PROJET RNTL ARCAD*
D4.1 Synthèse des différentes approches permettant la description d'une configuration logicielle

Coordonnateur : Daniel Hagimont
Projet Sardes - Laboratoire LSR-IMAG - INRIA Rhône-Alpes
655, avenue de l'Europe, Montbonnot
38334 St Ismier Cedex France

18 décembre 2002

1 Introduction

La conception, le développement et la maintenance des applications de grande complexité posent des problèmes difficiles à résoudre. Les concepteurs sont confrontés à des contraintes de réutilisation de code existant, d'installation d'applications dans des contextes matériels ou logiciels qui peuvent varier avec le temps, des contraintes d'administration, d'évolution des applications, *etc.* La conception d'applications par le biais des langages de programmation classiques (C, Ada...) permet difficilement de respecter ces contraintes. Ces langages sont intéressants pour programmer des composants logiciels d'une application mais sont inadéquats pour intégrer les composants en les faisant communiquer et coopérer. D'autre part, à l'opposé des applications traditionnelles, le réalisateur d'applications distribuées ne peut avoir une vision globale de l'état et du comportement de son application. De ce fait, des problèmes sont rencontrés pour l'intégration, l'installation et l'administration dans un environnement réparti.

Des études récentes [ISZ98, ADG98, OMT98] ont montré l'importance de la notion d'architecture. Cette dernière permet d'exposer de manière compréhensible et synthétique la complexité d'un système logiciel et de faciliter l'assemblage de composants logiciels. Les axes de travail autour de ce thème sont multiples, l'idée fondamentale est de profiter des avantages d'une architecture clairement explicitée pour ne laisser qu'en second plan la programmation d'applications. C'est pour cela que nous accordons dans ce délivrable D4.1 du projet ARCAD une certaine attention à la notion d'architecture et présentons une synthèse des langages de construction et de configuration d'architecture, les ADL (Architecture Description Language).

Dans la première section de ce délivrable, nous présentons ce qu'est la programmation constructive. Ensuite, après avoir relevé deux principales classes de langages de description d'architecture, nous nous intéresserons successivement dans les section 3 et 4 aux langages permettant la simulation d'architecture et aux langages générant des applications. Enfin, la section 5 proposera une synthèse des différentes approches.

*Le projet ARCAD (Architecture Extensible pour Composants Adaptables) a été labelisé en décembre 2000. Les différents partenaires sont France Télécom R&D (équipe ASR - Thierry.Coupaye@francetelecom.com), INRIA (projet Oasis - Denis.Caromel@inria.fr, projet Sardes - Daniel.Hagimont@inria.fr), Ecole des Mines de Nantes (équipe OCM - Thomas.Ledoux@emn.fr), le laboratoire I3S commun à l'Université de Nice - Sophie Antipolis et CNRS (projet Rainbow - Anne-Marie.Pinna@unice.fr). Le coordonnateur du projet est Michel.Riveill@unice.fr

2 Programmation constructive

La programmation constructive, dénommée aussi programmation par composition de logiciels, est au centre d'études récentes sur le développement d'applications distribuées. Cette méthode se propose de construire les applications, en termes d'assemblage de composants logiciels autonomes et hétérogènes.

2.1 Architecture

De nombreuses recherches ont permis d'introduire une dimension nouvelle à la notion d'architecture de logiciels. La définition généralement admise de ce concept est qu'une « *(...) architecture spécifie les modules du système (appelés composants du système) et l'interaction entre ces composants afin de satisfaire les besoins d'un système* » [LVM]. Cette notion est essentielle puisqu'elle est utile pour la construction et l'administration de systèmes. Plus particulièrement, elle permet l'évolution de l'application (modification des constituants d'un système, de leur placement...), la définition de contraintes sur le système, et facilite la mise en œuvre de l'application en automatisant son processus d'installation. En outre, la structure à un haut niveau d'abstraction exposant des collections de composants interagissants, permet la spécification de propriétés importantes du système (protocoles d'interaction, localisation des données...).

2.2 Programmation par composants

La programmation par composants est une technique de construction d'applications par intégration d'entités logicielles. Les composants sont conçus pour être réutilisés de manière homogène dans différentes applications et contextes d'exécution, sans modification de leur implémentation.

Cette approche permet ainsi d'envisager un processus de développement d'applications à deux niveaux : un premier correspondant à l'architecture globale du système et le second correspondant à l'écriture des composants. La phase de programmation macroscopique définit l'assemblage de composants logiciels formant l'application. Elle est à l'origine de l'architecture du système. Enfin, la phase de programmation microscopique permet la réalisation d'un composant logiciel donné à l'aide de langages de programmation classiques.

Nous pouvons constater qu'il manque, au niveau de la phase macroscopique, un modèle de spécification de l'assemblage de composants. Un grand nombre de chercheurs ont proposé des notations formelles pour représenter et analyser les concepts architecturaux. Souvent appelés « Architecture Description Languages », ces notations fournissent à la fois une base conceptuelle et une syntaxe concrète pour la caractérisation d'architectures logicielles.

2.3 Langages de description d'architecture

L'intégration d'entités logicielles requiert un langage commun, indépendant des langages de programmation. Sans un tel langage, le développeur d'un système distribué doit rendre lui-même les entités logicielles inter-opérables au sein d'un environnement hétérogène. Les ADL permettent de spécifier des architectures d'applications en offrant un moyen pratique et abstrait pour une description compréhensible d'un système complexe. Ils permettent l'expression, dans un mode déclaratif, de la configuration d'une application en termes de composition d'unités logicielles élémentaires et d'éventuelles contraintes de déploiement.

Chacun de ces langages fournit une manière de décrire l'architecture d'un logiciel selon son utilisation finale. Certains langages ont le mérite d'aider à la vérification formelle de propriétés, de permettre la simulation d'une architecture, ou alors d'aider de manière directe à la mise en œuvre et l'exécution du programme. Un ADL se compose communément [MT97] de trois éléments : les composants, les connecteurs et les configurations.

2.3.1 Composant

Le concept de composant est l'élément central du modèle d'assemblage. Le composant étend la notion d'objet, qui possède de nombreuses qualités, parmi lesquelles l'héritage et le polymorphisme. Cependant, les limites du modèle à objets commencent à apparaître, spécialement dans un contexte réparti. En ce qui concerne la lisibilité de l'architecture de l'application, ce modèle n'apporte que peu de solutions hormis la connaissance de l'ensemble des types et classes utilisés. Le schéma d'instanciation des objets et la politique de répartition ne sont pas clairement exposés. D'une manière générale, aucune vision claire de l'architecture n'est proposée. Il est de même impossible de définir des assemblages d'objets. L'héritage des classes est en effet insuffisant, car il reporte les structures de l'objet dans les objets qui en dérivent, brisant par là même le concept d'encapsulation. Une autre limitation est à remarquer. Les objets sont décrits par une interface, au même titre que les fichiers *.h* du langage C, ou les spécifications Ada. Cependant, cette interface reste incomplète, elle ne spécifie pas les méthodes externes nécessaires à l'objet.

La description de l'architecture n'est pas non plus découplée de l'implémentation de l'objet. Elle est exprimée directement dans le code des objets et entrave, de ce fait, la réutilisation et l'intégration de ces modules logiciels dans des systèmes différents.

Le composant se veut plus spécialisé, il n'implante que du code spécifique de l'application réalisée, encore appelé « code métier ». C'est avant tout une entité d'intégration et de structuration de logiciels, dans la mesure où il peut encapsuler du code ou permettre une hiérarchisation de l'application[1], spécifiée indépendamment de l'implémentation du composant.

Les composants logiciels sont, au même titre que les objets, décrits par une interface qui explicite la liste des procédures et données, les types et éventuellement les événements pouvant être visibles par les autres composants du système. Cette interface exhibe, en plus, les informations et les appels qui lui sont nécessaires chez les autres composants du système. Par ailleurs, les composants dérivent d'un type qui représente le mode de mise en œuvre, *e.g.* un processus, une bibliothèque à chargement dynamique, *etc.* Cet aspect de typage impose des modes d'accès particuliers vis à vis du composant et autorise des vérifications, dans la phase d'analyse, de la faisabilité et de la validité de l'architecture. La description d'un composant au niveau du langage permet de donner lieu à de multiples instances, créées dynamiquement lors de l'exécution de l'application.

2.3.2 Interaction

Le connecteur, entité de gestion des interactions entre les composants, permet de définir le comportement des communications. Toutefois, certains langages de description d'architecture ne fournissent pas de connecteur permettant de spécifier le mécanisme d'interaction. La modification du mode de communication entre ces composants n'est alors pas aisée. Un changement du mécanisme de communication entre sites demande effectivement un remaniement profond de la réalisation de ces objets distribués.

Les connecteurs, ou interconnecteurs, sont donc des abstractions essentielles pour la maintenance et la réutilisation des composants. Ils permettent de séparer la phase de programmation en deux niveaux : la programmation classique des différents modules logiciels et le codage des connexions entre ces modules. Ils contiennent les informations concernant les règles d'interconnexion de composants, telles que le protocole, la spécification des échanges de données ainsi que les transformations éventuelles du format d'échange. Chaque connecteur dérive, comme les composants, d'un type qui représente le mécanisme et les propriétés de communication : type de communication, règles de synchronisation, schéma d'ordonnancement, *etc.*

2.3.3 Composition hiérarchique

Une composition hiérarchique, ou *configuration*, est une description de l'ensemble des composants logiciels nécessaires pour le fonctionnement d'une application, ainsi que celle de leurs

[1] Dans la plupart des langages de description d'architecture, ces deux fonctionnalités découlent de la nature propre du composant (*primitif* ou *composite*).

communications. Elle permet de gérer la complexité de l'application en raffinant peu à peu son architecture en un ensemble hiérarchique. Une configuration contient en outre toutes les informations relatives à l'instanciation des composants logiciels et les interconnexions entre ces composants. De ce fait, sa manipulation permet de former facilement une nouvelle architecture.

3 Langages de simulation d'architecture

Cette section présente les langages de description d'architecture qui permettent la formalisation et la vérification d'architectures, à des fins de simulation.

3.1 UniCon

La particularité de UniCon est de supporter un large éventail de styles d'architecture, ou modèles, trouvés dans les applications d'aujourd'hui et de construire à partir d'eux des systèmes complexes. Une description d'architecture dans UniCon s'appuie sur deux concepts principaux, les composants et les connecteurs [SDZ96].

3.1.1 Modèle et notations

Le modèle décrit le système sous forme de deux entités principales, les composants et les connecteurs. Ils ont une structure analogue[2] (*cf.* figure 1) :
- un nom ;
- une spécification (*interface* ou *protocole*) contenant un type et une collection d'entités de composition (*players* ou *rôles*) ;
- une implémentation.

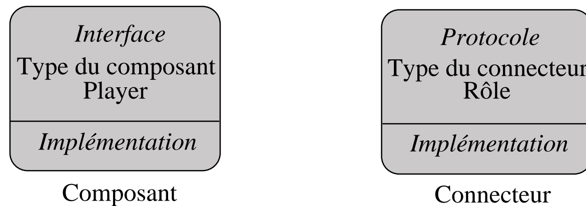


Figure 1: *Modèle d'architecture*

Composants

La première fonction d'un composant est d'intégrer du code logiciel. Son interface spécifie les fonctions que le composant exporte et importe dans son implémentation. Un composant définit aussi son type, ce qui permet de nombreuses vérifications au niveau de l'architecture. Comme dans la plupart des langages de description d'architecture, deux types d'implémentation de composants existent.

Les éléments primitifs font directement référence à du code source ou binaire. Les composants composites, quant à eux, ont une implémentation qui consiste en une liste de composants et connecteurs, des instructions de composition et des connexions avec le composite englobant. En fait, un composant composite encapsule les informations sur la structure globale d'un sous-ensemble d'une application. Il permet de structurer l'application, dans le sens où il présente la hiérarchisation du système en termes d'entités plus petites, sous-composants primitifs ou composites.

[2] A part pour les connecteurs composites qui n'existent pas encore.

Interface L'interface définit les propriétés du composant à plusieurs niveaux. Elle permet d'exhiber d'une part, les fonctions et données accessibles, et d'autre part, les fonctions et données requises par l'implémentation du composant. De plus, elle contraint la manière dont le composant doit être utilisé en imposant des modes d'accès aux fonctions et données qu'il possède. Ces modes d'accès sont spécifiés par des entités visibles, les *players*. Les players correspondent aux services[3] fournis et requis par le composant vis à vis des autres composants de l'application. Ils sont définis par un nom, un type et des attributs optionnels, comme la signature.

Type UniCon présente un système de typage fort pour ses composants, dans la mesure où chaque composant dérive d'un type qui le caractérise. Outre l'expression de la fonctionnalité globale du composant, les types expriment des restrictions sur les propriétés, en termes de formes prescrites pour la communication, le partage de données et autres interactions.

Notations La syntaxe des composants contient en premier lieu une interface. Celle-ci est composée de trois parties différentes : le type du composant, des propriétés optionnelles figurant sous forme de liste et enfin la définition des players. Le type du composant est l'expression d'une classe de fonctionnalités que le concepteur de l'architecture souhaite fournir au travers du composant. Il restreint les nombres, types et spécifications des players qui peuvent être définis. Les propriétés sont des attributs spécifiant des informations supplémentaires sur le composant dans sa globalité. Quant aux players, ce sont les unités sémantiques visibles à travers lesquelles les composants peuvent interagir avec d'autres.

Dans une deuxième partie, le composant contient son implémentation, c'est-à-dire le lien entre l'interface et le code logiciel. L'implémentation d'un composant peut prendre deux formes différentes, primitive ou composite. L'implémentation primitive fait référence à un fichier source, indépendant du langage de description, pouvant être exprimé dans un langage de programmation traditionnel, un script shell, *etc.* Voici la syntaxe de UniCon pour un composant primitif :

```
COMPONENT componentName
   INTERFACE IS
      TYPE componentType
      <Liste de Propriétés>
      <Liste de Players>
   END INTERFACE

   IMPLEMENTATION IS
      <Liste de Propriétés>
      <Implémentation Primitive ou Composite>
   END IMPLEMENTATION
END componentName
```

Pour l'implémentation composite, une description d'un ensemble de composants ainsi que leur composition au moyen de connecteurs sont spécifiées. Prenons maintenant l'exemple d'une application bancaire. C'est un composant composite constitué de deux composants primitifs : un titulaire d'un compte et son compte lui-même. Ces deux composants interagissent au moyen d'un connecteur qui les relie. Plus précisément, les rôles des connecteurs sont reliés aux players de chaque composant.
La figure 2 représente cette application. Ci-dessous est joint le code des composants `Titulaire` et `Banque`.

```
COMPONENT Titulaire
   INTERFACE IS
```

[3] Les services sont des points d'entrée et de sortie des composants, seuls éléments visibles de l'extérieur du composant.

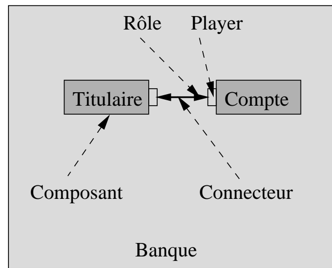


Figure 2: *Représentation de l'application bancaire dans UniCon*

```
      TYPE Process
      PLAYER consultation IS RPCDef
         SIGNATURE (“int”; “int”)
         END consultation
   END INTERFACE

   IMPLEMENTATION IS
      VARIANT titulaire IN “titulaire.c”
      IMPLTYPE (Source)
      END titulaire
   END IMPLEMENTATION
END Titulaire
```

Ce composant contient une interface et une implémentation primitive. Il est de type *Process* et offre un player possible pour l'accès à des fonctions par le mécanisme RPC. Il peut demander à un autre composant (notamment un compte) la consultation de son solde. Nous pouvons remarquer que la signature de players est écrite dans la syntaxe du langage utilisé pour la programmation du code source. L'attribut *IMPLTYPE* spécifie le langage de programmation du code source (C).

```
COMPONENT Banque
   INTERFACE IS
      TYPE General
   END INTERFACE

   IMPLEMENTATION IS
      // COMPUSES : Instanciation des composants
      USES titulaire INTERFACE Titulaire
         PROCESSOR (“caroline.inrialpes.fr”)
         ENTRYPOINT (titulaire)
         END titulaire
      USES compte INTERFACE Compte
         PROCESSOR (“dyade.inrialpes.fr”)
         ENTRYPOINT (compte)
         END compte
      // CONNECT : Connexion utilisée
      ESTABLISH RPC WITH
         titulaire.consultation AS appelant
         compte.solde AS appelé
         END RPC
   END IMPLEMENTATION
```

```
END Banque
```

Les éléments composites fournissent le mécanisme de construction d'un système à partir de composants primitifs, comme ci-dessus, ou de sous-systèmes qu'il inclut dans son architecture. Une composition est constituée des :
- instanciations des composants dans la partie *COMPUSES;*
- instanciations des connecteurs dans la partie *CONNUSES;*
- associations de players de certains composants avec les players du composant composite établies dans la partie *BIND;*
- définitions des interactions entre les composants dans la partie *CONNECT.*

L'application bancaire n'ayant pas besoin de typage précis, nous avons utilisé *General* qui sert à la structuration du logiciel. L'implémentation de l'application comprend l'instanciation des sous-composants primitifs `titulaire` et `compte`, de type `Titulaire` et `Compte` respectivement. Au cours de ces instanciations, des propriétés propres au type des composants peuvent être positionnées, telles le site d'exécution par l'attribut *PROCESSOR* (par défaut, c'est la site local), le point d'entrée pour le lancement des processus (par défaut, le *main* du module), le niveau de priorité, *etc.* Juste après la définition des composants utilisés, l'implémentation contient la déclaration des interactions entre ces composants. La clause *ESTABLISH* permet de décrire une interaction entre les deux composants par le biais d'un connecteur RPC.

Connecteurs

Un connecteur permet d'établir des connexions entre les composants. Il spécifie les protocoles d'interaction et tous les autres mécanismes nécessaires pour les communications : spécifications des échanges de données et choix du mode de communication. Il est défini par un *protocole* qui spécifie le type d'interaction que le composant fournit. Par ailleurs, le connecteur possède, comme le composant, une implémentation. Actuellement, seul un type d'implémentation de connecteur est possible, c'est le connecteur primitif.

Protocole Le protocole définit les interactions permises parmi la collection de composants de l'application et fournit des garanties sur ces interactions. En fonction du type du connecteur, le protocole définit des *rôles.* Ces entités représentent les points de branchement auxquels les players d'un composant peuvent se connecter. Un rôle est décrit au moyen d'un nom et d'un type. Il comporte aussi une liste d'attributs optionnels comme une signature, des spécifications fonctionnelles et des contraintes d'utilisation. Le protocole doit, par conséquent, contenir le type du connecteur, les assertions qui imposent des restrictions à ce connecteur (*p. ex.* des règles d'ordonnancement), et les *rôles* qui servent de points d'entrée ou de sortie du protocole.

Type Les rôles sont les unités sémantiques visibles à travers lesquelles les connecteurs font les interconnexions entre les composants. Les rôles identifient le type d'interaction qu'un connecteur peut établir, le type des composants avec lesquels ils peuvent être connectés, le type de players qu'ils supportent. Plus précisément, certains rôles requièrent des players particuliers.

Notations Dans l'exemple précédent, le connecteur n'était pas encore défini. Sa description est spécifiée ici, par le code suivant :

```
CONNECTOR RPC
   PROTOCOL IS
      TYPE RemoteProcCall
      ROLE appelé is definer
      ROLE appelant is caller
   END PROTOCOL

   IMPLEMENTATION IS
```

```
      BUILTIN
    END IMPLEMENTATION
  END RPC
```

Nous pouvons remarquer que cette notation est très similaire à celle des composants. Notamment au niveau de la structure, nous retrouvons la séparation entre le protocole (interface pour le composant) et l'implémentation. Le protocole contient dans sa définition le type dont dérive le composant, une liste optionnelle de propriétés et une liste de rôles. L'implémentation du connecteur ne figure qu'en deuxième partie. Celle-ci ne correspond qu'à des éléments primitifs, UniCon ne permettant pas encore l'utilisation de connecteurs composites.

La description de ce connecteur rend le renommage des rôles de *RemoteProcCall* possible. Il est à noter aussi qu'aucun mécanisme de UniCon ne permet de définir des implémentations personnalisées de connecteurs. Seule l'implémentation *BUILTIN* est disponible.

3.1.2 Atouts

UniCon est un langage qui présente un nombre non négligeable d'avantages. Parmi eux, nous pouvons retenir :

- Le système de typage fort est intéressant pour plusieurs raisons. Le compilateur de UniCon vérifie statiquement si les interconnexions sont valides et conformes au typage des entrées et sorties des composants et des connecteurs. De plus, ce typage impose un certain mode d'accès aux entités du système en restreignant leur accès. Enfin, chaque type de players et rôles correspond à un mécanisme d'accès aux composants et connecteurs, permettant aux outils de génération de produire le code associé ;
- Le mécanisme de communication n'est pas inclus dans le code des composants mais tient une place particulière au niveau du langage de description d'architecture. Par conséquent, une même définition d'un connecteur peut gérer les échanges entre des composants divers et variés, à partir du moment qu'il sont autorisés à interagir via ce type de communication ;
- Les sources, objets et exécutables peuvent être encapsulés dans les composants. Cette intégration à différents niveaux semblent offrir des mécanismes suffisants pour la conception d'applications. De nombreuses primitives de construction très utiles sont disponibles : les processus, filtres, pipes...

3.1.3 Limites

Malgré les nombreux avantages présentés par UniCon, un certain nombre d'inconvénients sont à relever :

- Bien entendu, la limitation majeure de UniCon correspond à la description statique de l'architecture qu'il fournit. Cela implique que tous les composants doivent être instanciés initialement et qu'aucun ajout ni aucune suppression ne sont autorisés par la suite ;
- Trop de rigidité, induite par le typage fort, est aussi à déplorer. Bien que les concepteurs d'applications réparties aient de bonnes abstractions informelles pour les interactions, ils sont restreints à l'utilisation d'abstractions par défaut, ou implicites, et ne peuvent effectuer un choix délibéré ;
- UniCon supporte seulement un ensemble prédéfini de types pour les composants et les connecteurs. L'ensemble n'est pas complet et demande de faire une taxonomie approfondie sur le sujet. Il est toutefois prévu de rendre cet ensemble extensible ;
- Bien que l'implémentation devrait supporter un bon nombre de langages, le C est actuellement le seul accepté. Les interfaces sont entièrement dépendantes du langage utilisé, *p. ex.* la signature est exprimée dans le langage du code encapsulé. Ainsi, un composant est difficilement réutilisable dans un contexte différent de celui de sa création ;
- Les seuls composants primitifs disponibles dans la version de [SDZ96] sont les unités de compilation *BUILTIN*, il n'est pas encore possible d'en définir des personnalisés. De même, UniCon supporte les composants composites mais pas encore les connecteurs composites.

3.1.4 Bilan

UniCon est un langage de description d'architecture qui semble intéressant sur de nombreux aspects, particulièrement pour la séparation entre le composant et le connecteur, mais son système de typage reste trop contraignant et manque de souplesse. De plus, une architecture UniCon ne présente pas d'abstraction au niveau du langage pour décrire la dynamique de l'application. Elle ne permet ni de définir la présence variable d'un composant, le marquant explicitement comme optionnel, ni de spécifier des composants alternatifs.

3.2 Wright

Wright est un langage de description orienté vers la vérification des protocoles entre les composants, plutôt que sur la correction fonctionnelle de l'architecture globale [ADG98]. Ce langage n'est pas dédié à la production d'une image exécutable de l'application, il porte effectivement son accent sur la vérification formelle du système.

3.2.1 Modèle et notations

Wright modélise des structures architecturales en utilisant des composants, connecteurs et configurations, chaque abstraction étant définie à l'aide d'un calcul proche de CSP [Hoa85]. CSP décrit le comportement d'une architecture logicielle à travers un modèle algébrique de processus. L'utilisation de ce calcul permet de vérifier la faisabilité d'une interconnexion entre les composants et les connecteurs. Des notions de CSP sont présentées en Annexe A.

Composants

Les composants intègrent du code logiciel. Par exemple, notre titulaire d'un compte en banque émet des requêtes pour consulter son compte et reçoit en retour son solde. Dans Wright, la description d'un composant possède deux parties, l'interface et la partie `COMPUTATION`. Une interface est constituée d'un ensemble de ports. Chaque port représente une interaction dans laquelle le composant participe. Dans notre exemple, le client possède un seul port, celui pour consulter son solde. La spécification d'un port est une description partielle du composant [All97], une projection du comportement du composant sur ce point particulier de l'interface. La spécification complète du composant se trouve dans `COMPUTATION`, lieu d'analyses des propriétés. La partie `COMPUTATION` d'une description décrit le véritable comportement du composant. Elle reprend les interactions décrites au sein des ports et montre les relations entre les ports ainsi que leur lien avec les actions internes du composant. Ci-dessous se trouve un exemple de description du composant `Titulaire` inspiré de [AG97] :

```
COMPONENT Titulaire
   PORT consultation = requete? x → reponse! y →
 consultation ⊓ √
   COMPUTATION = calculInterne→ consultation.requete? x →
 consultation.reponse! y → COMPUTATION ⊓ √
```

Le port `consultation` figure comme un point logique[4] d'interaction entre le composant `Titulaire` et son environnement. Il exprime le comportement que le composant fournira à travers ce point particulier de l'interface à son environnement. Ici, il souhaite pouvoir renouveler les actions « émission d'une requête x » et « réception du résultat y » ou terminer son processus quand il le désire (choix non déterministe ⊓).

La partie `COMPUTATION` ressemble beaucoup au codage du port `consultation` puisqu'il est le seul port présenté par le composant. Le comportement observé à travers ce port est mis en relation avec le comportement interne du composant. Les actions qui ne sont pas internes sont

[4] Les ports sont des entités logiques : il n'est pas nécessaire qu'un port réalise une fonction dans le système où il se trouve.

préfixées par le port où leur processus a lieu. Par exemple, `requete` et `reponse` sont préfixées par `consultation`.

Connecteurs

Un connecteur détermine les interactions parmi une collection de composants. Par exemple, un pipe représente un flot de données séquentiel entre deux filtres.

Une description Wright d'un connecteur est divisée en un ensemble de rôles et une spécification `GLUE`. Les rôles définissent le comportement local souhaité ou attendu pour les ports qui vont s'attacher aux extrémités du connecteur.

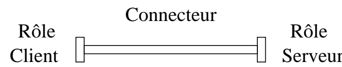


Figure 3: *Exemple de connecteur Wright*

Ainsi, le connecteur de la figure 3 admettra sur son rôle gauche la connexion d'un composant ayant le comportement d'un client et sur son rôle droit celui d'un serveur. Dans notre mise en scène, le client pourrait être le titulaire d'un compte en banque et le serveur pourrait être le compte lui-même. Les connecteurs sont en attente de propriétés particulières pour les participants potentiels à l'interaction. Cette attente est spécifiée au niveau des rôles. Ceux-ci définissent le protocole qui doit être spécifié par ceux qui s'y attacheront. En général, un port n'a pas besoin d'avoir le même comportement que le rôle qu'il remplit, il peut en effet utiliser seulement un sous-ensemble des capacités du connecteur. La `GLUE` d'un connecteur, quant à elle, décrit comment les participants interagissent au cours de cette communication. Comme pour `COMPUTATION`, elle relie les différentes actions comprises dans son interface.

```
CONNECTOR Connecteur
   ROLE client = requete ! x → reponse ? y → client ⊓ √
   ROLE serveur = demande ? x → resultat ! y → serveur □ √
   GLUE = (client.requete ? x → serveur.demande ! x →
serveur.resultat ? y → client.reponse ! y → GLUE) □ √
```

Le rôle du `client` décrit le comportement de l'utilisateur du service dans la communication. C'est un processus qui peut répétitivement appeler le service et recevoir le résultat, ou terminer. L'utilisation de l'opérateur décisionnel ⊓ représente un choix laissé au rôle lui-même.

Similairement au `client`, le `serveur` est défini comme un processus qui, soit accepte répétitivement une invocation, retournant par la suite une réponse, soit peut terminer avec succès. À cause de l'utilisation de l'opérateur déterministe □ , le choix est effectué par l'environnement de ce rôle (qui se résume à la glu et aux autres rôles). En comparant les deux opérateurs de choix, on peut s'apercevoir qu'une distinction formelle est faite entre les situations où un rôle donné est obligé de fournir des services (le cas du `serveur`) et la situation où il peut profiter s'il le désire de services (`client`).

Le processus `GLUE` coordonne le comportement entre les deux rôles en indiquant comment tous les événements interagissent ensemble. La glu décrit comment les activités des rôles `client` et `serveur` sont coordonnées et présente leur séquencement. Le lecteur pourra noter que les événements qui sont en entrée (*resp.* sortie) au niveau des rôles (`reponse ? y`) se trouvent en sortie (*resp.* entrée) dans la glu (`client.reponse ! y`). Ceci s'explique aisément en se ramenant aux spécifications des deux parties d'un connecteur. Un rôle décrit le comportement d'un composant qui s'y attache alors que la glu s'intéresse au comportement du connecteur.

Configuration

Pour décrire l'architecture complète d'un système, les composants et connecteurs d'une description Wright sont combinés dans une configuration. Une configuration est une collection d'instances de

composants reliés au moyen de connecteurs. Dans l'exemple suivant, des styles sont utilisés. Un style architectural permet la réutilisation d'un ensemble de types et de règles pour la description d'une architecture. Plus précisément, ils déterminent un vocabulaire à base de types de composants et connecteurs et spécifient les contraintes topologiques. Ces contraintes définissent un ensemble de prédicats que chaque configuration, conforme à ce style, doit satisfaire. Ici, les définitions précédentes des composants et du connecteur sont réutilisées. Les contraintes topologiques spécifient qu'un attachement entre toutes les instances des types `Compte` et `Titulaire` doit exister.

```
STYLE Titulaire-Compte
   COMPONENT Titulaire    // Définition du Titulaire
   COMPONENT Compte       // Définition du Compte
   CONNECTOR Connecteur   // Définition de Connecteur
   CONSTRAINTS
      ∃!c ∊ Component, ∀ t ∊ Component : TypeCompte(c) ∧
    TypeTitulaire(t) ⇒ connected(c, t)
END STYLE
```

La configuration du système est décrite en trois étapes. La première est la définition d'une ensemble de types de composants et connecteurs. Cette étape fait appel au style `Titulaire-Compte` prédéfini. Ensuite, un ensemble d'instances de ces types est déclaré, représentant les entités qui apparaîtront réellement dans la configuration. Il figure une seule instance `titulaire` dans cette illustration, ainsi qu'une seule instance `compte` et un connecteur. Dans la phase finale, les instances de composants et connecteurs sont combinées pour décrire quels sont les ports de composants attachés aux rôles des connecteurs. La consultation du `compte` par le `titulaire` est reliée au rôle `client` du connecteur. De même, le port `solde` est attaché au rôle `serveur`. Ainsi, le prédicat exprimé dans les contraintes du style `Titulaire-Compte` est vérifié. Le code suivant montre comment notre simple système de banque peut être défini dans la description Wright.

```
CONFIGURATION Banque
 STYLE Titulaire-Compte
 INSTANCES
   titulaire : Titulaire
   compte : Compte
   connecteur : Connecteur
 ATTACHEMENTS
   titulaire.consultation as connecteur.client;
   compte.solde as connecteur.server;
END Banque
```

3.2.2 Évolution dynamique

En guise d'illustration de la dynamicité de Wright, reprenons le style décrit précédemment. Pour cela, nous allons considérer maintenant une nouvelle configuration dans laquelle deux serveurs interviennent (*cf.* figure 4), représentant les comptes en banque : un serveur primaire qui peut tomber en panne à tout moment et un secondaire, répliqué, qui sert de serveur de secours. Initialement, le connecteur relie un serveur primaire et un titulaire de compte. Dès que le serveur primaire tombe en panne, la connexion est défaite. Elle est ensuite rétablie vers le serveur secondaire jusqu'à la restauration du primaire.

D'abord, des événements spéciaux de contrôle sont introduits dans l'alphabet des composants et connecteurs. Ces événements de contrôle sont utilisés dans une vue séparée de l'architecture,

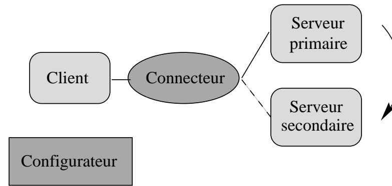


Figure 4: *Deux configurations possibles pour la topologie dynamique*

le programme de configuration. Ce dernier fait la correspondance entre les événements émis et les reconfigurations à effectuer sur l'architecture. L'idée de base est d'exprimer, au niveau des abstractions (composants et connecteurs), le contexte dans lequel elles admettent des reconfigurations ; la gestion de ces reconfigurations, elle, est effectuée au sein du contrôleur.

Dans la description suivante, le compte primaire indique dans quel état il est susceptible de tomber en panne avec `control.down` et se rétablir avec `control.up`. Effectivement, il peut s'interrompre à tout moment sauf au cours d'une transaction (entre `demande` et `retour`) et ne peut se rétablir qu'après un arrêt. Le connecteur peut, lui, être reconfiguré (`changeOk`) au cours d'une transaction (entre `request` et `reply`).

```
COMPONENT ComptePrimaire
   PORT p = √ □ (demande? x → resultat! y → p ⊓ control.down
     → (√ □ control.up → p))
   COMPUTATION = √ □ (p.demande? x → calculInterne →
    p.resultat! y → COMPUTATION ⊓ control.down →
    (√ □ control.up → COMPUTATION))

CONNECTOR Connecteur
   ROLE client = requete! x → reponse? y → client ⊓ √
   ROLE serveur = (demande? x → resultat! y
     → serveur ⊓ control.changeOk → serveur) □ √
   GLUE = (client.requete? x → serveur.demande! x
     → GLUE
     □ serveur.resultat? y → client.reponse! y → GLUE)
     □ √
     □ control.changeOk → GLUE
```

Dans ce programme de reconfiguration, le type du client est identique à celui utilisé précédemment. Le connecteur est modifié, ainsi que les comptes[5]. Ce nouveau style est donc utilisé dans le configurateur. Des modifications de la topologie de l'architecture seront effectuées en appliquant des actions de base comme `new`, `del`, `attach` et `detach` sur les instances de types architecturaux contenus dans le style.

Au début de la description du configurateur, le style utilisé est déclaré. Ensuite, la séquence initiale d'actions (`new` et `attach`) modélise la configuration initiale du système. `Arrêt` décrit deux situations possibles : le système peut marcher correctement et terminer avec succès ou une erreur peut apparaître avant la terminaison de l'application. Si le compte primaire dysfonctionne, le secondaire s'active et le connecteur passe en mode de reconfiguration. Le compte primaire est alors détaché du connecteur et il est remplacé par le serveur de secours. La nouvelle configuration est ainsi établie jusqu'au rétablissement du primaire. Enfin, `Rétablissement` se comporte de manière similaire, les rôles entre des deux comptes étant inversés.

[5] Le compte secondaire n'a pas été présenté ici. Il est assez similaire au compte primaire et ne présente, par ce fait, pas beaucoup d'intérêt.

Donc, le configurateur décrit les configurations possibles (une initiale et deux configurations qui s'alternent). L'annotation « style » spécifie l'ensemble de types de composants et connecteurs que le configurateur utilise ainsi que les contraintes qu'il doit satisfaire.

```
CONFIGURATOR ClientServeur
   STYLE Titulaire-Compte
   // Instances et attachements
   new.C : Client →
   new.P : ComptePrimaire →
   new.S : CompteSecondaire →
   new.Conn : Connecteur →
   attach.C.p to Conn.client
   attach.P.p to Conn.serveur → Arrêt
   WHERE
   Arrêt =(P.control.down → S.control.up →
     Conn.control.changeOk → Style Titulaire-Compte
     detach.P.p from Conn.serveur →
     attach.S.p to Conn.serveur → Rétablissement) → √
   Rétablissement = (P.control.up → S.control.down →
     L.control.changeOk → Style Titulaire-Compte
     detach.S.p from Conn.serveur →
     attach.P.p to Conn.server → Arrêt) → √
```

Ainsi, Wright centralise la gestion et le contrôle de la dynamicité dans le configurateur. Les abstractions architecturales, composants et connecteurs, exposent au sein de leur description les contraintes liées à leur reconfiguration.

3.2.3 Atouts

- Wright met l'accent sur la spécification du comportement dynamique des applications : création, suppression de composants, interconnexions qui se modifient dynamiquement en fonction de propriétés, *etc.* Des éléments de reconfiguration, constituant un vocabulaire de contrôle, ont pour cela été ajoutés au vocabulaire de base ;
- Wright présente les connecteurs comme des entités sémantiques particulières. L'essence de cette approche est de fournir une notation qui donne un statut explicite à la connexion, augmentant par ce fait leur indépendance vis à vis des composants. Ceci permet à l'architecte de comprendre la fonctionnalité d'un connecteur indépendemment du contexte dans lequel il sera utilisé ;
- Wright offre un formalisme qui permet de vérifier si les ports et les rôles peuvent être compatibles d'un point de vue échange de paramètres comme du point de vue modèle d'exécution de la communication. Il supporte aussi l'analyse statique d'absence d'interblocage.

3.2.4 Limites

- Il faut noter que Wright n'est pas dédié à la production d'une image exécutable de l'application. Il autorise les vérifications mais ne construit pas d'applications. Notamment pour la répartition, Wright s'occupe essentiellement de définir le comportement des composants et des connecteurs en termes de processus communiquants et oublie de les répartir sur des sites d'exécution. Si les composants sont sur des sites différents, Wright considère qu'il s'agit de composants dans des processus différents. Par ailleurs, contrôler la reconfiguration au sein d'une entité monolithique ne paraît pas être une solution adéquate si l'on souhaite déployer une configuration distribuée à grande échelle ;

- En outre, Wright n'apporte pas de solution pour la hiérarchisation et la structuration de l'application. Les composants composites qui existent dans la plupart des langages n'ont pas d'équivalent ici. Malheureusement, toutes les architectures devant passer à l'échelle ont besoin d'encapsulation de sous-systèmes dans des composants. Cette limitation réduit donc grandement la classe d'applications utilisables ;
- Wright s'appuie sur un modèle algébrique de processus, compréhensible dans des architectures simples mais qui se complique très rapidement pour décrire des applications plus complexes.

3.2.5 Bilan

Wright est un langage architectural qui se concentre sur le comportement du système. Celui-ci est caractérisé en termes d'événements significatifs qui peuvent prendre place dans les calculs des composants et des interactions entre ces composants (les connecteurs). Ce langage a été récemment étendu[ADG98] et propose maintenant une bonne solution pour l'expression de la dynamique des applications.

3.3 Rapide

Rapide est un langage de modélisation d'architectures, dont la sémantique est basée sur des échanges de messages caractérisés par des événements partiellement ordonnés. La simulation de modèles dans Rapide permet de valider une architecture à l'aide de cette sémantique et de vérifier certaines propriétés comme l'absence d'interblocage lors de l'exécution de l'application ou la conservation de l'ordre causal de délivrance d'événements.

3.3.1 Concepts

La principale caractéristique de Rapide par rapport aux autres langages de description d'architecture étudiés est qu'il utilise des règles d'interconnexion, déclenchables selon le comportement des composants. Dans cette partie, nous abordons le modèle événementiel et les règles de Rapide. Sont également présentés le modèle de composants et d'architecture, permettant de décrire la structure des applications.

Composants

Chaque composant, ou module, possède un ensemble d'interfaces qui décrivent les styles d'événements qui peuvent être échangés avec les autres composants de l'architecture.

Une interface est constituée d'un ensemble de services. Ces services sont des fonctions qui peuvent être appelées ou que le composant requiert. Ils peuvent être de type *provides*, *requires* ou *action*. Les provides peuvent être appelés de manière synchrone par les composants. Les requires sont les services que le composant demande de manière synchrone à d'autres composants. Un module peut donc communiquer de manière synchrone avec un autre module si leurs services requires et provides sont connectés. Le dernier type de service possible est l'action. Une action correspond à un appel asynchrone entre composants. Deux types d'actions existent : les *in* et les *out* qui sont respectivement des événements acceptés et envoyés par un composant.

Outre la définition des services, l'interface contient une section de description du comportement (clause *behavior*). Le comportement est la description du fonctionnement observable du composant, par exemple l'ordonnancement des événements ou des appels aux services. C'est grâce à cette description que Rapide est en mesure de simuler le fonctionnement de l'application.

Architecture

Une application est représentée par son architecture. Une architecture consiste en des déclarations d'instances de composants, des connexions entre ces composants et des contraintes sur le comportement de l'architecture. Voici la structure d'une architecture :

```
architecture name is
```

```
//Déclarations
connections
//Connexions
[constraints]
//Contraintes optionnelles
end name
```

Une architecture contient la déclaration des instances de composants ou modules. Toutes les instances sont représentées par des variables. Rapide introduit deux types de variables un peu particulières [LV95] : les *placeholder* et les *iterator*. Le nom des variables placeholder débute toujours par « ? », ce qui permet de les distinguer des variables classiques. Ce sont des variables typées, leur déclaration est similaire à des variables ordinaires. Cependant, elles ont une sémantique différente. Elles servent à la sélection dynamique, dans le sens où elles désignent un objet qui est susceptible d'être présent. Par exemple, `?c : Client` fait référence à une instance de `Client`. L'itérateur « ! » désigne, quant à lui, une conjonction d'instances d'un certain type. Ainsi, `!s : Serveur` désigne toutes les instances de `Serveur` présentes dans l'application. Ces déclarations de variables sont relativement dynamiques car on définit un objet devant être présent dans l'architecture ou un ensemble (borné ou non) d'objets de tel ou tel type. Il n'est pas obligatoirement nécessaire de définir exactement les instances de composants présents, car cela peut se faire dynamiquement au fur et à mesure de l'exécution.

Le reste de l'architecture contient la spécification de règles de connexions entre les objets et des contraintes optionnelles. Nous verrons par la suite les caractéristiques des interconnexions. Par contre, nous n'étudierons pas les contraintes, les détails de Rapide étant trop nombreux pour être tous abordés ici.

Événements

Le concept de base de Rapide est l'événement qui est une information transmise entre composants. L'événement permet de construire des expressions appelées *event patterns*. Ces expressions permettent de caractériser les événements circulant entre les composants. Par exemple, si A est un événement, A>B signifie que B sera envoyé après A. La construction de ces expressions se fait avec l'utilisation d'opérateurs permettant d'exprimer des dépendances entre événements. Parmi les opérateurs présents, on peut trouver l'opérateur de dépendance causale, d'indépendance, *etc.* L'ensemble de ces opérateurs est répertorié dans le tableau 1. Un événement peut correspondre à

Opérateur	**Sémantique**
A>B	B est envoyé après A
A->B	B dépend causalement de A
A||B	A et B ne sont pas causalement dépendants
A~B	A et B sont différents
A and B	A et B sont vérifiés simultanément

Table 1: *Opérateurs de dépendance entre événements*

une demande de service, à une valeur particulière d'un attribut, à une apparition d'un nouveau composant... Il ne s'agit pas seulement d'un envoi de message entre deux entités logicielles mais à une information quelconque portant sur le comportement des composants de l'application.

Règles

Une règle est composée d'une partie droite et d'une partie gauche. La partie gauche contient l'expression d'événements qui doit être vérifiée avant que les événements contenus dans la partie droite ne soient déclenchés, *i.e.* envoyés dans le système vers leurs destinataires. Entre ces deux parties, deux sortes de connexion sont possibles, les simples et complexes. Une connexion simple définit une interaction entre deux services (`to`). Une connexion complexe définit de manière plus générale les interconnexions de composants. Il existe deux principaux types de connexions complexes : les connexions *agent* (||>) et *pipe* (=>).

to connecte deux expressions d'événements simples, *i.e.* ne définissant qu'un événement possible vers un composant. Si la partie gauche est vérifiée, alors l'expression de la partie droite permet le déclenchement de l'événement vers l'unique composant désigné par cette expression.

||> connecte deux expressions quelconques. Dès que la partie gauche est vérifiée, tous les événements contenus dans la partie droite sont déclenchés. Ils sont envoyés vers l'ensemble des destinataires désignés dans cette expression. L'ordre d'évaluation de cette règle de connexion est quelconque, c'est-à-dire qu'un déclenchement de cette règle de connexion est indépendant des autres déclenchements antérieurs ou postérieurs. L'ordre d'observation de ces déclenchements n'est pas significatif.

=> possède le même rôle que l'opérateur précédent mais ici l'ordre d'évaluation des règles est contrôlé. Un déclenchement de cette règle est causalement dépendant des déclenchements antérieurs de cette règle. Cet opérateur de connexion est appelé opérateur *pipe-line*.

Essayons maintenant d'illustrer ces règles au sein de notre application pilote.

```
with Client, Serveur;
...
//Déclaration des instances des composants de l'application
  susceptibles d'exister
?s : Client; //Référence à une instance de Client
!r : Serveur; //Référence à toutes les instances de Serveur
?d : Data; //Référence à un bloc de paramètres d'un
             certain type Data
//Règle d'interconnexion
?s.Send(?d) => !r.Receive(?d);
//Si un client transmet un événement de type Send avec ce type
  de paramètres, alors l'événement est transmis à tous les
  serveurs de l'application avec ces paramètres.
```

L'interconnexion précédente spécifie l'existence d'un client qui émet `Send` avec n'importe quelles données de type `Data`, alors la partie droite de la règle est exécutée, *i.e.* tous les serveurs du système reçoivent la même donnée `?d`. L'opérateur ? indique que l'on choisit un composant du bon type parmi ceux présents dans le système alors que ! indique que tous les composants de ce type présents dans le système sont choisis. Les parties gauches et droites peuvent être interconnectées par trois sortes d'opérateurs (**to**, ||> et =>). Dans notre exemple, il ne peut y avoir de connexion simple car la partie droite définit un ou plusieurs composants destinataires, `!r` désignant tous les serveurs existants dans le système.

3.3.2 Évolution dynamique

Dans cette partie, nous allons développer un exemple un peu plus complet qui présente une architecture dynamique. Trois types de composants interviennent : des clients, des fournisseurs et un serveur de noms. Les fournisseurs peuvent s'enregistrer auprès du serveur de noms, indiquant les services qu'ils remplissent. Le serveur de noms enregistre leur référence ainsi que leurs fonctions. Un client peut à tout moment demander au serveur un fournisseur d'un service qui l'intéresse. Il obtient en résultat un nom de fournisseur qu'il ne peut déréférencer. Pour cela, il doit envoyer une requête à l'architecture, qui est le seul module pouvant effectuer les déréférençages et générer donc la bonne requête au fournisseur.

```
with Client, Provider, nameServer;
architecture Network is
//Déclarations
  ns : nameServer;
  clients : array[1..NUM_CLIENTS] of Client;
```

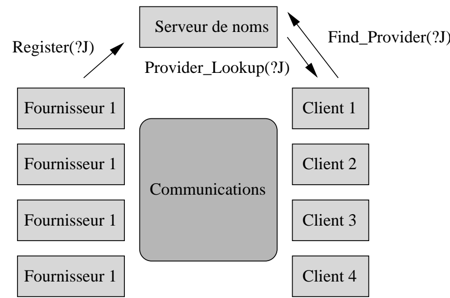


Figure 5: *Représentation d'un serveur de noms en Rapide*

```
   providers : array[1..NUM_PROVIDERS] of Providers;
  ?P : Provider;
  ?J : Job;
  ?C : Client;
  ?N : &Provider;
  ?param : Parameters;
//Connexions
   connect
  ?P.Register(?J) To ns.Register_Provider(?J, &?P);
  ?C.Find_Provider(?J) To ns.Provider_Lookup(?J);
  ?C.Request_Job(?J,?param,?N) To *?N.Do_Job(?J,?param);
end Network;
```

La première règle connecte toutes les demandes d'enregistrement (`Register`) des fournisseurs au service `Register_Provider` du serveur de noms. Cette interaction met en jeu plusieurs fournisseurs et un seul serveur de noms : l'utilisation de l'opérateur `to` est ainsi justifiée.

La seconde règle déclenche un événement `Provider_Lookup` vers le serveur de noms quand l'événement `Find_Provider` du client est détecté. Le client attend ici un résultat. Cette règle peut être considérée comme un alias de la fonction du client vers celle du serveur de noms.

Enfin, la dernière règle sert au déréférençage du nom des fournisseurs. Quand un client effectue une demande de service à l'un des fournisseurs, le serveur de nom déréférence le nom du fournisseur et renouvelle la demande de service correctement.

3.3.3 Atouts

- Rapide prend totalement en compte différents modèles d'exécution en proposant au niveau de l'interface des composants différents types d'appels de services (principalement synchrones et asynchrones) et en utilisant différents opérateurs d'interconnexion ;
- La simulation d'un modèle génère un ensemble d'événements apparaissant à l'exécution avec des relations causales et temporelles. Cela fournit de nombreuses opportunités pour l'analyse de modèles de systèmes complexes, particulièrement concernant les aspects de distribution et de comportement concurrent. Cette caractéristique est particulièrement appréciable puisqu'elle évite l'installation et le déploiement d'une application de grande taille pour effectuer les tests de validité de l'architecture ;
- Tout en se basant sur des abstractions identiques aux ADL (composants possédant une interface, interconnexions exprimées en dehors des composants...), Rapide permet d'exprimer toute la dynamique de l'application en terme d'ensembles d'instances de composants, de règles d'interconnexion qui évoluent en fonction du comportement des composants... Ceci est permis par l'utilisation de règles d'interconnexion qui sont déclenchées par le comportement des composants logiciels, que ce soit lors d'un changement d'état ou une demande de communication avec d'autres composants.

3.3.4 Limites

- Dans les articles [LKA+95, LVM, LV95, Luc96], aucun opérateur de création, suppression, migration de composants ou modification d'interconnexion n'est disponible. Cela est un frein à la dynamicité de l'application. En effet, il est regrettable de ne pas pouvoir reconfigurer l'application ;
- Comme Wright, Rapide ne propose aucun élément de structuration de l'application.

3.3.5 Bilan

Rapide est un langage de description d'architecture intéressant car il permet de modéliser une application en ne raisonnant que sur des ensembles de composants logiciels manipulés par une interface et sur le concept d'interconnexion entre composants. Certaines propriétés des applications, ainsi que certains de ses opérateurs nous semblent très intéressants et le système de règles paraît extrêmement prometteur.

4 Langages de configuration

Cette section présente différents langages de description d'architecture, qui autorisent la génération d'applications ou de squelettes d'applications. Ces langages, appelés langages de configuration, offrent ainsi un lien entre la description d'une architecture et son implantation.

4.1 Darwin

Darwin [ADG98] est un langage de description d'architecture fournissant une notation simple pour spécifier la structure de systèmes distribués construits à partir de composants. À partir de cette description, les composants peuvent être instanciés à des fins de former une architecture spécifique exécutable. Ce langage fournit une sémantique basée sur le π-calcul[6] et supporte la description d'une certaine dynamique de l'application en termes de schéma de création de composants logiciels en cours d'exécution.

La figure 6 illustre notre application bancaire en termes de composants et interconnexions décrits dans le langage Darwin. Dans un premier temps, nous ne nous intéresserons qu'à un seul `Titulaire` et un seul `Compte`. Ces composants sont contenus dans le composant `Banque` et sont liés au moyen d'une interconnexion. Le rond blanc représente un service requis et le noir un fourni.

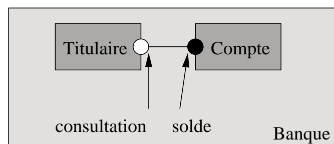


Figure 6: *Présentation des composants de l'application bancaire en Darwin*

La notion de service sera introduite dans la prochaine section.

4.1.1 Composants

Les principales abstractions gérées par Darwin sont les composants. Darwin les décrit en termes de services qu'ils fournissent et qu'ils requièrent pour leur permettre de communiquer avec d'autres composants.

Les composants primitifs

[6] Nous ne détaillerons pas dans ce document cette sémantique. Se référer à [MK96, page 5].

Les composants primitifs sont avant tout des entités d'encapsulation de fonctions et données d'un module logiciel. La description d'un composant est constituée de son nom et d'une interface qui déclare les services fournis et requis (`provide` ou `require`) par le composant. Voici la description des composants `Compte` et `Titulaire` dans le langage Darwin :

```
component Compte{
   provide solde <port, int> ;
}
component Titulaire {
   require consultation <port, int> ;
}
```

À partir de ce modèle décrivant sa nature et ses services, le composant primitif sera représenté à l'exécution par une classe C++. Celle-ci hérite d'une autre classe *process.* Elle est issue des classes de base de Régis [MDK94], support d'exécution réparti de Darwin.

Les composants composites

Un composant composite, lui, est construit à partir de composants primitifs et d'autres composites. La structure finale du système est représentée par un composant composite. La description d'un composant composite dans le langage est constituée de son nom suivi d'une liste de paramètres typés[7]. La portée de ces paramètres est l'implémentation du composant uniquement. Il est alors possible d'utiliser la valeur des paramètres à l'intérieur du composite pour décrire, par exemple, le nombre d'instances de composants à créer[8]. Dans la description de ce composant apparaissent aussi les services requis ou fournis. Vient ensuite l'implémentation du composant composite, vide dans le cas de composants primitifs (`Titulaire` et `Compte`), mais qui pour les composites (`Banque`) est constituée de déclarations d'instances et de schémas d'interconnexion. La définition des implémentations des composites s'appuie sur deux constructions syntaxiques de base : l'opérateur *inst* qui déclare une instance d'un composant (sur éventuellement un site particulier) et l'opérateur *bind* qui relie un port requis en partie gauche avec un port fourni en partie droite à l'aide du constructeur −−.

```
component Banque{
   inst
     titulaire : Titulaire ;
     compte : Compte ;
   bind
     titulaire.consultation--compte.solde ;
}
```

Nous n'avons pas encore abordé la répartition des composants sur des sites différents. Elle se fait lors de la création d'instances de composants en y ajoutant un numéro désignant le site de création. Ce numéro correspond à un site dans les tables d'administration internes du support Régis. L'ordre suivant permet de créer un client sur le site numéro 2 :

```
inst titulaire : Titulaire@2
```

Les services

Les composants interagissent au moyen de services. La notion de connecteurs n'apparaît pas dans ce langage. En effet, chaque interaction est représentée par un lien entre un service requis et un service fourni entre des composants différents. La notion de service s'apparente ici plus à des flots de communication qu'à la notion de fonctions que l'on retrouve dans Wright (*cf.* section 3.2). Les services n'ont aucune connotation fonctionnelle, ils désignent seulement le type d'objet de communication utilisé ou autorisé à venir appeler une fonction du composant. Ces objets de

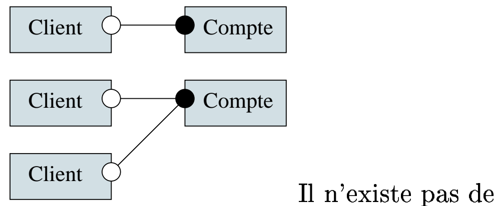
[7] Le composant figurant dans cet exemple ne possède pas une telle liste.
[8] Dans la suite du document, un exemple en fait usage.

communication sont fournis par le support Régis qui permet à des configurations Darwin de s'exécuter.

Le type des services est spécifié dans leur signature. L'usage du type `port` est le plus courant : il s'agit d'un objet envoyant des requêtes de manière synchrone ou asynchrone entre les composants répartis ou non. Darwin, par défaut, effectue un simple test d'équivalence des noms et des types, vérifiant que le service requis est compatible avec le fourni. Très peu de vérifications étant faites, nous pouvons considérer que l'aspect de typage dans Darwin n'est pas très bien développé.

4.1.2 Opérateurs particuliers

Darwin fournit des constructeurs dans le langage qui permettent de définir les schémas d'instanciations de composant plus complexes. Pour présenter ces différents opérateurs, nous allons nous appuyer sur notre exemple en l'étendant un peu. Considérons pour cela un nombre n de clients ayant chacun un compte qu'ils veulent consulter, sauf le dernier client qui partage son compte avec son conjoint. Une représentation graphique est proposée avec la figure 7.



Il n'existe pas de

Figure 7: *Représentation graphique de la configuration*

Dans l'exemple ci-dessous, nous utilisons les mêmes composants `Client` et `Compte` que ceux définis précédemment :

```
component Banque(int n) {
// Interface vide

// Implémentation
// Définition d'un ensemble d'instances de
   clients et comptes en banque
   array : Cl[n] : Client ;
   array : Cpt[n] : Compte ;
   forall k : 0..n
      inst Cl[k] ;
          Cpt[k] ;
      bind Cl[k].consultation--Cpt[k].solde ;
      when k=n-1
         bind Cl[k].consultation-- Cpt[k-1].solde ;
}
```

Deux nouveaux opérateurs sont utilisés ici. `Forall` permet de déclarer un tableau de clients et un tableau de comptes et faire des interconnexions entre leurs composants. Le constructeur conditionnel `when` permet d'effectuer des déclarations de création d'instances et d'interconnexions selon la validité de la condition.

4.1.3 Évolution dynamique

Darwin apporte une amélioration notable par rapport à des langages comme UniCon puisqu'il autorise la spécification des instants et des emplacements de création des composants et interactions d'une application plutôt que de considérer le système comme étant fixe par rapport à

sa phase de conception. Il fournit deux principaux mécanismes pour la description de structures dynamiques : l'instanciation paresseuse et dynamique.

Instanciation paresseuse

La première de ces instanciations est l'instanciation paresseuse dans laquelle un composant fournissant un service n'est pas instancié tant que l'utilisateur de l'application n'essaie pas d'accéder à ce service.

```
component Banque{
   inst
      titulaire : Titulaire ;
      compte : dyn Compte ;
   bind
      titulaire.consultation--compte.solde ;
}
```

Les instances non dynamiques sont créées normalement lors de l'instance du composant composite. Seules les instances déclarées dynamiques sont effectuées quand le service qu'elles fournissent est sollicité. Initialement, un seul `titulaire` est instancié. L'instance du `compte` n'est pas directe puisqu'elle est déclarée comme *lazy* par le mot clé `dyn`. Quand le `compte` est appelé pour la première fois par son service fourni par l'expression `compte.solde`, l'instanciation est déclenchée. L'instanciation paresseuse permet donc, par une spécification de structures dynamiques, de décrire la configuration potentielle à l'exécution en déclarant des instances qui seront peut-être créées par un appel à leur service. Ce type d'instanciation permet notamment de décrire des structures dont la taille ne peut être déterminée que dynamiquement. Malheureusement, ce mécanisme ne permet d'instancier qu'un seul composant par clause d'interconnexion, contrairement à l'instanciation dynamique que nous détaillerons par la suite.

Instanciation dynamique directe

L'instanciation paresseuse ne pouvant pas toujours être utilisée, Darwin propose alors le concept d'instanciation dynamique directe.

Dans cet exemple, un client de la banque réalise la création dynamique d'un nouveau compte. La déclaration de cette instanciation dynamique est réalisée dans la clause d'interconnexion `titulaire.creation--dyn Compte`. Pour cela, un service requis particulier (`creation`) du composant initiateur (`titulaire`) doit exister. Dès que le composant de la partie gauche (`titulaire`) demande une création au travers un service, le composant de la partie droite (`Compte`) est créé. Cette interconnexion ne sert à rien d'autre qu'à la création du composant, aucun appel de service n'est effectué.

```
component Banque{
   inst
      titulaire : Titulaire
   bind
      titulaire.creation--dyn Compte ;
}
```

Il est à noter que le lien est spécifié pour le type du composant à instancier (`Compte`) plutôt que pour une instance de ce type. L'instanciation dynamique directe, par ce fait, permet de multiples instanciations au moyen d'une seule clause. Le problème majeur de cette technique est l'inaccessibilité des composants une fois qu'ils sont créés dynamiquement. En d'autres termes, les composants qui sont créés statiquement ne peuvent pas communiquer avec ceux qui sont instanciés dynamiquement. L'inverse est cependant possible.

4.1.4 Atouts

Parmi les avantages de Darwin, nous pouvons noter :

- Tout d'abord, Darwin a la capacité de construire une configuration dynamique. En effet, ce langage présente deux constructions qui permettent de décrire un schéma d'instanciation dynamique de composants et de relier ces instances au reste du système par un moyen de communication. ;
- Des constructeurs du langage, comme *forall* et *when* permettent de spécifier des configurations complexes et paramétrables. Il nous semble que ces opérateurs sont appropriés lorsque la topologie des interconnexions ou la répartition sur les sites est connue à l'avance ;
- L'utilisation des mécanismes de communication entre les composants colocalisés ou distribués est complètement transparente pour le programmeur de l'application. Le facteur de répartition des composants est pris en compte lors de chaque instanciation de composants.

4.1.5 Limites

- La description de la dynamicité d'un ensemble de composants est limitée. Il n'est pas possible de supprimer des composants dynamiquement ni de permettre à un composant normal de communiquer avec des composants dynamiquement créés. En effet, seuls les composants dynamiques peuvent communiquer avec l'extérieur. Une sélection dynamique des intervenants d'une communication dynamiquement créés n'a malheureusement pas été envisagée ;
- Darwin ne fournit pas de base adéquate pour l'analyse du comportement d'une architecture. L'implémentation des composants sont des boîtes noires non interprétées. Les types de services dépendent de la plate-forme Régis et la sémantique est alors là-aussi non interprétée dans Darwin. Le modèle devrait fournir plus de moyens pour décrire des propriétés associées à un composant ou ses services. En outre, le typage dans Darwin n'est pas bien intégré. Une simple vérification du typage des services ne paraît pas suffisant ;
- Le support de Darwin pour les styles d'architecture est par ailleurs limité par sa faiblesse relative à la notion de connecteurs. Le schéma d'interaction peut être difficilement décrit indépendamment du composant qui le fournit. De plus, l'utilisation de divers modes de communication n'est pas configurable par le concepteur de l'architecture. Il faut que ce support d'exécution supporte les bons types de communication et il n'est malheureusement pas possible d'étendre la classe des types supportés.

4.1.6 Bilan

Darwin propose une vision de l'architecture claire et fonctionnelle. À travers l'utilisation de constructions conditionnelles et itératives, il autorise une distribution sur les sites plus aisée. En outre, l'utilisation de paramètres pour déterminer la structure du système à l'initialisation est possible. Mais l'apport principal de Darwin est de permettre des descriptions de structures dynamiques qui évoluent en cours d'exécution.

4.2 Olan

Olan [Bel97, BAKR95] est un environnement de programmation qui vise à faciliter le développement, la configuration et le déploiement d'applications réparties construites par assemblage de composants hétérogènes. Son langage de description d'architecture, OCL (Olan Configuration Language), repose sur un modèle d'assemblage de composants logiciels, dont la conception initiale a été inspirée de Darwin et de sa notion de composants interconnectés. Cependant, certaines limitations de ce langage ont été levées, avec entre autre, un début de spécification du comportement dynamique de l'application, et l'insertion de la notion de connecteur dans l'architecture. Nous allons présenter dans cette section le langage de description OCL, ainsi que l'architecture et l'implémentation du système d'exécution Olan, dédié à l'installation et l'exécution des applications.

4.2.1 Modèle de composants

Olan permet de spécifier les architectures d'applications dont les abstractions de base sont les composants, entités d'intégration et de structuration de logiciels ; et les connecteurs, entités de gestion de la communication entre composants. Un composant englobe la définition d'aspects statiques (au travers d'une interface) et dynamiques (au travers d'une réalisation et d'une configuration). Les composants décrits en Olan dérivent d'une classe `Component`. Cette classe Java peut donner lieu à de multiples instances, créées dynamiquement lors de l'exécution de l'application. Deux sortes d'implémentation sont disponibles : les primitifs encapsulent directement les modules logiciels ou classes écrits dans des langages de programmation comme Java, alors que les composites contiennent les interconnexions de composants, qu'ils soient composites ou primitifs.

Interface

Les dépendances fonctionnelles du composant avec le monde extérieur sont exposées au niveau de l'interface. Celle-ci permet de décrire de manière complète l'accès aux opérations et aux structures de données fournies par le composant. La terminologie associée à ces opérations est le *service* qui contient les indications sur le rôle de l'opération et le mode de communication devant être utilisé pendant l'exécution. La sémantique exacte d'un service dépend de la manière dont il sera défini et accessible à l'exécution. Le tableau 2 détaille des types de service aujourd'hui existants dans OCL. Voici un exemple de service qui utilise le support d'exécution A3[9].

```
service CompteService : A3Service{
   notifications = aaa.CompteNotificationSet;
   file = ${SRC_DIR}/aaa/CompteNotificationSet.java;
   }
}
```

Un service de type A3 spécifie toutes les notifications devant être gérées au sein du service. Une notification représente des données significatives dont la diffusion peut entraîner l'exécution de réactions de la part de composants présents et intéressés par ces données. La localisation du fichier Java implémentant ces notifications est donnée par le chemin d'accès `file`. Ce fichier permet de faire la correspondance entre les services fournis et requis spécifiés dans l'interface, et l'implémentation de ces services. [BPF] présente plus précisément les différents types de service existants.

Support	Type de service	Description des opérations
AAA	A3Service	Type Java sérialisable
Corba IDL	IDLService	Type IDL de l'OMG
Java RMI	JRemoteService	Type Java sérialisable
Java rARI	ARIRemoteService	Type Java sérialisable

Table 2: *Types de service d'Olan*

L'interface peut aussi contenir des définitions d'attributs publics. Les attributs sont des variables typées qui permettent de rendre accessibles au niveau du langage certaines données contenues initialement dans le code intégré et qui peuvent éventuellement changer en cours d'exécution. Certains attributs sont prédéfinis (tout composant possède par exemple un « site de chargement »), tandis que d'autres sont définis directement par le réalisateur. Voici l'exemple Olan de l'interface du composant Compte.

```
Interface CompteItf{
   attribute String personne;
   attribute Location localisation;
```

[9] Une description succincte de ce support est fournie en section ??

```
    provided CompteService solde ;
    provided CompteService init ;
}
```

Le compte en banque possède un attribut `personne` qui indique le nom de la personne propriétaire du compte, `localisation` indique le site d'exécution du composant. Les opérations `solde` et `init` sont des notifications implémentées au sein du fichier `CompteNotificationSet.java`.

Implémentation

Un type caractérise la nature de l'implémentation du composant. En fonction du type, l'interface et les propriétés doivent être soigneusement choisies. Le tableau montre les types disponibles ainsi que leurs services associés.

Type	Description	Services
A3Agent	Agent A3	A3Service
CORBAObject	Objet CORBA distribué	IDLService
RMIObject	Objet Java RMI	JRemoteService
ARIObject	Objet Java ARI	ARIRemoteService
CorbaAgentProxy	Passerelle entre objet CORBA et A3	A3 et IDLService

Deux sortes de composants sont identifiés : les composants primitifs et composites.

Composant primitif Un composant primitif est une entité logicielle encapsulant la mise en œuvre d'entités logicielles qui forment une application. La réalisation d'un composant primitif comprend son interface et une implémentation des entités programmées qu'il encapsule, écrites en Java ARI (modèle RMI en asynchrone), Java RMI, A3 ou CORBA. La syntaxe d'une implémentation est illustrée dans l'exemple suivant de la banque.

```
Component Compte : A3Agent{
  A3Server location ;
  Interface CompteItf{
     attribute String personne ;
     attribute Location localisation ;Il n'existe pas de
     provided CompteServive solde ;
     provided CompteService init ;
}
  Implementation{
     agent = aaa.Compte ;
     file = ${SRC_DIR}/aaa/Compte.java ;
  }
}
```

La mise en œuvre du composant dans le fichier `Compte.java` permet de faire une liaison explicite entre les services déclarés dans l'interface et ceux définis dans le code du composant.

Composant composite Les composants composites servent à la fois d'entités de description de la configuration et d'entités de structuration d'une application en composants coopérants. Les composites permettent de former une hiérarchie de composants partiellement ou totalement réutilisable dans diverses applications.

Chaque composite est formé d'une interface identique à celle des primitifs. De la même façon, un composite est composé d'une mise en œuvre (*configuration*) qui contient la déclaration des sous-composants nécessaires avec éventuellement des paramètres de création de ces composants, ainsi que les interconnexions entre eux. Ce schéma est essentiellement statique, car tous les sous-composants sont créés à un seul instant, lors de la création du composant composite qui les

englobe. L'utilisation du constructeur `instance` permet de déclarer les sous-composants à instancier. Un composite présente en plus les interconnexions entre eux. La figure 8 illustre le composite `Application` de l'application bancaire.

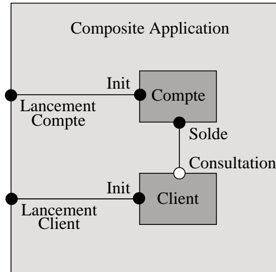


Figure 8: *Représentation Olan du composite Application*

Les lignes horizontales et verticales représentent les connecteurs dont la description est établie dans la prochaine section. Voici le code associé à cette figure :

```
Component class Application{
interface
   provided appService lancementCompte ;
   provided appService lancementClient ;
configuration
   Compte compte ;
   Client client ;
//Instanciation des sous-composants
   compte = new Compte(localisation = caroline.inrialpes.fr) ;
   client = new Client(localisation = dyade.inrialpes.fr) ;
//Connexions
   lancementCompte => compte.init ;
   lancementClient => client.init ;
   client.consultation => compte.solde using A3bus ;
}
```

Le composite exhibe au niveau de son interface deux services fournis pour configurer et lancer les composants `Compte` et `Client`, qui seront instanciés dans la partie `configuration`. Au cours de cette instanciation, l'attribut `localisation` est fixé, afin de définir le site d'exécution de ces composants.

4.2.2 Modèle d'interactions

Connecteur

Les connecteurs sont des objets qui contrôlent les interconnexions de composants et spécifient le protocole requis à l'exécution, identifiant le médium utilisé pour les communications locales ou distantes. Similairement aux notions de rôles introduits dans UniCon (*cf.* section 3.1), chaque connecteur Olan spécifie l'ensemble de composants dont il accepte la connexion. Des mots-clé identifient le type de service (`in` pour les services d'entrée et `out` pour ceux de sortie) attendu par le connecteur.

Actuellement, il n'existe pas de langage ou de formalisme, comme dans [AG94], pour exprimer les connecteurs. Ils sont fournis par le support d'exécution et sont utilisables directement dans le langage Olan. Quelques connecteurs prédéfinis (`builtin`) dans l'environnement Olan sont proposés mais, dans la version actuelle, il n'est pas permis d'intégrer des connecteurs propres à l'utilisateur :

```
connector A3bus{
   in A3Service input ;
   out A3Service output ;
   implementation {builtin}
}
```

Cet exemple de connecteur est utilisé pour la communication au sein des `A3Service`. La validité d'une interconnexion est effectivement vérifiée par les caractéristiques de chaque connecteur (services imposés, signatures compatibles). Par exemple, les services connectés aux extrémités du connecteur (`in` et `out`) doivent être de même type.

Connexion

Les connexions sont les communications effectives qui prennent place entre les composants. Elles peuvent être vues comme des instances d'un type de connecteur avec une implémentation spécifique dont les sources et destinations sont spécifiées. Chaque interaction contient l'initiateur de la communication et le ou les destinataires. Une connexion ne décrit jamais le moment exact de la communication. L'interconnexion a pour unique rôle de dire que dès lors l'initiateur effectue une demande de communication, celle-ci sera effectuée en utilisant tel mécanisme ou protocole pour envoyer une requête vers les services destinataires. La clause `using` d'une interconnexion indique le type de connecteur utilisé.

Reprenons l'exemple du composant composite `Application`. Son implémentation contient la définition d'une interconnexion entre les sous-composants `client` et `compte` :

```
client.consultation => compte.solde using A3bus ;
```

Dans cette clause, le client est connecté au composant `compte` par le biais d'un connecteur de type A3. Les services de type `A3Service` reliés étant `provided` et `required`, l'interconnexion est bien valide.

Liaison

Les liens définissent les correspondances entre les services du composite englobant, et les services de ses sous-composants définis à l'intérieur du composite. Ils correspondent aux traits horizontaux de la figure 8.

```
lancementCompte => compte.init ;
lancementClient => client.init ;
```

Dans notre illustration, deux liens entre services fournis sont établis. La syntaxe de ces liaisons est assez similaire aux connexions, si ce n'est que le nommage des services du composite n'est pas préfixé par le nom de ce composant.

4.2.3 Évolution dynamique

Deux types d'instanciation dynamique existent dans un programme OCL : l'instanciation paresseuse et dynamique. Ce sont des opérations comparables à celles qui ont été présentées dans Darwin. Nous ne faisons donc pas de rappel et présentons directement un nouveau schéma d'instanciation dynamique, introduit par les auteurs d'OCL. Ce concept offre à l'architecte de l'application le contrôle d'un ensemble de composants ayant la même interface.

Collection

Les collections, ou les groupes, sont des ensembles dynamiques de composants identiques. La désignation des composants peut prendre deux formes. Tout d'abord, les membres de la collection peuvent être dynamiquement déterminés par le fait qu'ils satisfont certaines propriétés (un composant récepteur d'un message, un composant soumis à certaines règles d'administration...), les

composants sont alors gérés implicitement. Une deuxième forme de désignation consiste à percevoir les composants comme une seule entité et la désignation concerne donc tous les membres de la collection. La clause suivante définit une collection de comptes en banque.

```
collection = Collection[0..n] of Compte ;
```

Les collections ont été introduites pour faciliter l'utilisation de multiples composants de même type. Son nombre à l'exécution repose sur un intervalle défini par un nombre minimum et maximum. Cette cardinalité est passée en paramètre au constructeur de la collection, au moment initial. À sa création, la collection est vide. On spécifie, par cet intervalle de valeurs, le nombre d'instances de composants que la collection pourra contenir à l'exécution. Deux opérateurs particuliers permettent d'ajouter ou de supprimer des composants d'une collection en cours d'exécution, ce sont `new` et `delete`, qui sont appliqués à un identificateur.

```
collection.new(idf) ;
collection.delete(idf) ;
```

De plus, un mécanisme spécial peut effectuer la création d'un nouveau composant de la collection avant d'accéder à un service spécifique. Quand l'initiateur demande à l'application de se lancer, une première instance du composant est créée et son service `init` est activé.

```
initiateur.lancement => collection.init using createInCollection ;
```

Enfin, pour faire un appel de service à distance du client vers un serveur de la collection, une méthode de nommage par association est utilisée, basée sur la vérification d'attributs existants à l'exécution. Le code suivant montre une utilisation possible.

```
client.envoiRequete(requete, idf) =>
       collection.receptionRequete(requete)
       where collection.idf = idf using methodCall ;
```

Ainsi, l'atout principal des collections est de créer des ensembles de composants ayant les mêmes fonctions mais dont le nombre varie. Cela autorise la manipulation d'entités qui peuvent évoluer dynamiquement et qui sont accessibles à travers une seule interface.

4.2.4 Atouts

- La description que le composant primitif fournit au travers d'une interface est homogène et indépendante de l'entité encapsulée, de son langage de programmation ou de sa plateforme d'exécution. Ils peuvent être utilisés dans n'importe quelle architecture à condition de satisfaire ses besoins fonctionnels et son modèle d'exécution. Ceci est une amélioration conséquente par rapport à UniCon où le langage de programmation n'était pas transparent au niveau des composants. De plus, OCL est plus souple en terme de définition de composants, son typage est moins contraignant. Il garde malgré tout la possibilité d'effectuer un grand nombre de vérifications sur l'architecture, en particulier avec les connecteurs et les règles d'interconnexion qui y sont attachées ;
- En outre, OCL apporte des spécifications pour la communication des composants par le biais de connecteurs, concept inexistant dans Darwin. Même si cette notion n'est pas encore bien définie, elle apporte déjà des réponses quant à la gestion des divers aspects de la communication, comme le protocole de communication utilisé entre les composants ;
- Le langage offre le concept de collections qui remplace avantageusement l'instanciation dynamique de Darwin, car elle peut être contrôlable par un connecteur et non par un service de création particulier comme Darwin. Elle permet de créer comme de supprimer des composants et surtout elle permet l'accès par des composants clients aux composants contenus dans

la collection, ce que ne permet pas l'instanciation dynamique de Darwin. Ainsi, concernant la gestion dynamique d'instances de composants, l'environnement Olan est assez complet ;
- Enfin, Olan offre un environnement complet de construction d'applications. Toutes les étapes du cycle de vie d'une application distribuée sont représentées, incluant la construction, l'installation, le déploiement, la configuration du système.

4.2.5 Limites

- Les inconvénients de l'approche Olan sont essentiellement liés au travail de création des composants. Il faut écrire les interfaces, les implémentations primitives dans un langage de programmation, et décrire l'application, les composites et primitifs, avec OCL. Ce travail est donc relativement long par rapport à la description Darwin ou UniCon ;
- Dans l'interface d'un composant est défini le type de service qu'il possède, définissant par la même le support d'exécution que le composant utilise (*cf.* tableau 2). Cette information est suffisamment pertinente pour anticiper le modèle de communication nécessaire. On aurait pu imaginer une correspondance directe entre le type de composant et le type de connecteur, évitant à l'architecte de préciser quel connecteur utiliser dans les clauses d'interconnexion.

4.2.6 Bilan

Olan est un langage qui permet aux applications d'être exprimées comme une vue hiérarchique de composants interagissants. Le modèle architectural est basé sur celui de Darwin [MDEK95], étendu à la notion de connecteurs et aux collections. Cet environnement est idéal pour le développement d'applications réparties puisqu'il génère, à partir d'intégrations de morceaux logiciels hétérogènes, une application déployable.

5 Synthèse des approches existantes

UniCon est le langage de définition d'architecture le plus complet que nous ayons étudié. Il permet de faire remonter au niveau du concepteur de l'application des abstractions qu'il manipule régulièrement, telles que la notion de processus, filtres, *etc.* Des règles d'interconnexion strictes autorisent ou n'autorisent pas l'utilisation d'un connecteur avec des composants. Ainsi, UniCon effectue toutes les vérifications d'assemblage des entités et mécanismes utilisés pour faire fonctionner une application répartie. L'approche de Darwin est un peu différente car il n'existe pas de typage de composants au sens de UniCon. Un composant est une entité logicielle homogène sur laquelle aucune contrainte comparable aux types n'existe. UniCon tente de fournir au concepteur de l'application un cadre et des règles pour automatiser et garantir l'intégrité de fonctionnement des différents mécanismes de programmation.

OCL et Darwin proposent, quant à eux, une description dynamique de l'architecture. Ces deux langages permettent d'exprimer une partie de l'évolution initiale de l'application, essentiellement le schéma d'instanciation des composants. UniCon considère ce schéma d'instanciation comme statique, défini une fois pour toute par l'architecte. Le tableau 3 présente les caractéristiques de tous les langages étudiés. Il s'intéresse aux aspects dynamiques, mais la comparaison de ces langages porte aussi sur la hiérarchisation de l'application ainsi que la présence de la notion de connecteur et la génération automatique d'une image exécutable de l'application distribuée. Tous ces aspects semblent essentiels pour le développement de systèmes répartis. D'une part, l'intégration de logiciels comporte une phase de structuration en proposant de modéliser une application comme une hiérarchie de composants. Cette modélisation permet de ne pas considérer l'application comme un ensemble plat de composants. D'autre part, la séparation du code de la communication et celui de la mise en œuvre des composants logiciels permet de créer des composants logiciels sans se soucier de l'utilisation ou de la programmation des communications distantes eIl n'existe pas dentre sites. Enfin, la description de l'architecture, associée au code des composants logiciels permet, dans certains langages, d'installer et de faire exécuter l'application

sur un système réparti. Cette caractéristique offre des facilités pour le déploiement, puisqu'une génération automatique de l'exécutable est effectuée.

Langage	Dynamicité	Hiérarchisation	Connecteur	Déploiement
UniCon	non	non	oui	non
Darwin	oui	oui	non	oui
Olan	oui	oui	oui	oui
Wright	oui	non	oui	non
Rapide	oui	oui	non	non

Table 3: *Caractéristiques des différents ADL*

Les caractéristiques d'instanciation statique des langages de description d'architecture sont intéressantes mais trouvent rapidement leur limite dans des applications réalistes de grande taille. Il est en effet impensable d'installer un ensemble de composants avant l'exécution si ces composants ne sont jamais utilisés. De plus, il peut être intéressant de créer des ensembles de composants ayant les mêmes fonctions mais dont le nombre varie pour des raisons de disponibilité par exemple. Darwin fournit des réponses partielles à ces soucis avec l'instanciation paresseuse d'une part et l'instanciation dynamique d'autre part. L'instanciation paresseuse est l'action de déclarer une instance en retardant l'instant de sa création au premier accès. L'instanciation dynamique est la possibilité de créer des instances n'importe où, quand par exemple, un client le demande via un service particulier de création. Nous avons vu que l'instanciation paresseuse de Darwin ne répond pas à tous les besoins de création dynamique et que l'instanciation dynamique possède des inconvénients majeurs tels que l'impossibilité par un composant client d'appeler des services des instances créées dynamiquement.

Dans OCL, les deux concepts existent. Le concept de « collection » a été aussi introduit, apportant des réponses concernant la gestion dynamique d'un groupe de composants (*cf.* tableau 4). Les collections sont des ensembles, bornés ou non, de composants ayant la même interface. La cardinalité de l'ensemble est contrôlable par l'architecte de l'application, car une collection permet d'ajouter ou de supprimer des composants en cours d'exécution. Il existe à cet effet deux connecteurs spécifiques, qui lorsqu'ils sont utilisés pour la communication, déclenchent la création ou la suppression d'une instance de composant dans la collection.

Langage	Composant	Groupe	Connexion	Attribut
UniCon	aucune	aucune	aucune	aucune
Darwin	instanciation	aucune	aucune	complète
Olan	instanciation	gestion	aucune	complète
Wright	complète	aucune	complète	modification
Rapide	complète	gestion	complète	modification

Table 4: *Dynamicité possible pour les abstractions architecturales*

Enfin, les langages de description d'architecture s'architecturent en deux classes distinctes : les langages de simulation (ex : UniCon, Wright, Rapide) et les langages de configuration (ex : Darwin, Olan). La première classe de langages vise à spécifier les architectures en vue de simuler leur comportement, alors que la seconde décrivent les architectures pour générer une image exécutable de l'application.

Rapide faisant partie des langages de la première catégorie, il ne prend pas en compte les éléments relatifs à l'intégration de code et de placement. L'effort de Rapide se porte autour de la description des architectures dont la dynamique des communications est très forte. Rapide offre un certain nombre de solutions originales à la description de solutions complexes qui évoluent fortement au cours de l'exécution. Chaque interconnexion est ainsi décrite comme une règle qui est évaluée et déclenchée dès que les conditions d'exécution s'y prêtent.

Enfin, concernant Wright, on peut noter qu'il n'est pas non plus dédié à la production d'une image exécutable de l'application. Il autorise des vérifications, de la manière à UniCon, mais ne construit pas d'application. De plus, la répartition n'est pas prise en compte, Wright s'occupe essentiellement de définir le comportement de composants et connecteurs en terme de processus communiquants. Pour la gestion de la dynamicité, Wright introduit dans son algèbre à base de processus un vocabulaire de reconfiguration. Toute la dynamicité introduite est gérée au sein d'une entité monolithique, le configurateur (*cf.* tableau **??**). Seules des spécifications et contraintes sont introduites dans la description des composants. Rapide, Olan et Darwin intègrent, quant à eux, tous les aspects dynamiques au sein des composants.

Pour conclure, nous considérons qu'un certain nombre de propositions sont intéressantes mais elles sont ciblées pour certaines utilisations. Wright et Rapide expriment bien la dynamique de l'application mais ne s'intéressent qu'à sa vérification comportementale, ils ne génèrent pas d'image exécutable et ne prennent pas en compte la répartition des composants. De la même manière, les langages Darwin et Olan proposent des schémas d'instanciation qui ne répondent pas à tous les besoins de création dynamique. Mais ils proposent de générer les applications ou squelettes d'applications, correspondant à la description fournie.

6 Conclusion

L'administration d'applications réparties demande une connaissance détaillée de l'architecture des applications et de ses possibilités d'évolution. L'ensemble de cette connaissance peut être initialement exprimé au niveau des langages de description d'architecture. Leur but est de fournir une certaine vision de l'architecture de l'application en termes d'entités logicielles nécessaires au fonctionnement de l'application et de leurs intercommunications. Ces langages sont traditionnellement utilisés dans un phase de construction de l'application.

Dans ce délivrable, nous avons présenté ce qu'est la programmation par composants. Ensuite, nous avons étudié quelques langages académiques significatifs, que nous avons classés en deux catégories : les langages de description d'architecture pour la simulation ou pour la génération de code.

Pour conclure, nous notons qu'il n'existe pas de modèle de composants universel qui prenne en compte tous les besoins des applications. Il est donc nécessaire de choisir la classe de langages nécessaire à ses besoins et ensuite sélectionner, au sein de cette classe, le langage de description d'architecture qui conviendrait le mieux.

Références

[ADG98] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998.

[AG94] R. Allen and D. Garlan. Formal connectors. *CMU Tech. Report CMU-CS-94-115*, March 1994. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA15213, USA.

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. In *ACM Transactions on Software Engineering and Methodology*, volume 6(3), pages 213–249, July 1997.

[All97] R. Allen. *A Formal Approach to Software Architecture.* PhD thesis, Carnegie Mellon University, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[BAKR95] L. Bellissard, S. B. Atallah, A. Kerbrat, and M. Riveill. Component-based programming and application management with Olan. In *Proceedings of Workshop on Object-based Parallel and Distributed Computation*, Tokyo, Japan, June 1995. LNCS Springer Verlag.

[Bel97] L. Bellissard. *Construction et Configuration d'Applications Réparties.* PhD thesis, Institut National Polytechnique de Grenoble, ENSIMAG, Décembre 1997.

[BPF] L. Bellissard, N. De Palma, and D. Fėliot. The Olan architecture definition language. Internal Report. Version 2.1.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[ISZ98] V. Issarny, T. Saridakis, and A. Zarras. A survey of architecture description languages. Technical report, C3DS Project, June 1998.

[LKA+95] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. In *IEEE Transactions on Software Engineering*, volume 21 n.4, pages 336–355, April 1995.

[Luc96] D. C. Luckham. Rapide : A language and toolset for simulation of distributed systems by partial orderings of events. In *Technical Report CSL-TR-96-705*, Stanford University, Computer Systems Laboratory, September 1996.

[LV95] D. C. Luckham and J. Vera. An event-based architecture definition language. In *IEEE Transactions on Software Engineering*, volume 21 n.9, pages 717–734, September 1995.

[LVM] D. C. Luckham, J. Vera, and S. Meldal. *Three Concepts of System Architecture.*

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Fifth European Software Engineering Conference (ESEC'95)*, Barcelona, September 1995.

[MDK94] J. Magee, N. Dulay, and J. Kramer. Regis : A constructive development environment for distributed programs. In *IEEE Distributed Systems Engineering Journal*, volume 1 n.5, pages 304–312, December 1994.

[MK96] J. Magee and J. Kramer. Dynamic structure in software architectures. In *ACM SIGSOFT'96 : Fourth Symposium on the Foundations of Software Engineering*, pages 3–14, San Fransisco, CA, October 1996.

[MT97] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *Sixth European Software Engineering Conference*, pages 60–76, Zurich, Switzerland, September 1997.

[OMT98] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *International Conference on Software Engineering (ISCE'98)*, Kyoto, Japan, April 1998.

[SDZ96] M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations for architectural connections. In *Third International Conference on Configurable Distributed Systems*, May 1996.

A Utilisation simplifiée de CSP

En CSP, les processus sont définis par des suites d'événements. Alors que cette algèbre possède un ensemble riche de concepts pour la description d'entités communicantes, nous n'utilisons qu'un sous-ensemble incluant :

Des processus et événements Un processus décrit une entité qui peut s'engager dans des communications événementielles. Les événements peuvent être primitifs ou ils peuvent être associés à des données (*p. ex.* `P.e? x` et `P.e! x` représentent des données d'entrées et de sorties respectivement sur le port `P`). L'événement $\surd$ est utilisé pour représenter la terminaison réussie d'un processus ;

Des préfixes Un événement `e` précédant le processus `P` est noté `e` $\rightarrow$ `P` ;

Des choix déterministes Le choix entre `P` ou `Q`, fait par l'environnement, est noté `P` $\Box$ `Q`. L'environnement correspond aux autres processus interagissant avec le processus courant ;

Des choix non-déterministes Le choix entre `P` ou `Q`, fait par le processus lui-même est noté `P` $\sqcap$ `Q`.

Dans les expressions de processus, $\rightarrow$ est associative à droite et prioritaire par rapport à $\square$ ou $\sqcap$.

De plus, pour pouvoir utiliser des noms locaux, les λ-expressions, telles que `P = let Q = expr in R`, sont introduites.