Chapitre IX- L'intégration d'une machine d'exécution Esterel.

Objectifs du chapitre :

- Permettre la modélisation des composants à contrôle prédominant par un langage adapté (Esterel : langage réactif synchrone).
- Démontrer un intérêt de l'approche modulaire par composants autonomes pour un environnement de simulation : intégration de plusieurs formalismes.
- Valider formellement le comportement de certains composants et ainsi réduire l'ensemble des tests nécessaires en simulation pour valider le modèle complet de l'architecture.

La modélisation de systèmes complexes nécessite très souvent l'utilisation de plusieurs formalismes **[Bal&co97]**. Cette nécessité doit donc être prise en compte par les environnements de conception et de simulation car aucun langage ne semble pouvoir modéliser efficacement tous les aspects d'un système complexe.

En particulier les langages synchrones **[Hal93]** comme Esterel **[Ber00]**, Lustre **[Cas&co87]**, Signal **[BeG90]**, SYNCCHARTS **[And96],** sont particulièrement bien adaptés à la description de systèmes réactifs à contrôle prédominant (cf. section 1-). Les langages impératifs classiques pour leur part offrent des possibilités intéressantes pour la gestion de structures de données. Les langages orienté-objets permettent l'encapsulation et la réutilisation de structures de données complexes. C'est pourquoi de nombreuses équipes proposent différentes approches d'intégration des aspects réactifs des langages synchrones dans le langage C (Reactive-C **[Bou91]**, ECL **[LaS98]**) , dans l'extension objet C++ (objets réactifs **[And&Co96]**) ou dans le langage Java (Sugar Cubes **[BoS98]**, Jester **[AnF99]**).

Ces langages offrent de puissantes primitives de modélisation du parallélisme et de la préemption. Ils permettent de modéliser efficacement les architectures logicielles destinées à des systèmes embarqués pour des applications réactives.

Nous avons vu notre proposition d'utiliser le langage Java pour la description d'architectures matérielles. Mais certaines parties des architectures sont principalement des parties de contrôle (décodeur d'instructions, contrôleurs de bus) et Java n'est pas le langage le mieux adapté à la description de ce type de comportement. En outre, notre approche de modélisation des architectures matérielles permet la description du comportement de composants indépendamment des autres composants. C'est pourquoi, ce chapitre a pour objectif d'introduire le mécanisme mis en place dans SEP afin de profiter de la puissance des langages synchrones pour les composants orientés contrôle **[MaB99]**. La contrainte que nous nous fixons est de ne pas remettre en cause les acquis.

Dans ce chapitre, la section 1- présente les hypothèses faites par les approches synchrones et les contraintes existantes pour l'intégration de ce paradigme dans un environnement asynchrone[1]. La section 2- montre comment la sémantique constructive par circuits d'Esterel nous permet d'intégrer une machine d'exécution **[Bou98]** dans un composant SEP. La prise en

[1] Ici le mot asynchrone est utilisé par opposition à la sémantique synchrone rappelée à la section 1-.

compte des tâches asynchrones Esterel est présentée dans la section 3-. En conclusion, les possibilités de validation de propriétés et le gain pour un environnement de simulation sont présentés.

1- Systèmes réactifs synchrones et machines d'exécution.

système réactif

Les insuffisances des systèmes transformationnels et des systèmes interactifs ont conduit à l'introduction de la notion de système réactif **[HaP85]**. Les systèmes transformationnels utilisent un ensemble de données d'entrée, pour effectuer un traitement sans interaction avec le monde extérieur. Après avoir fourni un résultat ils sont prêts pour un autre cycle. Les systèmes interactifs interagissent avec leur environnement mais le font à leur propre rythme. Par exemple, les interfaces graphiques et la plupart des systèmes d'exploitation sont des systèmes interactifs. Des événements trop rapprochés peuvent être perdus si le système est surchargé. Les systèmes réactifs, quant à eux, réagissent au rythme des événements émis par l'environnement. Tous les événements extérieurs sont pris en compte et c'est au système de choisir ceux qui ne seront pas traités, s'il est surchargé. Il fonctionne alors en mode dégradé.

instant

synchrone

Les langages synchrones modélisent les systèmes réactifs comme des machines d'états finis. Ils reposent sur une sémantique mathématique, rigoureuse, qui rend possible des validations formelles de propriétés de sûreté, de vivacité en utilisant des techniques de *model-checking* (cf. fin du chapitre). Ils considèrent un temps logique constitué d'une suite d'instants au cours desquels le système évolue. Dans ce cadre, deux événements sont simultanés s'ils sont présents dans le même instant. Un système est alors dit synchrone **[BeB91]** si, dans ce système, d'une part les réactions se font en temps (logique) nul et d'autre part les diffusions à l'intérieur du système sont instantanées. C'est-à-dire que les signaux émis pendant une réaction sont simultanés avec les signaux qui ont provoqué la réaction et qu'un signal émis est instantanément (dans le même instant) reçu par tous les destinataires avec la même valeur.

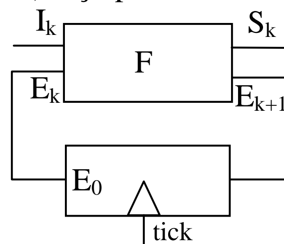

Figure 1- Machine d'états finis.

Sur la Figure 1, le système modélisé par cette machine d'états est dit synchrone si, le temps de calcul, de la fonction F qui calcule les sorties S_k et l'état suivant E_{k+1} en fonction des entrées I_k et de l'état courant E_k, est inférieur à la période du signal *tick*. Chaque front de montée du signal *tick* marque alors le début de l'instant k.

machine d'exécution

Les systèmes réactifs réagissant au rythme de l'environnement, c'est-à-dire au rythme des variations sur les entrées fournies par l'environnement, il est indispensable pour que l'hypothèse synchrone soit réaliste que les entrées du système soient stables au cours de l'instant. C'est ainsi que l'implémentation d'un modèle synchrone dans un environnement réel doit mettre en œuvre des mécanismes pour garantir cette contrainte. La machine qui va extraire par échantillonnage les entrées de l'environnement pour les donner au modèle synchrone et ainsi garantir que les entrées ne varient pas au cours d'un état, puis répercuter sur l'environnement les sorties fournies par le modèle, est appelée *machine d'exécution* **[Bou98]**. Notre travail et l'objet de ce chapitre consistent principalement à intégrer une

machine d'exécution dans un composant de l'environnement SEP. On réalise ainsi des composants synchrones.

Notons que nos composants synchrones sont différents des objets synchrones. En effet, nos composants sont représentés par des classes et pourront être instanciés plusieurs fois. Toutefois, une caractéristique importante des objets synchrones est leur dynamicité, c'est-à-dire leur aptitude à être instanciés, composés de façon synchrone et détruits dynamiquement. La composition dynamique de composants synchrones entraîne une diminution de la puissance d'expression standard des langages synchrones (i.e. pas de réaction instantanée à l'absence d'événements, détection dynamique des cycles de causalité). Dans SEP, nous modélisons essentiellement des architectures statiques, c'est-à-dire dont la topologie n'est pas modifiée en cours de simulation. De ce fait, la composition de modules synchrones peut être faite statiquement. En conséquence, la communication entre deux composants synchrones qui n'auraient pas été composés de façon synchrone est possible mais n'est pas synchrone. Il n'y a pas de diffusions instantanées entre deux composants synchrones, on peut considérer que la communication est synchrone faible. Sur la Figure 2, les entrées utilisées par le composant C2 à l'instant k – valides entre le $k^{ième}$ et le $k+1^{ième}$ front de montée du signal *tick* – sont les sorties du composant C1 à l'instant k-1 – émises entre le $k\text{-}1^{ième}$ et le $k^{ième}$ front de montée du signal *tick* –.

Cela s'explique très bien en reprenant la sémantique temporelle des services séquentiels : $\mathcal{F}_{so}(t_k) = H(\mathcal{F}_s(t_k - \varepsilon), \mathcal{F}_e(t_k - \varepsilon))$. Etant donné que les composants synchrones délivrent leurs sorties de façon synchrone avec l'horloge, elles ne varient pas entre deux *tick*s successifs. Ici ε est égal à la période de l'horloge *tick*.

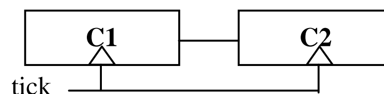

Figure 2 – Communication synchrone faible entre deux composants synchrones.

L'aspect de l'héritage de comportement à partir d'un composant synchrone n'a pas été abordé dans SEP. La notion d'héritage, définie dans le Chapitre II, n'a pas de sens ici car les composants synchrones ont un seul service (tick). Nous n'avons pas choisi d'implémenter l'héritage tel qu'il est défini pour les objets synchrones. En effet, il se limite au filtrage des entrées ou des sorties d'un module par interposition d'objets synchrones. Il s'agit pour nous de composition statique de modules synchrones. L'utilisateur qui désire réaliser l'héritage de modules synchrones devra le faire en utilisant les mécanismes fournis par le langage synchrone.

2- La sémantique constructive par circuits d'Esterel et le composant Strl.

Le langage Esterel est un langage impératif synchrone qui manipule des signaux purs (événement présent ou absent) ou valués (signaux purs auxquels est associée une valeur entière). Il permet de modéliser aisément le parallélisme et la préemption. Un programme Esterel est dit valide, s'il est possible au compilateur de trouver pour chaque variable locale ou de sortie, un et un seul état (présent ou absent, valeur unique) respectant la sémantique des instructions. Un programme Esterel pur valide peut donc être représenté par un ensemble d'équations booléennes. Un programme valide est dit constructif, si la validité peut être avérée en propageant uniquement des faits (signaux présents ou absents), c'est-à-dire sans faire aucune supposition qui pourrait être remise en cause par la suite. En fait, pour déterminer

la constructivité d'un programme, c'est-à-dire la valeur de toutes les équations booléennes qui représentent le programme, on ne peut pas se servir du raisonnement par l'absurde c'est-à-dire de l'assertion logique **'a ou non a est vrai'**. L'idée est que par la suite les programmes Esterel seront compilés en circuits logiques. Il est donc nécessaire que le circuit logique généré soit électriquement stable[2] indépendamment des délais éventuels dans les signaux ou dans les portes logiques. Dans **[Ber99],** G. Berry montre qu'un circuit logique est stable si et seulement si son programme Esterel équivalent est constructif.

SEP permettant la simulation d'architectures matérielles, nous nous proposons d'intégrer une machine d'exécution Esterel[3] dans un composant SEP en utilisant son équivalent en circuit logique. Le compilateur Esterel après vérification de la constructivité d'un programme génère un circuit logique équivalent. Le circuit logique produit peut l'être dans le format BLIF (Berkeley Logic Interchange Format)[4] ou dans le format SSC[5]. Le format BLIF ne permet de représenter que les programmes Esterel purs, c'est-à-dire qui ne contiennent que des signaux purs et aucune référence à une tâche asynchrone, alors que le format SSC permet de représenter tous les programmes Esterel.

Le compilateur vérifie la constructivité du programme mais la machine d'exécution doit gérer les tâches asynchrones ainsi que l'échantillonnage des entrées et la propagation des sorties vers l'environnement du système.

L'intégration dans SEP d'une machine d'exécution consiste alors à créer une classe de composants SEP (*component.basic.Strl*) qui sait lire les formats SSC ou BLIF, interpréter leur comportement et convertir les événements du monde SEP en signal Esterel. Cette classe représente les composants synchrones SEP (cf. Figure 3). Un tel composant est capable de fournir deux services. Le service *reset* commandé par le port de niveau 'reset' et qui permet d'initialiser les signaux de sortie et les signaux intermédiaires. Le service *tick* commandé par le port de niveau 'tick' et qui commande la réaction du système modélisé en Esterel. Cette réaction représente la réaction du programme Esterel au cours d'un seul instant. Les entrées sont photographiées pour s'assurer qu'elles seront stables au cours de l'instant, la valeur de chaque sortie au cours du même instant est calculée, puis propagée dans l'environnement SEP. Les services *reset* et *tick* sont donc soumis au même régime que tous les services commandés par un port de niveau. Les entrées et les sorties du module ne sont connues qu'après lecture du fichier BLIF ou SSC, les ports associés sont donc créés dynamiquement à partir d'un composant générique (cf. Chapitre IV). Deux sorties supplémentaires sont systématiquement fournies. La sortie *halt* indique la fin de l'instant Esterel et doit donc être présente après chaque réaction. La sortie *ret* indique la fin de l'exécution du module synchrone.

Le format BLIF permet de décrire deux types d'équations :

Les équations **logiques :** Ou, Et, Non
Les équations de **séquence** caractérisées par des bascules ('latches').

Le format SSC permet de plus de décrire deux types supplémentaires :

[2] Un circuit est électriquement stable si les sorties se stabilisent à un niveau logique lorsque les entrées sont fixées.
[3] Machine qui permet de convertir les événements entre le modèle Esterel et son environnement physique, et est capable de jouer son comportement
[4] http://kalliope.uni-trier.de/~wagner/html/bdd_html/docu/blif/blif.html
[5] Format propriétaire de l'équipe qui développe Esterel : http://www.esterel.org.

Les équations **arithmétiques** : addition, soustraction, comparaison.

Les équations de **saut** : appel de tâches asynchrones, ce mécanisme permet à un module synchrone de contrôler l'exécution d'un module asynchrone dont le comportement est décrit dans un langage compris par la machine d'exécution.

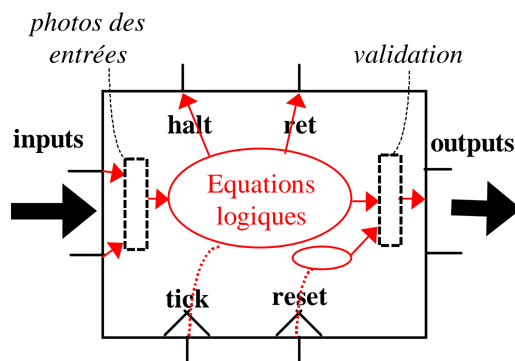

Figure 3 - Strl component

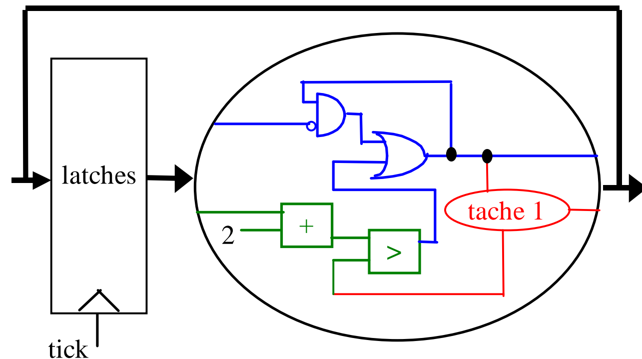

Figure 4 – Equations logiques

Les signaux purs sont représentés par un port et transporteront des valeurs de type LevelValue (High=présent, Low=Absent). Les signaux valués sont représentés par deux ports. Le premier port représente l'état c'est-à-dire la partie pure du signal. Le deuxième port représente la valeur entière associée et transporte des valeurs de type IntValue. Cette information sur les types est déduite lors de l'analyse des fichiers BLIF ou SSC et peut donc être utilisée dans la validation de la composition.

Il est possible de regrouper ces différents types d'équations en deux catégories. La première catégorie composée des équations logiques, arithmétiques et de saut constitue des blocs combinatoires de description. Tandis que la deuxième catégorie composée uniquement des équations de séquence permet de modéliser les instants de stabilité du programme Esterel et matérialise la commande synchrone du système. Cette classification est illustrée par la Figure 4.

Le cœur des composants Strl est constitué d'un système d'équations logiques qui est modélisé par l'interface *sep.sync.Archi*. Une *Archi* est en fait une liste ordonnée d'équations logiques[6] (interface *sep.sync.Equation*) mettant en relation des entrées (entrées du composant *Strl*), des sorties (sorties du composant *Strl*) et des signaux intermédiaires. Une *Archi* est évaluée (méthode *void evaluate ()*) par une évaluation unique de toutes les équations suivant un ordre calculé par le compilateur Esterel. La gestion du système d'équations logiques et de l'interaction avec l'environnement Sep est faite par la classe abstraite *sep.sync.ArchiAdapter*. La partie spécifique à la gestion des formats BLIF et SSC est faite respectivement par les classes *sep.sync.ArchiSsc* et *sep.sync.ArchiBlif*.

3- Gestion des tâches asynchrones.

Certaines équations logiques d'un fichier SSC consistent en l'exécution de tâches asynchrones. Ces tâches sont commandées à l'aide de cinq signaux de contrôle (start, activate, return, kill, suspend). Ces tâches peuvent avoir des paramètres modifiables ou non correspondant à des signaux intermédiaires, les paramètres sont validés par la présence du

[6] L'ordre est fourni par le compilateur Esterel. Il permet de choisir l'ordre d'évaluation des équations logiques afin d'obtenir le comportement souhaité. Chaque équation est évaluée une et une seule fois.

signal *start*. Les paramètres modifiables sont mis à jour et sont valides lorsque le signal *return* est présent. La tâche permet d'exécuter des actions asynchrones à partir du noyau synchrone Esterel. L'exécution de celle-ci n'est pas cadencée au rythme de l'instant Esterel, cependant le signal *activate* est présent à chaque instant. La fin de l'exécution d'une tâche est marquée par le signal *return* rendu présent par la tâche. Les signaux *kill* et *suspend* sont utilisés respectivement pour terminer ou suspendre la tâche asynchrone quel que soit son état d'avancement. Le signal *start* commande le début de l'exécution du comportement représenté par la tâche.

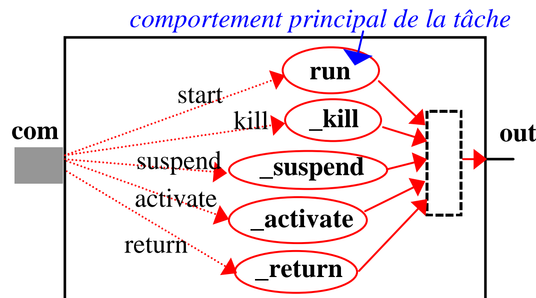

Figure 5 – Tâche asynchrone en Esterel.

Une tâche asynchrone est modélisée dans SEP (cf. Figure 5) par un composant avec cinq services (run, _kill, _suspend, _activate, _return) commandés par le port de service *com*. Ces composants sont définis par la classe *sep.sync.TaskStrl* qui hérite de la classe *sep.model.MaterialComponent*. On peut ainsi utiliser ces composants soit comme des composants standards, soit comme des tâches asynchrones commandées par un composant *Strl*. Le comportement des tâches étant complètement indépendant des autres composants SEP on utilise le mécanisme de tâches de Java (Thread) pour modéliser un composant *TaskStrl*. Le comportement principal de la tâche est décrit dans le service *run* (méthode run qui doit être redéfinie). Dans cette classe on dispose de certaines primitives qui permettent de récupérer les paramètres. En particulier, on dispose de la primitive *waitTick* qui utilise l'information du signal *activate* pour synchroniser la tâche asynchrone sur le début du prochain instant Esterel.

Un exemple illustrant l'utilisation d'un module Esterel dans un environnement asynchrone est présenté au chapitre XI. Cet exemple utilise des tâches asynchrones contrôlées par le module synchrone dont le comportement est spécifié en Esterel.

Il est intéressant de rappeler que grâce aux modèles rigoureux sur lesquels reposent les langages synchrones il est possible de valider formellement certaines propriétés. Les modules synchrones représentent des machines d'états finis. Les états sont représentés par des variables du système (signaux intermédiaires et sorties) booléennes ou valuées. Il est donc possible de vérifier une propriété sur le système en parcourant de manière explicite et exhaustive l'ensemble des états accessibles par un programme donné. Evidemment tous les états ne seront pas atteints systématiquement. Cela dépend des combinaisons successives des entrées au cours de l'évolution du temps logique. Ces techniques dites de *model-checking* **[CES86]** permettent d'établir des propriétés de sûreté : sous certaines conditions, il est impossible qu'une (mauvaise) action soit effectuée, ou de vivacité : il existe une combinaison d'entrées telle qu'une (bonne) action puisse se produire. Ces approches sont très rapidement soumises à une explosion de l'espace d'état, c'est-à-dire à un trop gros nombre d'états à parcourir quand le nombre de variables augmente. Toutefois, la taille des systèmes que l'on peut valider avec ces techniques a augmenté très nettement avec l'introduction des techniques de *model-checking* symbolique **[McM93]**. L'espace des états n'est alors plus explicitement

construit, il est représenté symboliquement par des équations logiques. Les bonnes propriétés de composition et de manipulation des arbres de décision binaires (BDD)[7] pour la représentation d'équations logiques permettent le traitement d'applications plus importantes. De nombreux outils accessibles dans l'environnement synchrone et qui utilisent cette technique permettent de valider les modèles Esterel par rapport à des propriétés exprimées en logique temporelle.

Le mécanisme d'intégration d'une machine d'exécution dans les composants SEP a été implémenté sans modification de l'environnement SEP et des composants existants. On montre ainsi que l'utilisation d'une méthode de modélisation basée sur l'utilisation de composants autonomes facilite l'encapsulation de nouveaux modèles de programmation dans un environnement de simulation. De plus, le modèle mathématique rigoureux sur lequel les langages synchrones reposent permet la validation de propriétés par utilisation des techniques de *model-checking* symbolique. La validation de ces propriétés permet de réduire l'ensemble des tests nécessaires en simulation afin de valider un modèle. Ainsi, l'utilisation de cette technique pour la validation de certaines parties orientées contrôle d'architectures matérielles assure localement un comportement sain. La simulation permet de valider ensuite le comportement du composant synchrone dans son environnement asynchrone.

Enfin, on peut noter que cette méthode d'encapsulation d'une machine d'exécution Esterel dans un composant a été réutilisée dans le cadre d'un travail sur l'utilisation et la validation de composants logiciels **[RaM00]**. Il s'agit d'aider l'utilisateur d'un *framework* logiciel à découvrir les services qui lui sont offerts. De plus, l'encapsulation d'une machine d'exécution Esterel dans le *bytecode* d'une classe Java rend possible la vérification du comportement par la machine virtuelle Java. En particulier, elle est alors en mesure de s'assurer dynamiquement que l'utilisation qui est faite des composants d'un *framework* est conforme à la spécification définie avec un syncChart[8] par le concepteur du *framework*. Ce travail ne rentre pas dans le cadre de cette thèse, le document présenté à un *workshop* lors d'une conférence est disponible en annexe.

[7] Les opérations logiques sur BDD sont peu coûteuses en temps de calcul et sont **en moyenne** peu coûteuses en terme d'occupation mémoire. En pratique dans la plupart des cas, selon certains ré-ordonnancements de variables, l'utilisation de BDD permet de représenter à moindre frais une équation booléenne.

[8] En effet, SyncCharts est la représentation graphique du langage Esterel. Un syncChart peut soit être compilé en Esterel, soit directement en équations logiques.