Modélisation orientée-objet d'architectures numériques matérielles/logicielles hétérogènes.
L'approche SEP.

F.Mallet, F.Boéri. Rapport de recherche numéro RR 99-06.
Laboratoire I3S, UPRES_A 6070 CNRS, Université de Nice-Sophia Antipolis.
E-mail : fmallet@unice.fr, boeri@unice.fr

1. Introduction.

La complexité des architectures numériques logicielles/matérielles augmente très rapidement. Un des problèmes majeurs des concepteurs d'architectures est donc de disposer d'outils de haut niveau permettant une conception et une évaluation rapide des performances pouvant être obtenues et une réutilisation maximale de tout ou parties des architectures précédemment validées. C'est pourquoi, notre objectif est de fournir aux concepteurs d'architectures un environnement de simulation d'architectures logicielles/matérielles numériques utilisant un modèle logiciel de haut niveau. La simulation permet d'estimer les performances d'une architecture matérielle/logicielle. De plus, la technique de modélisation objet utilisée permet au concepteur une réutilisation maximale des composants validés en mettant à sa disposition un environnement de programmation par composants. Le projet Simulation et Evaluation des Performances (SEP) vise à remplir ces objectifs.
La plupart des techniques classiques utilisent pour la description des parties matérielles les possibilités bas niveau de langages comme VHDL ou Verilog. Mais ces langages fortement typés et destinés à la synthèse d'architectures nécessitent souvent la prise de décisions secondaires et trop précises dès le début de la conception. De plus ces langages ne possèdent pas les bonnes propriétés de flexibilité et les facilités de réutilisation liées à la programmation orientée objet. Plus récemment des tentatives d'adaptation des techniques objets à ces langages ont été effectuées : OO-VHDL [ScN95] [BeD97]. Cependant ces langages restent destinés à la synthèse et nécessitent donc de nombreux paramètres souvent inutiles dans les phases de recherche ou d'amélioration comme par exemple, le choix des familles de composants, les temps de propagation, le choix de types particuliers.
Nous avons déjà proposé une technique [MBD98a] et un outil [MBD98b] afin de permettre une modélisation incrémentale permettant le choix d'un niveau d'abstraction adapté aux besoins du concepteur ainsi qu'une bonne réutilisation et flexibilité des modèles réalisés. Cet objectif avait été atteint grâce à la définition d'un modèle objet suffisamment générique pour ne dépendre que des caractéristiques générales des architectures visées. Le chapitre 2 propose un rappel du modèle générique et de son pouvoir de description et montre l'utilité et l'originalité d'un tel modèle par rapport à d'autres techniques.
Pour la description logicielle du comportement de parties du système deux 'écoles' se distinguent nettement. Certaines équipes s'intéressent de préférence aux structures de contrôle et aux aspects réactifs du comportement tandis que d'autres équipes ne s'intéressent qu'aux parties flots de données et aux traitements transformationnels. Des langages spécifiques adaptés à l'un ou à l'autre des points de vue existent et sont très performants : Esterel, Synccharts pour les parties réactives ; VHDL, Verilog, Lustre, Signal ou des langages impératifs classiques pour les parties transformationnelles. Cependant, les concepteurs manquent de moyens pour combiner les deux aspects efficacement en conservant les propriétés liées à leur spécificité. La généricité de la description de notre modèle objet nous permet de relier aisément les deux approches. D'autres approches similaires sont en cours d'étude (Intégration de la sémantique constructive d'Esterel dans les CFSMs du projet POLIS [BeS98]), notre position par rapport à ces projets sera discutée dans le paragraphe 2.4. Le chapitre 3 décrit la technique de simulation utilisée pour l'ordonnancement des réactions aux événements et la propagation des données. Puis, le chapitre 0 présente notre proposition pour l'intégration de modèles hétérogènes synchrones et asynchrones dans un environnement de modélisation de haut niveau. Pour finir, le chapitre 5 illustre l'utilisation de cette méthode sur un exemple de système de contrôle d'un arrosage automatique.

2. Modèle objet générique.

2.1 Pourquoi un modèle générique ?

Notre modèle objet générique constitue un environnement d'intégration d'architectures numériques. Le mot générique signifie que l'on ne fait pas d'hypothèses relativement au domaine d'application des architectures numériques modélisées. On peut aussi bien modéliser des processeurs de tout type (RISC, DSP, CISC, VLIW) que des systèmes de contrôle ou des architectures multiprocesseurs voire distribuées. Ce modèle consiste en fait en la définition d'une interface minimale de communication entre les différents composants du modèle. Cette interface minimale permet alors une propagation des événements et données entre les composants d'une façon indépendante du système d'exploitation ou du langage utilisé. Ce modèle générique permet d'une part la description du comportement des composants élémentaires et d'autre part l'agencement de ces composants ainsi que des voies de communication (signaux de contrôle, bus de données). Les différents composants peuvent alors être conçus de manière autonome et sont donc parfaitement réutilisables. Nous devons alors assurer au concepteur la non ambiguïté de la description des composants afin qu'elle ne soit pas dépendante de l'agencement. L'interface minimale de chaque composant est illustrée par la Figure 1. Tout composant à des ports de communication qui garantissent l'autonomie de sa description. Le comportement est alors encapsulé dans le composant est commandé par les ports.

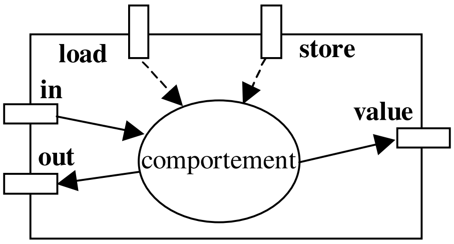

Figure 1 – Interface du registre Load/Store

Le cœur de ce modèle générique est décrit par les classes du paquetage modele.simulation dont les différentes associations sont décrites en détail dans [MBD98a].

Un environnement graphique mettant en avant les propriétés de ce modèle a été créé par l'ajout de nouveaux paquetages essentiellement liés à l'interface homme machine (modele.awt, gui, modele.awt.event) [MDB98b].

De plus, la prise en compte des possibilités des langages synchrones a été effectuée par l'ajout d'un nouveau paquetage (modele.sync) qui est construit sur la base du cœur.

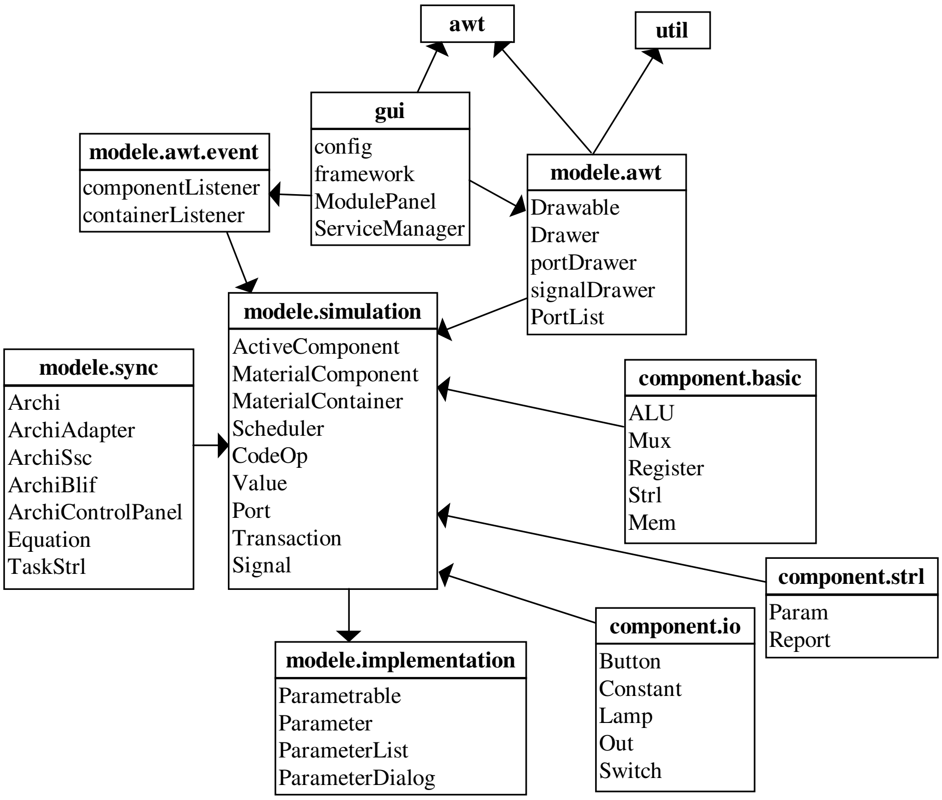

Figure 2 - modèle statique de paquetage

Enfin une bibliothèque de composants réutilisables a été constituée par l'ajout de nouveaux paquetages (component.basic, component.io, component.misc). Cette bibliothèque peut évidemment être complétée.

La Figure 2 présente le modèle de paquetage actuel constituant l'environnement de modélisation et de simulation proposé. Ce modèle de paquetage est décrit à l'aide du langage de modélisation UML. L'utilisation d'une modélisation objet pour la construction de ce modèle permet une évolution et un enrichissement simples.

2.2 Comportement : Modèle structurel.

La première manière de décrire le comportement est de faire une description structurelle. Les composants dont la réaction est décrite à l'aide du modèle structurel sont des conteneurs (modele.simulation.materialContainer) dont la hiérarchie est décrite par le modèle récursif de relation conteneur – contenu (Figure 3).
Un composant est un objet pouvant communiquer à travers des ports et ayant un certain comportement. Ce comportement peut être décrit par l'agrégation d'autres composants plus simples eux-mêmes reliés entre eux et dont les entrées/sorties constituent les entrées et les sorties du conteneur. L'objectif est de rassembler des composants collaborant à la réalisation d'une fonctionnalité commune et de haut niveau suffisamment générique et autonome pour pouvoir être réutilisée.
Il apparaît sur ce modèle que la profondeur d'imbrication est a priori infinie. De plus, materialComponent étant une classe abstraite, il apparaît aussi que toutes les feuilles de l'arbre d'imbrication doivent être du type ActiveComponent, c'est-à-dire que leur comportement doit être décrit à l'aide d'un langage de programmation.

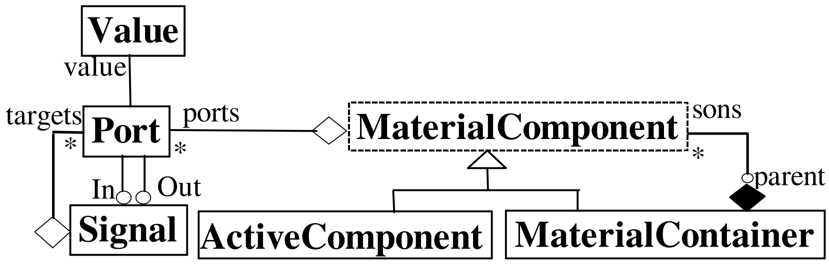

Figure 3 - Modèle récursif de relation conteneur - contenu

Ces composants autonomes présentent une interface avec l'extérieur sous la forme d'une liste de ports. Ces ports possèdent une valeur (Value) et sont connectés entre eux à l'aide de signaux. Tout port peut avoir au plus une voie de communication (Signal) vers l'extérieur : Out. Les ports des conteneurs constituent simplement une interface entre les composants extérieurs reliés au signal Out et les composants contenus reliés au signal In. Les signaux peuvent délivrer des événements ou des données vers plusieurs ports cibles par une technique de diffusion (cf. chapitre 3).
La Figure 4 présente une illustration d'un registre load/store, dont le comportement est modélisé à partir d'une description structurelle.

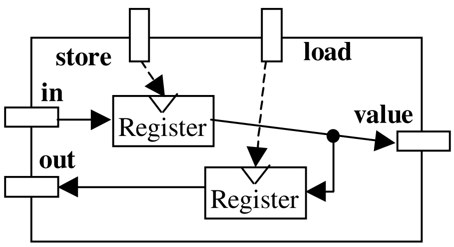

Figure 4 – Le registre load/store: Une description structurelle.

2.3 Comportement : Modèle logiciel.

La généricité du modèle objet permet une représentation multi-formalisme du comportement. Il suffit de respecter l'interface élémentaire de communication. Le paragraphe précédent explique comment il est possible d'utiliser un formalisme structurel. Ce paragraphe montre qu'il est aussi possible de décrire le comportement d'un composant à l'aide d'un modèle logiciel en utilisant un langage de programmation.
L'implémentation de l'environnement de programmation que nous proposons a été faite dans le langage Java. Ainsi il est naturel d'utiliser Java comme langage de description du comportement. Celui-ci nous permet aussi

bien une programmation de type impérative qu'orientée objet. Afin d'utiliser Java comme langage de description du comportement, un certain nombre de primitives de communication à travers l'interface constituée de ports ont été définies. Ces primitives seront détaillées ultérieurement.
Nous avons déjà dit que le langage Java pouvait être adapté à la description de comportements de type transformationnel, il l'est beaucoup moins pour les comportements dits réactifs. Un langage comme Esterel est alors plus adapté. C'est pourquoi, nous avons développé une implémentation de l'interface élémentaire définie par notre modèle objet pour le langage Esterel. En fait, il est possible de respecter la sémantique d'Esterel avec une description à l'aide d'un ensemble de portes logiques et de bascules. Ces résultats ont été démontrés par les travaux sur la sémantique constructive électrique d'Esterel dans [Ber98]. Or nous avions jusqu'à présent un environnement de simulation d'architectures numériques. Nous avons alors développé une passerelle entre les équations générées par le compilateur Esterel et notre environnement de simulation. Pour des raisons d'efficacité, nous n'avons pas instancié un composant pour chaque porte logique, mais un seul composant pour chaque module Esterel.
Les bonnes propriétés du programme Esterel sont vérifiées à l'aide de l'environnement classique de développement Esterel. La solution du programme Esterel est construite en simulation par propagation des faits sous forme de niveaux logiques. L'ordre d'ordonnancement et d'évaluation des équations est donné par le compilateur Esterel, en effet celui-ci génère une liste ordonnée d'équations logiques. Cependant, notre environnement de simulation aurait convergé de la même façon en supposant qu'il y ait une solution unique, ce qui est vérifié par le compilateur Esterel.
Une illustration du registre load/store décrit en utilisant le language Java est présentée par l'Exemple 1.

```
package component.basic;
public class LSRegister extends modele.simulation.EdgeComponent {
    public LSRegister() {
        addInactivePort("in", LEFT);
        addInactivePort("out", LEFT);
        addRisingEdgePort("load", TOP).setService("load");
        addRisingEdgePort("store", TOP).setService("store");
    }

    public void load() {
        setPriority(1);
        emit("out", read ("value"));
    }
    public void store() {
        setPriority(0);
        emit("value", read ("in"));
    }
}
```

Exemple 1 – Le registre Load/Store: description Java.

2.4 *Les autres approches.*

POLIS est un environnement de conception conjointe logicielle/matérielle de systèmes embarqués basé sur un modèle de programmation formel appelé CFSMs (Codesign Finite State Machine). Ce modèle proposé par l'université de Berkeley, Californie est basé sur une communication entre des tâches synchrones représentées par des machines d'états finis et formalisées en utilisant le langage synchrone Esterel. La communication entre les différentes tâches synchrones est faite avec des délais non bornés. Les objectifs énormes qui avaient été définis ne sont pas encore atteints, ils ont étaient réduits pour se limiter aux systèmes orientés contrôle. Le cycle complet de conception prévoyait de partir d'une spécification fonctionnelle la plus formelle possible, puis de plaquer la description des unités fonctionnelles sur les architectures matérielles disponibles. Ce qui permet alors d'effectuer une évaluation des performances de l'application construite. Cette approche est prévue pour être la plus formelle possible afin de garantir une implémentation correcte par construction. La validation que ce soit par simulation ou par vérification doit intervenir au plus haut niveau d'abstraction possible. Pour la simulation POLIS fait appel à Ptolemy qui est aussi un outil proposé par Berkeley. Ptolemy est un environnement de co-simulation d'architectures hétérogènes. Cet environnement est supposé permettre l'intégration de divers modèles de programmation afin d'effectuer des simulations hétérogènes. Dans ce but, l'utilisateur doit définir des passerelles

(worm-hole) entre les différents domaines qu'il souhaite utilisé. En pratique, seules quelques passerelles avaient été écrites et l'environnement était difficilement accessible pour des utilisateurs extérieurs.
Bien qu'ayant les mêmes objectifs notre point de vue a été un peu différent, c'est pourquoi nous sommes parvenus à une solution légèrement différente qui sur de nombreux points converge. En effet, nous avons donné la préférence à la conception des architectures matérielles et donc impliquant des calculs transformationnels de préférences aux systèmes de contrôle. Avec cet objectif, il nous a semblé intéressant de partir d'une description orientée objet par composants plutôt que d'une approche fonctionnelle. Cette approche objet devait nous permettre d'améliorer la réutilisabilité et la flexibilité des modèles, ce qui est le cas pour les exemples étudiés et présentés dans des articles antérieurs. Par notre approche, nous souhaitions permettre une simulation rapide d'un modèle construit de façon incrémentale et possédant un niveau d'abstraction adapté. Nous nous sommes donc intéressés en priorité aux aspects de validation par la simulation avant de nous intéresser à la vérification par preuve. De plus, la nature transformationnelle des applications visées et l'utilisation d'une technique de modélisation objet nous ont conduit à choisir Java comme langage de description des composants. Cependant, grâce à la flexibilité du modèle générique ainsi créé, il a été facile d'intégrer la possibilité de décrire les composants avec un modèle synchrone Esterel. L'intégration d'un autre modèle de programmation est alors rendu facile et passe par l'écriture d'un composant abstrait pour servir de modèle et de classe de base aux autres composants. Ces composants mettront en œuvre l'interface minimale de communication inter-composants en adaptant dans la mesure du possible le nouveau modèle de programmation à celui par défaut.
Ceci a été rendu possible grâce à la description détaillée du modèle générique et la description du modèle de classes et de paquetages. Cette description a aussi été effectuée depuis peu par le projet Ptolemy2 qui a été marquée par l'utilisation du langage Java en préférence du langage C++.

Des outils commerciaux tels que CARDtools (computer aided real-time design tools) et Coware sont des outils graphiques de conception et de simulation conjointes. Ils permettent la description d'un système complet indépendamment de l'implémentation et fournissent des résultats intéressants à partir d'une simulation. Ils utilisent une technique similaire de regroupement en modules et sous modules reliés par des signaux. Cependant dans ces outils, la description du comportement et l'interaction entre les différents modules est basée sur des langages impératifs (VHDL, C, TBM). Ils ne fonctionnent donc pas avec une bonne autonomie ce qui entraîne des limites dans la réutilisation. En effet, les descriptions fonctionnelles entraînent la plus part du temps, des références aux modules proches pour la description du comportement d'un module. Ceci implique que le module ainsi créé dépendant de son environnement et ne pourra donc être réutilisé que dans le même type d'environnement.

Avec notre méthode, nous voulons démontrer que l'utilisation de techniques objets et basées sur une programmation à partir de composants pour la modélisation et la simulation d'architectures numériques. La généricité et la flexibilité de notre modèle nous a permis une rapide extension du modèle et la prise en compte d'un nouveau domaine d'application. En effet, nous avons pu intégrer l'utilisation d'un modèle réactif synchrone par l'encapsulation d'une machine d'exécution Esterel dans un de nos composants.

3. Simulation par propagation de données.

3.1 Environnement de simulation.

L'environnement de simulation proposé permet l'agencement des différents composants qui constituent le système étudié et de leurs voies de communication. On peut alors y ajouter le contexte de simulation, c'est-à-dire une ensemble de composants extérieurs au système qui fournissent les entrées ou stimulations et qui permettent éventuellement la récupération des résultats. Ces excitations extérieures peuvent être aussi bien une horloge pour les systèmes synchrones que des actionneurs ou des capteurs pour les systèmes logiciels. La simulation du système consiste alors à le faire réagir de façon cohérente par rapport aux stimuli de son contexte et à observer des propriétés de réaction du système par rapport au scénario des stimuli. Les stimulus engendrés par le contexte définissent les instants de simulation. Deux stimulus sont toujours supposés non simultanés c'est-à-dire que l'on considère que le système fonctionne en mode fondamental. S'ils sont non simultanés, on peut donc les ordonner, on effectuera alors la réaction provoquée par le premier arrivé, puis au deuxième. Attention, il est tout de même possible de construire des réactions simultanées au niveau macroscopique. Il suffit pour cela de créer un seul stimuli d'entrée qui représente une horloge. Cette horloge engendrera plusieurs données pouvant provoquer deux réactions simultanées.
La simulation ne permet de tirer que des conclusions partielles dépendantes des entrées. Dans le cas d'un système ne réagissant qu'à une horloge comme un processeur, le scénario simulé est de fait, l'unique scénario possible puisque toutes les entrées (horloge, programme et instants de réaction) sont connues. Il faut cependant supposer

que les temps de propagation non explicités ne sont pas un obstacle au bon fonctionnement du système. Pour ce dernier point il est de la responsabilité du concepteur de choisir des composants physiques et une fréquence d'horloge afin qu'ils soient adaptés et d'inclure dans le modèle tous les paramètres pertinents. En spécifiant les temps de propagation critiques de ses composants réels, le concepteur pourra mettre en évidence sur les chronogrammes les délais éventuellement non respectés. Dans le cas où tous les temps de propagation seraient connus et que l'on posséderait une description complète du système de niveau RTL, il serait conseillé d'utiliser une méthode de simulation classique à base de VHDL ou de Verilog. En effet, notre modèle est mieux adapté pour la phase de conception par utilisation d'un modèle flexible de haut niveau.
Dans le cas d'autres systèmes, par exemple les systèmes réactifs, la simulation ne permet d'étudier qu'un nombre fini non exhaustif de scénarios. On ne peut donc pas obtenir une garantie de bon fonctionnement du système indépendamment du contexte, il faudrait pour ce faire, utiliser un outil de preuves formelles. On peut tout de même se convaincre que les scénarios les plus courants respectent un certain nombre de bonnes propriétés. Dans ce but, on peut utiliser des observateurs dynamiques qui vérifient le respect de certaines règles. Un observateur dynamique est un composant normal qui par exemple vérifie dynamiquement à la simulation qu'une valeur sur un bus appartient toujours à un certain domaine de valeurs ou qu'une succession de valeurs de plusieurs signaux vérifie un certain ordre partiel.

3.2 Composants distribués : sensibilité des ports de communication.

Les composants sont entièrement autonomes, ils sont conçus indépendamment les uns des autres et ne se connaissent pas. Toutes les connexions sont donc autorisées mais certaines peuvent être déraisonnables. Il serait bon tout de même de se limiter et de pouvoir aider l'utilisateur à se limiter aux connexions raisonnables. Cet objectif peut facilement être atteint par une technique d'assertion [Col98]. Une technique d'assertion permettrait d'identifier les connexions pertinentes ou de vérifier que certains domaines de valeurs sont compatibles. L'implémentation d'une technique similaire peut faire l'objet de travaux ultérieurs.
Lors de la communication entre plusieurs composants, les données émises peuvent être interprétées de façons différentes suivant les récepteurs en fonction de la sensibilité de leurs ports. C'est toujours le composant cible qui décide de l'utilisation qu'il fait des données qui lui parviennent du monde extérieur. Dans ce but, une sensibilité est associée aux ports, cette sensibilité déclenche l'exécution du comportement associé à un service (méthode) du composant cible. Il y a trois catégories différentes de composants. Les composants séquentiels peuvent avoir des ports sensibles sur front : modele.simulation.EdgeComponent. Pour ces composants l'ordre relatif de réaction des différents services est important, il ne doit donc pas dépendre de l'ordonnancement des données. Les composants combinatoires peuvent avoir des ports sensibles sur niveau : modele.simulation.LevelComponent. Pour ces composants, l'ordre d'exécution des services n'est pas important. Quel que soit l'ordre d'ordonnancement, on doit obtenir le même résultat final même s'il est possible que les résultats transitoires soient différents. En effet, dans un système combinatoire la sortie ne dépend que des entrées. Enfin les composants sensibles au temps n'ont aucun port sensible : modele.simulation.TimeComponent. Ces composants réagissent spontanément au temps qui passe. Ils permettent de représenter les phénomènes continus comme une pompe ayant un débit particulier, une horloge ayant une fréquence connue. Ces composants annoncent une sensibilité temporelle qui permet de rendre discret leur comportement afin d'établir une relation temporelle cohérente entre les différents TimeComponent et avec les autres composants.

Grâce à l'utilisation de l'introspection, on peut encapsuler dans l'objet Port toute la gestion du service qui lui est associé en utilisant éventuellement des ressources communes aux autres ports du composant concerné. En effet, il suffit de spécifier à un port le nom du service qui lui est associé ainsi que sa sensibilité comme le montre l'exemple suivant :

add**RisingEdge**Port("rst", TOP).**setService**("reset");

Ici, on définit le port "rst" comme étant un port sensible sur front montant associé au service "reset". Dans ce cas, la réception en séquence des valeurs LOW (niveau bas) et HIGH (niveau haut) provoquera la réaction du composant associé. Cette réaction est décrite par la méthode "reset" du composant correspondant au nom du service déclaré : setService("reset"). Si aucun service n'est déclaré, le service par défaut porte le nom "report".

En dehors de ports sensibles sur front (modele.simulation.EdgePort), il existe deux autres types de sensibilité pour les ports. La sensibilité sur niveau (modele.simulation.LevelPort) correspond à la sensibilité des composants combinatoires. La sensibilité de service (modele.simulation.ServicePort) correspond à une sensibilité sur niveau particulière. En effet, dans ce dernier cas, c'est la valeur transportée qui détermine le service concerné. Il permet en fait de multiplexer plusieurs ports sensibles sur niveau qui collaborent à un même objectif. L'utilisation de ce type port sera illustrée dans le paragraphe 4.3.

La Figure 5 illustre l'utilisation de cette notion de service pour rendre modulaire la description des différents comportements d'un composant. Le code complet du composant Registre illustré sur cette figure est présenté au paragraphe 3.4.

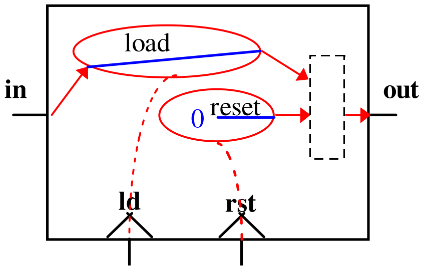

Figure 5 - Registre

Ces services utilisent deux primitives élémentaires pour communiquer à travers les ports : emit et read. La primitive emit permet l'émission d'une valeur sur un canal de transmission (port) : **emit**("out", 12). La primitive read permet la lecture de la valeur présente sur un port : **read**("in"). Suivant le type des composants, la réalisation des actions de lecture ou d'émission peut être assez complexe. Ce mécanisme ainsi que la différence de réalisation sont totalement transparents pour l'utilisateur grâce au polymorphisme.

La description d'un service peut être effectuée de la manière qui convient à l'utilisateur. En particulier, il est possible de décrire tout en Java pour les comportements transformationnels, mais il est aussi possible de décrire le service dans d'autres formalismes en développant des passerelles. Par exemple, pour la description d'un service localement synchrone ou réactif, on peut utiliser le langage esterel comme le montre le chapitre 0. On récupère alors localement toutes les propriétés du langage esterel en particulier la possibilité de preuve et la sémantique rigoureuse. **C'est l'utilisation d'un langage objet et d'une technique de simulation par composants qui a permis l'intégration rapide du langage esterel sans aucune modification du modèle alors qu'il n'avait pas été conçu pour cela.**

3.3 Planification et propagation des données.

Le système réagit aux événements extérieurs avec une technique data-driven. Ce n'est pas une technique demand-driven qui permet de n'exécuter que les parties utiles mais qui retarde le traitement et concentre les exécutions en entraînant des exécutions plus complexes. Une technique data-driven permet de ne faire réagir que les composants qui ont des opérations à effectuer contrairement à une technique cycle-based. Afin de conserver l'autonomie des composants nous n'avons pas utilisé une technique event-driven. En effet, chaque composant émet des données vers le monde extérieur et c'est le récepteur qui détermine si cette donnée, ou éventuellement cet ensemble de données constitue un événement et nécessite une réaction. Par exemple, un composant peut émettre une séquence de niveaux haut et bas qui peuvent être interprétés différemment suivant les récepteurs. On peut les interpréter comme une suite de niveaux, comme une impulsion, comme une occurrence de fronts montants ou descendants ou éventuellement les ignorer. Un ensemble de données est transformé en événement en fonction de la sensibilité des ports du composant destination.

Lorsqu'un événement extérieur se produit, des propagations de valeurs sur des signaux sont planifiées. Ces valeurs peuvent exciter un composant actif en attente de certains types d'événements, ce qui peut engendrer la planification de nouvelles données. On propage ainsi les réactions jusqu'à l'obtention de la stabilité du système modélisé. La stabilité sera obligatoirement obtenue même si le système comporte des cycles, à condition qu'un nombre fini de valeurs par signal soient autorisées entre deux événements extérieurs du contexte (démonstration par l'absurde). Cette condition est satisfaite en utilisant une valeur spécifique déterministe appelée 'Undefined' dans le cas où l'ordonnanceur devrait planifier pour un même signal deux valeurs valides et différentes. Le signal ne pourra alors être affecté à une valeur valide différente de Undefined qu'à l'instant suivant, c'est-à-dire au plus tard lors de l'occurrence du prochain événement extérieur. Attention la stabilité de la simulation n'est pas équivalente à la stabilité du système réel. Dans le cas d'un système oscillant en temps physique nul, c'est-à-dire sans délai de propagation spécifié, la simulation convergera vers la valeur Undefined et le concepteur pourra ainsi identifier le problème. Si le système oscille et que des délais de propagation sont spécifiés, le système sera simulé pour une durée définie par l'utilisateur et un chronogramme mettra en évidence l'oscillation.

Il est important de laisser l'environnement de simulation réagir jusqu'à la stabilité, en effet nous supposons le mode fondamental et on ne peut donc prendre en compte qu'un seul stimulus provenant du contexte de simulation à la fois. Dans le cas où l'utilisateur n'attendrait pas la stabilité, l'environnement de simulation garantie l'atomicité des réactions mais pas la réactivité. C'est-à-dire que certains événements pourront être

perdus. Il ne s'agit pas de prouver que l'environnement de simulation est parfaitement réactif (ce qui peut ne pas être vrai pour de gros systèmes), mais il s'agit de se convaincre que le système simulé ou plutôt que le modèle du système étudié possède un comportement réactif. Il faut pour cela laisser le temps à l'environnement de simulation de réagir à l'occurrence des événements extérieurs. Même si la réaction à ces événements extérieurs est supposée être instantanée, le simulateur nécessite un temps non nul pour les simuler puisqu'il n'y a pas de phase de compilation. Cependant, il pourra tout de même afficher un chronogramme présentant des réactions instantanées et des résultats simultanés.
Certaines réactions peuvent être définies comme n'étant pas instantanées c'est-à-dire utilisant des délais de propagation. Il s'agit pour l'ordonnanceur de respecter tout de même un ordre partiel sur la prise en compte des différentes données. Toutes les données planifiées à une même date seront activées dans le même instant logique en respectant l'ordre et les dates d'ordonnancement.
L'ordonnanceur (**scheduler**) fonctionne suivant l'algorithme qui suit défini comme une des attributions de la classe ***modele.simulation.Scheduler*** :

```
Incrémenter le temps logique.
Tant qu'il y a des opérations planifiées
    liste <= opérations à la date courante.
    Si le temps physique doit évoluer pour qu'on puisse progresser, il faut propager
    l'évolution vers les composants sensibles au temps.
    Pour tous les éléments op de liste
        calculer l'effet de l'opération op au temps logique
            et propager l'effet des composants non sensibles sur front (Résoudre
            les conflits lorsque deux données sont propagées sur le même signal)
        Mettre les cibles sensibles sur front dans la liste toBeExecuted
    fin Pour
    Pour toutes les opérations op de la liste toBeExecuted
        Valider l'exécution de l'effet pour op.
    fin Pour
fin Tant que
```

Remarque la validation des effets aussi bien que la propagation peut planifier d'autres données qui seront exécutés sans modification du temps logique, ni du temps physique.

La validation du comportement des composants sensibles sur front est effectuée en appelant la méthode validate de chaque EdgeComponent qui a joué un rôle. Pour ces composants la réaction se passe en deux phases. Lors de la première phase, chaque composant supposé réagir sur front fait une photo de ses entrées et calcule la réaction. Celui-ci peut avoir à calculer plusieurs réactions si plusieurs services sont appelés simultanément. Puis lors de la deuxième phase, la propagation résultant de la réaction la plus prioritaire est effectivement planifiée, c'est le comportement réalisé par défaut par la méthode validate. Afin de définir le niveau de priorité du service, on doit utiliser la primitive '**setPriority**' illustré lors de l'exemple du paragraphe 3.4. Par défaut la méthode validate suppose que les différents services n'effectue pas d'effets de bord sur l'état interne du composant. Dans le cas contraire, l'utilisateur doit réaliser sa propre politique de priorité. Une politique particulière a par exemple été développée pour la création du composant Strl (cf. chapitre 4.1).

3.4 Utilisation de l'héritage pour une conception plus rapide.

Nous avons déjà entrevu la description d'un registre, il est assez simple d'utiliser la technique d'héritage pour construire un nouvel objet de type registre à décalage.

Ce registre possède un service 'reset' qui permet la mise à zéro de la sortie. Un service 'load' qui permet de charger la valeur de l'entrée sur la sortie et de la mémoriser. Cette valeur est de type quelconque héritant de la classe Value. Elle peut aussi bien être un bit, le registre se comporte alors comme une bascule D, qu'un entier, une chaîne de caractère ou une instruction assembleur.

```
package component.basic;
public class Register extends modele.simulation.EdgeComponent {
    public Register() {
            addInactivePort("in", LEFT);
            addInactivePort("out", RIGHT);
            addRisingEdgePort("rst", TOP).setService("reset");
            addRisingEdgePort("ld", BOTTOM).setService("load");
    }

    public void reset() {
            setPriority(1);
            emit("out", 0);
    }

    public void load() {
/* On donne la priorité au reset */
            setPriority(0);
            emit("out", read ("in"));
    }
}
```

Un service fait référence à la méthode appelée lors de l'excitation du port.

Un registre à décalage possède en plus le service de décalage pour décaler la valeur mémorisée d'un bit sur la droite. Ceci est toujours possible car tous les objets qui héritent de la classe abstraite Value peuvent être décomposés sous forme de bits. Pour un entier signé, on utilise la notation classique, le bit de poids fort détermine le signe et on utilise la notation binaire pour coder la valeur absolue. Pour les instructions assembleurs, on peut considérer que la décomposition sous forme de bits correspond au codage du langage machine associé. Après le décalage le nouveau bit de poids fort est fixé en utilisant la valeur du port 'cin' supposée représenter un bit. La valeur de l'ancien bit de poids faible est alors affectée au port 'cout'. C'est un registre à décalage vers la droite possédant une entrée 'in' et une sortie 'out' parallèle commandée par le service 'load' ainsi qu'une entrée 'cin' et une sortie 'cout' série commandée par le service 'shift'.

```
package component.basic;

public class ShiftRegister extends Register {
    private int size;
    public ShiftRegister(int size) {
            super();           // Les port "in", "out", "rst", "ld" sont hérités.
            this.size = size;
            addRisingEdgePort("shr", BOTTOM).setService("shift");
            addInactivePort("cout", RIGHT);
            addInactivePort("cin", LEFT);
    }
    public ShiftRegister() { this (8); } // 8 est la taille par défaut du registre à décalage

    // Les services load et reset sont hérités.

    public void shift() {
            setPriority(-1); // Moins prioritaire que le load et que le reset.
            IntValue content = read("out");              // content = valeur courante
            emit("cout", content.trunc(1).getLevelValue());    // cout <= bit de poids faible
            content = content.shr().add(read("cin").getIntValue().shl(size-1)); // Effectue le décalage
            emit("out", content);      // out <= ancienne valeur décalée avec comme bit de poids fort cin.
    }
}
```

Pour réaliser le registre à décalage à partir d'un registre normal, on peut donc utiliser une technique d'héritage. Il suffit en effet de rajouter 3 ports ('cin', 'cout' et 'shr') et le service 'shift'. Ce service peut se matérialiser par l'ajout d'une méthode en effet on hérite les méthodes load et reset. Ce service doit avoir une priorité inférieure aux deux autres services.

Le code du registre à décalage en réutilisant au maximum le code du composant registre est présenté ci-dessus : ShiftRegister.

4. Implémentation des modèles synchrones dans l'environnement asynchrone.

4.1 Le composant Strl

L'objet Strl est un composant dont les réactions sont synchrones sur les fronts montants du signal tick. Les objets de ce type héritent de la classe ***component.basic.Strl***. Tout composant a un certain nombre d'entrées et de sorties. Il a de plus, une entrée pour un reset asynchrone et deux sorties particulières qui marquent la fin d'un instant (halt) et la fin de l'exécution de l'instruction synchrone (ret). Le comportement de ce composant est décrit en utilisant le langage synchrone esterel. Ce composant constitue en fait une «machine d'exécution » d'un programme synchrone. Afin de conserver les bonnes propriétés de preuves d'esterel, on n'utilise en réalité que les équations logiques générées par le compilateur esterel pour modéliser le comportement. Ces équations logiques peuvent être décrites sous différents formats, ceux actuellement acceptés pour la description du comportement de l'objet Strl sont SSC et BLIF. Il y a quatre différents types d'équations valides :

- Les équations **logiques :** Ou, Et, Non
- Les équations de **séquence** caractérisées par des bascules ('latches').
- Les équations **arithmétiques** : addition, soustraction, comparaison.
- Les équations de **saut** : appel de tâches externes.

Les deux derniers types (arithmétiques et saut) ne sont pas acceptés par le langage blif. En effet, ce dernier ne peut être utilisé que pour les programmes esterel synchrone pur sans signaux valués ni tâches. En fait, 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 un bloc 'combinatoire' de description. Tandis que la deuxième 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 7.

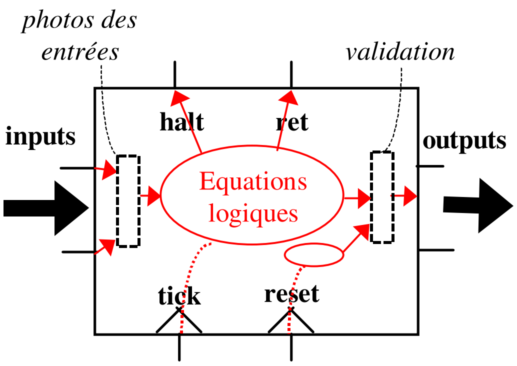

Figure 6 - Strl component

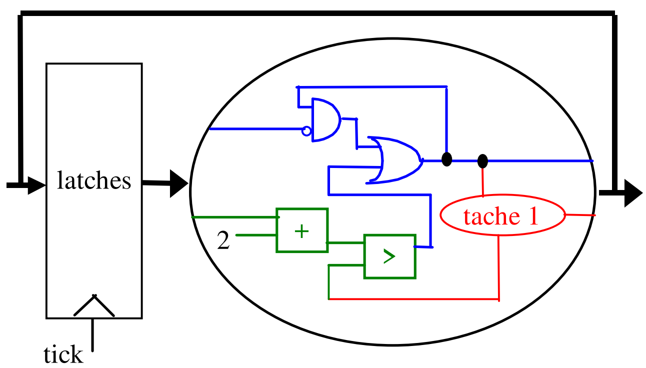

Figure 7 - Equations logiques

4.2 Le modèle constructif électrique et notre implémentation.

Le modèle constructif électrique garantit par construction que la sémantique du programme esterel est conservée lors de la compilation en équations logiques. Cependant, ce modèle laisse un certain nombre de points délicats à la charge de la machine d'exécution. En effet, une partie de la gestion des tâches ainsi que le processus de synchronisation du monde asynchrone ne sont pas traités. Cependant, ces processus doivent vérifier certaines propriétés pour respecter la sémantique du programme esterel.
Le cœur des composants Strl est constitué d'un système d'équations logiques qui est modélisé par l'interface ***modele.sync.Archi***. Une Archi est en fait une liste ordonnée d'équations logiques (interface ***modele.sync.Equation***) mettant en relation des entrées (entrées du composant Strl), des sorties (sorties du composant Strl) et des signaux intermédiaires. La fonctionnalité principale d'une archi (hors mis les accesseurs)

est son évaluation, il s'agit d'évaluer une et une seule fois toutes les équations dans un ordre prédéfini par l'utilisateur.
Les formats de fichiers acceptés pour la construction d'une Archi sont les formats Ssc et Blif. La lecture des fichiers de ce type est effectuée respectivement par les classes modele.sync.ArchiSsc et modele.sync.ArchiBlif. La partie commune à toutes les Archi (parcours de fichiers, accès aux valeurs) est effectuée par la classe abstraite modele.sync.ArchiAdapter.

4.3 Les tâches

Certaines équations logiques d'un fichier Ssc consistent en l'exécution de tâches esterel. Ces tâches sont commandées à l'aide des 5 signaux de contrôle (start, activate, return, kill, suspend). Ces tâches peuvent avoir des paramètres passés par valeur ou par référence et correspondant à des signaux intermédiaires. Ces paramètres sont passés lors du 'start'. Les paramètres passés par référence sont mis à jour lors du 'return'. La tâche permet d'exécuter des actions asynchrones dans le noyau synchrone esterel. L'exécution de celle-ci n'est pas cadencée par le tick esterel, mais la fin de l'exécution d'une tâche est marquée par un signal esterel de return qui est rendu présent par la tâche elle-même.

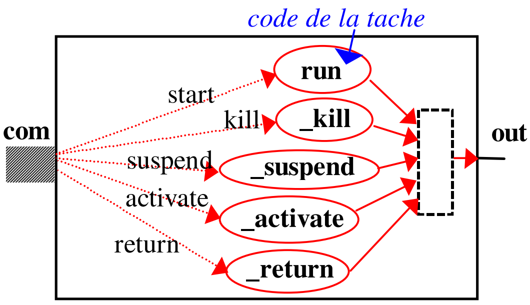

Figure 8 - Task Strl

Lorsqu'un utilisateur doit écrire une tâche, il peut construire un objet héritant de la classe ***modele.sync.TaskStrl***. En effet les objets de cette classe mettent en place un mécanisme permettant l'utilisation des méthodes _start, _return, _kill, _suspend, _activate, qu'il suffit donc de surcharger. De plus, cette classe implémente la fonctionnalité ***java.lang.Runnable*** ce qui permet de la considérer comme une Thread. Et enfin, cette classe hérite de modele.simulation.materialComponent, ce qui permet de l'utiliser pour elle-même dans l'environnement Java et indépendamment d'un objet component.basic.Strl. De plus une méthode de synchronisation par rapport au monde esterel est disponible : waitTick(), qui attend le prochain instant esterel. Dans la majorité des cas, les tâches sont complètement indépendantes du monde esterel excepté pour le contrôle (lancement, terminaison, suspension, passage de paramètres), l'intérêt de cette fonction est donc très relatif.

5. Système d'arrosage automatique

5.1 Présentation de l'arroseur

Ce système tiré de [Gaf91], contrôle le fonctionnement d'un arroseur automatique. Cet arroseur est chargé de deux tâches devant s'effectuer en parallèle :

1. Préparer le mélange qu'il faut disperser : Mélanger l'engrais avec l'eau pour obtenir une bonne concentration appelée concentration de référence. Préparer suffisamment de mélange pour avoir une réserve minimum mais pas trop grande.
2. Disperser le mélange en commandant une pompe pour conserver l'humidité nécessaire aux plantes arrosées. Cette humidité est appelée humidité de référence.

Aussi bien la concentration que l'humidité de référence sont lues lors du lancement du système dans une base de donnée. De plus, cet arroseur automatique doit conserver des valeurs qui correspondent à la quantité de mélange consommée pour obtenir la bonne humidité par rapport au type des plantes arrosées.

La Figure 9 présente l'environnement commandé par le contrôleur que l'on essaye de définir. Les noms qui y apparaissent en gras sont des signaux (entrées ou sorties) ou des capteurs nécessaires pour le fonctionnement du contrôleur.

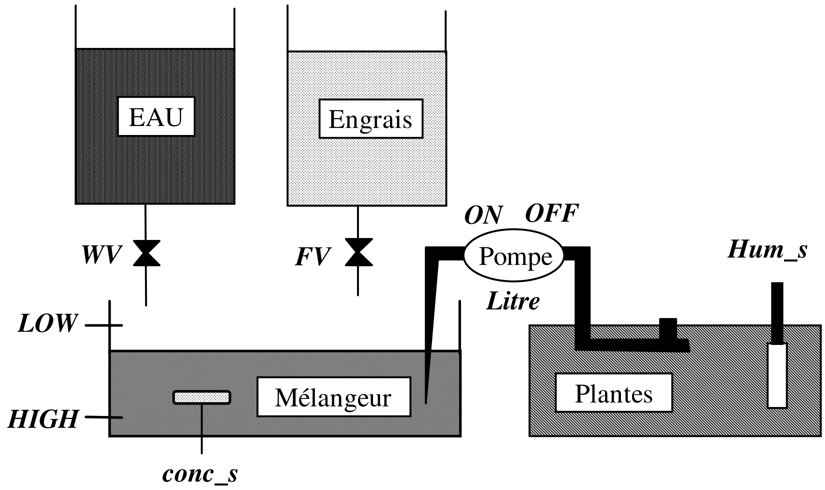

Figure 9 - Système d'arrosage automatique

5.2 Analyse du problème

Ce problème assez simple fait apparaître plusieurs difficultés que l'on rencontre couramment. En effet, on peut identifier une partie contrôle de diverses activités concurrentes : Surveiller les niveaux dans le mélangeur, les indicateurs d'humidité, de concentration et de consommation, commander les différents robinets. Il est souhaitable que cette partie de contrôle soit réactive, c'est pourquoi les langages réactifs synchrones comme esterel sont adaptés à la description de tels systèmes.
De plus, il apparaît aussi une partie transformationnelle, en effet, les paramètres d'humidité et de concentration de référence sont gérés par une base de données. Pour le traitement de données, les langages impératifs ou objets classiques sont plus adaptés. Mais étant donné le caractère synchrone des accès à la base de données, il est possible de déléguer cette gestion à un langage impératif sous le contrôle synchrone d'un langage comme esterel par l'utilisation de tâche. Pour ce faire, il est nécessaire que la modélisation soit effectuée dans un langage connu par la machine d'exécution synchrone comme C ou Java.
Enfin, il y a aussi un certain nombre de processus continus non discrets qui entre en jeu comme la pompe ou les robinets et dont le comportement dépend essentiellement du temps absolu et qui établit des corrélations entre les divers signaux. En effet, les résultats peuvent être complètement différents en fonction des vitesses respectives de ces processus.
Afin de simuler de façon complète un tel système et afin de pouvoir envisager plusieurs scénarios intéressants, il est nécessaire d'avoir un environnement permettant la simulation mixte de ces différents processus hétérogènes qui prend en compte tous les paramètres pertinents extérieurs au système.

5.3 Solution proposée

La solution que nous proposons consiste à utiliser les bonnes propriétés du modèle générique proposé pour permettre la simulation complète de ce système. Grâce à l'intégration de la prise en compte des langages synchrones dans l'environnement proposé, il nous est possible de décrire le comportement du contrôleur de l'arroseur automatique en utilisant le langage Esterel. La description de ce comportement bénéficie alors de toutes les bonnes propriétés de preuve et de validation du langage Esterel. De plus nous avons décrit en Java les tâches permettant l'accès à la base de donnée pour fournir des paramètres et enregistrer des résultats, on bénéficie alors de toute la souplesse du langage pour la description de processus transformationnels. En particulier, l'environnement Java fournit des 'Application Programming Interface' permettant une gestion très facile et rapide des bases de données qu'elles soient localisées ou distribuées (Java Data Base Connection). Ces tâches restent sous le contrôle complet du contrôleur réactif synchrone par le mécanisme de 'tâche Esterel' (cf. chapitre 4.3), mais la réaction est autonome et gérée par un mécanisme de processus légers.
Enfin, nous pouvons modéliser les composants dont le comportement est continu en utilisant les composants sensibles au temps : modele.simulation.TimeComponent. Ainsi, nous avons modélisé des composants comme des récipients, des pompes, des robinets et une horloge. L'utilisateur définit la fréquence de l'horloge et le débit de la pompe et des robinets. Ces composants élémentaires ont été ajoutés à la bibliothèque de composants. De plus, nous avons défini un nouveau type de données (LiquidValue), qui représente un certain volume de liquide avec un certain pH. Il est facile d'ajouter de nouveaux paramètres à ce type en utilisant l'héritage.

L'interface du récipient est présentée par la Figure 10. Ce composant possède un port d'entrée sensible sur niveau qui s'appelle 'control'. Ce port est supposé recevoir une valeur qui exprime la variation de volume demandée. Cette variation peut par exemple être exprimée en utilisant le type LiquidValue. De plus, ce composant exporte des informations concernant son état : plein (full), vide (empty), quantité de liquide présent (quantity) et montant de la dernière variation (real). La pompe et le robinet sont représentées par le même composant : Tap. Son interface est présentée par la Figure 11. Ce composant possède deux ports d'entrée sensibles sur front qui permettent l'ouverture et la fermeture du robinet. Il possède aussi un port de sortie qui exprime la quantité de liquide à transférer. Enfin, ce composant possède un attribut pour régler son débit de sortie.

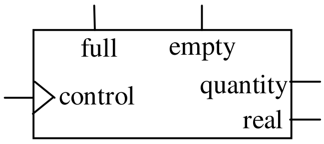

Figure 10 – Le composant récipient.

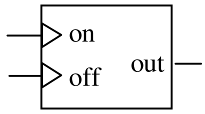

Figure 11 – Le composant Tap.

Le système complet a été modélisé et simulé dans l'environnement SEP. Une illustration de ce système dans l'environnement est présentée par la Figure 12.

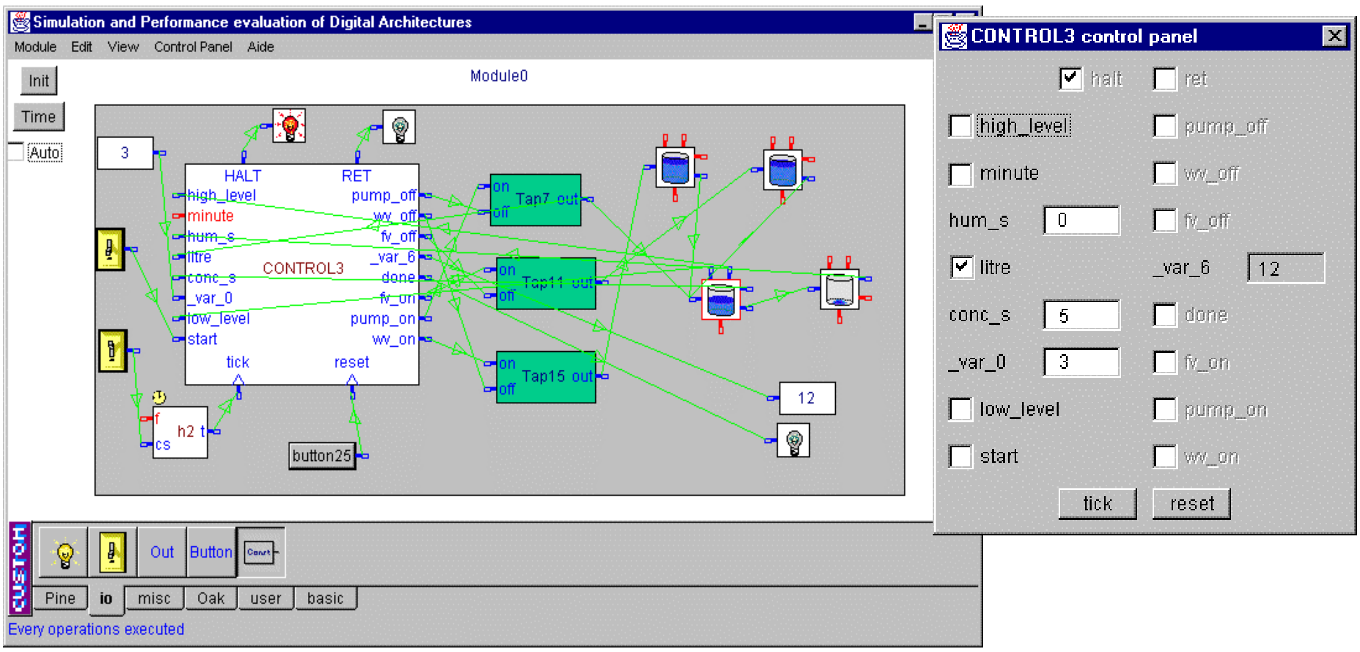

Figure 12 - L'arroseur automatique dans l'environnement SEP et le panneau de contrôle du module esterel.

6. Conclusion

Ce travail montre que l'utilisation des bonnes propriétés orientées objet de l'environnement SEP nous ont permis d'y intégrer facilement un nouveau type de programmation : la programmation réactive synchrone. Il nous est désormais possible de simuler des systèmes hétérogènes complexes composés de composants à caractère calculatoire, de composants orientés vers le contrôle et de composants à comportement continu. Un utilisateur peut désormais utiliser, pour la construction de son système, des composants dont les propriétés sont prouvées de façon formelle. Il peut alors concentrer son action sur les autres composants ainsi que sur l'agencement des divers composants.

Dans un travail futur, il peut intéressant d'étudier les possibilités existantes pour récupérer les informations obtenues sur les composants Esterel afin de déduire de nouvelles propriétés pour les autres composants. En effet, les relations d'ordre logique obtenues sur les événements gérés par les modules Esterel peuvent certainement être propagées vers les autres composants.

De plus, nous modélisons actuellement un ASIC composé d'un processeur de type RISC et d'un processeur de traitement du signal. Nous avons déjà modélisé avec succès un processeur de traitement du signal industriel, mais il nous semble intéressant d'étudier les possibilités d'utiliser le langage Esterel pour la description de la partie contrôle des processeurs et en particulier pour la description des séquenceurs.

7. Bibliographie

[BeD97] ***« Object-Oriented Extensions to CHDL : The LaMI Proposal »***
J. Benzakki, B. Djaffri. IFIP 1997. Chapman & Hall. p. 334-347.

[Ber98] ***« The esterel V5 langage Primer »***
G. Berry. Rapport interne Ecole des Mines et INRIA. Avril 1997. http://www.cma.fr

[BeS98] ***« An Implementation of Constructive Synchronous Programs in POLIS »***
G.Berry, E.M. Sentovich. http://www.inria.fr/meije/esterel.

[Col98] ***« Un modèle foncdé sur les assertions pour le génie logiciel et les bases de données »***
P.Collet. Thèse de doctorat, Université de Nice-Sophia Antipolis. Décembre 1997.

[Gaf91] ***« Conception d'un exécutif temps-réel pour esterel »***
D. Gaffé. Rapport de stage de DEA, Université de Nice-Sophia Antipolis. Septembre 1991.

[MBD98a] ***« Hardware Modelling and Simulation using an Object-oriented Method. »***
F. Mallet, F. Boéri, J-F. Duboc. Proceedings of the European Simulation Multiconference (SCS), Juin 98. ISBN 1-56555-148-6 p.166-168.

[MBD98b] ***« Hardware Architecture Modelling using an Object-oriented Method »***
F. Mallet, F. Boéri, J-F. Duboc. Proceedings of the Euromicro'98. IEEE Vol I, p.147-153. ISSN 1089-6503.

[ScN95] ***« Inheritance Concept for Signals in Object-Oriented Extensions to VHDL. »***
G. Schumacher, W. Nebel. Proceedings of the EURO-DAC'95 with EURO-CHDL'95.