Modélisation d'architectures d'applications - Moyens et formalismes

Domenico Cavaliere, Françoise Simonot-Lion, Ye-Qiong Song

Modélisation d'architectures d'applications

Moyens et formalismes

Domenico Cavaliere
Françoise Simonot-Lion
Ye-Qiong Song

Juin 1999

Rapport interne LORIA

Résumé : Ce document présente un état de l'art des moyens de description d'architectures d'applications informatiques qui peuvent s'appliquer dans le contexte des systèmes d'automatisation de la production du domaine manufacturier. Certains langages de description d'architecture (ADL) sont analysés. Les techniques de description formelle (FDT) sont étudiées sous le même point de vue. Puis, le pouvoir d'expression de ces formalismes pour la vérification de propriétés des applications prenant en compte la distribution et les performances du support matériel est discuté. Enfin, ils sont confrontés aux concepts de « middleware » et de « CORBA ». L'annexe rassemble les tableaux comparatifs sur certains ADL analysés.

Mots clés : architectures opérationnelles, ADL, FDT, CORBA, modélisation, vérification.

1. Objectifs du document

Ce document présente un état de l'art des moyens de modélisation d'architectures opérationnelles en vue de leur validation a priori, c'est-à-dire sur leur modèle, par des techniques d'évaluation de performances. Le processus de modélisation est nécessaire pour réaliser une description du système conforme aux besoins d'analyse ou d'étude de ce système, comme par exemple :

- évaluer a priori les performances d'un système
- évaluer a priori sa sûreté de fonctionnement
- spécifier son fonctionnement
- valider son fonctionnement
- dimensionner / optimiser le système
- déterminer les « meilleurs » composants à choisir pour réaliser l'application
- observer le système réalisé et le maintenir
- .....

Pour un coût optimal d'investissement et d'exploitation, cette architecture doit garantir le respect des contraintes d'applications (temps réel et sûreté de fonctionnement) ainsi que les performances et la pérennité du système d'information industriel.

Les problèmes qui se posent sont liés à la fois à la modélisation d' une architecture et à l'utilisation de son modèle.
Dans le cadre de la modélisation, les travaux portent sur :

1. *l'identification des attributs caractérisant des composants matériels et des fonctions logicielles,*
2. *l'expression de l'interfaçage entre fonctions logicielles et composant matériel,*
3. *l'expression des propriétés à prouver*
4. *la recherche d'un formalisme approprié permettant de construire et de simuler une architecture opérationnelle [REN-99]*

Le choix des moyens et des langages de modélisation doit alors être fait en fonction de ces objectifs de développement, exploitation et vérification. Il est donc nécessaire, au préalable, de mettre en évidence le domaine d' action (pouvoir de modélisation – puissance d'exploitation du modèle – type des résultats « extractables » du modèle) de chaque langage/méthode de modélisation.

Dans la suite du document, nous présentons plusieurs moyens pour modéliser des architectures. Ces moyens peuvent se classer en :

- les « langages de description d'architecture » classiques (Architecture Description Language – ADL) ; ceux-ci sont présentés au paragraphe 3.
- les langages de description formelle de processus communicants ; leur description figure au paragraphe 4.

Puis, nous étudierons, au paragraphe 5, les moyens de vérifier, plus ou moins directement, des propriétés de performances à partir de ces modèles.

Enfin, au paragraphe 6, nous positionnerons rapidement ces langages par rapport aux concepts de « middleware » ou à celui de CORBA.

2. Introduction

2.1 Contexte et domaine d' actions de l'étude

Le contexte de l'étude est celui des « **Systèmes Automatisés de Production** » (SAP). Ceux-ci réalisent la fonction de production à l' intérieur du système global de l'entreprise. Le SAP représente, donc, le domaine racine de notre étude.

Le Système de Production Automatisé (SAP) est constitué de l'ensemble des moyens destiné à l'élaboration de produits conformément aux objectifs économiques et techniques [ROB-93]. Il associe trois entités [BAY-95]:

- **la partie opérative** : elle regroupe les équipements réalisant le processus de production ; l'ensemble de ces équipements est organisé dans le but d'effectuer les opérations de transformation, de stockage, de transport, etc. sur les flux de produits selon un procédé déterminé (suite d'opérations à respecter).
- **le système d'automatisation** : son rôle est de fournir une aide à la commande, la surveillance, la gestion technique du processus de production, en fonction des objectifs et des contraintes de production. Le système d'automatisation est constitué
  - d'une **architecture matérielle** , c'est à dire, des dispositifs matériels (les calculateurs, les régulateurs, les automates programmables et leurs systèmes d'exploitation), des dispositifs d'actions et d'observations du procédé (les capteurs et les actionneurs), des dispositifs de mémorisation (les mémoires, les disques) et les dispositifs de communication (les lignes des communications, les réseaux locaux) assurant les fonctions de communication.,
  - supportant une **architecture logicielle**, c'est-à-dire l'ensemble des programmes communicants implantant l'**architecture fonctionnelle**.
- **les opérateurs** : au moins trois classes d'opérateurs sont identifiés ; les premiers (agents de maîtrise) interviennent sur les processus de production, au travers du système de décision, au fin de l'optimiser, de corriger les dérives quand des problèmes apparaissent. Les agents de maintenance interviennent également sur le processus pour des opérations d'entretien et de maintenance, tandis que les agents qualité opèrent sur le suivi de la qualité du produit et de sa production.

La figure ci-dessus (Figure 1) montre une représentation du système de production en utilisant un modèle de référence hiérarchique [1]. Ce modèle a pour but d' identifier les liens de communication entre hiérarchies différents du système de production.

Le contexte de notre étude est le système d'automatisation restreint aux niveaux 1, 2 et éventuellement 3 de l'architecture CIM :

- niveau « terrain » (capteur, actionneurs, robots,…)
- niveaux « contrôle » (applications de contrôle / automatismes réflexes - gestion / optimisation).

Avant de présenter l'état de l'art, objet de ce document, nous précisons, au paragraphe 2.2, quelques définitions.

[1] Ce modèle n'est rien d'autre que le modèle CIM privé du niveau « Management ».

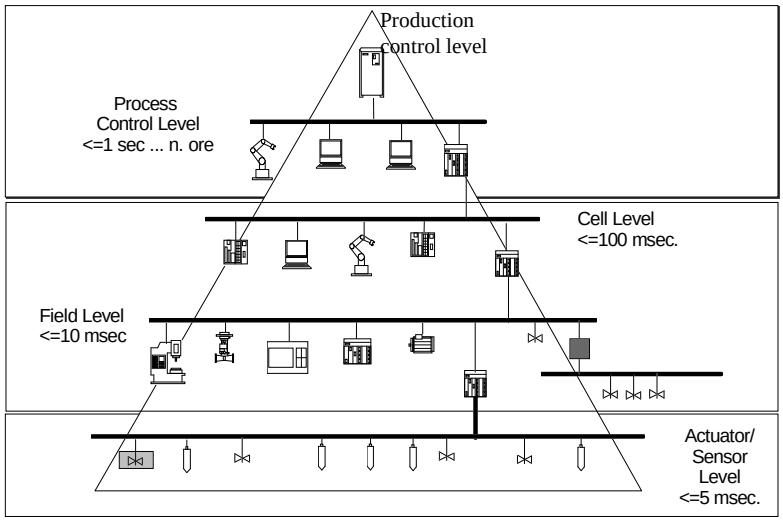


Figure 1 - Modèle hiérarchique du Système Automatisé de Production

2.2 Architectures – Définitions [2]

Définition d'une architecture informatique support, dimensionnement de celle-ci, affectation de fonctions à certains nœuds (calculateurs, automates, …), configuration des flux d'informations transitant par les réseaux de communication, etc. sont des activités à assurer pour l'élaboration d'une application d'automatisation de la production ; si on adopte une démarche de validation a priori, la conception de celle-ci repose sur l'exploitation de modèles. Bien qu'elles ne soient pas toujours clairement identifiées en tant que telles dans la pratique courante, les **architectures fonctionnelle, opérationnelle et matérielle** présentées ci-dessous, sont trois de ces modèles qui offrent trois points de vue sur le système.

2.2.1 Architecture Fonctionnelle

Une **Architecture Fonctionnelle** (AF) est un modèle de la structure et du comportement des fonctions de l'application. C'est une description de la solution fonctionnelle envisagée pour répondre aux exigences du cahier des charges. Le terme « fonctionnel » signifie ici l'ensemble des fonctions du système assurant son « bon » fonctionnement et incluant notamment toute fonction de prévention, détection de défaillances et/ou correction d'erreurs. L'architecture fonctionnelle modélise, en fait, l'ensemble des traitements, les flux d'informations et/ou de contrôle entre ces traitements dans tous les modes de marche de ce système.

Dans [BAY-95], chaque fonction (ou traitement élémentaire du système ou encore **atome)** est caractérisée par les services qu'elle rend. La structure représente l'ensemble des activités et les liens de données entre activités (échanges d'informations). Le comportement de l'ensemble des activités est décrit par l'ensemble des liens de contrôle. Ce comportement représente l'organisation des activités dans le temps (synchronisation, causalité, …).
L'atome est défini, dans ces travaux, comme une entité non décomposable (au sens de la distribution sur des machines) de l'ensemble des traitements du système d'automatisation. Dans [AKA-96], l'atome possède trois propriétés :

- l'identité fonctionnelle : à chaque atome correspond un service ; si un traitement doit être redondé pour des objectifs de sûreté de fonctionnement, il sera réalisé par deux atomes identifiés,
- l'identité géographique : un atome sera implanté sur une seule machine,
- l'identité d'exécution : un atome ne peut rendre un service partiellement ; une conséquence de cette propriété est la garantie de cohérence des données mises à jour par un atome et consommées par d'autres atomes.

L'entité « atome » de l'architecture fonctionnelle y est caractérisée par :

- le traitement réalisé pour assurer le service (un algorithme),
- les données consommées,
- les données produites,

[2] Ces définitions sont extraites de « Maîtrise de la conception et de la validation des applications temps réel distribuées »- habilitation à diriger les recherches – Françoise Simonot-Lion, à paraître octobre 1999. Il en est de même des paragraphes marqués d'un trait vertical dans la marge.

- les états de l'atome, l'automate de changement d'état,
- les requêtes à destination d'autres atomes (activation, désactivation, …) ; l'ensemble de ces requêtes pour tous les atomes de l'architecture fonctionnelle représente les liens de contrôle.

Pour chaque caractéristique, il est possible de définir des attributs évaluables lors de la spécification de l'architecture fonctionnelle et exprimant, par exemple, la complexité des traitements (nombre d'itérations, suite des instructions, …) ou la taille des données. D'autres attributs, comme la durée de vie d'une donnée, sa date de production, sa date de consommation par un atome donné, la date d'activation d'un atome, sa date de fin d'activation sont des variables qui ne seront évaluées (instanciées) qu'après construction de l'architecture opérationnelle. C'est sur ces attributs temporels que s'expriment certaines contraintes de l'architecture fonctionnelle, comme, par exemple, les contraintes d'échéances de traitement (durée d'un traitement, date de fin d'activation d'un atome, durée de vie d'une donnée, …bornées). Il se peut également que ces attributs soient des critères à optimiser : une durée de traitement doit être la plus petite possible.

A ce niveau de décision, dans le développement des systèmes, certaines fonctions peuvent être réalisées sous forme de circuits intégrés ou sous forme de logiciels. Les travaux présentés ici portent essentiellement sur le deuxième type de réalisation, aussi, le terme **Architecture Fonctionnelle** peut-il être compris, dans ce qui suit, comme équivalent à celui **d'Architecture Logicielle**.

2.2.2 Architecture Matérielle

Il s'agit du modèle de l'architecture informatique support de l'architecture fonctionnelle (logicielle). Une **Architecture Matérielle** (AM) est constituée

- d'un ensemble de machines munies de systèmes d'exploitation ; une machine est un support matériel qui peut être mono ou multiprocesseurs ; chaque processeur bénéficie de sa mémoire et de ses entrées-sorties propres ; une machine multiprocesseurs peut avoir accès à une mémoire commune.
- d'un ensemble de moyens de communication les connectant, munis des protocoles nécessaires,
- de la définition des connexions des machines sur les moyens de communication ; il s'agit de la topologie de l'architecture.

La définition de cette architecture (topologie, dimensionnement) repose sur la caractérisation de ses constituants (coûts, capacités, performances, …) ; certaines preuves peuvent être faites uniquement sur l'architecture matérielle, par exemple le respect de contraintes budgétaires d'investissement en matériel, l'interopérabilité des constituants, …D'autres seront faites sur l'architecture matérielle considérée comme un support de l'application d'automatisation, c'est-à-dire sur l'architecture opérationnelle que nous définissons ci-dessous ; les attributs envisagés de l'architecture matérielle nous servirons alors à calculer les valeurs d'attributs de l'architecture fonctionnelle (la puissance d'un processeur permet d'évaluer la durée d'un traitement, si celui-ci est décrit par un nombre d'opérations). De la même manière que précédemment, les attributs de l'architecture matérielle peuvent être soumis à des contraintes ou être utilisés comme des critères d'optimisation.

2.2.3 Architecture Opérationnelle

Une **Architecture Opérationnelle** (AOP) désignera le résultat d'une projection de l'architecture fonctionnelle (logicielle) sur une architecture matérielle.
L'**Architecture Opérationnelle validée et optimisée** est le résultat de l'étape de conception. Il s'agit de la "meilleure" architecture opérationnelle au sens d'un ou plusieurs critères. Elle est validée dans le sens où elle est conforme au cahier des charges, c'est-à-dire qu'elle respecte toutes les contraintes énoncées.

3. ADL – Langage de description d'architectures

Au fur et à mesure que les applications informatiques devenaient de plus en plus importantes, est apparu le besoin de passer du stade artisanal d'écriture de programmes à un stade plus « industriel » où on dispose d'infrastructures et de méthodes pour la conception, la programmation, la gestion de versions d'applications, l'assistance au travail coopératif, … Celles qui nous intéressent ici portent sur les moyens de décrire une application à un niveau d'abstraction qui offre une précision suffisante pour en appréhender toute la structure et un niveau de granularité tel qu'on puisse en extraire des éléments réutilisables. Les travaux pour ce faire, passent par la spécification de langages de description d'architectures (Architecture Description Languages – ADL). Ci-dessous figure un résumé sur ce qu'est un ADL, que doit-il contenir et comment caractériser les différents langage. Ce résumé a été alimenté par des états de l'art et analyses diverses de ces formalismes ; en particulier, dans [VES-93] et [ABD-94], les analyses sont menées suivant une optique purement génie logiciel tandis que dans [BIL-91], [DEP-92], [THO-97] et [DUR-98], elles sont guidées par le besoin de modéliser des applications temps réel et tolérantes aux fautes. Un guide d'évaluation de langages de description d'architectures est disponible dans [KOG-94].

Dans la terminologie DSSA[3] [HAY-94], une architecture de système, et ceci est vrai pour tout type de système, est définie comme un ensemble de composants, les interfaces de chacun d'eux et la topologie d'interconnexion de ceux-ci. Les ADL en fournissent les moyens de leur description, en offrant trois concepts de base : les composants, les connexions et connecteurs, les architectures.

- **Composants**

Suivant les ADL, ceux-ci peuvent être modélisés :
- par des boites noires dont on ne spécifie que les interfaces ; c'est le cas par exemple par exemple pour UNICON [SHA-96] ou DARWIN [MAG-95], …
- par la description de leur comportement interne celui-ci peut être décrit par le code source à exécuter, comme dans METAH, ou par une abstraction de ce code ainsi qu'il est fait dans WRIGHT [ALL-97], RAPIDE [LUC-95], BASEMENT [HAN-96], VOLCANO [TIN-98], CLARA [DUR-95], …

Presque tous les ADL offrent la notion de hiérarchisation de composants permettant des vues sur une architecture à plusieurs niveaux d'abstraction. Les exceptions à la règle sont les langages qui sont très fortement associés à des outils de vérification de propriétés temporelles, telles l'ordonnançabilité des traitements sur une machine ou des messages sur un réseau (BASEMENT, VOLCANO). METAH [VES-94] et CLARA offrent des possibilités de regroupement de composants élémentaires au sein de « macro » composants ; néanmoins, la vérification de propriétés temporelles repose sur une mise à plat de l'architecture.
Enfin la sémantique des interfaces des composants peut être plus ou moins définie ; les interfaces d'échanges de données sont caractérisées par les types de celles-ci ; lorsqu'on désire dépasser les simples vérifications structurelles d'architecture, il est indispensable d'intégrer la notion d'événements, et la sémantique associée, aux points d'interface du composant ; ceci est fait en particulier pour Wright (CSP) et CLARA (automates) [DUR-98].

- **Connexions / connecteurs**

Une connexion permet de spécifier comment sont liés un ou plusieurs composants dans une architecture. Cette notion de connexion peut être plus ou moins formalisée ; la spécification d'une connexion peut se résumer à la simple description d'un lien entre points d'interface de composants. Elle peut également se décrire de façon formelle par des éléments spécifiques appelés connecteurs (ou canaux de communication). Ceux-ci modélisent le protocoles de communication entre les composants liés ; on trouve ces notions, en particulier, dans RAPIDE et CLARA. On peut remarquer que, dans ce niveau de formalisation, les connecteurs sont des composants particuliers.

- **Architectures**

Au niveau des architectures, deux points, entre autres, permettent de caractériser les ADL. Le premier concerne ce qu'on appelle les styles architecturaux ; cette notion permet à un concepteur d'application de dériver une partie d'architecture à partir de modèles de coopération de référence (par exemple : client / serveur, tableau noir, filtres et tubes, …) ; elle est implantée, entre autres, dans Wright.
La possibilité de décrire des architectures qui évoluent dynamiquement est le deuxième point qui permet de caractériser des ADL, au niveau des architectures décrites. Ce service est relativement rare ; on le trouve par exemple dans Darwin et dans RAPIDE où doivent être spécifiées toutes les architectures possibles. Par contre, dans aucun des cas ne sont décrits les mécanismes de reconfiguration ; ce problème est pourtant crucial dans tous les systèmes tolérants aux fautes. La notion de mode de fonctionnement, définie à tous les niveaux de la hiérarchie, est également présente dans CLARA.

Les ADL sont donc des langages adaptés à la description structurée d'une **Architecture Logicielle**. Cette structuration facilite la **compréhension** des fonctions exécutées par cette Architecture et peut permettre un certain type de validation (cohérence des types dans les connexions, mais aussi, par construction d'une machine d'exécution, cohérence d'un comportement de l'architecture logicielle).
L'évaluation des performances de l'**Architecture Opérationnelle**, c'est à dire :
- la validation de l'exécution des fonctions dans les contraintes de temps spécifiées,
- l'évaluation de l'utilisation de ressources,

n'est pas un objectif classiquement intégré dans ces langages

Dans ce document nous introduirons non seulement les ADL proprement dits, mais aussi des langages qui nous appellerons DSSAL (Domain Specific Software Architectures Langages). Ces derniers supportent, en parallèle à la description de l'**Architecture Logicielle** des éléments de description de l'**Architecture Matérielle** support et de la **distribution**.

[3] DSSA (Domain Specific Standard Architecture) est un projet soutenu par le ministère de la Défense des Etats Unis

Six langages ont été étudiés : Wright [ALL-97], Aesop, Rapide [LUC-95], [RAP-96],[RAP-97], Chiron2 [MED-96a], [MED-96b], [MED-97], [TAY-97], Unicon [SHA-96], MetaH [VES-94]. Les conclusions sur leur étude sont résumées dans les tableaux présentés en annexe 1.

4. Les techniques de description formelle (FDT) – le cas de SDL

La deuxième classe de formalismes qui semblent intéressant d'étudier sous l'angle de modélisation d'architectures, est constituée par les techniques de description formelle (FDT) [TUR-93]. Le but de ces techniques est de fournir des langages formels de spécification et des méthodes rigoureuses pour développer des systèmes complexes, intégrant des activités concurrentes, à contraintes strictes de sûreté de fonctionnement (sécurité-innocuité, sécurité-confidentialité, fiabilité du système) et maintenables. En particulier dans le domaine de l'OSI (Open System Interconnection), les standards développés doivent être non ambigus et indépendants des implantations afin d'en conserver les qualités d'ouverture. Les approches formelles sont un moyen de le garantir. Deux approches sont utilisées : les automates d'états finis (ESTELLE), le typage algébrique de données (LOTOS) ou une intégration de concepts des deux précédents (SDL). ESTELLE et SDL permettent tous deux une représentation hiérarchique des systèmes. En raison de la maturité des outils supportant le langage SDL, c'est ce formalisme qui sera confronté, ici, aux langages usuels de description d'architectures.

La version 1988 de SDL (Specification and Description Language) est connue comme la recommandation Z.100 du CCITT[4] ; elle a été étendue en 1992 pour intégrer une approche objet et la possibilité de modéliser l'indéterminisme. Ce formalisme est largement employé actuellement comme support d'activités de conception d'applications, c'est-à-dire d'activités où on se préoccupe plus de la structure de l'application que de la programmation elle-même. Les mots clés sont : structuration, communication entre les composants modélisés et description du comportement du système par spécification du comportement de ses composants. Si on décline SDL suivant les caractéristiques énoncées dans le paragraphe 3 (architecture, composants, connexions), on obtient les conclusions ci-dessous.

- **Structuration - architecture**

SDL fournit les notions de systèmes, sous-systèmes et processus (dernier élément de la décomposition hiérarchique) qui communiquent par l'intermédiaire de canaux de communication. Le langage graphique fait apparaître une topologie de connexions d'entités à tous les niveaux de la hiérarchie. Il permet de spécifier des sous-systèmes qui peuvent s'exécuter en parallèle et coopérer par échanges de messages ou partage de variables. On est donc bien au niveau d'une description architecturale. Notons que SDL permet de modéliser des configurations d'architecture qui évoluent dynamiquement à l'aide d'actions de création de processus.

- **Composants**

Par contre, on n'a pas réellement une description en termes de composants réutilisables car la notion d'interface de composants n'est pas attachée au composant lui-même ; elle n'est spécifiée que dans le sous-système qui l'englobe.

- **Communication entre composants - connexions**

Cet aspect est décrit par des canaux de communication qui peuvent transporter des signaux dans un ou les deux sens. Les signaux sont valués. Les types des valeurs associées aux signaux sont définis à l'intérieur des composants élémentaires (processus). Une connexion joue un simple rôle de lien entre composants. La communication modélisée est asynchrone ; tout composant est supposé disposer d'une file non bornée de signaux en réception.

Si sur le plan de la réutilisabilité de composants, SDL n'offre pas les qualités attendues des langages de description d'architecture, il en possède néanmoins l'aspect structuration et composition de composants. Son intérêt est le plus évident réside dans la possibilité de décrire les composants élémentaires de manière formelle et, donc, non ambiguë. La spécification du comportement du système se fait sous forme d'automates étendus communicants. Pour ceci, chaque composant élémentaire (processus) est décrit à l'aide d'un langage fournissant les moyens de décrire les états et les transitions ainsi que les manipulations de variables et les déclenchements et annulation d'horloges. Les interactions avec les autres composants de l'architecture se font par l'intermédiaire des signaux émis et reçus par les composants et transitant via les canaux de communication. A l'intérieur d'un processus, le comportement peut dépendre de la valeur de variables (décisions). Celles-ci sont internes au processus qui les déclarent.
Grâce à cette description formelle du comportement de composants élémentaires, un modèle SDL supporte des activités de validation. Tout d'abord, grâce à un outil de simulation, il est possible de vérifier que le comportement du système correspond bien au comportement spécifié dans le cahier des charges. Enfin, par exploration exhaustive de l'espace des états, un outil d'analyse permet de vérifier que le système n'a pas des comportements non souhaités (interblocage, …).

[4] Comité Consultatif International de Télégraphe et de Téléphone (IUT-T).

5. Exploitation des modèles d'architecture pour la vérification de propriétés de performances.

Les deux paragraphes précédents ont introduit la notion de spécification d'architectures d'applications, soit à l'aide d'ADL proprement dits, c'est-à-dire de langages précisément définis dans ce but, soit à l'aide d'autres formalismes, tels SDL. Quel est l'apport de ces différents travaux dans le contexte de la conception d'architectures opérationnelles distribuées et temps réel ? Rappelons que le terme « conception » signifie :

- définition du placement, optimal suivant certains critères, d'une architecture fonctionnelle-logicielle sur une architecture matérielle
- et vérification des propriétés de sûreté de fonctionnement, de temps …

On analyse ci-dessous les langages étudiés dans les paragraphes précédents sous deux angles :

- les moyens d'évaluer des propriétés temporelles sur les modèles d'architectures décrits à l'aide de ces langages,
- les moyens de prendre en compte les performances d'une plate forme matérielle et d'une distribution.

- **Evaluation de propriétés temporelles**

Aux langages étudiés en 3, sont généralement associés des outils qui favorisent des activités de validation, telle l'analyse structurelle de l'architecture ou la vérification de propriétés sur son comportement. Ce dernier point présente un intérêt tout particulier dans le contexte de ce document ; c'est le seul étudié ici. Parmi les ADL cités précédemment, nous retenons ceux qui intègrent les moyens de faire des vérifications de propriétés comportementales :

- Par exécution de modèles

  C'est le cas, par exemple pour RAPIDE avec la mise en œuvre de la théorie des POSETS. La simulation, ou exécution de modèle, est guidée par la spécification du comportement des composants (les POSETS contraignant le comportement dans RAPIDE) et des événements sont datés par rapport à une horloge supposée globale (activation d'une action, appel et retour d'une fonction dans RAPIDE). Les propriétés temporelles sont alors vérifiées sur les traces produites.

  Pour les trois FDT cités plus haut, ont été développés des travaux qui permettent de traduire un modèle FDT en un modèle exploitable par un outil de simulation : ESTELLE vers QNAP2[5] [MRA-96], LOTOS vers QNAP2[5] [VAL-93] ou plus récemment SDL vers OPNET. A chaque fois, des règles précises sont fournies pour traduire les objets d'un modèle dans ceux de l'autre. Une simulation permet ensuite de générer des traces d'événements et de les exploiter pour valider l'architecture.

- Par analyse de modèles

  Ceci permet, par exemple, de prouver des propriétés d'absence d'interblocage. Dans Wright, le formalisme des CSP est utilisé, les propriétés sont prouvées par exploration du graphe de classes. SDL, comme cela a été dit ci-dessus permet une exploration de l'espace des états. Il s'agit de propriétés sur l'ordre et non sur les dates d'événements.

  Dans des domaines plus proche du temps réel, sont menées des preuves d'ordonnançabilité. METAH montre l'ordonnançabilité de tâches si l'architecture matérielle est centralisée, VOLCANO démontre celle des messages dans le cas où le réseau utilisé est CAN et BASEMENT prouve celles des tâches et des messages ; chacun de ces trois types de preuves repose sur des hypothèses de périodicité des traitements.

- **Prise en compte des performances des matériels et des exécutifs supports**

La vérification de propriétés de sûreté de fonctionnement, en particulier de propriétés temporelles, sur une proposition de placement doit tenir compte des performances des matériels informatiques supports. On peut remarquer que dans la plupart de ces travaux, la prise en compte d'une architecture matérielle support de l'application logicielle est explicite en ce qui concerne le placement des composants de l'architecture logicielle sur les machines (par exemple dans METAH), mais l'est très peu en ce qui concerne l'implication d'une performance du matériel sur le modèle exécutable ou analysable de l'architecture logicielle. En particulier, les mécanismes exécutifs sont rarement pris en compte. L'impact de la puissance d'un processeur sur l'exécution des traitements est indiqué sous forme de taux d'utilisation de ce processeur (une durée de traitement dans la plupart des cas, un retard sur le tir d'une transition en SDL ou une relation temporelle entre événements d'un POSET dans RAPIDE). Le mécanisme d'ordonnancement des actions est supposé être de type « à échéance » pour BASEMENT et les calculs sont faits sous cette hypothèse. Dans METAH, la durée caractérisant un traitement pour un processeur donné intègre,

[5] QNAP2 est un langage pour l'évaluation de performances, diffusé par la société Simulog SA.

mais ce n'est pas explicitement dit, les temps d'éventuelles préemptions. Pour VOLCANO, les mécanismes protocolaires du réseau CAN sont modélisés, par contre, on ne connaît des traitements que leur périodicité.

En conclusions, il y a pléthore de moyens pour décrire des architectures d'applications ; ces moyens sont efficaces et utiles pour apporter des facilités de réutilisation de composants. Certains d'entre eux offrent des possibilités de faire des preuves mais sous de fortes hypothèses de performances du matériel (ressources infinies). S'il est impossible de garantir ces hypothèses, ils ne permettent pas de modéliser des architectures opérationnelles.

6. Autres moyens de structuration – modélisation d'architectures

Dans de nombreux domaines, la variabilité du support exécutif est assez faible. Pour des exigences normatives internes à une entreprise ou à un domaine d'aplications, les systèmes exécutifs et les réseaux de communication, par exemple, sont imposés. Par contre, il subsiste pratiquement toujours des besoins d'évolutivité d'application, de réutilisation de composants logiciels (portabilité), de dimensionnement optimal d'une application en fonction de la distribution des composants logiciels sur le matériel.
Un moyen pour assurer ceci est de garantir une **indépendance entre les composants logiciels et les matériels et exécutifs**. La solution à ce problème réside dans l'existence d'une couche interfaçant les deux types de composants ; cette couche est couramment désignée sous le terme de « **middleware** ».

On identifie deux aspects fondamentaux dans un « middleware » :

- L'**interface** ou API : elle permet au programmeur de spécifier et programmer ses composants logiciels sans hypothèse sur l'implantation qui en sera faite. Toute communication – synchronisation entre composants logiciels passe par les mêmes API. Par exemple, soient deux composants logiciels – un producteur / un consommateur – se synchronisant par boîte aux lettres (FIFO sans écrasement) ; leur spécification – programmation se fera à l'aide des API ou services « SEND » et « RECEIVE » et ce, indépendamment de leur affectation à une station.
- Les **services** assurés par l'ensemble du middleware (et surtout, la qualité de service associée):
    - service de cohérence des copies multiples de données,
    - service de cohérence des horloges,
    - services de synchronisation entre entités distantes,
    - …

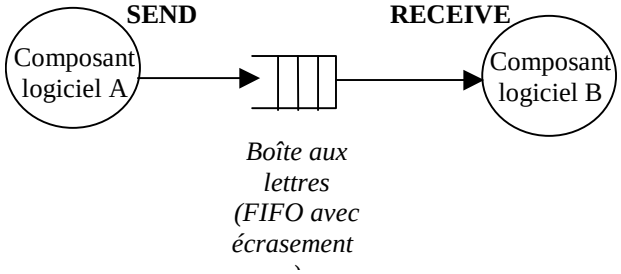


Figure 2 - architecture logicielle - coopération de deux composants

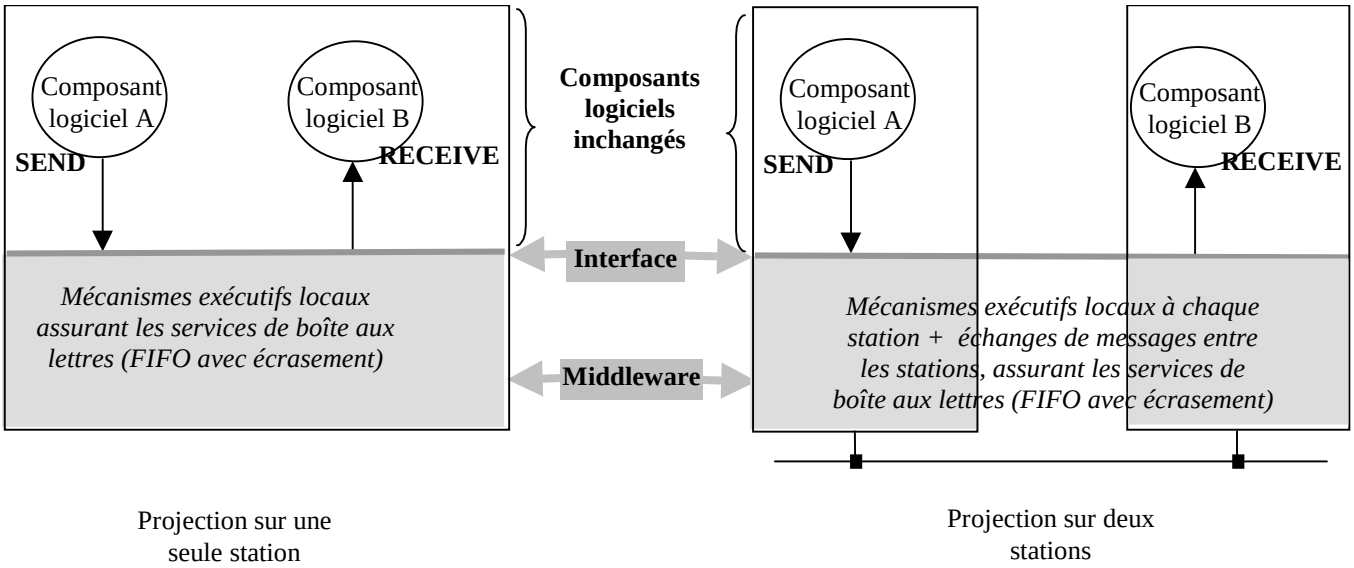


Figure 3 – deux architectures opérationnelles

La structuration d'une application et la modélisation d'une architecture de composants logiciels peut alors reposer sur l'ensemble des services disponibles dans le middleware. L'intérêt est confiné alors à la portabilité des composants logiciels. Mais, tant qu'un modèle de comportement et de performances de l'implantation du middleware lui-même n'est pas disponible, il est impossible de vérifier des propriétés de performances d'une architecture opérationnelle.

Quelle est la position de **CORBA**, dans ce contexte ? On le confronte souvent :

- d'une part, à un « middleware » : si on fait abstraction de son implantation, ce sont deux notions distinctes ; en particulier, on ne peut caractériser CORBA seul comme un composant exécutif,
- d'autre part, à un ADL : CORBA ne fournit pas en lui-même les concepts suffisants pour décrire une architecture ; en effet, il ne définit explicitement aucune notion de composant – de connecteur – d'architecture.

En fait, l'intérêt premier de CORBA est d'apporter une **interface normalisée**. Par contre, rien, dans CORBA, ne permet de spécifier comment sont implantés les services. La même remarque que celle faite sur les middleware peut être donc formulée.

7. Conclusions

L'état de l'art fait dans ce document a porté sur des moyens de décrire des architectures d'applications en vue de vérification de certaines propriétés. Deux classes de moyens sont identifiés : les ADL et les techniques de description formelle. Nous avons, d'autre part, présenté, comment, en exploitant des modèles exprimés dans certains de ces formalismes, il est possible de vérifier des propriétés. Enfin, nous avons rapidement positionné les concepts de « middleware » ainsi que CORBA par rapport à ces moyens.

Un travail supplémentaire à faire est d'étudier comment des approches objet, en particulier UML, peuvent apporter des éléments de solution à la modélisation d'architectures. UML propose un formalisme de modélisation de systèmes (quelque soit le type de système) permettant de décrire plusieurs points de vue sur le système ; ceci est concrétisé par les 9 types de diagrammes de UML. Un de ces diagrammes permet de décrire le comportement des objets en termes de systèmes « états-transitions » en utilisant un formalisme inspiré des StateCharts ; par ailleurs, les diagrammes de séquencement et de collaboration apportent des informations complémentaires sur le comportement du système. Deux remarques peuvent être faites à ce niveau. D'une part, UML ne gère aucuns liens sémantiques entre les différents diagrammes (les entités manipulées dans un diagramme ne sont pas fortement liées aux entités d'un autre diagramme). D'autre part, même si un modèle de comportement est disponible, il permet difficilement de modéliser une architecture opérationnelle (voir paragraphe 2.2.3) et encore plus difficilement d'en évaluer les performances. Par contre, UML apportant un moyen efficace de communication entre acteurs d'un projet et de documentation, il ne semble pas déraisonnable d'étudier comment s'appuyer sur ce formalisme pour en déduire un modèle sur lequel faire de l'évaluation de performances. Ceci fera l'objet de l'étude suivante.

8. Bibliographie

[ABD-94] Abd-Allah A., *Architectural Description Languages*, Software Architecture Tutorial, University of Southern California, center for Software Engineering, juin 1994.

[AKA-96] Akaichi J., Systèmes automatisés à intelligence distribuée : des stratégies de répartition basées sur une approche de classification, doctorat de l'Université des Sciences et Techniques de Lille, novembre 1996

[ALL-97] Allen R ., Garlan D., *A Formal Approach for Architectural Connection* PhD thesis,school of Computer Science, Carnegie-Mellon University, Pittsburgh, 1997.

[BAY-95] Bayart M., Simonot-Lion F., « impact de l'émergence des réseaux de terrains et de l'instrumentation intelligente sur la conception des systèmes d'automatisation de la production », 132 pages, contrat MESR 92-P-239, février 1995.

[BIL-91] Biland P., Deplanche A.-M., *Langages de configuration pour les systèmes temps réel répartis*, Technical Report, LAN-ENSM, n° 91.05, 1991.

[DEP-92] Deplanche A.-M., *Configuration des applications réparties : quelques approches*, Technical Report, LAN-ENSM n° 92.03, mars 1992.

[DUR-95] Durand E., Deplanche A.-M., Creusot D., *A Configuration Language to Describe Real-Time Applications*, in Proceedings Real Time'95, Ostrava, République Tchèque, septembre 1995.

[DUR-98] Durand E., Description et vérification d'architectures d'application temps réel : CLARA et les réseaux de Petri temporels, thèse de doctorat de l'Ecole Centrale de Nantes.

[HAN-96] Hansson H.A., Lawson H.W., Strömberg M., Larsson S., *BASEMENT : A Distributed Real-Time Architecture for Vehicule Applications*, Real-Time Systems, Vol.11, 223-244, 1996.

[HAY-94] Hayes-Roth R., Tracz W., *DSSA Tool Reaquirements For Key Process Functions*.

[KOG-94] Kogut P., Clements P., *Features of Architecture Representation Languages*, Technical report, Software Engineering Institute, Carnegie Mellon University, décembre 1994.

[LUC-95] Luckham D., Vera J., *An Event-Based Architecture Definition Language*,IEEE Transactions on Software Engineering, vol. 21(4) :336-355, Avril 1995.

[MAG-95] Magee J.,Dulay N., Eisenbach S., Kramer J., *Specifying Distributed Software Architectures*, in Proceedings of the 5th European Software Engineering Conference (ESEC'95), Sitges, Espagne, septembre 1995.

[MED-96a] Neno Medvidovic, "A Classification and Comparaison Framework for Software Architecture Description Languages", Technical Report UCI-ICS-97-02, University of California, Irvine Février 1996

[MED-96b] Nenad Medvidovic, "Formal definition of Chiron-2 software architectural style", Technical Report UCI-ICS-95-24, Departement of Information and Computer Science, University of California, Irvine, Avril 1996

[MED-97] Richard N. TAYLOR, Nenad Medvidovic, "A Framework for Classyfing and Comparing Architecture Description Langages", Departement of Information and Computer Science, University of California, Irvine, 1997

[MRA-96] Mrabet R., *Modèle de simulation QNAP2 pour l'étude de performances de spécifications Estelle hiérarchiques*,Colloque Francophone de l'ingénierie des Protocoles, CFIP'96.

[RAP-96] Program Analysis and Verification Group Computer System Lab, "The Rapide 1.0 Full Syntax Reference Manual", Stanford University, September 5, 1996

[RAP-97] Program Analysis and Verification Group Computer System Lab, "Draft, Guide to the Rapide 1.0", Language Reference Manual, Stanford University, juillet 17, 1997

[REN-99] Olivier RENAN, Annexe Technique, "Descriptif détaille du projet de R&D dans le cadre d'une thèse CIFRE coencadree par l'industriel PSA Peugeot-CITROEN et le laboratoire de recherche LORIA", PSA 1998

[ROB-93] M. Robert, M. Marchandiaux, M. Porte., "Capteurs intelligents et méthodologies d'évaluation", Ed Hermes, 1993

[SHA-96] Shaw M., Garlan D., Software Architecture – Perspectives on an Emerging Discipline, Prentic-Hall, 1996.

[TAY-96] Richard N. TAYLOR, Nenad MEDVIDOVIC et al., "A Component and Message Based Architectural Style for GUI Software", IEEE Transaction on Software Engineering vol. 22, n. 6, Juin 1996

[THO-97] Thomas L., Jourdan J., Simonot-Lion F., Bayart M., Choukair C., Peraldi M.-A., André C., Deplanche A.-M., Trinquet Y., rapport intermédiaire à 6 mois du contrat COVADIS 96448 DRET-DGA/ 96C0076 DGRT-MENESR, juillet 1997

[TIN98] Tindell, Volcano, CIA'98

[TUR-93] Turner K.J., *Using Formal Description Techniques : An Introduction to ESTELLE, LOTOS and SDL*, John Wiley & Sons, 1993.

[VAL-93] Valderruten Vidal A., Hjiej O.,Benzekri A., Gazal D., *Deriving Queuing Networks Performance Models from Annotated LOTOS specifications*, 6th International Conference on Modelling Techniques and Tools for Performance Evaluation, Edimburgh University Press, août 1993.

[VES-93] S. Vestal, "Scheduling and Communicating in MetaH", Real -Time System Symposium, p.194-200, Rleigh-Durham (NC) Decembre 1993

[VES-94] Vestal S., *MetaH Reference Manual*, Technical Report,Honeywell Technology Center, mars 1995.

Adresses internet

Chiron2:	*www.ics.uci.edu/pub/arch/c2.html*
Aesop:	*www.cs.cmu.edu/afs/cs/project/able/www/able.html*
UniCon Langage Reference Manual:	*www.cs.cmu.edu/afs/cs/project/vit/www/unicon/reference-manual*
Rapide:	*www.parg.stanford.edu/rapide/*
Argo:	*www.ics.uci.edu/pub/arch/uml/v06/index.html*
ControlH et MetaH:	*www.htc.honeywell.com/projects/dssa/dssa_tools.html*
MetaH:	*www.htc.honeywell.com/metah*
Wright:	*www.cs.cmu.edu/afs/cs/project/able/www/wright/wright_bib.html*
DARPA (Defence Research Projects Agency):	*www.arpa.mil/organization.html*
Une Liste de adls:	*www.asset.com/stars/lm-tds/Papers/sysdev/section6_1_2.html*
DSSA (Domain-Specific Software Architectures) References:	*www.htc.honeywell.com/projects/dssa/dssa_refs.html*
FDR2:	*www.formal.demon.co.uk/FDR2.html* *ftp.comlab.ox.ac.uk/pub/Packages/FDR*

1. SOMMAIRE

Annexe 1 :

Etude comparative d'ADL

Les tableaux figurant dans cette annexe permettent de comparer plusieurs ADL en fonction d'un certain nombre de caractéristiques. L'étude est menée selon deux points de vue :

- le formalisme de modélisation,
- les outils supportant le langage.

Les caractéristiques étudiées pour le formalisme de modélisation sont :

- les composants
- les connecteurs
- les types manipulés
- la sémantique associée
- la capacité à décrire des architectures opérationnelles
- les contraintes spécifiant le comportement

pour les outils support :

- l'ergonomie de l'outil
- les services de l'outil
- les moyens de se procurer et les domaines d'application de l'outil.

Dans les différents tableaux, nous avons séparé les ADL classiques (Rapide, Aesop, C2, Wright) et ceux qui permettent de décrire le support matériel d'exécution et la distribution (MetaH, UNICON).

Quelques conclusions sur ces tableaux

- Les solutions comme Wright, Aesop, C2 n'offrent pas des moyens satisfaisants pour modéliser des comportements temporels de l'architecture et il est donc impossible d'évaluer le respect de contraintes temporelles. Rapide permet, lui, de représenter et gérer le temps et ce temps est un temps physique, caractérisé donc par une origine temporelle fixé et une unité de mesure.
- Rapide et WRIGHT offrent des moyens (syntaxe formelle, définitions de contraintes) pour l'évaluation de l'architecture modélisée. C2, MetaH, UNICON permettent la réalisation d'architectures caractérisées par un style bien défini ; le processus de validation est lié aux contraintes introduites par le style (par exemple en C2 est chaque porte représente une seule connexion connecteur/composant)
- Dans le cas de C2 le choix d' un style est obligé et peut limiter les capacités de description du langage.
- Rapide, est le seul langage, entre ceux étudiés capable de représenter des architectures dynamiques, c'est à dire des architectures capables de changer de topologie et de comportement *en ligne*.
- MetaH est le seul langage à introduire le concept de *mode*, c'est à dire la possibilité de fixer un ensemble énuméré des configurations de l'architecture, et de changer *en ligne* à l'intérieur de cet ensemble.
- Une caractéristique commune à la plupart des langages considérés (Rapide, C2, UNICON, Aesop) est que les outils à disposition sont encore en phase d'évaluation. L'application de ces outils pour l'évaluation de prestations de systèmes réels, normalement complexes et de grandes dimensions, représente donc un choix risqué.
- Les solutions UNICON et MetaH répondent aux besoins de représentation du système sur lequel l'application tournera. UNICON est un langage plus expressif que MetaH et, à la différence de ce dernier, propose séparément la notion de connecteurs. UNICON ne se limite pas à un type de matériel support, tandis que MetaH est conçu pour un processeur particulier: le processeur i80960MC.

Langage	*Composant*	*Connecteur*	*Type*	*Formalisme d'expression du comportement*	*Spécification des systèmes informatiques de support*	*Contraintes su la cohérence statique d'architecture*	*Propriétés temporelles de l'architecture (à respecter par la machine d'exécution)*
WRIGHT	Déclaration explicite. L'interface du composant est spécifiée par des portes d'entrée sortie. A chaque porte il peut être associé un protocole de communication	Déclaration explicite. L'interface du connecteur est spécifiée par des points de connexion (rôles). A chaque rôle il peut être associé un protocole de communication.	L'ensemble de base des types de données est extensible.	Basé sur CSP.	Non supporté.	Définition des contraintes sur portes et rôles.	La description des propriétés temporelles n'est pas supportée.
AESOP	Déclaration explicite. L'interface du composant est spécifiée par des portes d'entrée sortie. A chaque porte il peut être associé un protocole de communication.	Déclaration explicite. L'interface du connecteur est spécifiée par des point de connexion (rôles). A chaque rôle il peut être associé un protocole de communication.	L'ensemble de base des types de données est extensible.	Acun (optionnel: utilisation de CSP pour la déclaration de portes et de rçoles).	Supporté que pour le style architectural *"Real Time Message Passing"* .	Définition des contraintes sur portes et rôles.	La description des propriétés temporelles est supportée seulement par le style *"Real Time Message Passing"* .
CHIRON2	Déclaration explicite. L'interface du composant est spécifiée par des portes d'entrée sortie.	Déclaration explicite. Un seul point de connexion est spécifié pour chaque composant connecté.	L'ensemble de base des types de données est extensible.	Règles de filtrages des messages pour les connecteurs.	Non supporté.	Ensemble des contraintes implicites imposés par le style (ex. un seul lien amissible entre composant et connecteur).	La description des propriétés temporelles n'est pas supportée.
RAPIDE	Déclaration explicite d'un module composant avec ou sans spécification d'interface.	Déclaration implicite.	Existence d'un langage à part dédié à la déclaration ou construction de types de données. Les données sont paramétrables.	Basé sur la théorie de Posets (ensembles partiellement ordonnés d'événements).	Non supporté.	Existence d'un langage à part dédié à la déclaration des contraintes.	La description des propriétés temporelles est supportée.
UNICON	Déclaration explicite. L'interface du composant est spécifiée en associant de module de communication fixés (players) .	Déclaration explicite.	L'ensemble des types de données est prédéfini.	Définition des traces amissibles d'événements.	Supporté.	Ensemble de contraintes implicites imposés par le style. (ex. Limitations des liens possibles entre différents types des composants).	La description de propriétés temporelles est supportée seulement par certains types de composants (ex. composant *SchedProcess*).
METAH	Déclaration explicite sous forme de Processus. L'interface d'un Processus est spécifiée par des portes d'entrée sortie.	Déclaration implicite.	L'ensemble des types de données est prédéfini.	Définitions du comportement via Attributs	Supporté.	Ensemble de contraintes implicite imposés par le style. (ex. Modes)	La description des propriétés temporelles est supportée pour synchroniser l'activité des composant l'architecture ou pour caractériser le système de support.

Langage	*Outil d'édition*	*Outil d 'analyse*	*Génération de Code*	*Système*	*Partner et Sponsor*	*Applicabilité*	*Licence*
WRIGHT	traducteur WRIGHT textuel vers CSP	FDR2 (Failure - DivergenceRe finement) parser , consistance connexions, analyse de deadlock sur connecteurs	non disponible	• Sun Sparc (SunOS 4.1.3 or later , Solaris 2.4 or later) • HP PA-RISC (HP-UX 9.0.5) • IBM RS/6000 (AIX 3.2.5) physical Memory > 32Mb virtual Memory > 64Mb	•Advanced Research Projets Agency (Prototech programm) •AIR Force Office of Scientific Research •National Security Agency •Office of Naval Research (AASERT Programm)	Recherche et Applications industrielles (VLSI, Fault Tolerance in embedded systems , Protocols and security)	8,500 UKP prix pour une license commerciale permanente. FTP exemples fto .comlab .ox .ac.uk /pub/ Pa ckages /FDR
AESOP	AESOP graphique et textuel	Parser ; différents outils de validations pour chaque style	non disponible disponible un outil de glue C++ pour le style Pipe Filtre	//	•California Institute of Technology •Jet PropulsionLaboratory •Carnegie Mellon University	générique	FREEWARE version bêta à l 'adresse: http://www - msim.jpl.nasagov/ ~alan/AESOP/ aesophtml alan@elroy.jpl . nasagov
CHIRON2	Argo graphique et textuel	Argo Parser ; messages de contrôle pour une modélisation correcte	**Argo** génération de code C/C++, ADA, Java prévue actuellement disponible: C	//	//	générique	*//*
RAPIDE	Rap Arch Tools graphique et textuel	•Raptor Animation Tools •POV Poset Browser •Rapide Constraint Checker Parser, analyse des traces et des contraintes	Rapide Compiler & Runtime System génération d'exécutables C, C++, ADA, VHDL, Rapide	SunOS 4.1.3 Linux 2.0 Solaris > 5	•UK DefenceEvaluation & Research Agency •Rolls-Royce Aerospace(CONTESSE programme) •C;S. DraperLaboratory of Cambrige (ONR SBIR programme)	générique	FREEWARE Download à l 'adresse: www .pargstanford. edu/rapide/tools- releasehtml

Langage	*Outil d' édition*	*Outil de validation*	*Génération de Code*	*Système*	*Partner et Sponsor*	*Applicabilité*	*Licence*
UNICON	UNICON éditeur graphique édition guidé par *error warning*	UNICON parser , Rate Monotonic Analisys	génération de code C	Windows (Juillet 98) SunOS > 4.1.3, Solaris , Ultrix (Mars 97) Linux 2.0 (Janvier 97)	•Carnegie Mellon University School of Computer Science and Software Engineering Institute •Siemens Corporate Research •Wright Laboratory •Aeronautical Systems Center •Air Force Materiel Command •USAF •Advanced Research Projects Agency (ARPA)	générique	FREEWARE Download software à l 'adresse: www.cscmu.edu/ ~UniCon/ distribution.html informations: UniCon-Distribution@cs.cmu.edu
METAH	MetaH éditeur graphique édition guidé par *error warning* et contraintes sur les menus de choix	MetaH analyse de l 'ordonnance ment, modèle pour analyse de la robustesse	génération de code ADA, génération de code C en cours d'étude	Windows NT 4.0 mémoire> 64 megabytes	•Honeywell Technology Center •DSSA group •ITO (Information Technology Office)	Avionique Contrôle et Conduite	FREE sur contrat fixé de collaboration adresse: www .htc .honeywell . com / metah à cet adresse une copie d évaluation est disponible (60 jours) metah - info@ htc.honeywell .com

ANNEXE 2 :

Fiches techniques d'ADL (Architecture Description Languages)

Dans cette annexe, nous présentons des fiches techniques qui synthétisent les ADL et les outils associés suivants que nous avons étudié dans le document principal :

- *Fiche 1 : WRIGHT*
- *Fiche 2 : AESOP*
- *Fiche 3 : CHIRON 2*
- *Fiche 4 : RAPIDE*
- *Fiche 5 : UNICON*
- *Fiche 6 : METAH*

Ces fiches ne suivent pas toutes la même forme de présentation car chaque langage a sa spécificité et il est naturel de le faire apparaître.

A. Glossaire

Architecture Fonctionnelle / Logicielle
L' Architecture est un modèle de la structure de l' application qui met en évidence l' ensemble de fonctions participantes au correct fonctionnement de l' application.

ADL (Architecture Description Language) :
Un ADL est un langage qui permet de structurer la description de l' Architecture en termes de Composants, Connecteurs, Configuration.

Composant
Un composant contient la description d' une fonction composant l' Architecture. Il contient un module indépendant de calcul nécessaire à exécuter cette fonction.

Connecteur
Il contient la description des fonctions liées à la gestion de la communication entre composants. Un choix est de laisser le composant gérer indépendamment le protocole de communication avec les autres composants connectés ou laisser le connecteur gérer la communication et sa synchronisation.

Interface Composant
Un composant peut avoir ou non une interface. Une interface explicite les points d' accès au composant en termes de services fournis ou demandés ou en termes des règles sur la modalité d' interaction.

Interface Connecteur
Un connecteur peut avoir ou non une interface. Une interface explicite les points d' accès au connecteur en termes de services fournis ou demandés ou en termes des règles sur la modalité d' interaction.

Configuration
La configuration décrit la topologie de l' Architecture. La description d' une Architecture peut expliciter ou moins une partie configuration. Si explicitée, elle contient la liste de connexions entre composants ou entre composants et connecteurs.

Style
Un style architectural est une description d' architecture réutilisable. C' est à dire, une description suffisamment générique pour permettre la réutilisation du style et suffisamment détaillées pour en garder un ensemble des propriétés caractéristiques qui distinguent un style d' un autre.

Validation
Etape nécessaire à déterminer
- si le comportement du système est cohérent avec ses spécifications
- si son fonctionnement est correct et libre d' erreurs ou de défauts

B. Fiche 1 : WRIGHT

B.1 Introduction

Le langage WRIGHT est un ADL créé par Allen [ALL97] et Garlan. Il est basé sur les concepts architecturaux de : composant, connecteur, configuration. Wright s'appuie sur le langage formel CSP (Communicating Sequential Processes) pour la description des comportements (ou des contraintes).

Après une brève description de CSP nous résumerons dans les paragraphes suivants les caractéristiques du langage WRIGHT.

B.2 CSP

CSP est un langage formel qui a été décrit pour la première fois en 1978 par C.A.R. Hoare [HOA78] et a été mis à jour en 1985 par A.W. Roscoe [ROS85].

En CSP le comportement est spécifié au travers de la description d'un ensemble d'événements, et cet ensemble est appelé processus. La description d'un système en CSP est réalisée à travers un ensemble de composants (processus) actifs indépendants. Ces composants communiquent par le biais de canaux prédéfinis. La technique utilisée dans CSP pour modéliser la coopération des processus est la *composition parallèle*.

En CSP il est possible de définir une relation d'équivalence entre processus, cette propriété est appelée raffinement (*refinement* en anglais). Si le processus P1 est le raffinement du processus P2 alors son comportement est contenu par le comportement du processus P2. Par le biais de cette propriété on peut exécuter une étape d'analyse formelle d' un modèle CSP. Si en effet on définit un processus qui a comme propriété celle d'être libre de « deadlock », et on vérifie que le modèle global est un raffinement de cette propriété, ce modèle global est alors libre de « deadlock », parce qu' il doit contenir les propriétés caractéristiques du processus dont il est le raffinement.

B.3 La description de l' Architecture

Dans le paragraphe suivant nous présentions comment du langage Wright permet de décrire une architecture au travers des concepts de : composant, connecteur, configuration.

B.4 Composant

Un composant décrit un calcul indépendant.

La description d'un composant (voir Figure 1.1) est composée de la description d'une part du calcul et d'autre part de son interface.

La partie calcul représente la description du comportement du composant, c'est à dire l'ensemble des taches qu'il doit exécuter.

Une interface est un ensemble de portes, et chaque porte représente une interaction à laquelle le composant peut participer.

Une porte représente d'une part l'attente du composant sur le comportement de l'environnement avec lequel il interagit et d'autre part la description du comportement que l'environnement attend du composant en regardant à travers la porte.

Il faut préciser qu'en WRIGHT la partie calcul comprend la description entière du comportement du composant et la partie de comportement décrite par le port est un sous-ensemble de la partie calcul qui décrit les interactions du composant avec l'environnement.

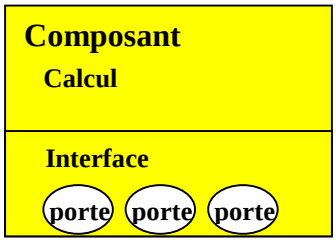


Figure 1.1 : Composant

B.5 Connecteur

Un connecteur représente une interaction entre une collection de composants. Il décrit comment le composant est connecté aux autres dans le contexte actuel de configuration.

La description d'un *connecteur* est composée de la description des *rôles* et d'une *glu* (voir Figure 1.2).

Le *rôle* spécifie le comportement attendu des participants intervenants à l'interaction.

La *glu* spécifie comment le connecteur gère l' interaction des plusieurs connecteurs.

En effet la *glu* décrit le comportement entier du connecteur, la partie de comportement décrite par les *rôles* est contenue dans sa description. Mais les *rôles* explicitent les propriétés demandées aux participants pour garantir la compatibilité de communication.

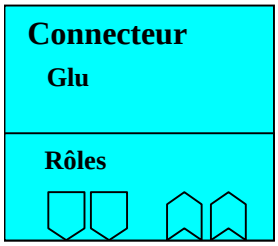


Figure 1.2 : Connecteur

La structure du connecteur répète celle du composant et la distinction entre composants et connecteurs semble superflue. Mais la nécessité de distinguer les connecteurs des composants dérive de l' exigence de mettre en évidence, dans la description de l' architecture, les entités dédiées au contrôle de l' échange des donnés et celles destinés au seul traitement. Cette distinction simplifie la compréhension du comportement globale de l' architecture.

B.6 Configuration

Une configuration est une collection d' instances des composants reliés par le bais de connecteurs (voir Figure 1.3).

Une configuration est composée par la définition des types composants, des connecteurs participants à la construction du système, des instances des types définis et des attachements.

Un attachement décrit l'association entre une porte composant et un rôle connecteur.

La liste des attachements décrit la topologie de la configuration.

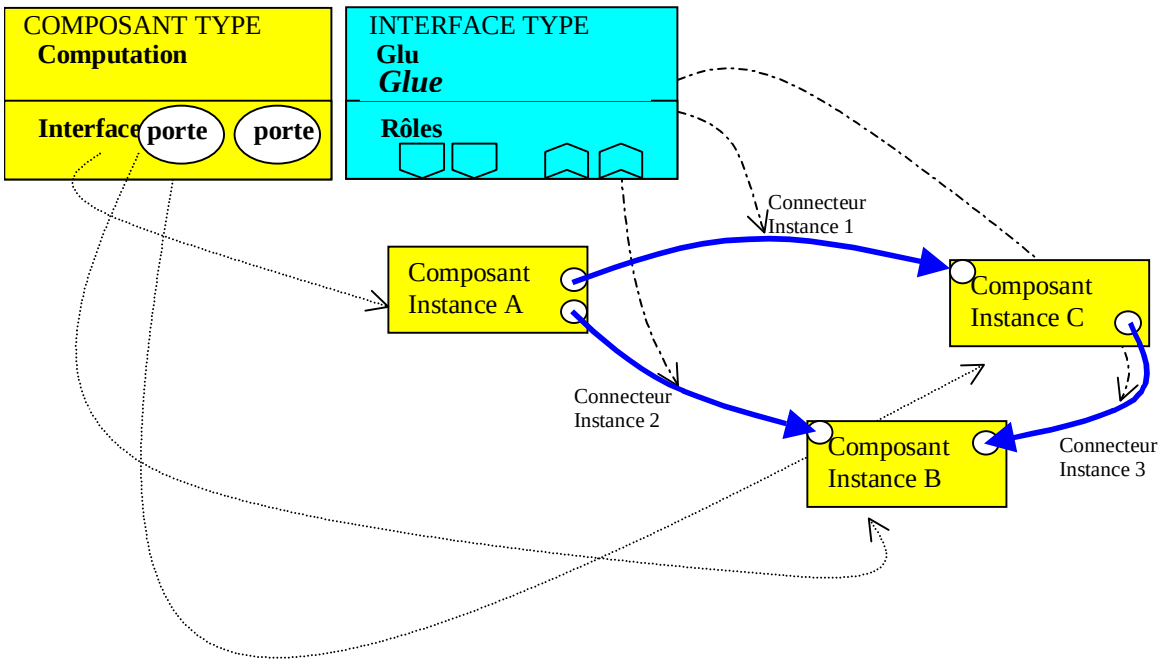


Figure 1.3 : Configuration

B.7 Un Exemple

Dans l' exemple suivant, je proposerai une simple architecture décrite en Wright. Les mots clés appartenant à la syntaxe du langage sont reportés en gras. Le comportement de l' architecture est décrit à un niveau d' abstraction haut, proche du langage naturel.

L' architecture imaginée pour cet exemple s' appelle *Cours*. Elle contient deux types de composants, *Maître* et *Elève*. Le composant *Maître* dialogue avec le composant Elève par le biais d' un connecteur, appelé Voix.

Le connecteur décrit la façon d' intervenir de la communication entre *Maître* et *Elève*.

Cet exemple met en exemple comme même un non expert en techniques de programmation peut comprendre la structure d' une architecture ou la concevoir en utilisant un ADL, et laisser aux experts l' étape d' implémentation. Par exemple, dans le cas du langage Wright, le comportement des composants de l' architecture peut être détaillé en utilisant le langage CSP.

Exemple d' architecture "Cours"

Definition du composant "Maître"

Component Maître
Porte Mots [Sortie de mots nécessaires à expliquer le cours]
Calcul [Se rappeler la dernière leçon traitée. Suivre le programme du cours. Trouver les mots nécessaires à gagner l' attention de l' élève]

Definition du composant "Elève"

Component Elève
Porte Oreille [Entrée de mots nécessaires à apprendre le cours]
Calcul [Se rappeler la dernière leçon traitée. Suivre le programme du cours. Comprendre le sens des mots de la leçon en cours]

Définition du connecteur : voix

Connector Voix
Role Air [délivre continuellement les données par transmission des vibrations sonores]
Glue [Transmettre passivement les donnés émis par une source sonore. Ces donnés son codés en vibrations]

Pour réaliser une configuration il est nécessaire de définir les attachements entre les portes des composants et les rôles des connecteurs.

Configuration Cours
Component Maître
Component Elève
Connector Voix
Instances
Françoise : Maître
Luc : Elève
Discours : Voix
Attachements
Maître.Mots **as** Discours.Air
Discours.Air **as** Elève.Oreille
End Cours

B.8 Les aides à la description de l' Architecture

Les concepts de Hiérarchie, Style, Paramétrage, etc. sont des aides pour la description d' architectures complexes. Dans la session suivante on détail ces concepts.

B.9 Hiérarchie

Wright supporte des descriptions hiérarchisées. Une partie du calcul (composant, glu, connecteur) peut être spécifiée avec un processus CSP ou avec une architecture imbriquée.

Dans ce dernier cas la description de la configuration est complétée par la définition des connexions (*Bindings)* entre les portes internes à un certain niveau hiérarchique et celles au niveau supérieur (voir figure 1.4).

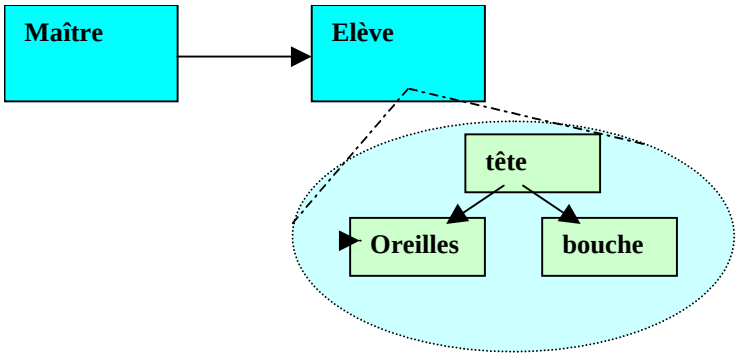


Figure 1.4 : exemple de la hiérarchie

B.10 Style

Wright a été conçu avec l'idée d'encourager l'implémentation de styles architecturaux. Plus précisément on entend par styles architecturaux des descriptions des composants ou systèmes suffisamment génériques pour permettre leur réutilisation et suffisamment détaillées pour en garder les propriétés caractéristiques. Pour cette raison il faut introduire les concepts de contraintes, sous-style, paramétrage et interface.

B.11 Contraintes

Un style doit déclarer des propriétés qui doivent être respectées par toutes les configurations appartenantes à un même style.

Exemple :

Style Cours
Component Maître
Component Elève
Connector Voix
Contraintes [Un seule instance de composant Maître doit être active]
End Elève Model

B.12 Interface Types

Une Interface type représente une déclaration des contraintes sur les portes d'une interface. Il est une sorte de généralisation du concept de style appliqué aux règles d'interactions entre composants.

La déclaration d'une interface facilite l'utilisation répétitive des *rôles* ou *interfaces* déjà décrits pour la description des Interfaces ou Connecteurs différents.

Exemple :

Interface Type DataInput = [lire continuellement les données jusqu'à leur terminaison]

B.13 Sous-Styles

Un Sous-Style hérite toutes les propriétés du style dont il dérive. Cette possibilité représente une façon rapide d'étendre un Style en ajoutant des contraintes ou des propriétés.

Pour déclarer un sous-style, il est suffisant de déclarer le nom du sous-style après celui du style.

Exemple :

Soustyle Elève.Diligent

Contraintes [Pendant que le maître parle, il faut rester attentif et prendre toujours des annotations]

End Elève.Diligent

B.14 Paramétrage

Des parties de la description d'une porte, d'un rôle, d'un calcul ou d'un type peuvent être décrites à travers un *placeholder* qui sera substitué pendant la phase d'instanciation par les paramètres actuels.

Exemple de composant dont le numéro des portes est paramétré :

Component Monstre (n_in : 1 .. n)

Porte Oreille1.. n_in = DataInput

Computation = [écoute continuellement des Oreilles toutes les données]

B.15 Validation sur une architecture Wright

L'étape d'analyse sur l'architecture logicielle a comme but de vérifier la correction du comportement fonctionnel du système. Cette étape peut être divisée en la vérification de la *consistance* du système, c'est à dire la vérification qu'aucun comportement contenu dans la description du système n'est contradictoire, et en sa *complétude*, qui consiste à vérifier le respect ou l'absence de certaines propriétés.

Wright prévoit explicitement 11 tests, listés ci-dessous, pour la validation d'une architecture logicielle, mais comme on verra dans la suite (paragraphe suivant), actuellement seule une partie des tests est exécutable automatiquement.

Port-Computation Consistency (test sur le composant) :

Test pour vérifier la consistance (non-contradiction) entre le comportement d' un composant (computation) et le comportement des ces portes.

Connector Deadlock-free (test sur le connecteur) :

Test pour détecter si le comportement du connecteur (Glue) est sans blocage.

Roles Deadlock-free (test sur le rôle) :

Test pour détecter si un *rôle* est sans blocage.

Single Initiator (test sur le connecteur) :

Test pour détecter l' état de conflit dans l' initialisation d' un événement. Un conflit se produit si plus d' un processus initialise le même événement.

Initiator Commits (test sur le processus) :

Test pour détecter un état de blocage interne à un processus.

Parameter Substituotion (test sur une instance) :

Test pour vérifier la consistance entre le type déclaré pour un paramètre et le type des valeurs assignées aux paramètres pendant l' exécution.

Range Check (test sur une instance) :

Test pour détecter le respect des bornes fixées sur les variables numériques.

Port-Rôle Compatibility (test sur les attachements) :

Test de compatibilité entre les portes et les rôles attachés.

Style Contraints (test sur une configuration) :

Test qui vérifie l' appartenance d' une architecture à un style.

Style Consistency (test sur un style) :

Test qui vérifie la consistance d' un style, c' est à dire la possibilité de dériver au moins une architecture du style déclaré.

Attachment Completeness (test sur une configuration) :

Test pour vérifier si tu les connexions portes - rôles de l' architecture sont configurées correctement.

B.16 Outils de validation disponibles sur WRIGHT : FDR (Failure-Divergence Refinement)

FDR est un atelier de modélisation pour le formalisme CSP. Il a été développé par *Oxford University*.

Il permet de tester, sur le système modélisé, l'appartenance des propriétés particulières (exprimées comme des processus CSP) en vérifiant la propriété de raffinement, en effet si le modèle *refine* la propriété en question alors cette propriété lui appartient.

Il permet aussi de valider certaines propriétés standards déjà définies par le biais d'un test de raffinement comme l'existence de deadlock ou de boucles infinies.

FDR met à la disposition une interface graphique pour analyser les traces qui ont causé une erreur.

Pour appliquer FDR sur WRIGHT il faut traduire un fichier de spécification WRIGHT dans le format accepté par FDR et traduire les propriétés à tester en tests de raffinement.

Pour cette raison actuellement seulement les tests *Port-Computation Consistency, Connector Deadlock-free, Roles Deadlock-free, Port-Rôle Compatibility* sont exécutables automatiquement.

Ils existent des contraintes pour l'utilisation de FDR :

- Les spécifications doivent être un automate à états finis. Il n'existe pas une façon de tester si un processus est un automate à état finis à moins de limiter la description d'un processus à un sous-ensemble de WRIGHT (en empêchant par exemple l'utilisation récursive de l'opérateur de composition parallèle des processus).
- Il est nécessaire de calculer la version déterministe du processus P, *det*(P). WRIGHT, ou plus précisément CSP, permet dans la définition d'un processus d'introduire des opérateurs non déterministes. Ces opérateurs empêchent la vérification de la propriété de raffinement.

Il faut préciser que l'outil FDR conseillé sur Wright n'est pas le seul permettant de tester d'un modèle CSP, plusieurs autres existent comme ProBE et Deadlock Checker, Casper.

B.17 Conclusions

Un des buts des ADL est de faciliter la construction d'un système à partir de la liste des ses spécifiques fonctionnelles, et de permettre une implémentation.

Wright en introduisant les concepts des composants et connecteurs, atteint cet objectif. Et en plus, en introduisant les connecteurs, il permet de séparer les propriétés sur les interactions de composants et les propriétés sur le comportement des composants.

Wright invite à l'utilisation de styles, c'est à dire l'utilisation des familles de systèmes ou composants caractérisés par la définition d'un ensemble des contraintes, cette démarche est utilisée pour simplifier l'étape de configuration d' une architecture. La configuration se réduit à détailler tous les aspects que le style décrit seulement de façon générique, également l' appartenance à un style permet d' héritier toutes ses propriétés.

Wright résout l'exigence de formalité en s'appuyant sur CSP. Cette base formelle donne la possibilité d'exécuter sur le modèle une étape de validation automatique des propriétés fonctionnelles du système en utilisant l'outil FDR qui vérifie la propriété de raffinement. Mais pour exécuter l'étape de validation il faut traduire toutes les propriétés à valider dans un test de raffinement. Il faut noter que plusieurs tests proposés par Wright ne sont pas encore automatiques à cause de la difficulté de traduction de ces derniers dans un test de raffinement.

Le système validé est, en réalité, une approximation du système réel décrit, car les tests sont exécutés sur la version déterministe du Processus.

Il est souhaitable que un ADL puisse permettre l'extensibilité et la révision du système décrit. Wright atteint cet objectif en utilisant les concepts de paramétrage, sub-type, hiérarchie.

Par contre WRIGHT est un langage statique, la topologie du système reste invariable pendant toute son exécution et ne contient pas d'informations ou des facilités sur la gestion du temps.

Donc sur une architecture écrite en Wright il n'existe pas de possibilité de validation du respect de contraintes temporelles.

Pour terminer, aucune référence n' est donnée au problème de la génération du code, et aucun outil automatique d'édition n'est disponible.

B.18 Bibliographie

[ALL97] Robert J. Allen, "A Formal Approach to Software Architecture", School of Computer Science, Carnegie Mellon University, Pittsburg, Mars 1997

[HOA78] C.A.R. Hoare, "Communicating Sequential Processes", Prentice-Hall 1972.

[ROS85] A.W. Roscoe, "The Theory and Practice of Concurrency", Prentice-Hall 1985.

Wright: *www.cs.cmu.edu/afs/cs/project/able/www/wright/*

FDR2: *www.formal.demon.co.uk/FDR2.html*
ftp.comlab.ox.ac.uk/pub/Packages/FDR

C. Fiche 2 : AESOP

C.1 Introduction

AESOP est un projet de Carnegie Mellon University (Pittsburgh) sous la direction de Ralph MELTON [MEL99].

Le système AESOP est un environnement graphique pour l'édition et l'analyse d'architectures logicielles.

Plutôt qu'un vrai ADL, AESOP est un support pour l'utilisation et l'intégrat ion d'outils et langages différents. AESOP n'offre aucune syntaxe de support à la description des comportements des éléments composants de l'architecture. Chaque élément a une liste d'attributs (couples nom-valeur) mais aucun attribut n' est associé à une fonction interprétée par AESOP. Ils représentent seulement des informations pour les utilisateurs ou l'éventuelle utilisation d'autres outils.

AESOP propose un style architectural analogue à celui défini dans WRIGHT qui est basé sur les concepts des composants dotés des portes, connecteurs dotés de *rôles* et configuration.

Ce style est proposé comme squelette commun à tous styles supportés par cet environnement.

La figure suivante représente la structure de l'environnement AESOP.

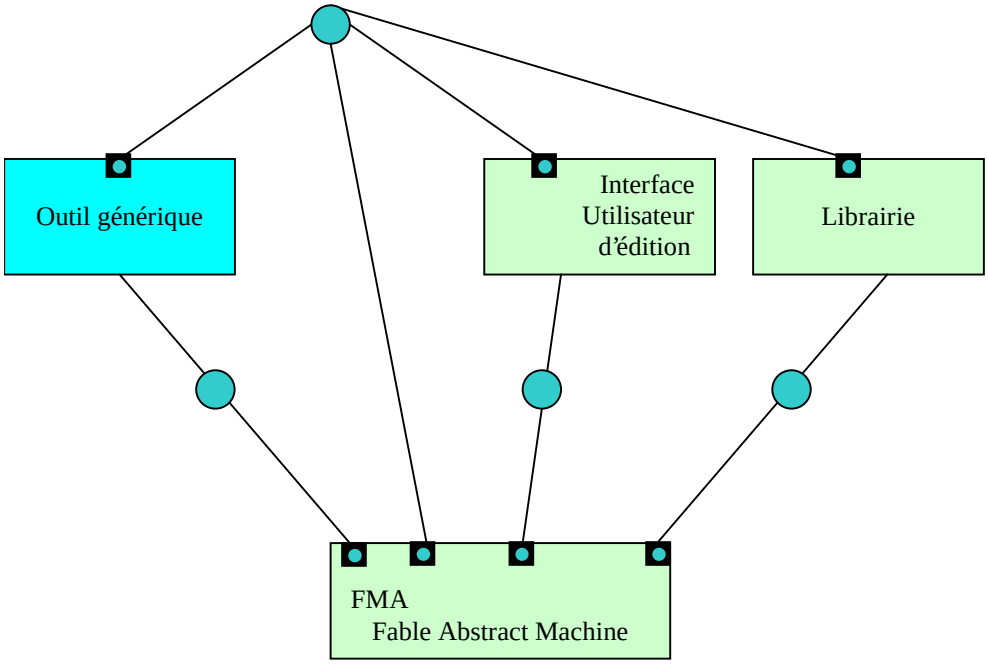


Figure 2.1 : Structure AESOP

- **Interface Utilisateur d'édition :** est l'interface graphique d'édition.
- **Libraire :** est une base de données contenant composants et connecteurs à réutiliser dans l'étape de l'édition.
- **Outil Générique :** est un outil de support à l'environnement AESOP. Aesop est un environnement ouvert qui prévoit la collaboration d'outils pour exécuter des étapes d'analyse des architectures, ou d'édition des modèles, ou nécessaires à la production de code, etc.…
- **FMA :** est une base de données pour styles architecturaux et elle offre une interface pour la réutilisation des données.

Pour la présentation d'AESOP nous avons utilisé la même sémantique prévue pour la construction d'une architecture, sémantique qui sera détaillée dans les paragraphes suivants.

C.2 La description de l' Architecture

Dans le paragraphe suivant nous présentions comment du langage Aesop permet de décrire une architecture au travers des concepts de : composant, connecteur, configuration.

C.3 Composant

Il représente un calcul ou des données. Il est représenté graphiquement par un rectangle.

Un composant peut avoir des **portes**. Une porte représente une interaction du composant avec l'environnement. Une porte est représentée graphiquement par un carré noir sur les bords du composant.

C.4 Connecteur

Il définit les interactions entre les composants. Il est représenté graphiquement par une ligne contenant un cercle au milieu. Un connecteur a des points d'accès appelés rôles. Les rôles, représentés par des ronds aux extrêmes du connecteur, définissent les règles à respecter sur la connexion.

Le cercle au milieu du connecteur contient la description du comportement du connecteur et donc le protocole d'interaction fixé.

C.5 Configuration

Par configuration nous entendons une composition particulière de connecteurs et de composants.

Si les portes et les rôles sont décrits en utilisant WRIGHT alors il est possible de tester la consistance sur les attachements.

C.6 Un Exemple

Nous proposons l' exemple de l' architecture *Cours* (p. 9) dans le cas du langage Aesop.

Dans la figure est montrée l' architecture et les contenu des ses composants.

En Aesop le choix du langage pour la description d' un calcul est arbitraire au moins d' utiliser un style particulier (voir le chapitre Style). Au niveau de conception, on se limite à décrire de façon informelle le comportement de composants.

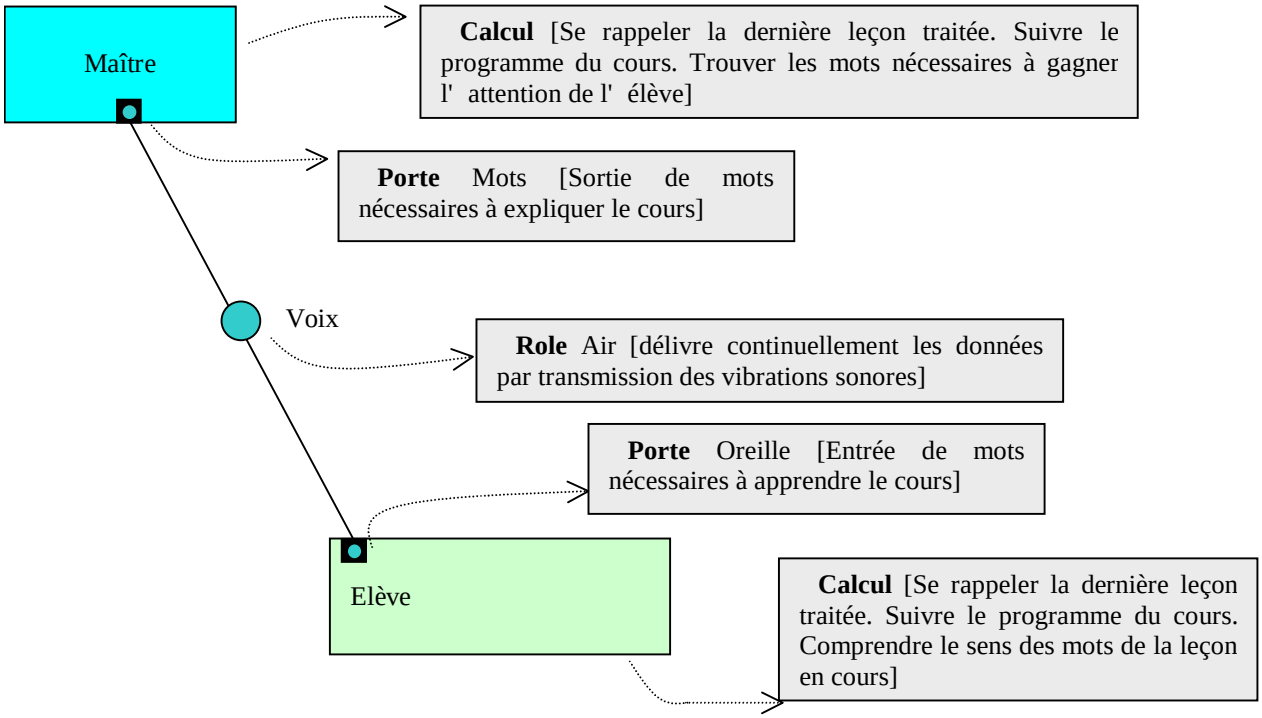


Figure 2.2 : Architecture Cours

C.7 Les aides à la description de l' Architecture

Les concepts de Hiérarchie, Style, sont des aides pour la description d' architectures complexes. Dans la session suivante on détail ces concepts.

C.8 Hiérarchie

Un composant ou un connecteur peut être spécifié sur plusieurs niveaux de description imbriqués.

Dans ce cas, selon le modèle proposé par WRIGHT, la description de la Configuration est complétée par la définition des connexions (*Bindings)* entre les portes internes à un certain niveau hiérarchique et celles au niveau supérieur (voir figure 1.4)**.**

C.8.1.1. Style

Actuellement AESOP se réduit à une collection non homogène de trois styles. On présente ci-dessous ces trois styles pour mieux détailler leurs caractéristiques :

Le style Pipe et Filtre

Dans ce style, les seuls connecteurs qu'on puisse utiliser sont des pipes. Les filtres sont les composants les plus utilisés (et en effet aussi si fortement conseillés, il n'existe pas la contrainte de ne pas utiliser d'autres types de composants).

Un filtre est un composant qui se limite à transformer des chaînes de caractères.

Un *Pipe* gère la communication séquentielle des données entre filtres.

Aucune sémantique ou guide n' est offerte pour ce style ni dans son implantation ni pour une éventuelle validation.

Le style *Pipes et Filtre de l'Unix*

Ce style respecte les contraintes fixées par le style *Pipe et Filtre*, mais en introduisant la contrainte supplémentaire d'utiliser des fichers UNIX.

Les types des composants utilisables sont limités à quatre :

- *StdFilter* : filtre générique.
- *UnixFilter* : filtre Unix (par exemple stdin, stdout). Ce filtre peut être édité en utilisant un outil sur AESOP appelé *Filter Editor*.
- *UnixBinary* : un fichier standard binaire Unix.
- *UnixFile* : fichier normal de test Unix.

L'outil appelé *Build* compile le modèle en produisant un exécutable C seulement si des contraintes supplémentaires (par exemple un fichier doit être toujours connecté à un filtre) sont respectées.

Le style Real-time Message-Passing

Des contraintes fixent ce style.

L'architecture est mono-processeur. Sa capacité de traitement est fixée (cycles par second).

Chaque instruction nécessite un traitement égale à un nombre entier des cycles processeur.

Seulement trois types de composants sont prévus :

- *Equipements* : sont des producteurs périodiques de messages.
- *Processus* : sont des filtres qui transforment des messages en des autres messages. Chaque transformation demande un nombre fixé de cycles de traitement.
- *Ressources* : représentent l'utilisation d'une ressource générique qui est modélisée comme un délai de traitement.

Sur ce modèle il est possible d'exécuter un test d' ordonnancement. Le résultat de ce test donne informations de l' existence de conflit d' accès à une ressource par processus différents.

C.9 Conclusions

Aesop est un environnement en phase expérimentale.

Il définit une sémantique de description des architectures, mais aucune syntaxe de description des comportements, et aucune étape n' est fixée pour l'analyse de cette architecture.

Seule la consistance de connexion est vérifiable si on détaille en WRIGHT (ou mieux en CSP) la description des rôles et des portes. Mais il n'est pas claire si cette étape est implantée dans l'environnement Aesop ou exécutée par collaboration d'outils externes (par exemple FDR).

AESOP ne définit pas un langage pour détailler les comportements des composants de l' architecture.

AESOP est un environnement qui accueille des outils différents pour chaque style. Pour le style générique *Pipe et Filtre* aucun moyen de description du comportement d'éléments et de validation de ce comportement est offerts. Le style *Pipe et Filtre sur Unix* inclut la possibilité d'utiliser un compilateur C associé qui fonctionne seulement au coût d'ajouter des contraintes externes au style. Au contraire, aucune référence n'est donnée pour un éventuel générateur de code pour le style *Real Time Messages*. L'implantation des styles non homogènes, ni compatibles, supportés par outils avec des prestations différentes, réduit la portabilité et la réutilisation des architectures.

AESOP semble donc un langage insuffisant, au moins dans sa première version, à satisfaire aux besoins de description du comportement des composants de l' architecture, de validation, de génération de code.

C.10 Bibliographie

[MEL99] Ralph Melton, "The Aesop System: A Tutorial", School of Computer Science, Carnegie Mellon University, Pittsburg, 1999

Aesop: *www.cs.cmu.edu/afs/cs/project/able/www/able.html*

D. Fiche 3 : CHIRON 2 (ou C2)

D.1 Introduction

C2 est issue d'un projet sponsorisé par *Air Force Command*, *Rome Laboratory*, et *Advanced Research Projects Agency* (Contract Number F30602-94-C-0218). Il est basé sur les concepts architecturaux de composant et connecteur. Il a une syntaxe graphique.

Pour la description du comportement interne des composants aucune contrainte syntaxique n'est fixée.

En effet, à priori, C2 est un ADL multi-langages, il laisse la possibilité de choisir entre plusieurs langages de programmation. Mais l'environnement d'édition et de validation sur C2, appelé ARGO, supporte pour l'instant seulement une implémentation en C++.

Il est affirmé dans [TAY96] que les composants peuvent "tourner" aussi dans un environnement distribué sous l' hypothèse qu'ils soient dotés d'un espace de mémoire partagée, mais il est clair que cette propriété dérive de l'indépendance des composants, de l'action intermédiaire des connecteurs et du fait qu' il n'existe aucune relation entre l'architecture conceptuelle C2 et une éventuelle implantation.

Une caractéristique du style architectural C2 est sa description hiérarchique par niveaux indépendants.

Les composants appartenant à un niveau n'ont pas la connaissance du niveau du dessous. En fait les composants et les connecteurs ont un domaine supérieur et un domaine inférieur et seules les connections entre domaines adjacents sont permises.

Dans la figure ci-dessous nous représentons une architecture typique C2 construite en définissant les connexions (Comunication Link) entre connecteurs et composants.

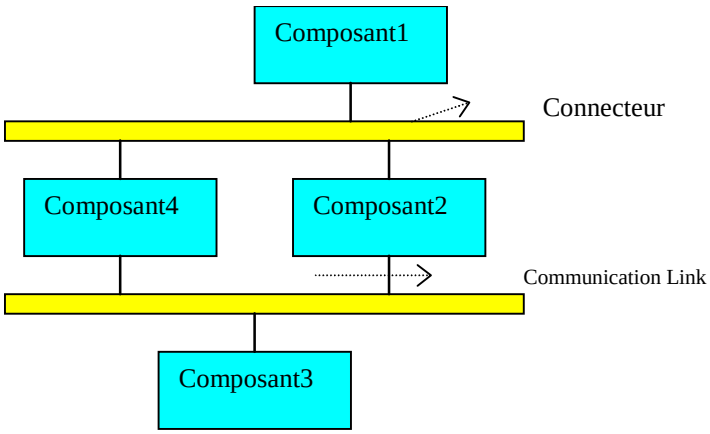


Figure 3.1 : Architecture C2

Dans cette architecture les composants peuvent communiquer seulement par le biais des connecteurs. La communication est de type asynchrone et elle est exécutée à travers d'échanges de messages. Un message est caractérisé par un nom et une liste de paramètres typés. Les messages sont distingués entre notifications et requêtes. Une notification contient des informations sur l'état du composant, donc la valeur des variables caractérisant cet état.

Les notifications vers les couches inférieures sont envoyées sans faire aucune hypothèse sur les récepteurs de ces notifications.

L'extension d'une architecture est simplement exécutée en ajoutant des nouveaux composants ou une sous architecture à un connecteur par le biais de nouveaux liens.

Dans les paragraphes suivants on résumera les propriétés du langage C2.

La Communication dans l' architecture C2

Un ensemble de règles caractérise la communication dans une architecture C2 pour garantir l' indépendance entre une couche à un niveau *n* et les couches aux niveaux inférieurs :

- Un composant peut envoyer une notification seulement aux composants des couches supérieures.
- Un composant reçoit des requêtes de service (appel de fonctions ou de procédures, changement de variables, etc.) seulement du bas et donc, évidemment, il ne peut exécuter une demande de services que vers le haut.
- Un composant peut décider de répondre à une requête avec l' envoie d' une notification.

D.2 Composant

La description d'un composant est caractérisée par deux domaines appelés TOP et BOTTOM.
Le TOP domaine spécifie l'ensemble des notifications aux quelles le composant peut répondre et l'exemple des requêtes qui peuvent être émises.
Le BOTTOM domaine spécifie les notifications qui peuvent être émises vers le bas, dans l'architecture, et les requêtes provenant des couches inférieures de l' architecture, auxquelles il peut répondre.
L'activité des deux domaines est indépendante, ils n' existent pas de relations entre les messages du TOP domaine et du BOTTOM domaine.

Dans la figure ci-dessous, on montre la structure d'un composant C 2.

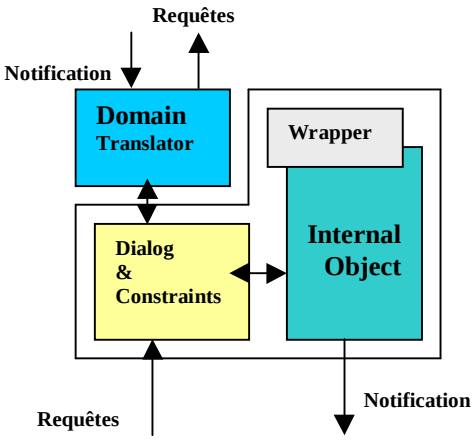


Figure 3.2 : Un composant C2

- **Domain Translator** : block destiné à la projection des noms des messages et des paramètres pour gérer la communication entre blocks ayant une sémantique différente (*TOP domaine*).
- **Internal Object** : block qui contient la description du comportement du composant.
- **Wrapper** : block transforme les informations en sortie de l'objet (valeur, changement d'états) en Notification à envoyer vers les niveaux inférieurs (*BOTTOM domaine*).
- **Dialog &Constraints :** block qui gère l'accès à l'objet et le respecte des contraintes sur cet accès. Une raison d'accès à l'objet peut être une requête ou une réaction à une notification.

D.3 Connecteur

Les composants appartenants aux différents niveaux ne peuvent communiquer que par le biais des connecteurs.
Egalement un connecteur est caractérisé par un domaine supérieur et inférieur.
Les règles de connexion dans une architecture C2 sont les suivantes :

- Le TOP domaine d' un composant doit être connecté au BOTTOM domaine d' un seul connecteur
- Le BOTTOM domaine d' un composant doit être connecté au TOP domaine d' un seul connecteur
- Il n' existe pas de limite au nombre de composant et connecteurs qui peuvent être connectés à un seul connecteur
- Pour lier deux connecteurs, on attache le TOP domaine d' un connecteur au BOTTOM de l' autre

Un connecteur peut être connecté à d'autres connecteurs. Le connecteur distribue les messages avec une politique Producteur/Consommateur et il gère le flux des données à travers l'application de quatre politiques de filtrage :

- Aucun filtrage : Distribution Producteur/Consommateur vers le bas pour une notification, vers le haut pour une requête.
- Notification de Filtrage : les messages sont envoyés seulement aux composants inscrits pour les recevoir.
- Filtrage Prioritaire : Une priorité est associée aux composants connectés. Les notifications sont envoyées en suivant l'ordre de priorité jusqu'au respect d'une condition de terminaison.
- Message Sync : Tous messages sont ignorés.

Dans le langage C2 une Configuration, qui représente les liens entre les composants et les connecteurs est réalisée définie en configurant les connecteurs. Un connecteur distribue les requêtes reçues de couches inférieures vers la couche supérieure. Le connecteur attend les notifications de réponse et il applique la règle de filtrage.

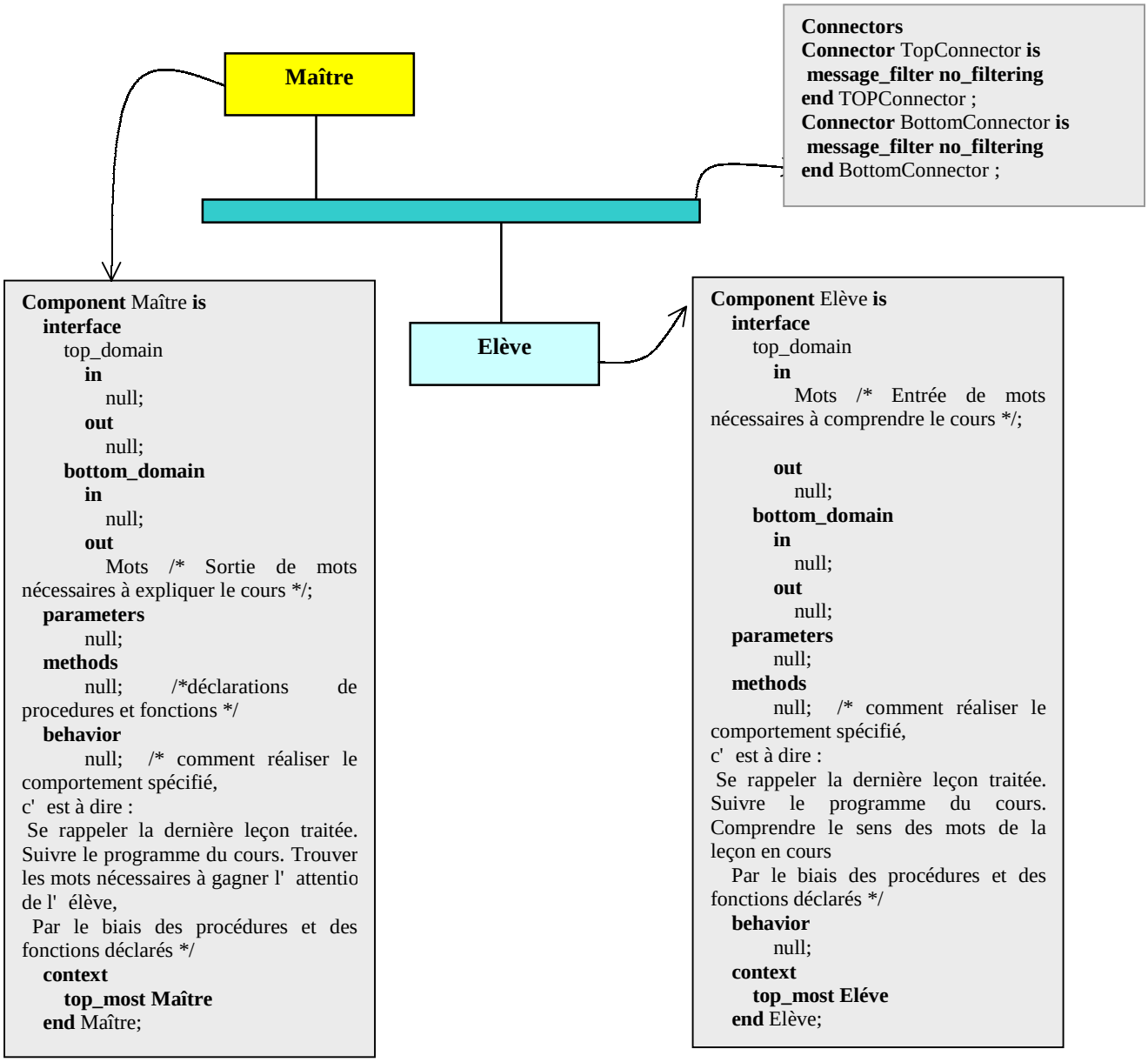
D.4 Un exemple

Egalement dans le cas de C2, nous somme intéressés à concevoir l' architecture : "Cours" (voir p.9).

La première étape est de la réaliser graphiquement : les composants Maître et Elève sont connecté par le biais d' un connecteur.

Il faut définir la politique de filtrage du connecteur. Dans notre cas, tous les messages envoyés par le Maître doivent être reçus par le composant Elève ou plusieurs composants Eléves connectés. Pour cette raison la politique choisi est : Aucun Filtrage (no_filtering). On remarque dans la description du connecteur la présence d' un TOPdomaine et un BOTTOMdomaine qui peuvent avoir politique de filtrage différent.

On définit d' abord le comportement de façon abstraite, en suite il faudrait implémenter les procédures et les fonctions nécessaires à l' implémenter.



D.7 ARGO, un environnement complet d'implantation d'architectures C 2

C 2 utilise l'environnement de validation et d' édition appelé ARGO. Il s'agit d'un environnement graphique. On construit l'architecture en définissant les connexions entre composants et connecteurs. L'interface de chaque composant est représentée comme une collection des messages, c'est à dire les notifications et requêtes relatives respectivement aux domaines supérieurs et inférieurs.

Chaque composant est une application indépendante avec un espace de mémoire propriétaire. L'application conceptuelle décrite dans la syntaxe prévue par l'environnement est traduite en une instance exécutable C++.

Une fois l'architecture réalisée, selon le style C2, on peut entraîner une étape de validation syntaxique et sémantique.

Il est possible de vérifier la consistance entre les services demandés et les services offerts (correspondance du nombre de paramètres, correspondance entre les noms des messages, etc.). D'autres tests sont aussi possibles (vérification de l' état de *deadlock*).

D.8 Conclusions

C2 est un langage graphique qui offre la notion de composant et de connecteur.

Il n'offre pas une syntaxe bien définie pour la description du comportement des composants. D'une certaine façon cette possibilité permet de choisir le langage à utiliser pour décrire le comportement mais représente une limite pour réaliser la vérification du comportement globale du système.

Il n'est pas claire s'il existe en C2 la possibilité d'imbriquer la description des composants (par exemple le comportement d' un composant peut être décrit au travers d' une architecture complète). Pourtant cette propriété est nécessaire pour gérer des architectures de grandes dimensions qui nécessitent une description repartie sur plusieurs niveau d' abstraction.

La critique la plus importante est que C2 supporte un modèle de communication de type Producteur Consommateur. Il n'est donc pas naturel de définir des styles de communication différents comme par exemple celui Client/Serveur.

C 2 n'offre aucun moyen de décrire ou gérer le temps dans la modélisation de l'architecture.

Il possède un outil d'édition et de validation, et la génération du code est prévue.

D.9 Bibliographie

[TAY96] Richard N. TAYLOR, Nenad MEDVIDOVIC *et al.*, "A Component and Message Based Architectural Style for GUI Software", IEEE Tranaction on Software Engineering vol. 22, n. 6, Juin 1996

[MED96] Nenad Medvidovic, "Formal definition of Chiron-2 software architectural style", Technical Report UCI-ICS-95-24, Departement of Information and Computer Science, University of California, Irvine, Avril 1996

Chiron2: www.ics.uci.edu/pub/arch/c2.html
Argo: www.ics.uci.edu/pub/arch/uml/v06/index.html

E. Fiche 4 : RAPIDE

E.1 Introduction

Rapide est issue d' un projet du *Computer System Lab* de l'*Université de Stanford* (David Luckman) [RAP96], [RAP97]. Le projet Rapide a été proposé pour la première fois au *Advanced Research Projects Agency* (ARPA) en 1990. Le groupe TRW (Frank Belz) participe au projet pour l'évaluation de son applicabilité au monde industriel.

Rapide est basé sur les concepts architecturaux d'interface, module et configuration. Ce ADL est composé de 5 sous-langages[6] :

- *Types Languages*

Ce langage permet la déclaration de types composants, interfaces, etc.

Il existe un ensemble de types prédéfinis. Il est possible de définir des types utilisateur à partir de ces types. Il existe le concept de **sous typage**. C' est à dire qu' un sous type hérite de toutes les caractéristiques et propriétés du type d'origine de telle manière qu'un type soit toujours remplaçable pour son sous type.

- *Pattern Language*

Ce langage permet de décrire les relations entre des événements en utilisant un ensemble de règles et d' opérateurs.

Ce langage définit aussi les règles de dépendance, les relations d'ordre et de temps.

Comme le met en évidence la figure 4.1, le langage de définition de contraintes utilise le Pattern Language. En effet nous pouvons imposer, par exemple, un ensemble amissible d' évènements et la contrainte que la trace de simulation doit contenir cet ensemble, c' est à dire que nous définissons une contrainte à aide du langage de description des événements.

- *Architecture Language*

Ce langage permet de configurer l'architecture entière par le biais de trois types d'informations :

✓ *Comportements*

A ce niveau le comportement est définit comme un automate, c'est à dire comme des états (collections d'objets) et les règles de transition entre les états.

✓ *Règles de Connexion*

Une connexion est définie en fixant une association entre différentes interfaces. Cette association met en relations les événements produits par une interface avec les événements produit par l' interface connectée. Par exemple, un événement produit par une interface peut devenir la cause qui franchis l'exécution d' un ensemble d' événements définis dans l' interface connectée.

Des relations plus complexes peuvent être fixées. Par exemple le type Service définit une ressource disponible qu' une interface serveur peut offrir à une interface client.

Un lien plus fort est un *Dual Service*, qui définit dans une même interface soit les ressources offertes soit celles nécessaires pour réaliser la connexion.

✓ *Imbrication d'architecture et Mapping*

Définition des niveaux hiérarchiques et des correspondances entre architectures distinctes.

- *Constraint Language*

Ce langage permet de définir des contraintes sur les comportements.

Définir un ensemble de contraintes sur un composant ou une architecture permet aussi de définir de classes d'appartenance, ce qui aide à la classification et la réutilisation des composants.

Ce langage est en cours d'évolution.

- *Reactive Programming Language ou Executable Language*

Ce langage contient des structures communes de contrôles typiques de langages de programmation.

Un but du projet Rapide est aussi de remplacer complètement ce langage par le *Pattern Language*.

La nécessité de diviser Rapide en 5 langages est dû à la nature expérimentale du projet Rapide, elle permet l'évolution séparée des différents langages.

Les relations entre ces langages sont présentées dans la figure ci-dessous. La flèche indique une extension syntaxique, c' est à dire $a \rightarrow b$ indique que le langage *b* est une extension du langage *a* et *a* est contenu dans *b*.

Par exemple chaque déclaration est exécutée en utilisant le *Type Language*, chaque langage contient une partie déclaration et pour cette raison le *Type Language* est la racine du schéma ci-dessous.

[6] Pour chacun de sous-langage, un manuel de référence séparé est disponible à l' adresse *www.parg.stanford.edu/rapide/*

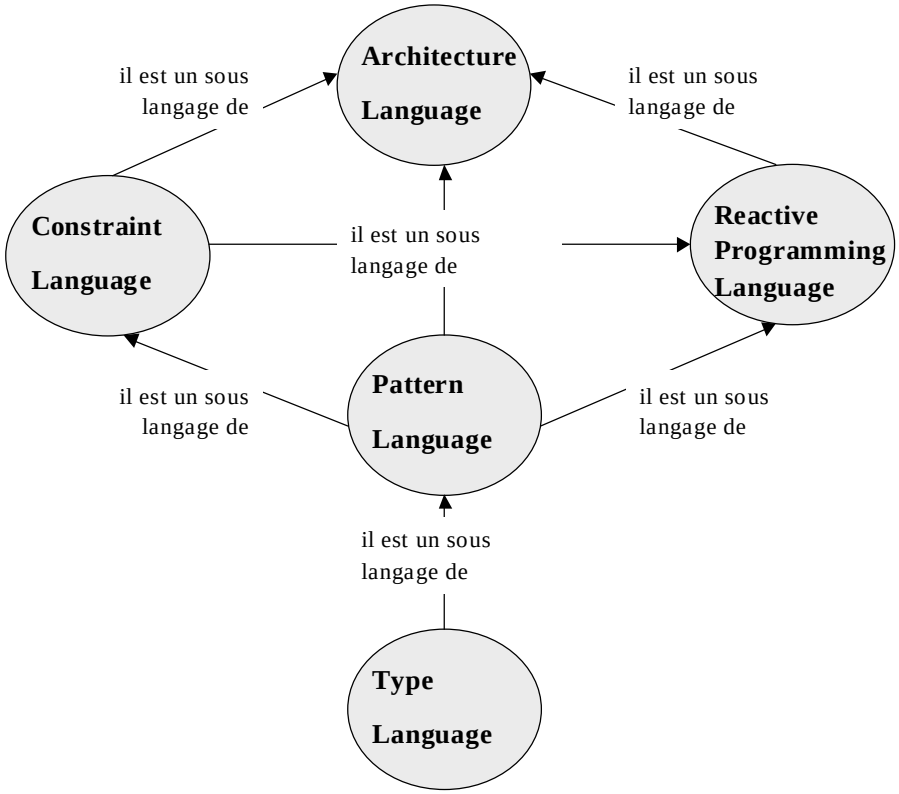


Figure 4.1 : Relation entre les cinq langages de RAPIDE

La description de comportements ou calculs de tous les éléments composant une architecture Rapide se fait en fixant les règles réactives entre événements individuels. Une règle réactive définit des relations de causalité entre ensembles d'événements. Un événement est donc l'unité élémentaire de la description d'un comportement.

Les relations de causalité sur événements peuvent être datées par rapport au temps. Le temps de simulation est un temps physique, il existe en effet une horloge commune de référence.

La simulation d'une Architecture Rapide se réduit en dernière instance en traces d'événements partiellement ordonnés (*posets*).

Les événements exécutés par les connexions individuelles sont indépendants sauf d'éventuelles relations de dépendance explicitement spécifiées**.** Comme pour les connexions, l'exécution d'événements qui décrivent les Comportements/Processus est locale au module et aux interfaces.

Rapide prévoit aussi un ensemble de règles de *Mapping* qui transforme l' exécution d' une architecture ou d' une interface en l' exécution d' une autre. Cette projection est définie par une fonction d' association d' un ensemble d' événements domaine (architecture ou interface d'appartenance) en un ensemble image (l' architecture ou interface associée).

E.2 Événements

Un événement est un objet typé qui est généré en appelant une action.

Les constituants de l'objet événement sont :

- L'action génératrice d' un événement
- Les paramètres nécessaires à l'exécution de l'action
- Les informations sur les éventuelles relations de dépendance entre événements
- La durée de l'événement notée par le couple (t1, t2), où t1 est inférieur ou égale à t2, indique le moment de commencement et de terminaison de l'événement.

Un comportement Rapide garantie la consistance de l'exécution par rapport au temps et une relation de dépendance. Les propriétés suivantes de consistance sont toujours respectées (autrement le modèle ne sera pas correct au niveau syntaxique et donc non compilable) :

Un événement ne peut jamais être dépendant d'un événement futur, et cette indépendance doit être respectée n' importe quel temps est considéré comme origine de la simulation. Si par exemple l'événement *a* dépend de *b*, *a* précèdera *b* sur toutes les origines de temps définissables.

E.3 Temporisation

Un type générique ou un module peut être temporisé en lui associant un timer.
Il est en effet défini en Rapide le type Clock.
A chaque événement est associé un temps de départ par rapport à l'origine commune des temps, et une durée t2 - t1.
Il est possible d'associer à une action une durée déterminée, c'est à dire que l'évent produit par l'exécution de cette action aura cette durée. Si aucune durée n' est pas fixée, l'événement est exécuté avec une durée nulle.

E.4 Composant

Un composant est soit une interface, soit un module doté de sa propre interface, soit une sous-architecture dotée d' interface.
L'**interface** définit la partie visible du composant et un composant peut se réduire à la spécification de la seule interface. Elle se compose (comme montré en figure 4.2) de sept parties toutes optionnelles.

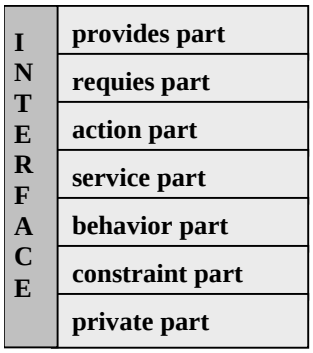


Figure 4.2 : Composition d'une interface

- *provides part*: contient les déclarations des fonctions, modules, types que le composant met au service de l' environnement.
- *requires part*: contient les déclarations des fonctions, modules, types que le composant doit posséder pour pouvoir collaborer avec l' environnement à la construction de l' architecture.
- *actions part*: peut être une *action in* ou une *action out* et indiquent les messages avec lesquels le composant peut communiquer de façon asynchrone avec l' environnement.
- *service part*: contient la description de services (composition des messages, protocole) que le composant met à disposition. En cas de *dual services*, les services qui sont nécessaires au composant pour garantir la consistance d'éventuelles connexions sont explicités aussi.
- *behavior part*: contient la description du comportement de l' interface. Cette description est donnée comme un ensemble des règles réactives sur les évenements.
- *constraint part*: contient les conditions que le comportement de l' interface doit respecter (conditions sur les entrés et les sorties, sur successions d' évents admissibles, etc…).
- *privat part*: contient une partie de description de l' interface visible seulement aux composants qui possèdent la même interface. Cette partie encourage l' interoparibilité des modules différents dotés de la même interface.

E.5 Description Générique d' une interface

```
type NOM is interface
provides
    function <function> ;
end NOM ;

type NOM(paramètres) is interface
      action out NOMaction_out(paramètres);
      action in NOMaction_in(paramètres) ;
behavior
 règles réactives
end NOM ;
```

Le **module** contient la description du comportement (exécutable) du composant et il est composé des sept parties optionnelles. La description de l' exécutable est décomposée en trois parties : initiale, processus, finale.

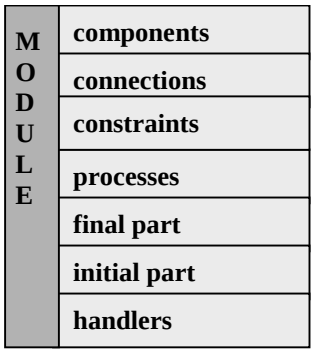


Figure 4.3 : Composition d'un composant

- *Components* : //Liste de composants constituant le module.
- *Connections* : //.Liste des liens entre les composants constituants le module.
- *Constraints* : Les contraintes défins sur les comportements du Module.
- *Processes* : //Corps de l' exécutable
- *Final part* : //Part initiale de l' exécutable.
- *Initial part* : //Part final de l' executable
- *Handlers* : //Des gestionnaires d' exemptions (*handlers*) sur l' exécution du code du module

E.6 Description Générique d' une interface

module NOM *(NOMinterface,…NOMinterface)* **return** …type… **is**
function NOMfunction **is** …. **end** ;
begin
…
body
….
end NOM

E.7 Architecture

Une architecture est constituée des composants connectés. Sa description est complétée par la définition de contraintes sur la configuration.

Un composant est soit une interface, soit un module avec sa propre interface.

Ils existent deux types de connexions : connexion de base et connexion complexe.

Une connexion de base représente une identité entre une pair de fonctions, actions, services listés dans les deux interfaces.

Une connexion complexe est un type interface qui définit le comportement à travers d' un ensemble de règles réactives qui gèrent le flux d' événements en entrée et en sortie.

Rapide définis deux connections complexes :

- Pipe, c'est à dire une connexion pipeline,
- Agent, c'est à dire un connecteur multi-threaded.

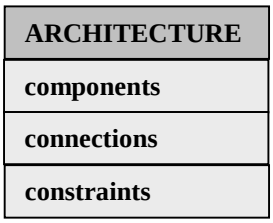


Figure 4.4 : Architecture RAPIDE

- *Components* : Les Interfaces et / ou les Modules qui réalisent l' Architecture.
- *Connections* : La liste des connexions entre les interfaces.
- *Constraints* : Les contraintes défins sur l' Architecture.

E.8 **Description Générique d' une interface**

architecture NOM **return** NOMtype **is**
liste d'interfaces(paramètres instantiés)
connect
Interface.action_out => Interface.action_in /* règles réactives de connexion */
end architecture NOM ;

type NOM(parametres**) is interface**
provides
functions NOM(paramètres) ;
requires
function NOM(paramètres) ;
end NOM ;

module NOM(paramètres**) return** type **is**
function NOMfunction **is** …. **end** ;
begin
…
end NOM ;

Avec Rapide il est possible de réaliser des architectures dynamiques, dans le sens où le nombre de composants peut varier pendant l' exécution de l' architecture et les connexions entre les interfaces peuvent exister ou non en fonction des conditions d'exécution. Cette possibilité est offerte parce qu' il est possible de soumettre la création d' instances et les connexions à des règles réactives.

L' existence d' une partie de déclaration des contraintes permet la réalisation d' architecture de référence ou styles architecturaux. En effet la description de l'architecture est donnée par un ensemble des contraintes que toute architecture appartenante au style doit respecter. Dans le style, la description du comportement exécutable est optionnelle.

Description des contraintes

Dans le premier exemple le comportement de l' interface n' est pas explicité.
Une partie déclaration définis les contraintes sous forme de règles réactives.
Cette interface peut servir de référence pour la déclaration d' interfaces, appartenant au même type, dont le comportement sera spécifié selon exigences.

type NOM(paramètres**) is interface**
action out NOMaction_out(paramètres);
action in NOMaction_in(paramètres) ;
behavior
…Vide…
constraint
observe from
NOMaction_out, NOMaction_in
match
liste de règles réactives ;
end ;
end NOM ;

Dans cet exemple une architecture de référence est définie à l' aide de la déclaration d' une partie contraintes.

```
architecture NOM return NOMtype is
 liste d'interfaces(paramètres instantiés)
connect
…Vide…
constraint
  observe from
    NOM_interface.NOMaction_out, NOM_interface.NOMaction_in
    match
    liste de règles réactives ;
   end ;
  observe from
    NOM_interface.NOMaction_out, NOM_interface.NOMaction_in
    match
    liste de règles réactives ;
   end ;
end architecture_NOM ;
```

E.9 Un Exemple

Pour concevoir en Rapide l' architecture choisie comme exemple, c' est à dire l' architecture "Cours" (voir p.9) on fixe d' abord les interfaces composantes l' architecture. Le comportement est spécifié en première instance avec des commentaires. Ce comportement sera en suite substitué par un ensemble de règles basées sur la théorie des *posets*. Dans une étape encore successive, on peut décider, en cas de comportement complexes, de réaliser de modules implémentant les fonctionnes qui sont appelées dans les interfaces. La topologie de l' architecture est fixée en réalisant les liens entres les interfaces.

```
architecture Cours return Courstype is
 Maître
 Elève
connect
Mots.action_out => Oreilles.action_in
end architecture Cours ;

  type Maître is interface
        action out Motsaction_out(paramètres);
  behavior
   /* Se rappeler la dernière leçon traitée. Suivre le programme du cours.
  Trouver les mots nécessaires à gagner l' attention de l' élève */
  end Maître ;

  type Elève is interface
        action out Oreillesaction_in;
  behavior
   /* Se rappeler la dernière leçon traitée.
  Comprendre le sens des mots de la leçon en cours */
  end Maître ;

end Cours ;
```

E.10 Une famille d' outils sur RAPIDE

Un ensemble d' outils est prévu pour réaliser les étapes d' édition, de validation, de compilation d' une architecture Rapide.

Dans la figure ci-dessous nous représentons les relations entre ces outils et les fonctions respectives :

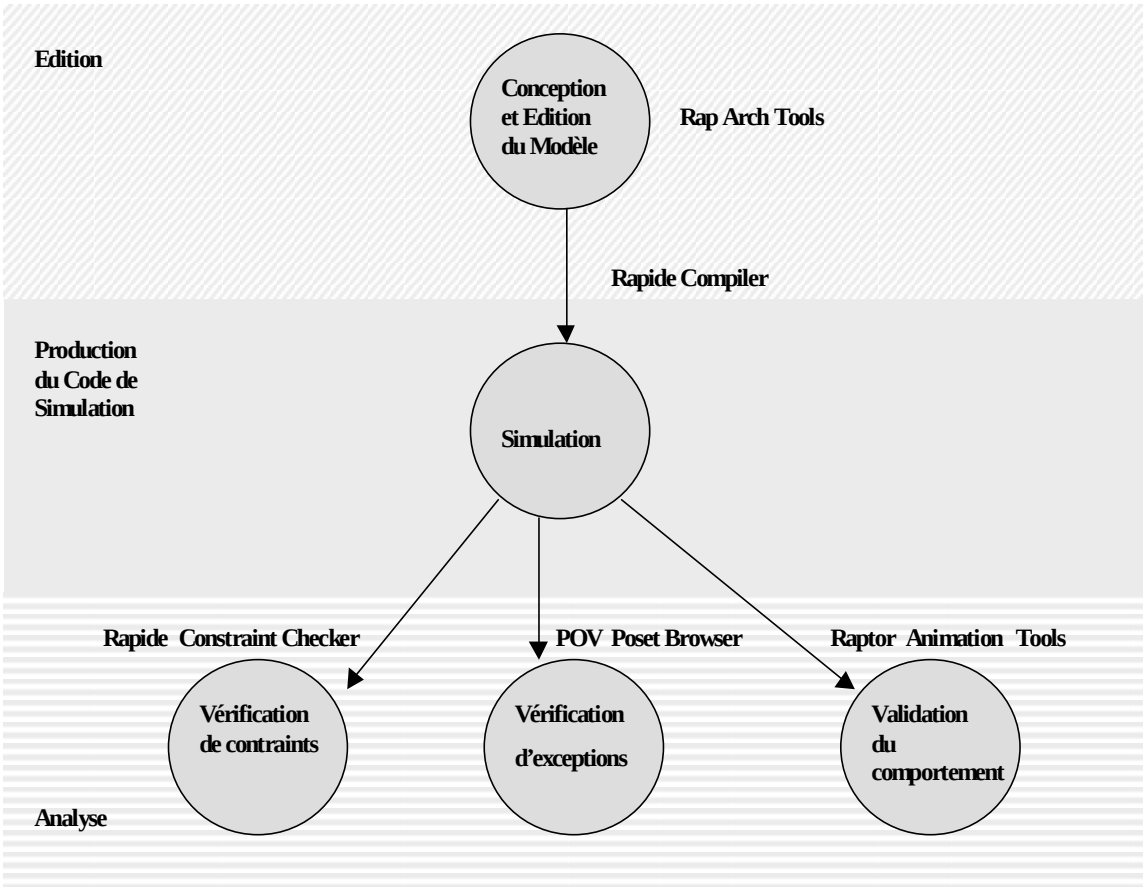


Figure 4.5 : Relations entre outils supportants RAPIDE

Deux parmi eux sont disponibles actuellement :

- ***Rapide Compiler*** ("rpdc") : cet outil traduit les programmes écrits en Rapide-1.0 en modules exécutables ou en informations relatives à la librairie utilisables pour construire des exécutables. En effet il existe la possibilité de gérer[7] une librairie Rapide contenant des modules compilés pour une leur utilisation différée.

  Le code généré par compilation d' un programme écrit en Rapide est le C++.

- ***Partial Order Viewer*** ("pov") : cet outil permet de visualiser un ensemble partialement ordonné d' événements produit par un calcul Rapide. Il est possible de filtrer les événements à visualiser pour en faciliter l' étude.

Ces outils sont en phase expérimentale.

E.11 Conclusions

Rapide est un langage intéressant, complet, qui permet de séparer toutes les étapes de description de l'architecture logicielle, de description du comportement et de la configuration globale (Architecture Language, Constraints Language), de description de comportements individuels (Pattern Langage), de réutilisation du squelette architectural défini (Constraints Language). Il permet la hiérarchisation (lisibilité) et le sous typage (reutilisabilité, lisibilité), le paramétrage (reutilisabilité). Il permet aussi la description temporelle des comportements. Par contre, comme tous les ADL, Rapide ne permet pas de compléter la description de l'architecture avec des paramètres liés au système support physique utilisé.

[7] Les services disponibles sont : créer, effacer, examiner une librairie de modules.

E.12 Bibliographie

[RAP-96] Program Analysis and Verification Group Computer System Lab, "The Rapide 1.0 Full Syntax Reference Manual", Stanford University, September 5, 1996

[RAP-97] Program Analysis and Verification Group Computer System Lab, "Draft, Guide to the Rapide 1.0", Language Reference Manual, Stanford University, July 17, 1997

Rapide: www.pavg.stanford.edu/rapide

DARPA (Defence Research Projects Agency): *www.arpa.mil/organization.html*

F.Fiche 5 : UNICON

F.1 Introduction

Le projet UNICON est supporté par le Carnegie Mellon University School of Computer Science et le Software Enginnering Institute[8]. Il est subventionné par Siemens Corporate Research, Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF (United States Air Force), Advanced Research Projets Agency (ARPA).

UNICON est un DSSAL[9] basé sur les concepts des composants et connecteurs décrits de façon symétrique. UNICON est aussi un environnement graphique d'édition.

Dans l'image ci-dessous on montre un diagramme typique réalisé par cet éditeur représentant deux composants disjoints.

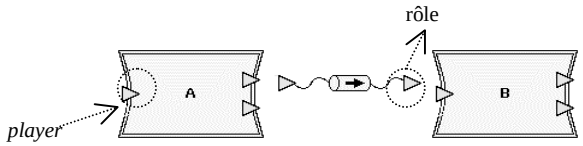


Figure 5.1 : Exemple d'édition dans UNICON (image tirée de : *www.cs.cmu.edu/afs/cs/project/vit/www/unicon/reference-manual)*

Mais une caractéristique propre de UNICON est que chaque composant est caractérisé par une partie spécification et une partie implémentation.

UNICON n'est pas un outil de validation proprement dit, l'architecture réalisée doit être conforme à la sémantique du langage. Le choix des combinaisons des composants est souvent obligé par des styles architecturaux implicites, dans le sens où seulement un ensemble de combinaison est valable.

En fin UNICON ne permet pas de décrire une architecture au même niveau d' abstractions des ADL étudiés dans les fiches précédentes (c' est à dire par exemple en faisant abstraction de la façon de décrire le comportement de composant l' architecture. Pour cette raison nous renoncerons a présenter l' exemple de l' architecture *Cours* (p.9).

F.2 Composants

Les Composants sont spécifiés en définissant leur interface.

Une interface contient les services que le composant s'engage à offrir à l'environnement, et les contraintes que l'environnement s'engage à respecter pour utiliser le service du composant.

Une interface contient trois types d'informations :

1. ***type du composant*** : chaque type est associé à une fonction, il est possible de choisir entre un numéro prédéfini de types. UNICON possède huit types prédéfinis.
2. ***propriétés* :** une propriété est un couple attribut/valeur, nécessaire pour définir les assertions ou contraintes sur le composant. UNICON définit 35 propriétés différentes, à chaque type est associé un sous-ensemble de ces propriétés. Une propriété peut être nécessaire ou optionnelle. Une propriété optionnelle peut rester non spécifiée, elle sera substituée par une valeur par *défaut* en phase de compilation.

Exemples des propriétés sont :

- ✓ *Processor* : utilisée pour assigner l'exécution d'un composant sur un processeur particulier définit dans l'environnement d'exécution.
- ✓ *Variant* : utilisée pour spécifier les fichers sources qui implémentent le composant. Plusieurs alternatives sont donc définissables, le compilateur choisira la première disponible dans l'ordre de la liste.
- ✓ *InstFormals* : utilisée pour déclarer des variables (de type chaînes de caractères) qui seront visibles localement dans le composant ou connecteur qui contient leur déclaration.
- ✓ *Signature* : utilisée pour spécifier les types de données, des arguments ou des valeurs retournées et utilisées par les *players*.
- ✓ …

[8] Sponsorisé par le U.S. Departement of Defense

[9] "Modélisation d' architecture d' application, Moyens et Formalismes", p.5

3. ***définitions des players*** *:* les *players* représentent la partie visible du composant, et les portes nécessaires pour interagir avec un composant.

La définition d'un *player* contient :

- Le type : chaque type est associé à un protocole d'interaction du composant. Il existe 14 types de *players* prédéfinis.
- Les propriétés : c'est à dire la définition des attributs et contraintes que le *player* doit respecter.

Ci-dessous nous décrirons un exemple de types composants UNICON.

Description du composant de type *Module*

Module : il représente une unité de compilation écrite dans un langage général de programmation.

L'interface d'un composant module peut avoir les sept *players* suivantes qui définissent les règles d'accès aux fonctions exécutées par le composant, au données, …

Players :

RoutineDef : définition de fonction ou procédure exportée
RoutineCall : appel à une fonction ou procédure externe
ReadFile : service de lecture d'un ficher,
WriteFile : service d'écriture d'un ficher,
GlobalDataDef : référence à une déclaration de données externe
GlobalDataUse :exportation d'une déclaration de données
Plbundle : collection de players *RoutineDef, RoutineCall, GlobalDataDef, GlobalDataUse.*

Propriétés : *InstFormals, Variant, Processor, EntryPoint, Library.*

Implémentation : le code source d'implémentation d'un Module est simplement une collection externe des fonctions ou donnés prédéfinis, en C

Un type intéressant de composant est le **SchedProcess** qui représente un processus géré selon un algorithme d'ordonnancement fixe. L'algorithme gère un ensemble des processus périodiques en compétition pour une ressource processeur et avec nécessité de respecter des contraintes de temps. Dans la définition des propriétés, il faut spécifier des informations sur la période d'exécution de processus, leur priorité et sur combien des segments de cycles CPU le processus peut être décomposé.

F.3 Connecteur

Le connecteur spécifie le protocole d'interaction entre composants. La description d'un connecteur est composée de trois parties :

♦ *type du connecteur* : chaque type est associé à un protocole, il est possible de choisir entre sept types prédéfinis.

♦ *propriétés* : une propriété est un couple attribut / valeur, nécessaire pour définir les assertions ou les contraintes sur le connecteur.

♦ *définitions des rôles :* les rôles représentent la partie visible du connecteur. Ils sont les points de connexion avec les composants. Un rôle définit les caractéristiques et les responsabilités demandées à un *player*.

La définition d'un rôle contient :

1. **Le type** : chaque type est associé à un protocole d'interaction diffèrent. Ils existent 11 types de *rôles* prédéfinis.
2. **Les propriétés** : c'est à dire la définition d'attributs et contraintes que le *rôle* doit respecter. Dans la définition d'un rôle, les propriétés spécifient quel *players* peuvent être acceptés pour réaliser la connexion et le nombre minimum et maximum de connexions *player*/*rôle* acceptées.

F.4 Implémentation

Il faut distinguer implémentation d'un composant et d'un connecteur.

L'implémentation d'un composant a deux formes possibles :

- *primitive* : l'implémentation est définie par un pointeur à un fichier source externe à UNICON. A priori le fichier peut être un ficher source écrit dans un langage de programmation quelconque. Mais actuellement UNICON supporte seulement le langage C.L' outil UNICON effectue pendant la génération du code l'association des éléments (composant, connecteur,…) sur la base des fichiers sources disponibles en librairie (types prédéfinis).Chaque élément UNICON possède la propriété

appelée VARIANT, cette propriété n'est autre que la définition d'un pointeur à un fichier source (dans un certain système d'exploitation) qui implante le type en question.
UNICON ne vérifie pas la consistance entre le type et le fichier source qui l'implante.

- *composite* : c' est une implémentation composée de plusieurs *primitives*.

Une implantation *composite* est composée de trois informations :

- ✓ *pieces :* représente la liste de composants/connecteurs dont est composée la configuration.
- ✓ *informations sur la configuration :* décrivent les connections entre connecteurs et composants.
- ✓ *Informations abstraites :* contiennent les informations sur la façon d'implanter les *players* abstraits de l'élément *composite* à travers les instances des *players* réels qui le composent.

Dans le cas d'un connecteur, seule l'implémentation de type *primitive* est possible, c'est à dire des types prédéfinis et non composés. L'implémentation des connecteurs est interne à UNICON, c'est à dire que le compilateur exécute directement l'association connexions code sans faire référence aux fichiers sources externes.

F.5 Conclusions

UNICON a toutes les caractéristiques d' un ADL, les concepts architecturaux de composant et de connecteur, il explicite les interfaces soit du composant soit du connecteur, permettant ainsi de construire des architectures imbriquées.

La description des composants de l' architecture ne fait pas abstraction de l' implémentation et le lien entre l' architecture fonctionnelle et l'architecture matérielle support est réalisé à l' aide des propriétés.

Ces propriétés contiennent les informations nécessaires à la distribution des applications sur les processeurs, à l' ordonnancement des processus, à la projection entre les types composants et les fichiers exécutables.

L' utilisateur peut composer les types à sa disposition pour réaliser son architecture, mais cette composition est aussi guidée par des contraintes encore définies par les propriétés des connecteurs.

Ces contraintes sont essentiellement des limitations entre la composition des *players* et *rôles*, (consistance entre interface composant et connecteur) et l' impossibilité de réaliser des types utilisateur.

F.6 Bibliographie

UniCon Langage Reference Manual: www.cs.cmu.edu/afs/cs/project/vit/www/unicon/reference-manual

G. Fiche 6 : MetaH

G.1 Introduction

Ce langage a été élaboré par *Vestal* [VES93] et son équipe à *Honeywell Technology Center*. C'est l' un de projets suivis par le groupe DSSA (Domain-Specific Software Architectures). DSSA dans le domaine du contrôle-commande a été fondée par le Defense Research Projects Agency (DARPA). MetaH est un langage à la fois textuel et graphique basé sur le concept d'objets composants. Pour la même raison que dans le cas de UNICON (§ *Introduction* p.31)nous renoncerons à présenter l' exemple de l' architecture *Cours* (p.9) en utilisant MetaH.

G.2 Composant

La description d'un composant est caractérisée d'une part par son implémentation et d'autre part par son interface.

Plusieurs implémentations alternatives peuvent être définies pour une même interface.

Les unités élémentaires de description du comportement du composant sont appelées processus. Il est possible de définir des Macros, c'est à dire des connexions de processus traitées comme une unique entité.

Des processus périodiques et apériodiques sont supportés. La différence entre les deux est que le premier est réexécuté périodiquement en fonction de la période fixée, le deuxième est activé sur un événement fixé.

La description de l' implémentation d'un processus consiste-en :

- définir l' adresse du ficher source qui l' implémente
- spécifier un ensemble des paramètres comme par exemple la période d'exécution.

L'interface d'un processus décrit les portes d'entrée / sortie. Des variables buffer d'entrée/sortie sont associées à chaque porte. Les processus sont connectés en définissant les liens entre les portes.

Les portes sont typées et il est possible de connecter seulement des portes du même type. En effet, connecter deux portes est équivalent à fixer une opération d' assignation entre les variables de sortie d' une porte et les variables d' entrée de la porte connectée. Cette opération est admissible seulement entre variable du même type (type associé à la porte). L' opération d' assignation est exécutée périodiquement selon la valeur de la période fixée pour le processus.

En MetaH,il est possible de synchroniser l'accès des processus à une ressource commune par activation de sémaphores.

Meta H offre la possibilité de partager des parties de code et des données sous forme de *packages* communs. Par exemple, des interfaces différentes qui contiennent la référence à un *package* feront référence à la même copie physique qui l' implémente. Cette possibilité permet donc de gagner espace mémoire.

Egalement la description d'un processus peut être détaillée hiérarchiquement, un composant peut contenir plusieurs composants imbriqués et un *packages* peut être décomposé en sous-programmes.

G.3 Mode

MetaH introduit le concept syntaxique de *mode*. Un mode représente une configuration particulière d'un ensemble de processus connectés. Sur plusieurs modes définis sur un même ensemble de processus seulement un à la fois peu être actif. Il est également possible de fixer des règles de transition d'un mode à l'autre.

La définition d'un mode représente la possibilité de paramétrer la configuration d'un ensemble de processus et de gérer dynamiquement cette configuration en définissant des règles de transitions entre modes différents.

G.4 Description du matériel

La plus importante caractéristique de MetaH est la possibilité qui offre de décrire de façon homogène software et hardware. MetaH offre la possibilité de modéliser l'application et le système sur lequel l'application s'exécute. Un avantage évident de cette approche est la possibilité d'obtenir par simulation une évaluation des prestations de l'ensemble application/système support, c'est à dire des informations sur l'ordonnancement de taches, le pourcentage d'utilisation de ressource, etc.

Les composants de l'architecture matérielle sont : processeurs, mémoires, canaux de communication entre processeurs.

G.5 MetaH Outil Environnement

La figure ci-dessous montre l'environnement MetaH actuel.

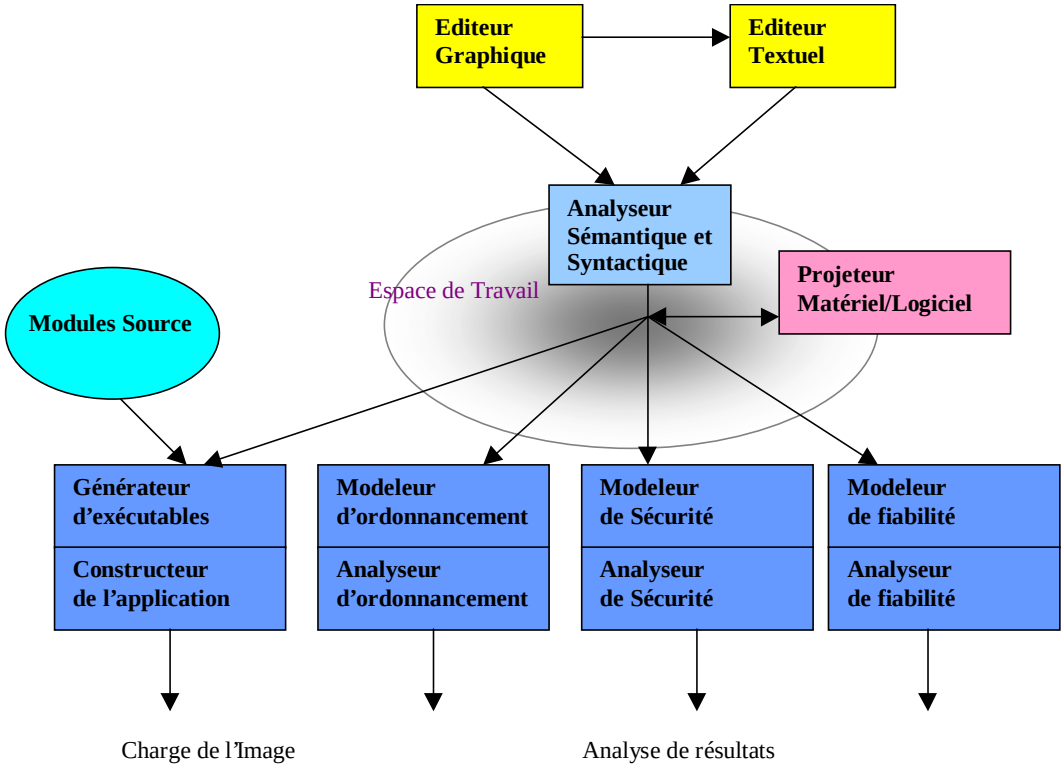


Figure 6.1 : Environnement MetaH

- **Editeur Graphique/Textuel** : permet l' édition du modèle MetaH. Les deux représentations graphique et textuelle sont équivalentes et la conversion automatique de l'une à l'autre est supportée.
- **Projeteur Logiciel/ Matériel** : représente un guide sur une projection correcte d'une entité logicielle (programme, connexion porte/porte, etc.) sur une entité matérielle (processeur, mémoire, etc.).
- **Analyse de l'ordonnancement** : permet l' analyse de l'ordonnancement de l'application si l' application est caractérisée par des processus tous périodiques gérés par un ordonnancement « rate monotonic »[10].
- **Analyse de la sécurité** : cet outil, qui est encore en phase d'expérimentation par le groupe du projet MetaH, offre la possibilité de spécifier un modèle d'erreur sur l'application. Pour chaque composant il est possible de choisir la fréquence d'occurrence d'une erreur et son type en choisissant entre un ensemble prédéfinis de classes d'erreurs : réversible, irréversible, aucun erreur. Il est donc possible d'analyser les traces d'erreurs. L'outil basé sur les chaînes de Markov fournit un résultat probabiliste sur les erreurs propagées à partir d'un ensemble préfixé.
- **Analyse de la robustesse** : Cet outil est déclaré par les concepteurs comme encore en phase d'expérimentation. MetaH propose trois mécanismes pour garantir la fiabilité des applications,
  - ✓ Chaque processus a son espace mémoire virtuel protégé. L'accès des processus aux espaces de mémoire communs est possible seulement sous le contrôle d'un agent de synchronisation.
  - ✓ Chaque processus a un nombre limité de mode de fonctionnement et son comportement est contraint seulement aux modes spécifiés.
  - ✓ Des classes de sécurité sont accordables aux processus de façon qu'un processus appartenant à une classe mineure ne puisse interférer avec l'ordonnancement de celui

[10] "Rate Monotonic" est une technique d' ordonnancement qui considère la situation idéale suivante :
- toutes les taches à ordonner sont périodiques
- une tache n' est pas synchronisée avec d' autres
- une tache ne peut pas être suspendue pendant son exécution
- une tache peut être anticipée instantanément par une tache de priorité plus haute
- une tache est interrompue âpres expiration de son période.

appartenant à une classe plus haute. On peut aussi fixer des limites temporelles sur l'executabilité des étapes d'initialitation, exécution, etc. d'un processus.

- **Génération automatique du code** : Chaque processus est traduit dans un programme qui est compilé dans son propre espace de mémoire virtuel. Les liens entre les différents processus compilés sont encore réalisés en phase de compilation. Le générateur de code produit une application superviseur coordinatrice des processus produits. En effet le générateur de code ne produit pas seulement un ensemble d'exécutables mais également des tables nécessaires à l'ordonnancement, au partage des ressources, aux communications, etc.

Le code généré doit utiliser les primitives de service offertes par l' OS cible, donc le générateur de code doit tenir compte de ces primitives. L' outil «Application Builder » a été construit pour l'ensemble d'outils "*Tartan Ada"* disponibles pour la compilation et le *debugging* sur le processeur i80960MC.

G.6 Un environnement complet pour les Architectures Embarquées

Comme déjà précisé dans le document principal, dans le contexte du Système de Production Automatisé (SAP) le processus de production regroupe un ensemble d' équipements organisés dans le but d'effectuer les opérations de transformation sur les flux de produits. Le système d'automatisation contrôle ce processus de production, en fonction des objectifs et des contraintes de production données.

Le SAP est constitué des dispositifs matériels et logiciels de traitement, d'observation du procédé, des dispositifs de mémorisation et de communication.

Dans l' implantation du SAP ils existent des exigences de

- Modélisation des processus physiques qui constituent le procédé pour la réalisation des algorithmes de contrôle et conduite. Dans ce contexte l' alphabet nécessaire à la modélisation est composé de blocks mathématiques de traitement des signaux (intégrateurs, filtres, etc.) et des générateurs des signaux d' entrée aux blocks. La représentation des processus est exécutée à travers des équations différentielles, ou d' équations aux différences, ou dans l' espace d' états (opérations sur matrices).
- Réalisation des logicielles de contrôle et d' analyse et leur projection sur le système matériel à disposition. Dans ce cas les formalismes qui peuvent être utilisés sont les langages de programmation ou les ADL.

Il existe donc la nécessité d' avoir un pont de communication entre les deux exigences de modélisation et les deux formalismes adoptés. En effet, les processus de contrôle sont directement implementables sur hardware ou ils sont traduisibles dans un ensemble d' applications logicielles (filtres digitaux, traitement des signaux échantillonnés, etc.), et ce dernier choix est normalement le plus économique.

Le projet conduit par *Vestal* permet la possibilité de tracer une correspondance entre les algorithmes de contrôle et leur traduction en applications logicielles**.**

Le langage ControlH est alors utilisé pour le contrôle et la modélisation des processus physiques et *une projection entre éléments de composition du système ControlH et objets MetaH est également possible.*

Cette projection donc permet d' étudier (ordonnancement, utilisation de ressource) l' implantation des applications de contrôle sur le système.

Dans la figure 6.2 nous montrons les domaines d' action et les relations entre ControlH et MetaH.

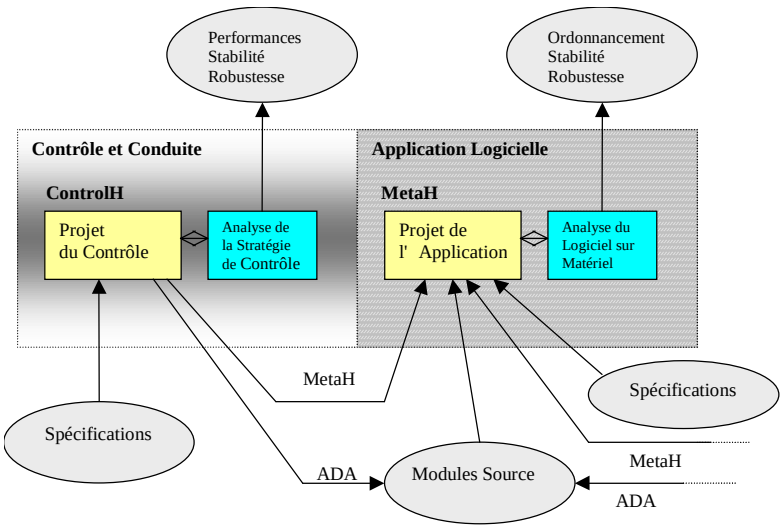


Figure 6.2 : Relation entre ControlH et MetaH

G.7 Conclusions

MetaH est un DSSAL[11] graphique. Il est basé sur le concept de composant, mais ne prévoit pas l'utilisation de connecteurs. En effet les connexions entre composants sont réalisées simplement à l'aide d'assignation de variables.

Le lien entre l' architecture fonctionnelle et matérielle est réalisé en introduisant dans la description de l' architecture des parties implémentations. Ces parties décrivent les périodes des processus, la localisation des fichiers sources relatifs aux éléments utilisés dans la description, etc. Toutes ces informations contribuent à donner une image du système utilisé (processeur, mémoire) et à réaliser une analyse de l' ordonnancement des tâches sur les ressources utilisées. Mais en réalité, actuellement, l'analyse de l'ordonnancement est exécutée seulement sur une famille limitée d'application (processus tous périodiques, ordonnancement « rate monotonic »).

L' utilisateur peut réaliser son architecture en composant les types prédéfinis, mais cette composition est aussi guidée par des contraintes liées au système utilisé.

Ces contraintes sont essentiellement des simplifications sur la description de processus pour faciliter la projection sur le système cible. Le concept d' interface est par exemple réduit à celui d' une variable typée et celui de connexion à une assignation entre variables.

Ce qui est intéressant est l' introduction dans MetaH du concept de Mode, qui représente une sorte d' état global de l' application. La description de modes dégradés à cause d' erreurs ou mauvais fonctionnement ou la reconfiguration en-ligne (run-time) de l' application en fonctions des exigences de contrôle du procédé sont donc facilitées.

MetaH offre la possibilité d' implémenter directement une stratégie de contrôle d' un processus physique decrit en ControlH dans une application logicielle et de la valider en utilisant les outils définis sur son espace de travail.

Les applications de contrôle sont donc évaluées une première fois en fonction du respect des spécifications données sur le processus (ControlH) et une deuxième en fonction d'implementabilité sur le système support (MetaH).

[11] "Modélisation d' architecture d' application, Moyens et Formalismes", p.5

G.8 Bibliographie

[VES93] S. Vestal, "Scheduling and Communicating in MetaH", Real -Time System Symposium, pp.194-200 Rleigh-Durham (NC), December 1993.

ControlH et MetaH: *www.htc.honeywell.com/projects/dssa/dssa_tools.html*
MetaH: *www.htc.honeywell.com/metah*
DSSA (Domain-Specific Software Architectures)
References: *www.htc.honeywell.com/projects/dssa/dssa_refs.html*
Une Liste d'adresses utiles : *www.asset.com/stars/lm-tds/Papers/sysdev/section6_1_2.html*