux qui font intervenir des occurrences très proches de signaux d'entrée de la machine d'exécution. Ces résultats mettent en évidence les problèmes dûs au caractère arbitraire de la discrétisation du temps. Ces constatations peuvent conduire dans certains cas à modifier le programme intial afin de le rendre moins sensible aux petites incertitudes temporelles sur les signaux d'entrée.

Perspectives

Plusieurs prolongements sont envisageables à nos travaux :

En ce qui concerne les réalisations à **court terme**, il convient d'enrichir les bibliothèques actuelles par de nouvelles classes de contrôleurs d'entrée/sortie. Un autre point intéressant est le portage de l'environnement sur d'autres exécutifs. En particulier, le portage des bibliothèques sur des exécutifs permettant une meilleur maîtrise du temps devrait faciliter les évaluations temporelles. Du point de vue de la convivialité, une interface graphique pour la configuration des machines devrait remplacer le format textuel actuel.
En ce qui concerne les problèmes d'acquisition des signaux d'entrée de la machine, nous avons été en mesure de quantifier les distorsions qui existent entre les dates d'occurrences réelles et les instants de prise en compte par la machine. Il apparaît en fait qu'en raison de la discrétisation du temps, l'exécution ne conserve pas les *distances temporelles*. Le cas de deux occurrences arbitrairement proches est un cas particulier pour lequel la distance réelle est nulle alors que la distance logique peut être d'un instant. Le modèle proposé est une approximation du

comportement de l'interface de la machine d'exécution. Il pourrait être raffiné pour les différentes classes d'objets d'interface.

À plus long terme, une automatisation de la traduction "propriété de programme" vers "propriété de l'implémentation" est à envisager. Pour ce qui est des preuves logiques, il s'agit de faciliter le passage d'un programme valué à un programme pur. D'autres possibilités sont envisageables, en particulier concernant les preuves faisant intervenir des types énumérés et non plus seulement des booléens. En ce qui concerne les preuves temporelles, il faut d'une part disposer de cibles d'exécution à comportements temporels prévisibles et d'autre part développer des outils pour évaluer de façon réaliste les différentes durées d'exécution. De telles études sont utiles pour les industriels proposant des applications offrant des qualités de service sur une cible donnée.

Avec le développement des applications mettant en œuvre des réseaux, il faut également proposer des implémentations de machines d'exécution réparties. Des résultats concernant la répartition de programmes flot de données synchrones ont déjà donné des réalisations opérationnelles (outil SYNDEX de Y. Sorel [GMP+90], répartition de code LUSTRE de A. Girault [CCGJ96]). Par contre, les programmes synchrones impératifs posent des problèmes encore ouverts.

Annexe A

Syntaxe des SyncCharts

SyncCharts est un formalisme graphique synchrone basé sur la notion d'états/transitions et qui supporte la hiérarchie, le parallélisme et la préemption. Une présentation complète du modèle peut être trouvée dans un article de C. André, S. Dissoubray et de l'auteur [ABD98]. Nous donnons ici un résumé de la syntaxe.

La figure A.1 résume les éléments graphiques qui composent les SyncCharts. Les SyncCharts ont été définis avec des vocables anglais. Les éléments de base sont *star*, *constellation* et *macro-state*[1].

Une étoile (star, Figure A.1.B) est plus qu'un état : c'est un état et les façons de quitter cet état. Les arcs d'avortement issus d'un état constituent les “rayons” de l'étoile.

Une constellation (Figure A.1.C) est un ensemble d'étoiles tel qu'une et une seule étoile est active à l'intérieur d'une constellation.

Un macro-état (Figure A.1.A) est constitué d'une ou plusieurs constellations en parallèle.

Les arcs sont de deux types :

- les arcs d'avortement, entre deux étoiles ou macro-états,
- les arcs auxiliaires, depuis un connecteur vers une étoile ou un macro-état.

Les arcs sont étiquetés. La forme générale d'une étiquette est :

`trigger[condition]/effect` avec

- `trigger` : une expression booléenne (utilisant les opérateurs `and, or, not` et des parenthèses) portant sur la présence des signaux,
- `condition` : un prédicat pouvant faire intervenir des valeurs de signaux,

1. Un macro-état est parfois appelé “firmament” pour des raisons d'homogénéité.

– `effect` : une liste de signaux, éventuellement valués, à émettre lors de la transition.

Ces différents champs sont optionnels. Le défaut pour le trigger est le signal `tick`, un signal prédéfini présent à chaque instant. Le trigger peut avoir un effet immédiat si il est préfixé par le symbole `#`.

`A; /O; T [?T >nextTime]` sont des exemples d'étiquettes. La première étiquette déclenche la transition sur l'occurrence du signal `A`. Dans la deuxième, la transition est déclenchée au prochain instant avec émission de `O`. La troisième étiquette déclenche la transition sur l'occurrence du signal `T` si la valeur de ce signal (disons de type `time`) est supérieure à la valeur de la constante `nextTime` du même type.

Un syncChart a un *comportement entièrement déterministe*. Dans le cas où il existe plusieurs façons de quitter une étoile, on doit donner un ordre d'évaluation. Les entiers $1, 2, \cdots, n$ situés près de l'origine d'un arc précisent la priorité de celui-ci (le plus petit entier est le plus prioritaire)[2].

2. Avec la version v2, la règle a été modifiée : les avortements forts sont plus prioritaires que les avortements faibles, qui sont eux-mêmes plus prioritaires que la terminaison normale. Les priorités permettent de choisir parmi les avortements de même "force" (faible ou fort).

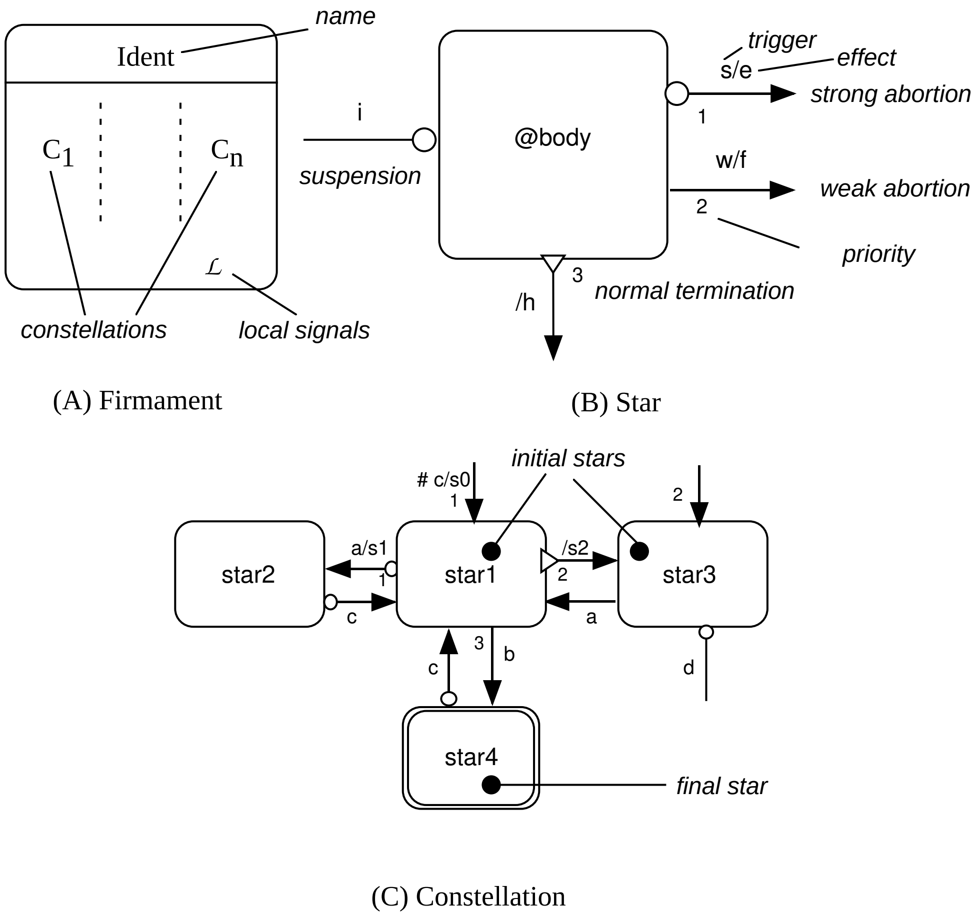


Fig. A.1 - SyncCharts*: Éléments de syntaxe*

Annexe B

Contrôle d'un atelier de production automatisé

B.1 Présentation de l'application

L'atelier a déjà été présenté dans le chapitre 2. Les deux paragraphes qui suivent peuvent éventuellement être omis.

Il s'agit d'un atelier d'usinage de pièces métalliques. Cet exemple a été proposé par les chercheurs de la FZI[1] comme modèle pour l'évaluation des différentes méthodes formelles de développement des systèmes réactifs [LL95]. L'atelier existe réellement, mais nous utilisons pour le test un logiciel de simulation qui a été développé à la FZI. C'est une fenêtre graphique TCL-TK (figure B.1) qui simule les différents dispositifs de l'atelier et leurs évolutions.

L'atelier est constitué de 6 éléments : un convoyeur d'alimentation, une table rotative-élévatrice, un robot à deux bras, une presse, un convoyeur de dépôt et une grue mobile. Le processus d'usinage se déroule de la manière suivante : chaque pièce est acheminée par le premier convoyeur vers la table rotative. Le robot est ensuite chargé de la transférer vers la presse. Une fois la pièce pressée, le robot effectue le transfert vers le deuxième convoyeur. Cet atelier étant conçu pour l'expérimentation, le processus doit pouvoir évoluer sans l'intervention d'un opérateur extérieur. Ainsi, la presse ne modifie pas réellement la forme des pièces et celles-ci sont réintroduites dans la chaîne en fin de parcours : la grue mobile déplace les pièces usinées du convoyeur de dépôt vers le convoyeur d'alimentation.

La tâche du contrôleur est de commander les différents dispositifs et d'enchaîner les différentes opérations :

– Convoyeurs : ils peuvent être mis en marche ou arrêtés. Une cellule photo-

1. Forschungszentrum Informatik de Karlsruhe

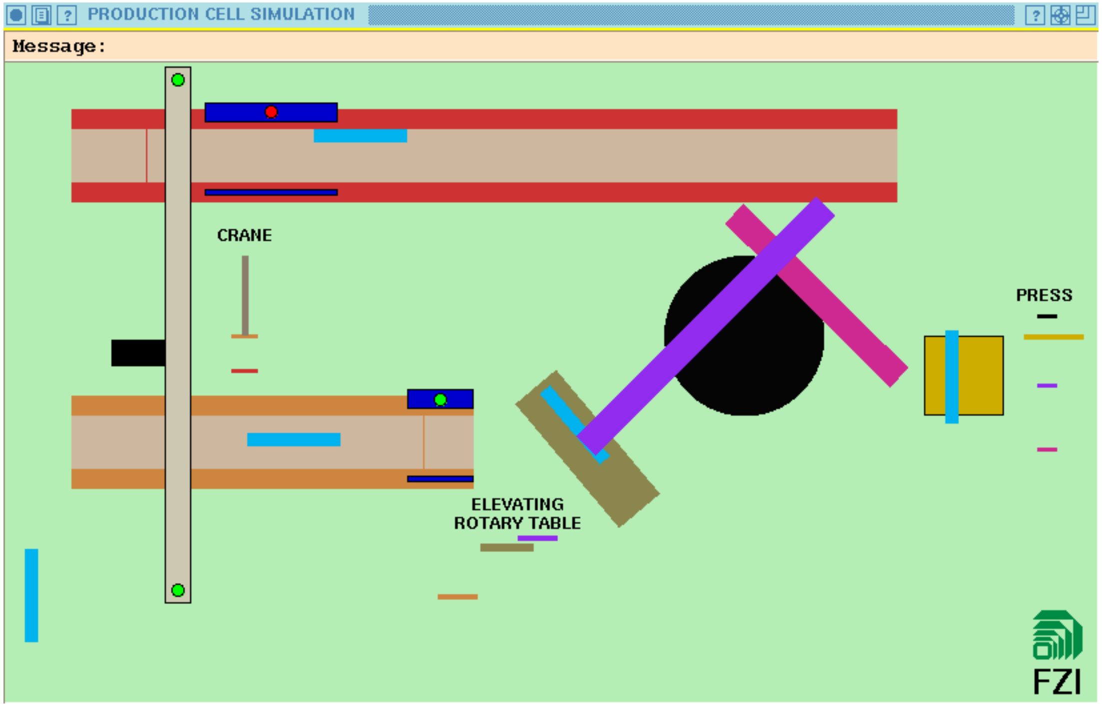


FIG. B.1 - *Représentation graphique de l'atelier*

électrique installée à leur extrémité indique si une pièce est entrée ou sortie de la partie terminale.

- Table rotative-élévatrice : son rôle est de faire pivoter les pièces de 45^o et de les porter à une hauteur où elles peuvent être saisies par le premier bras du robot.

- Robot : il comprend deux bras orthogonaux. Pour des raisons techniques, les deux bras ne sont pas à la même hauteur. Chaque bras peut s'étendre ou se rétracter horizontalement. Ils sont solidaires du même axe autour duquel ils peuvent pivoter. Les extrémités des deux bras sont pourvues d'électro-aimants qui leur permettent de saisir les pièces métalliques. La tâche du robot est de transférer les pièces de la table rotative vers la presse et de la presse vers le convoyeur de dépôt.

- Presse : elle est constituée de deux plateaux horizontaux, le plateau inférieur pouvant se déplacer verticalement pour forger les pièces.

- Grue mobile : son rôle est de ramasser les pièces du convoyeur de dépôt et de les décharger sur le convoyeur d'alimentation. Un électro-aimant permet la saisie des pièces. Il peut se déplacer horizontalement (d'un convoyeur vers l'autre) et verticalement (les convoyeurs ne sont pas à la même hauteur).

Le contrôleur connaît l'état du système grâce à un ensemble de capteurs (14 au total). Ce sont soit des capteurs *tout-ou-rien* (contacts ou cellules photo-électriques), soit, pour certains dispositifs, des potentiomètres (angle de rotation de la table, extension des bras du robot,...).
Les mouvements des dispositifs sont commandés par 13 actionneurs.

B.2 Conception du contrôleur en syncCharts

Le contrôleur

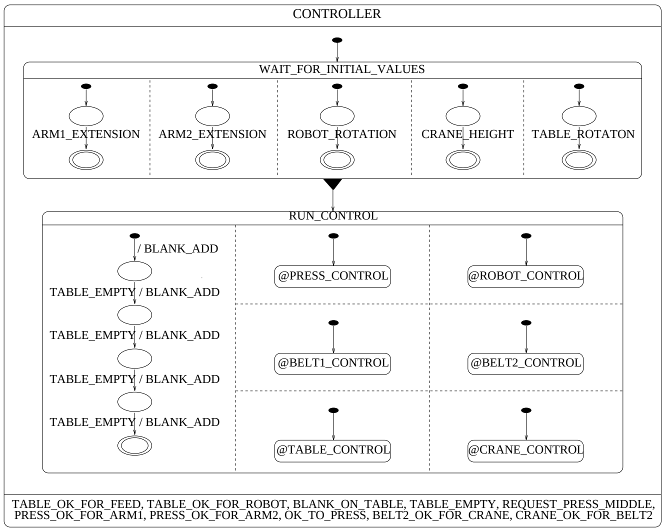


FIG. B.2 - *SyncChart du contrôleur*

Les particularités de cet atelier font qu'une décomposition fonctionnelle peut se calquer sur une décomposition structurelle. On distingue différentes unités comme le robot, la grue, ... À chaque unité est associé son propre contrôleur. La coordination de l'ensemble se fait par échange de signaux entre les contrôleurs. Ainsi l'interface d'un contrôleur comprend les signaux échangés avec les capteurs et les actionneurs du dispositif, plus des signaux de synchronisation. On a donc 6 contrôleurs : `belt1_control`, `belt2_control`, `table_control`, `robot_control`, `press_control` et `crane_control`.
La figure B.2 représente le syncChart de niveau 0 (*toplevel*) du contrôleur global. Il contient 2 macro-états en séquence : `WAIT_FOR_INITIAL_VALUES` spécifie la phase d'initialisation et `RUN_CONTROL` correspond au contrôle proprement dit. Le passage de la phase d'initialisation à la phase de contrôle se fait spontanément, dès que la première phase est terminée (utilisation d'un arc de terminaison normale). Le cahier des charges du simulateur ne prévoit pas de phase d'arrêt. Celle-ci pourrait se faire en ajoutant un arc d'avortement menant de `RUN_CONTROL` à un macro-état

d'arrêt. Grâce à l'orthogonalité de la préemption, cette modification ne remettrait pas en cause le corps du macro-état `RUN_CONTROL`.

La phase d'initialisation consiste à attendre que tous les capteurs numériques aient communiqué leur valeur initiale. Chacune des 5 branches parallèles attend l'occurrence d'un signal (`ARM1_EXTENSION`,...) puis passe dans un état final. Quand les 5 constellations sont dans leur état final, le macro-état `WAIT_FOR-_INITIAL_VALUES` termine normalement.

Le macro-état `RUN_CONTROL` comprend une constellation pour chaque contrôleur. La constellation supplémentaire, à gauche sur la figure, se charge d'introduire 5 pièces dans la chaîne (`BLANK_ADD`). Les signaux de synchronisation entre les différents contrôleurs sont déclarés comme des signaux locaux et mentionnés dans la partie inférieure du macro-état. Ces signaux ne proviennent pas de capteurs ; ils ont été introduits par le concepteur.

Chaque macro-état qui apparaît dans le macro-état `RUNCONTROL`, fait l'objet d'une description séparée. À titre d'exemple, nous allons détailler le contrôleur de la table rotative.

La table rotative-élévatrice

Le syncChart du contrôleur de la table est représenté figure B.3. La table transfère des pièces du convoyeur d'alimentation vers le robot. Elle alterne entre 2 positions : une position basse dans laquelle elle est prête à recevoir une pièce provenant du convoyeur et une position haute qui permet au bras du robot de saisir la pièce. La table doit d'abord se mettre en position de chargement (macro-état `GoToLoadPos`) puis signaler sa disponibilité au convoyeur. Cette signalisation est faite en SyncCharts par le maintien du signal de synchronisation `TABLE_OK-_FOR_FEED`. Une fois la pièce chargée (occurrence du signal `BLANK_ON_TABLE`), la table doit se mettre en position de déchargement (`GoToUnloadPos`) puis signaler sa disponibilité au robot (signal `TABLE_OK_FOR_ROBOT`). Une fois la pièce enlevée (occurrence du signal `TABLE_EMPTY`), la table retourne en position initiale et ainsi de suite.

Les macro-états `GoToLoadPos` et `GoToUnloadPos` ont des structures et des fonctionnalités analogues. Les SyncCharts permettent de définir des modules génériques et de les instancier.

Macro-état `GoToLoadPos`

Le macro-état `GoToLoadPos` (figure B.4) comporte 2 branches parallèles : une pour le déplacement vertical et une pour la rotation. On doit sortir de ce macro-état dès que les deux branches sont terminées. Pour spécifier ce comportement, les SyncCharts utilisent les étoiles finales.

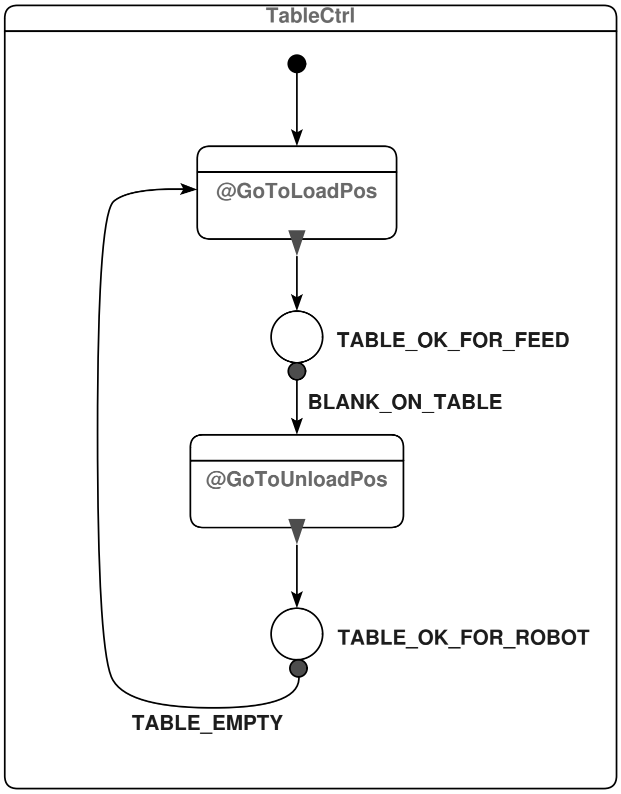


FIG. B.3 - *SyncChart du contrôleur de la table*

Comportement de chaque branche :

- *La position verticale* de la table est connue grâce à 2 contacts : `TABLE_AT_TOP` et `TABLE_AT_BOTTOM`. Les signaux `TABLE_UPWARD`, `TABLE_DOWNWARD` et `TABLE_STOP_V` commandent les déplacements verticaux de la table.

  Positionner un dispositif physique en un point donné est une tâche qui revient souvent dans les automatismes. Pour spécifier ce comportement, nous avons créé un macro-état générique appelé `basicCtrl` (voir figure B.5).

  Le principe est de déclencher le mouvement d'un dispositif (signal de commande `START`) et de l'arrêter (signal de commande `STOP`) quand la position désirée a été atteinte (occurrence du signal `REACHED`). Si le dispositif est initialement dans la position voulue, le signal `REACHED` est immédiatement pris en compte (signal déclencheur précédé du # sur le syncChart) et l'état final est atteint sans émission de `START`.

- Pour *la rotation*, le macro-état `rotateToBelt` est aussi une instance d'un module générique "`servoCtrl`". Ce module a été conçu en SYNCCHARTS

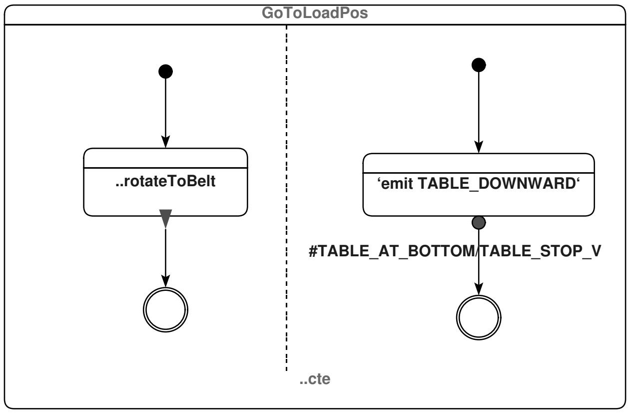


FIG. B.4 - *macro-état GoToLoadPos*

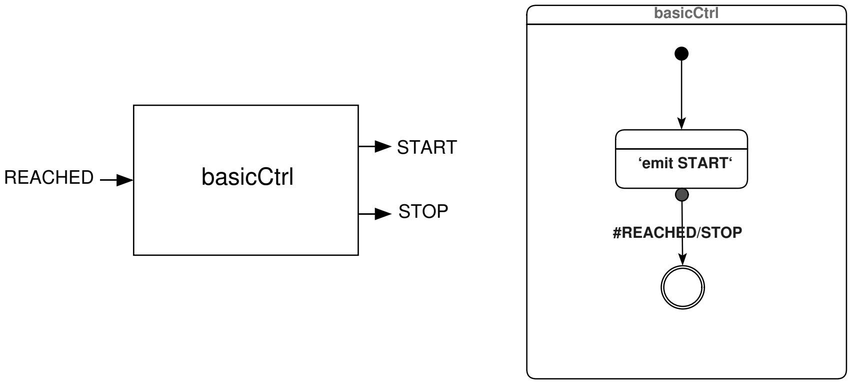


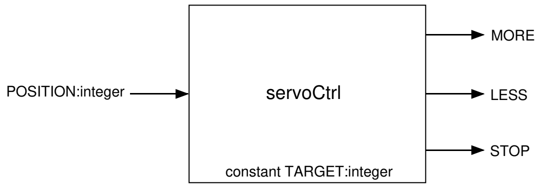
FIG. B.5 - *Module générique basicCtrl*

séparément et traduit automatiquement en ESTEREL. Nous le réutilisons dans cette application en faisant des "`run`" du module ESTEREL.

Ce module a été défini pour commander les appareils dont la position est repérée par un capteur numérique. Il peut être instancié avec 3 signaux de commande purs, un capteur numérique d'entrée et une valeur numérique *cible*. Il est utilisé pour amener un dispositif dans la position cible à l'aide des 3 commandes d'actionneurs (typiquement avancer, reculer et arrêter).

Il est instancié ici avec les signaux `TABLE_ROTATION`, `TABLE_RIGHT`, `TABLE_LEFT`, `TABLE_STOP_H` et la constante `TABLE_ANGLE_FOR_BELT` pour asservir la position angulaire de la table. Ce module est aussi utilisé dans les contrôleurs du robot et de la grue.

Remarque : certains modules génériques peuvent être directement exprimés en code ESTEREL, ce qui permet de réutiliser des conceptions antérieures.




```
run rotateToBelt/servoCtrl[
    signal  TABLE_ROTATION/POSITION,
            TABLE_RIGHT/MORE,
            TABLE_LEFT/LESS,
            TABLE_STOP_H/STOP;
  constant  TABLE_ANGLE_FOR_BELT/TARGET]
```

FIG. B.6 - *Module générique* `servoCtrl`

B.3 Vérifications

Un autre point important concerne l'aspect preuves formelles. Pour garantir le bon fonctionnement de l'atelier, il serait intéressant d'établir des propriétés de sûreté du type “il ne peut pas y avoir de collision entre le robot et la table rotative”. Dans le cas de l'atelier, ces propriétés font intervenir un grand nombre de signaux valués correspondant aux positions des différents dispositifs. Nous avons vu dans le chapitre 6 que les outils de preuves dont nous disposons ne s'appliquent pas à ce type de signaux. Le passage à un programme pur nécessiterait, ici, une modélisation du comportement d'une grande partie de l'atelier. Ce type de modélisation sort du cadre de nos travaux.

B.4 Couplage avec la simulation

Le simulateur graphique de l'atelier est un processus Unix qui lit les commandes à destination des actionneurs sur `stdin` et écrit les valeurs des capteurs sur `stdout`. Le format textuel de ces informations est clairement spécifié par les concepteurs du simulateur. Le couplage entre le contrôleur et la simulation nécessite donc d'établir la correspondance entre ces informations textuelles et les signaux ESTEREL du contrôleur.

La modélisation SYNCCHARTS est d'abord compilée en ESTEREL puis en `C++`. La machine d'exécution est configurée avec des handlers sur *pipe* Unix en entrée et en sortie. Des modules de codage/décodage sont chargés de multiplexer et de démultiplexer les signaux sur le pipe.

Une fois le contrôle mis au point par simulation interactive, le couplage avec l'atelier réel se fait en remplaçant les handlers pour entrées/sorties sur pipe par des handlers de signaux physiques et en recompilant la machine d'exécution.

Annexe C

Environnement sous Solaris 2

C.1 Architecture en couches

La figure C.1 représente l'architecture en couche des bibliothèques développés :

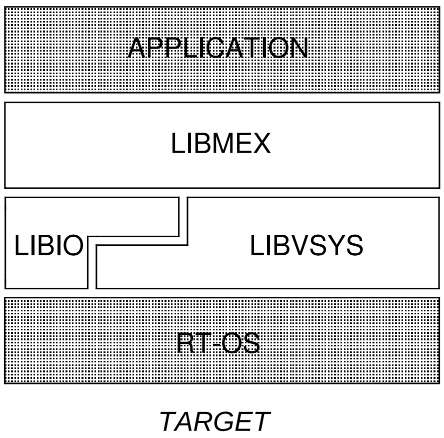


FIG. C.1 - *Couches logicielles*

La bibliothèque `libmex` contient l'architecture générique des machines sous la forme d'un ensemble de classes `C++` abstraites. Elle repose sur la bibliothèque synchrone `libSync` de F. Boulanger.

La bibliothèque d'entrée/sortie `libIO` fournit des classes de contrôleurs permettant divers types de connexion entre signaux ESTEREL et environnement.

La bibliothèque `libvsys` regroupe l'ensemble des services que doit fournir l'exécutif sous-jacent. Il s'agit en particulier des threads, des synchronisations et des entrées/sortie. Actuellement, certaines classes de contrôleurs d'E/S accèdent directement aux services de l'exécutif sans l'intermédiaire de `libvsys`.

C.2 Aperçu des classes de `libmex`

La figure C.2 est un diagramme de classes à la OMT représentant les classes de `libmex`.

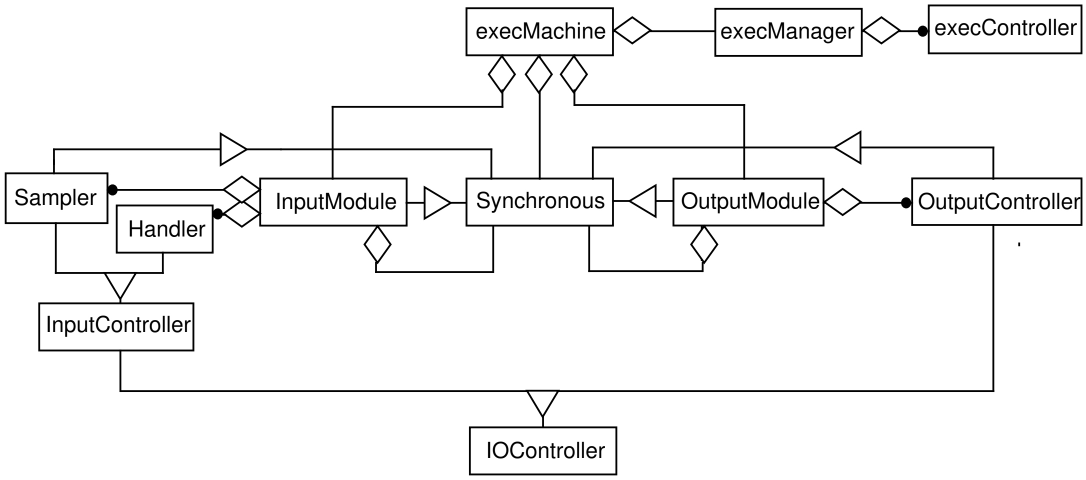


FIG. C.2 - *Classes de `libmex`*

Les losanges représentent la relation d'agrégation : les objets de la classe `execMachine` par exemple, sont constitués d'un module d'entrée (`InputModule`), d'un objet synchrone à exécuter (`Synchronous`), d'un module de sortie (`OutputModule`) et d'un contrôleur de tâches (`execManager`). Un rond noir à l'extrémité d'un arc d'agrégation indique la multiplicité du lien. Ainsi un module de sortie (`OutputModule`) est composé de plusieurs contrôleurs de sortie (`OutputController`). Les triangles représentent la relation d'héritage : les échantillonneurs de signaux d'entrée (`Sampler`) sont des contrôleurs d'entrée particuliers (`InputController`) qui eux même sont des contrôleurs d'entrée/sortie particuliers (`IOController`). On notera que contrairement aux `Sampler`, les `Handler` ne sont pas des objets synchrones. Ceci vient du fait que les objets synchrones sont *control driven* alors que les `handlers` sont purement *input driven*.

Toutes les classes de `libmex` sont abstraites à l'exception de la classe `execManager`. Les classes concrètes sont :

- Celles des contrôleurs d'E/S contenues dans `libIO`. Ces classes sont dérivées de `OutputController`, de `Sampler` ou de `Handler`.

- Les autres classes concrètes sont dépendantes de l'application à exécuter et sont produites automatiquement.

C.3 Chaîne de compilation

La figure C.3 représente la chaîne de compilation sous Solaris 2.

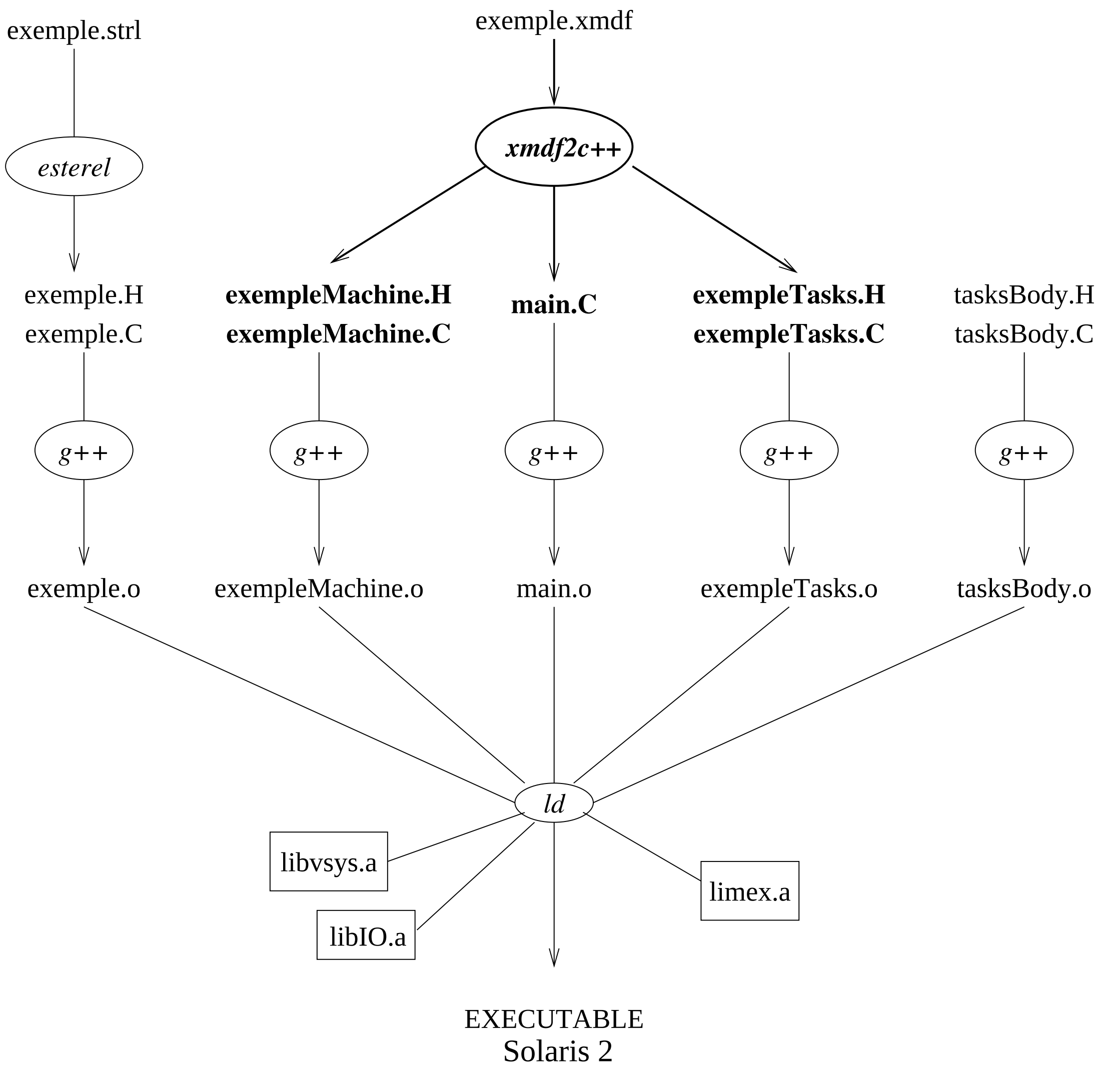


FIG. C.3 - *Chaîne de compilation*

Le générateur `xmdf2c++` produit 5 fichiers `C++` à partir d'une description textuelle au format `xmdl` (eXecution Machine Description Language) :

- `exempleMachine.H, exempleMachine.C` : ces fichiers contiennent la spécification et l'implémentation de la structure de la machine sous la forme des classes concrètes `exempleInputModule`, `exempleOutputModule` et `exempleExecMachine`.

- `exempleTasks.H, exempleTasks.C` : ces fichiers contiennent les contrôleurs de tâches concrets. Ces contrôleurs font l'interface entre le code `C` ou `C++`

fourni par l'utilisateur (fichiers `tasksBody.H` et `tasksBody.C`), l'`exec` ESTEREL correspondant et les services de l'exécutif.

– `exempleX.C` : programme principal.

Le compilateur ESTEREL puis le processeur `occ++` de F. Boulanger sont utilisés pour produire les fichiers `exemple.H` et `exemple.C` qui contiennent la classe synchrone de l'application.
La machine exécutable est obtenue par compilation des fichiers `C++` puis par édition des liens avec les bibliothèques.

Bibliographie

[ABD97] C. André, M. Bourdellès, and S. Dissoubray. Synccharts/esterel : Un environnement graphique pour la spécification et la programmation d'applications réactives complexes. In *Actes GL97*, number 46 in GL97, Paris (F), décembre 1997. Génie Logiciel.

[ABD98] C. André, H. Boufaïed, and S. Dissoubray. SyncCharts : un modèle graphique synchrone pour systèmes réactifs complexes. In *Real-Time Systems (RTS'98)*, pages 175–196, Paris (France), Janvier 1998. Teknea.

[ABGM96] C. André, H. Boufaïed, D. Gaffé, and J.P. Marmorat. Environnement pour la programmation synchrone des systèmes réactifs. In *Real-Time & Embedded Systems (RTS&ES'96)*, pages 27–41, Paris (France), Janvier 1996. Teknea.

[ABM+93] C. André, R. Bernhard, J-P. Marmorat, M-A. Péraldi, J-P. Rigault, and J-M. Tanzi. Etude d'une machine d'exécution pour le langage synchrone esterel. Technical Report CNET 91 5B 058, CERICS/CMA/I3S, Sophia-Antipolis, France, Mai 1993. Rapport Final, I3S TR 93–02.

[ABM+97] C. André, H. Boufaïed, J-P. Marmorat, J-P. Rigault, and J-M. Tanzi. Réalisation de machines d'exécution pour les langages parallèles synchrones. Technical Report Marché CNET 94 1B 111, I3S/CMA, Sophia-Antipolis, France, November 1997. Rapport Final.

[AG94] C. André and D. Gaffé. Evénements et conditions en grafcet. *A.P.I.I.*, 28(4):331–352, 1994.

[And96] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille (F), July 1996. IEEE-SMC.

[AP96] C. André and M-A. Péraldi. Predictability of a rtx2000-based implementation. *Real-Time System*, 10(3):223–244, May 1996.

[Bee94] M. von der Beeck. A comparison of STATECHARTS variants. In *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. FTRTFT'94, Springer-Verlag, 1994.

[Ber89] G. Berry. Real-time programming: Special purpose or general purpose languages. In 11^{th} *IFIP World Congress*, pages 11–17, San Francisco, 1989.

[Ber91] G. Berry. A hardware implementation of pure ESTEREL. Miami, January 1991. ACM Workshop on Formal Methods in VLSI Design.

[Ber92] G. Berry. Preemption in concurrent systems. *Proc FSTTCS, Lecture notes in Computer Science*, 761:72–93, 1992.

[Ber97] G. Berry. *The Esterel v5 Language Primer.* not yet published, available on the web, `www.inria.fr/meije/esterel`, Sophia Antipolis (F), 1997.

[Ber98] G. Berry. The foundations of Esterel. In C. Stirling G. Plotkin and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, 1998. To appear.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[Boo87] G. Booch. *Software engineering with* ADA. Benjamin/Cummings, California, 1987.

[Bou93] F. Boulanger. *Intégration de Modules Synchrones dans la Programmation par Objets.* PhD thesis, Supélec / Université de Paris-sud, Centre d'Orsay, Décembre 1993.

[Bou97] A. Bouali. XEVE: an ESTEREL verification environment. Technical Report draft, CMA - ENSMP, December 1997.

[BRS93] G. Berry, S. Ramesh, and R-K. Shyamasundar. Communicating reactive processes. 20ème ACM Conference on Principles of Programming Languages (Charleston, USA), January 1993.

[Bur84] Bureau d'orientation de la normalisation en Informatique. SCEPTRE: proposition de noyau normalisé pour les exécutifs temps-réel. *Techniques et Sciences Informatique*, 3:45–62, 1984.

[CCGJ96] B. Caillaud, P. Caspi, A. Girault, and C. Jard. Un modèle pour la répartition d'automates réactifs sur réseau asynchrone de processeurs. In Afcet, editor, *Modélisation des systèmes réactifs*, pages 343–350, Brest, Mars 1996.

[CHPR91] P. Caspi, N. Halbwachs, D. Pilaud, and P. Raymond. The synchronous data flow programming language LUSTRE. *Proceeding of the IEEE*, 79(9):1305–1320, September 1991.

[Dis96] S. Dissoubray. *Une approche synchrone pour l'intégration du contrôle.* PhD thesis, Université de Nice - Sophia Antipolis, Juin 1996.

[Dou98] B. P. Douglass. *Real-Time UML: developing efficient objects for embedded systems.* Object technology series. Addison-Wesley, Reading, Massachusetts, 1998.

[Ell91] J-P. Elloy. Les contraintes du temps réel dans les systèmes industriels répartis. *RGE*, (2):26–30, Février 1991.

[GMP+90] N. Ghezal, S. Matiatos, P. Piovesan, Y. Sorel, and M. Sorine. SYNDEX: un environement de programmation pour multi-processeur de traitement du signal. mécanismes de communication. Technical Report 1236, INRIA, Juin 1990.

[Hal93] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer Academic Publishers, Amsterdam, 1993.

[Har87] D. Harel. STATECHARTS: A visual formalism for complex systems. *Science of computer programming*, 8:231–274, 1987.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems in logic and models of concurrent systems. *NATO ASI Series, K.R Apt Ed., Springer-Verlag*, 13:477–498, 1985.

[HP88] D.J. Hatley and I.A. Pirbhai. *Strategies for Real Time Specification.* Dorset House, New York, 1988.

[HS91] W.A. Halang and A.D. Stoyenko. *Constructing Predictable Real-Time Systems.* Kluwer Academic Publishers, Amsterdam, 1991.

[LA90] S.T. Levi and A.K. Agrawala. *Real-Time System Design.* Mc Graw-Hill, New York, 1990.

[LGLL91] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceeding of the IEEE*, 79(9):1321–1336, September 1991.

[LL95] C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems*, volume 891 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1995.

[Mar90] F. Maraninchi. ARGOS*: un langage graphique pour la conception, la description et la validation des systèmes réactifs.* PhD thesis, Université Joseph Fourier, Grenoble I, Janvier 1990.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3), July 1983.

[PAB94] M-A Péraldi, C André, and H Boufaïed. Distributed synchronous processes for control systems. In *12th Workshop on Distributed Computer Control Systems*, pages 41–46, Toledo (Spain), September 1994. IFAC.

[Par92] J-P. Paris. *Exécution de tâches asynchrones depuis* ESTEREL. PhD thesis, Université de Nice-Sophia Antipolis, Juillet 1992.

[Per93] M-A. Peraldi. *Conception et Réalisation de Systèmes Temps-Réel par une Approche Synchrone.* PhD thesis, Université de Nice-Sophia Antipolis, Juillet 1993.

[Pot96] O. Potonniée. *Etude et prototypage en* ESTEREL *de la gestion de processus d'un micro-noyau de système d'exploitation réparti avec garantie de service.* PhD thesis, Université Pierre et Marie Curie, Paris VI, Avril 1996.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederic Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, 1991.

[SBM96] J-L. Scharbarg, J. Battut, and L. Marcé. Un automate programmable fondé sur l'approche synchrone du GRAFCET. In *Modélisation des Systèmes Réactifs*, Brest, Mars 1996. Afcet.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling.* J. Wiley Publ., 1994.