UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS - UFR Sciences

École Doctorale de Sciences et Technologies de l'Information et de la Communication

THÈSE

pour obtenir le titre de

Docteur en Sciences

de l'UNIVERSITÉ de Nice-Sophia Antipolis

Discipline : Informatique

présenté et soutenue par

Antonio CANSADO

FORMAL SPECIFICATION AND VERIFICATION OF DISTRIBUTED COMPONENT SYSTEMS

Thèse dirigée par **Eric MADELAINE**

soutenue le 4 décembre 2008

Jury:

Président du Jury	Jean-Paul RIGAULT	Université de Nice-Sophia Antipolis, France
Rapporteurs	Carlos CANAL	Universidad de Málaga, Espagne
	František PLÁŠIL	Charles University, République Tchèque
Examinateurs	Tomás BARROS	NIC Chile - Universidad de Chile, Chile
	Fréderic LANG	INRIA Rhones-Alpes, France
	Lionel SEINTURIER	Université de Lille, France
Directeur	Eric MADELAINE	INRIA Sophia-Antipolis, France

Acknowledgements

I would like first to thank my advisor, Eric Madelaine, who with great patience guided me throughout this thesis. His advices were always of excellence and made this work possible.

Next, I would like to highlight the help of Ludovic Henrio, who unofficially acted as my co-advisor giving me innumerous comments and pointers on the work.

I would like to thank the Chilean Commission in Research and Technology (Conicyt), and INRIA who granted me with a scholarship for doing my studies in France.

To all people at INRIA and particularly to the OASIS team, I'm deeply grateful. From the very start they welcomed me and let me feel as part of a long friendship.

Last, but not the least, to all my friends. They were the ones that pushed me all through this adventure.

Contents

List of Figures

Part I

Résumé étendu en français (Extended french abstract)

1 Introduction

Le génie logiciel est encore une discipline immature. Le problème semble être que nous ne savons pas développer des logiciels de manière fiable. Il est impressionnant de voir combien de logiciels sont infestés des bogues et de comportements non-déterministes.

La question qui se pose est : pourquoi les ingénieurs peuvent construire des bâtiments très solides mais qu'ils échouent quand il s'agit de logiciel ? Pour répondre à la question, un aspect essentiel est que dans la plupart des domaines il est plus ou moins facile(ou au moins bien connu) de concevoir un système dont les différentes parties peuvent être construites par plusieurs personnes de façon indépendante, et plus tard assemblées pour obtenir le produit final. De plus, le produit final n'est pas construit à partir de zéro mais en réutilisant des solutions précédemment testées. En informatique, au contraire, les ingénieurs ont tendance à reconstruire les mêmes applications plusieurs fois, et, par conséquent,la production coûte cher et il existe un risque élevé d'erreurs.

Alors, pourquoi n'arrivons nous pas à transférer cette expertise des ingénieurs, consolidée par des années de pratique, vers le cycle de vie du logiciel ? Un des problèmes du logiciel est que c'est un produit très abstrait: il est donc difficile de prévoir son comportement et les ressources nécessaires pour son fonctionnement. À première vue, il semble être un simple modèle mathématique, mais la réalité est très différente. Le développement logiciel a tellement de différents axes de conception que la majorité du design d'une application repose sur la créativité des ingénieurs. Comme chaque ingénieur peut avoir des idées brillantes, mais totalement différentes pour le même problème, l'assemblage des différentes parties du logiciel devient un véritable défi.

Pour réutiliser du logiciel, on doit l'imaginer comme une collection de modules. Cependant, on doit également considérer les influences de l'environnement car il est inutile de penser à un module logiciel qui fonctionne parfaitement découplé de son environnement. De plus, l'environnement peut très bien changer au cours du temps. Il y a deux idées importantes ici : (i) nous devons définir, de façon abstraite, ce qui est attendu de l'environnement et ce qui est fourni à l'environnement ; et (ii) nous devons concevoir des logiciels de manière que les hypothèses faites sur l'environnement soient les seules nécessaires pour utiliser le logiciel. Cela nous permettra de construire un logiciel qui peut travailler dans n'importe quel contexte qui correspond à nos hypothèses.

Il existe un excellent exemple qui nous montre que les hypothèses sur l'environnement sont souvent mal faites. L'agence spatiale européenne (ESA) a lancé une fusée en 1996 qui a explosé à cause d'un bogue logiciel [Ben-Ari 01] dans le système de navigation inertielle (INS). Le module avait été précédemment utilisé dans Ariane 4 avec succès, mais quand il a été utilisé dans Ariane 5, il est tombé en panne. Les spécifications d'entrée d'Ariane 5 étaient différentes de celles d'Ariane 4, ce qui a causé un débordement lors de la conversion d'une chiffre de 64 bits à 16 bits.

Personne n'a rien remarqué jusqu'à ce que la fusée explose.

Les ingénieurs ont abordé des problèmes similaires depuis le tout début du développement logiciel. Au début, ils ont été appelés "appels de procédures" et "bibliothèques" d'une part, et "spécifications formelles" ou "semi-formelles" d'autre part. En fait, ces sont les fondements qui ont donné lieu à la programmation orientée objets dans les années 60 et UML (pour Unified Modeling Language) [OMG 04] dans les années 90.

Aujourd'hui, nous sommes face à des projets beaucoup plus larges que d'habitude. Si l'on prend l'exemple du développement des systèmes d'exploitation, nous pouvons voir que la croissance du nombre de lignes de code source est énorme. Dans la Figure 1.1, nous montrons quelques valeurs approximatives [Wikipedia 08] sur le nombre de lignes de code nécessaires pour créer différentes versions de Microsoft Windows. [*]

Année	*Systèmes d'Exploitation*	*Lignes de Code*	*Nombre estimée de bougues*
1993	Windows NT 3.1	6 millions	120 milles
1994	Windows NT 3.5	10 millions	200 milles
1996	Windows NT 4.0	16 millions	320 milles
2000	Windows 2000	29 millions	580 milles
2002	Windows XP	40 millions	800 milles
2007	Windows Vista	50 millions	1 million

Figure 1.1: *Nombre de lignes de codes pour des differentes versions de Microsoft Windows*

Même si nous ne supposons pas que le rapport de bogues est constant, il est évident que quand il s'agit de projets de 50 millions de lignes de code, la qualité de développement du logiciel est très importante.

Le seul moyen de construire des logiciels de haute qualité, c'est à travers une bonne spécification du système. Par ailleurs, pour faire face aux projets de grande taille, il faut se raccrocher sur des méthodes de raisonnement compositionnel et des outils de vérification automatique qui nous donnent des garanties sur le comportement de notre programme.

[*] L'estimation du nombre de bugs ne sont qu'à titre indicatif. Ils sont basés sur des estimations faites par l'Université Carnegie Mellon CyLab Sustainable Computing Consortium de 20 à 30 bogues toutes les 1000 lignes de code en géneral.

1.1 Component Based Software Engineering

Récemment, il y a eu un intérêt majeur pour les paradigmes qui apportent une meilleure structuration des logiciels, parmi eux, les composants logiciels (CBSE) [Szyperski 02]. Ce qui différencie CBSE est la composition et l'accent mis sur la réutilisation du logiciel. Composition ici signifie construire logiciels plus complexes à partir d'autres logiciels. Ces logiciels sont appelés *composants* parce qu'ils peuvent être composés ensemble, et résultant dans un nouveau composant, plus complexe.

Un problème commun à CBSE est qu'il n'existe pas de consensus concernant ce que le composants logiciel est réellement. Plusiers définitions ont été proposées dans la littérature, mais aucune n'est considerée comme *la* définition. Toutefois, parmi les définitions les plus acceptées, nous citons [Szyperski 02] :

> *"Un composant logiciel est une unité de composition avec des interfaces spécifiées par contrat et des dépendances explicites par rapport au contexte. Un composant logiciel peut être déployé de façon autonome et est soumis à la composition par une tierce partie."*

Ce que nous pouvons extraire de cette définition est que les composants établisent des contrats avec l'environnement, et que les composants sont une sorte de modules indépendants qui peuvent être utilisés par un tiers. Le contrat avec l'environnement fixe des frontières bien définies que nous appelons *interfaces*. Une différence majeure avec le paradigme de la programmation objets est que dans CBSE nous définissons non seulement les interfaces fournies, mais aussi les interfaces nécessaires. Chaque élément a un ensemble d'interfaces et les composants peuvent être composés et assemblés par l'application de règles de composition sur les interfaces. Lorsque deux éléments sont liés, il doit y avoir une correspondance entre les services requis par un élément et ceux fournis par les autres, au moins concernant les interfaces que nous voulons connecter.

La définition de la compatibilité des liaisons entre les interfaces dépend du degré de détail que nous voulons observer aux interfaces. D'une part, les interfaces doivent représenter un haut niveau d'abstraction de ce qui se passe au sein du composant, d'autre part, ils doivent être assez précises pour que nous puissions être sûrs qu'il n'y aura pas de surprises une fois que le système est déployé. Au-delà du typpage des interfaces, il existe une grande variété d'exigences à fixer dans les interfaces, en commençant par le protocole de messages échangés entre les composants, les exigences sur l'intégrité physique telles comme le temps de réponse, sur l'infrastructure, sur l'encodage des messages, et ainsi de suite. De plus, les communications entre les composants peuvent être faites de plusieurs façons, par exemple avec des messages synchrones ou asynchrones, appels de méthode distants, streaming de données, appels MPI, etc.

Les interfaces peuvent également fournir un haut niveau d'abstraction du comportement des composants. Le but est d'établir un contrat entre les composants qui fournissent seulement l'information nécessaire pour utiliser le composant. L'interface définit également une implémentation de base pour les services fournis et requis par les composants. Ce que vise ce contrat est de permettre aux autres développeurs d'utiliser les composants dans leurs propres projets avec un minimum d'efforts et avec un comportement garanti. De plus, les contrats disent à d'autres développeurs ce qu'ils peuvent developper indépendamment de notre système. Cela correspond à ce que le développement de logiciel a envisagé depuis longtemps: un paradigme dans lequel il est possible d'acheter des composants et les connecter.

Néanmoins, comment ces contrats sont-ils définie? Une simple énumération des services fournis et requis par les composants ne suffit pas parce qu'une définition du comportement est également nécessaire. Par exemple, un composant peut exiger une initialisation avant que ces clients puissent l'interroger et une fermeture des session pour éviter la consommation illimitée de ressources.

1.2 Vers une spécification comportementale des composants

Le moyen plus simple de préciser le comportement c'est avec une description informelle : la documentation. Elles sont généralement bien adaptées pour des comportements simples dans lesquels le concepteur peut prédire l'effet global d'une composition. Lorsque le système grossit, elle devient rapidement inappropriée. Si les spécifications sont informelles, elles finissent avec un comportement incomplèt ou encore pire, les spécifications peuvent être sujettes à différentes interprétations. De plus, comme la spécification est informelle, on ne peut pas aider le développeur avec une assistance de vérification numérique.

Le domaine de la vérification formelle peut être date des premières œuvres de Floyd. Les méthodes formelles utilisent les mathématiques pour préciser et ensuite raisonner sur des systèmes informatiques. Donc ils enlèvent l'ambiguïté, l'incomplétude, et l'incohérence des dessins et des modèles. Pour ces raisons elles ont reçu un intérêt particulier dans la sécurité des systèmes critiques et pourquoi pas dans CBSE.

Selon le type de formalisme, il est possible de vérifier les erreurs de compatibilité en utilisant une large gamme de techniques de vérification telles comme la preuve automatique des théorèmes et le model-checking. Le model-checking a longtemps été utilisé par la communauté. Intel par exemple a adopté le model-checking depuis le fameux bogue sur les Pentium qui a coûté environ US$475M en 1994. Au sein de la communauté des développeurs, le succès de model-checking a été limitée. Généralement le logiciel est si complexe que même des petits systèmes peuvent tomber dans le problème d'explosion d'état [Valmari 98]. Une façon de faire face à cette problème est d'utiliser l'abstraction vers un système plus simple, qui détient la propriété désirée, et puis de vérifier ce modèle simple. Si l'abstraction est une bonne approximation du système d'origine, les propriétés vérifiés dans le modèle (simple) sont également vraies dans le système d'origine.

Quand il s'agit de la spécification des composants, de nombreuses théories classiques peuvent être adaptées pour soutenir la description formelle du comportement. Par exemple, les algèbres de processus, telles que le CCS de Milner [Milner 90, Milner 89]; CSP de Hoare [Hoare 85, Brookes 84] sont utiles car un composant a ses frontières des communications bien définies. La communication externe donne des actions observables, et la communication interne donne le comportement interne.

Les méthodes formelles peuvent répondre à la plupart des questions évoquées ci-dessus, même si l'adoption est encore limitée dans le CBSE. Parmi les principales raisons, nous pouvons mettre en évidence l'écart entre les compétences requises pour les méthodes formelles et celles trouvées dans l'expertise des ingénieurs logiciels. Ce dont nous avons besoin est d'aider le développeur avec des outils qui peuvent masquer la complexité des méthodes formelles.

1.3 Les composants distribués

La sémantique des composants dépend du modèle de composants choisi. Comme il n'existe pas une seule définition de ce qui est un composant, nous pouvons trouver des composants logiciels qui considèrent chaque objet (comme dans le paradigme orienté objets) comme un composant, comme des système à composants où l'ensemble de l'application est un composant. Nous pouvons aussi trouver des éléments qui sont déployés dans une seule machine ou distribués dans des milliers de machines.

Une propriété intéressante de CBSE est que les composants peuvent être utilisés pour définir les unités de distribution et pour déployer des applications sur des milliers de machines. C'est le cas du Grid Component Model, qui cible des composants s'exécutant sur des grilles de calcul. Les grilles de calcul sont des réseaux de machines très hétérogènes qui offrent une alternative de bas coût pour le calcul de haute performance. Ils utilisent des ressources réparties sur l'Internet au lieu d'utiliser des clusters dédiés qui sont trés coûteux.

Lorsque les composants sont distribuées et sont déployées sur des grilles, il est nécessaire que les composants soient faiblement couplés. En d'autres termes, les critères de conception sont détournés vers des systèmes avec un minimum de synchronisations afin de prendre avantage du parallélisme. Un moyen classique de minimiser les synchronisations est d'adopter des communications asynchrones, fournies par le modèle à composants. Par conséquent, les composants distribués présentent un complexe entrelacement d'événements en raison du parallélisme et en raison de l'asynchronisme.

Déduire le résultat de la composition des ces composants est intrinsèquement difficile. Malheureusement, il y a peu ou aucun soutien de langages de spécification pour faire face à ce type de composants. Le développement d'outils et de théories ont mis l'accent sur les services techniques pour le déploiement de composants distribués, mais sont insuffisants pour soutenir le concepteur.

1.4 Contribution

Globalement, cette thèse s'intéresse à la conception et l'analyse de systèmes à base de composants, et à la génération de composants avec une garantie de comportement.

Langage de spécification. La principale contribution est la définition d'un langage de spécification qui permet aux ingénieurs logiciels d'exprimer des synchronisations complexes et, plus tard, de vérifier ces systèmes. Nous proposons un langage de spécification qui apporte les avantages des méthodes formelles de plus proches de son expertise.

Formalisme et modèles de comportement. En plus du langage de spécification, nous travaillons également sur des formalismes et des modèles de comportement qui nous permettent de vérifier les spécifications de composants distribués.

Les composants garantis. Nous proposons de définir des composants qui utilisent ce langage de spécification, de vérifier leur comportement avec le soutien d'un outil, et ensuite de générer des composants avec du code certifié.

1.5 Structure du résumé

Le résumé est structurée comme suit:

Le Chapitre 2 présente un formalisme paramétrique qui permet d'exprimer des modèles comportementaux de composants distribués. Ce formalisme nous donne une représentation compacte du système, ainsi que la possibilité de de vérifier les modèles avec des techniques de model-checking.

Le Chapitre 3 présente la base de la thèse : un langage de spécification adapté aux composants distribués. Le langage de spécification est doté de suffisamment formalise pour permettre une approche constructive. Concrètement, la génération de modèles de comportement basé sur le formalisme, et la génération de squelettes de code avec le code de contrôle des composants.

Le Chapitre 4 résume les principales contributions de nos travaux.

2
Modèle Théorique

Nous donnons la définition formelle de notre langage intermédiaire que nous appelons *parameterized Networks of Synchronised Automata (pNets)*. pNets est un formalisme générique ayant pour ayant but de spécifier et de synchroniser le comportement d'un ensemble d'automates. Nous avons construit ce modèle avec deux objectifs: donner une base au modèle de génération et construire un modèle qui serait plus proche de la machine et qui servirait de format interne polyvalent pour différents outils.

Le produit synchronisé présenté par Arnold & Nivat [Arnold 94] est à la fois simple et puissant, parce qu'il vise directement le cœur du problème. Un des principaux avantages de son haut niveau d'abstraction est que presque tous les opérateurs parallèles rencontrés dans la littérature concernant les algèbres de processus deviennent des cas particuliers d'un concept très général: les vecteurs de synchronisation. Nous synchronisons la structure des vecteurs comme faisant partie d'un *réseau de synchronisation*. Le réseau permet des reconfigurations dynamique entre les différents ensembles avec un LTS *transducteur*. Notre définition du produit synchronisé est sémantiquement équivalent à celui donné par Arnold & Nivat.

De plus, nous utilisons l'approche proposée par Lin[Lin 96] pour rajouter des paramètres dans ses événements de communications de les systèmes de transitions et les réseaux de synchronisation. Ces événements peuvent avoir des conditions sur leurs paramètres. Nos agents peuvent aussi être paramétrés pour encoder des ensembles d'agents équivalents. Cela nous conduit à la définition des pNets. Ils correspondent à la façon dont les développeurs programment leur systèmes : la structure du système est paramétrée et décrite dans un mode fini (le code est fini), mais une instance est déterminée pour chaque exécution, ou même varie dynamiquement.

2.1 Systèmes de Transitions Étiquetés Paramétrés

Definition 1 (Actions paramétrées)
Soit V un ensemble de noms, $\mathcal{L}_{A,V}$ une algèbre de termes construite sur V, y compris l'action constante τ.

Nous appelons:

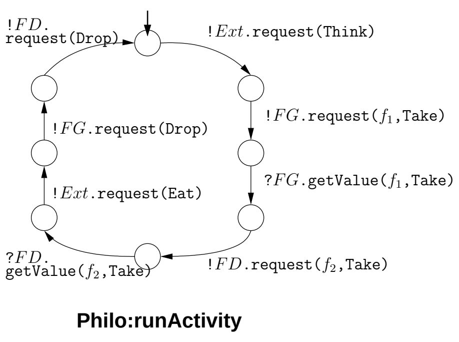
- *$v \in V$ un paramètre,*
- *$a \in \mathcal{L}_{A,V}$ une action paramétrée,*
- *$\mathcal{B}_{A,V}$ l'ensemble des expressions booléennes (gardes) sur $\mathcal{L}_{A,V}$.*

Exemple

Dans le *Value-passing CCS* de Milner [Milner 89] l'algèbre d'action a pour constructeurs "`tau`", "`a`" pour les actions d'entrée, "`'a`" pour les actions de sortie, "`a(x)`" pour les action paramétrées. Puis, "`'out(3)`" est un terme d'action de sortie fermé, "`a(x,y)`" est un terme d'action d'entré ouvert avec des paramètres `x` et `y`, et "`x+y=3`" comme une garde.

DEFINITION 2 (SYSTÈMES DE TRANSITIONS ÉTIQUETÉS PARAMÉTRÉS)
Un Système de Transitions Étiquetés Paramétrés (pLTS) est un tuple $(V, S, s_0, L, \rightarrow)$, où:

- *V est un ensemble fini de paramètres, à partir duquel on construit l'algèbre de terme $\mathcal{L}_{A,V}$,*
- *$pA_G \subseteq \mathcal{L}_{A,V}$ est la Sorte*
- *S est un ensemble d'etats; chaque etat $s \in S$ est associé à un ensemble indexé fini de variables libres $fv(s) = \tilde{x}_{J_s} \subseteq V$,*
- *$s_0 \in S$ est l'état initial,*
- *L est l'ensemble des étiquettes, $\rightarrow$ la relation de transition $\rightarrow \subset S \times L \times S$*
- *Les étiquettes sont de la forme $l = \langle \alpha,\ e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}} \rangle$ telle que si $s \xrightarrow{l} s'$, alors:*
  - *α est une action paramétrée.*
    - *l'action définit les variables d'entrée $iv(\alpha)$, possiblement en définissant de nouvelles variables $iv(\alpha) \subseteq V$;*
    - *l'action définit les variables de sortie $oe(\alpha)$ en utilisant des expressions d'actions.*
  - *$e_b \in \mathcal{B}_{A,V}$ est la garde* optionnelle,
  - *les variables $\tilde{x}_{J_{s'}}$ sont affectés au cours de la transition par les $\tilde{e}_{J_{s'}}$ optionnels avec les contraintes :*
    - *$fv(oe(\alpha)) \subseteq iv(\alpha) \cup \tilde{x}_{J_s}$*
    - *$fv(e_b) \cup fv(\tilde{e}_{J_{s'}}) \subseteq iv(\alpha) \cup \tilde{x}_{J_s} \cup \tilde{x}_{J_{s'}}$*



PhiloRunActivityLTS = $\langle V, S, s_0, L, \rightarrow \rangle$
with:

$V = \{f_1, f_2\}$

$S = \{s_i\}$, `i ∈ [0:7]`

L=\{ `!`*Ext*`.request(Think)`, `!`*Ext*`.request(Eat)`,
`!`*FG*`.request(`f_1`,Take)`, `?`*FG*`.getValue(`f_1`,Take)`,
`!`*FD*`.request(`f_2`,Take)`, `?`*FD*`.getValue(`f_2`,Take)`,
`!`*FG*`.request(Drop)`, `!`*FD*`.request(Drop)` \}

$\rightarrow$ such that:

s_0 : `!`*Ext*`.request(Think)` $\rightarrow s_1$,
s_1 : `!`*FG*`.request(`f_1`,Take)` $\rightarrow s_2$
...

Figure 2.1: *Exemple d'un pLTS*

Exemple

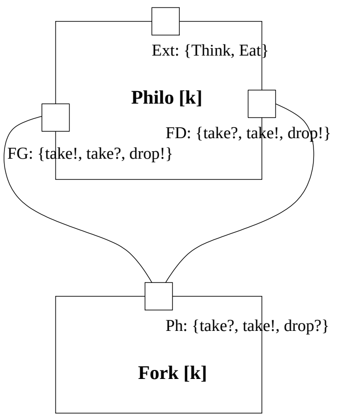
La Figure 2.1 est basée sur une implémentation du problème de philosophes dans ProActive. Elle représente le pLTS pour le comportement d'un composant Philo. L'alphabet d'action utilisé ici reflète le schéma de communication : chaque requête distante a la forme "`!`*dest*`.request(`$f, \mathcal{M}(\tilde{arg})$`)`", où *dest* est la référence distante, $\mathcal{M}$ est le nom de la méthode, avec des paramètres $\tilde{arg}$ et f est une référence de futur. Plus précisément, f est l'identificateur du proxy du futur. Les requêtes qui ne nécessitent pas une réponse n'utilise pas le proxy.

2.2 Réseaux de Synchronisation Paramétrés

DEFINITION 3 (RÉSEAUX PARAMÉTRÉS)
*Un réseau paramétré (***pNet***) est un tuple* $\langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T\rangle$*, où:*

- V *est un ensemble de paramètres,*
- $pA_G \subset \mathcal{L}_{A,V}$ *est un ensemble d'actions externes (paramétrées),*
- J *est un ensemble fini de trous, chaque trou* j *est associé avec (au plus) un paramétre* $p_j \in V$ *et avec une sorte* $O_j \subset \mathcal{L}_{A,V}$.
- T *est le (LTS) transducteur* (S_T, s_{0T}, L_T, T_T)*, où les étiquettes de transition* ($\overrightarrow{v} \in L_T$) *sont les vecteurs de synchronisation de la forme :* $\overrightarrow{v} = \langle a_g, \{\alpha_t\}_{i\in I, t\in B_i}\rangle$ *telles que:*
  - $I \subseteq J$
  - $B_i \subseteq \mathcal{D}om(p_i)$
  - $\alpha_i \in O_i$
  - $fv(\alpha_i) \subseteq V$

Un pNet *statique* a un état unique, mais il a des variables d'état qui codent certaines notions de mémoire interne qui peuvent influencer la synchronisation. Les pNets statiques ont la bonne propriété d'être facilement représentées graphiquement.


Ext: {Think, Eat}
Philo [k]
FD: {take?, take!, drop!}
FG: {take!, take?, drop!}
Ph: {take?, take!, drop?}
Fork [k]

PhiloNet = $\langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T\rangle$ with:
$V = \{k, f_1, f_2\}$
pA_G = {`Think(k), Eat(k), !TakeG(k), !TakeD(k), ?TakeG(k), ?TakeD(k), DropG!k, DropD!k`}
J = {`Philo, Fork`}
$p_{Philo} = k, p_{Fork} = k$
O_{Philo} = {`!`*Ext*`.request(Think), !`*Ext*`.request(Eat), !`*FG*`.request(`f_1`,Take), !`*FD*`.request(`f_2`,Take), ?`*FG*`.getValue(`f_1`,Take), ?`*FD*`.getValue(`f_2`,Take), !`*FG*`.request(Drop), !`*FD*`.request(Drop)`}
O_{Fork} = {`?`*Ph*`.request(`f_1`,Take), ?`*Ph*`.request(`f_2`,Take), !`*Ph*`.getValue(`f_1`,Take), !`*Ph*`.getValue(`f_2`,Take), ?`*Ph*`.request(Drop)`}

Ce pNet est statique, T a un état unique, et transitions avec les étiquettes :
L_T = {
⟨`Think(k), !Philo[k].`*Ext*`.request(Think)`⟩
⟨`Eat(k), !Philo[k].`*Ext*`.request(Eat)`⟩
⟨`!TakeG(k), !Philo[k].`*FG*`.request(`f_1`,Take), ?Fork[k].`*Ph*`.request(`f_1`,Take)`⟩
⟨`!TakeD(k), !Philo[k].`*FD*`.request(`f_2`,Take), ?Fork[k+1].`*Ph*`.request(`f_2`,Take)`⟩
⟨`?TakeG(k), ?Philo[k].`*FG*`.getValue(`f_1`,Take), !Fork[k].`*Ph*`.getValue(`f_1`,Take)`⟩
⟨`?TakeD(k), ?Philo[k].`*FD*`.getValue(`f_2`,Take), !Fork[k+1].`*Ph*`.getValue(`f_2`,Take)`⟩
⟨`DropG(k), !Philo[k].`*FG*`.request(Drop), ?Fork[k].`*Ph*`.request(Drop)`⟩
⟨`DropD(k), !Philo[k].`*FD*`.request(Drop), ?Fork[k+1].`*Ph*`.request(Drop)`⟩ }

Figure 2.2: *Exemple d'un pNet*

Exemple

Le dessin de la Figure 2.2 montre un pNet (statique) représentant le problème classique des philosophes, avec 2 trous paramétrés (indexés par la même variable k) pour les philosophes et les fourchettes. Sur le côté droit sont représentés les éléments correspondants au pNet formel, dans lequel nous énumérons explicitement les sortes des trous (O_{philo} et O_{Fork}), et les vecteurs de synchronisation sont paramétrés sur l'index k et les ids de futurs f_1 et f_2.

Les sortes de nos structures paramétrées sont des ensembles d'actions paramétrées.

DEFINITION 4 (SORTE PARAMÉTRÉE)

- *La sorte d'un pLTS:* $Sort(V, S, s_0, L, \to) = \left\{\alpha \mid \exists l \in L.\ l = \langle \alpha,\ e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}} \rangle\right\} = pA_G$
- *La sorte d'un pNet:* $Sort\langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T\rangle = pA_G$

2.3 Conclusion

Ce type de modèle sémantique est largement utilisé dans des ensembles d'outils d'analyse et de vérification, car il fournit un format intermédiaire compact et bien défini. En ce qui concerne les systèmes concurrents distribués, les modèles intermédiaires font souvent des hypothèses fortes sur le type de synchronisation sur des mécanismes de communication. Notre choix est d'avoir des primitives de bas niveau (LTS + vecteurs de synchronisation) qui sont en mesure de représenter de nombreux mécanismes possibles.

3 Langage de Spécification

Les composants distribués ont tendance à former de grandes unités de composition, et sont souvent faiblement couplés. Par la suite, nous présentons un langage de spécification sous forme d'une extension d'un sous-ensemble de Java pour la spécification de ces composants. Le langage décrit à la fois l'architecture et le comportement, et est suffisamment riche de sorte que nous sommes capable de :

- d'une part *vérifier l'exactitude du système* (Chapitre 6) : nous construisons un modèle comportamental qui peut être vérifié par du model-checking;
- d'autre part *d'assurer la sécurité des composants* (Chapitre 8) : nous voulons générer le code de contrôle des composants qui garanti le respect de la spécification.

Nous avons opté pour un langage proche de Java pour plusieurs raisons: (i) il est proche de l'expertise des ingénieurs, en utilisant une syntaxe connue telles que les appels de méthode et de classes utilisateur; (ii) il permet d'embarquer une partie de la spécification dans des squelettes de code; (iii) il emploie les même types de données que l'implémentation, en garantissant que les opérations sur les données sont directement utiles.

Architecture. La définition de l'architecture est lié à des ADL classiques. Nous allons fournir un syntaxe "à la" Java pour la définition de l'architecture, qui est utilisé pour définir le type du composant et sa structure. En d'autres termes, nous allons définir ses sous-composants et ses connexions internes.

Décomposition du comportement en services. Nous proposons de définir le comportement du composant comme une spécification boîte noire. La spécification est donnée par un ensemble de services, chacun étant une activité indépendante. Pour chaque service, nous définissons une politique et puis nous détaillons son comportement.

La première partie d'un service est appelé la *politique de service*; elle définit la manière dont un composant sélectionne les requêtes en fonction de son état interne, et tout comportement que le composant déclenche par ses propres moyens. La deuxième partie du service précise ce que chaque service exposés à la politique de service fait; elle s'appelle méthode de service.

Futurs. L'utilisation des futurs transparents dans le langage de spécification apporte les mêmes avantages que dans le langage de programmation : le développeur n'a pas besoin de se demander si une variable peut contenir un futur, ou plus précisément, il n'existe aucun mécanisme de synchronisation explicite pour des variables qui contiennent parfois un futur. Donc, les futurs transparents étendent la réutilisation des spécifications car ils peuvent être utilisés dans plusieurs contextes, où les valeurs sont calculées localement ou à distance.

Types de Données et d'Abstractions. Les types de données utilisées dans JDC sont des classes Java standard. De cette façon, les squelettes de code obtenu par nos outils de génération seront directement utilisables. D'une part les types de données arbitraires ont souvent de grand domaines (peut-être infini) qui ne peuvent pas être vérifiées directement. D'autre part, le type de propriétés comportementales que nous cherchons ont besoin seulement d'une abstraction de ces données.

Par conséquent, pour faire une vérification, la spécification inclu aussi une abstraction des types utilisateur. De cette façon, nous sommes capables de générer une spécification plus simple qui ne contient que des variables avec domaines "simples".

3.1 Spécification de l'Architecture

Dans les prochaines sections, nous présentons des éléments de la syntaxe abstraite et concrete du langage JDC. Chaque boîte définit une partie de la syntaxe du langage, en utilisant les conventions suivantes :

- mots-clés en gras (e.g. **component**);
- symboles terminal écrit entre guillemets simples (e.g. '{');
- symboles non terminal en police monospace (e.g. `Services`);
- parties optionnelles avec des expressions crochets (e.g. [`expr`]);
- choix avec | (e.g. `expr1` | `expr2`);
- enchaînements de zéro (resp. un) ou plusieurs expressions avec * et $^+$ (e.g. `expr`*, `expr`$^+$);
- identifiants comme $\underline{\text{id}}$.

3.1.1 Définition d'un Composant

La définition d'un type de composant comprend ses interfaces externes et une spécification de son comportement. Le comportement est donné soit par une spécification boîte noire sous forme d'un ensemble de services, ou soit par une composition de composants, ou par les deux. Cela est illustré dans la Figure 3.1.

Chaque interface dans un composant a un rôle (serveur ou client), un type (une interface Java comme dans la plupart des IDLs), et un nom.

`Component` →	**component** id '{'	*≪définition de composant≫*
	external interfaces	
	`Interface`*	*≪ensemble d'interfaces≫*
	[`Services`]	*≪spécification boîte noire≫*
	[`Architecture`]	*≪description du contenu≫*
	'}'	
`Interface` →	**server** | **client**	*≪rôle de l'interface≫*
	interface `InterfaceType` id ';'	*≪type et nom≫*

Figure 3.1: *Syntaxe pour la définition d'un* composant

3.1.2 Composition des Composants

La composition des composants se fait à l'intérieur de l'architecture, voir la figure 3.2. Elle expose le contenu d'un composant avec le contenu de ses sous-composants, de ses interfaces internes, et de ses liaisons internes. Les sous-composants sont nommés et typés. Le type du composant est donné soit par une définition externe, ou par une définition en ligne (*inline*). Les liaisons connectent soit deux interfaces des composants internes ou soit des interfaces de sous-composants avec le composant.

`Architecture` →	**architecture**	
	contents	
	`Subcomponent`*	*≪ensemble de sous-composants≫*
	internal interfaces	
	`Interface`*	*≪ensemble d'interfaces≫*
	bindings	
	`Binding`*	*≪ensemble de liaisons≫*
`Subcomponent` →	**component** `ComponentType` id ';'	*≪sous-composant≫*
	| `Component`	*≪définition inline≫*
`ComponentType` →	id	*≪référence à un type≫*
`Binding` →	**bind** '(' `SourceItf` ','	
	`TargetItf` ')' ';'	*≪liaison d'un paire d'interfaces≫*

Figure 3.2: *Syntaxe pour la définition d'une* architecture

Exemple

L'exemple CoCoME [Rausch 08] a été implementé en utilisant GCM / ProActive, et est détaillée dans l'annexe A. Il s'agit d'un système de point de vente dans lequel la caisse traite une vente. La caisse et ses périphériques sont implementés en tant que composants. Dans la figure 3.3, nous montrons un extrait de l'exemple, où un composant appellé Application a un seul périphérique appelé Scanner; ce dernier est contrôlé par un composant.

3.2 Spécification de Comportement

Nous proposons de spécifier directement le comportement acceptable par les interfaces, on appelle cela une spécification boîte noire du comportement d'un composant.

```
component CashDesk {
 external interfaces
   server interface ApplicationIf appIf;
   client interface ScannerIf scannerIf;
   // ... external interfaces
 architecture
   contents
     component Application application;
     component Scanner scanner;
   internal interfaces
      server interface ApplicationIf appIf;
      // ... internal interfaces
   bindings
     bind(this.appIf, application.appIf);
     // ... bindings
}
```

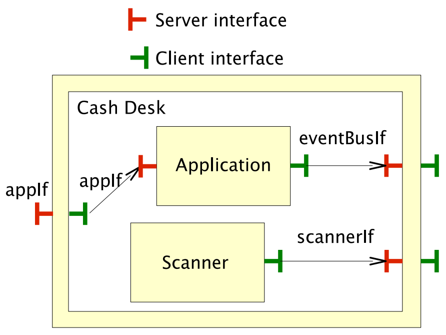


Figure 3.3: *Exemple d'une spécification* d'architecture

La concurrence en JDC est spécifiée par un ensemble de services dans le bloc *Services* (voir la figure 3.4). Chaque *service* définit un processus séquentiel avec son propre ensemble de variables locales. Un processus séquentiel est composé de la politique de service (*service policy*) qui définit le protocole du service, et d'un ensemble de *méthodes de service* qui donnent les détails du comportement des méthodes exportées par le composant.

`Services` →	**services**	
	`Service`$^+$	≪*un ou plusieurs services concurrents*≫
`Service` →	**service** '{'	
	`LocalVariableDecl`*	≪*variables du composant*≫
	policy '{' `Policy` '}'	≪*politique de service*≫
	`ServiceMethodDecl`*	≪*méthodes de service*≫
	`LocalMethodDecl`*	≪*méthodes locales*≫
	'}'	

Figure 3.4: *Syntaxe pour la définition d'un* service

La spécification du comportement du composant est une abstraction du flot de contrôle, du flot de données, et de l'accès aux données.

3.2.1 Politique de Service

La politique de service définit la manière dont les requêtes sont choisies dans la queue en fonction de l'état interne du composant, et tout comportement déclenché à l'intérieur du composant.

Le comportement du composant peut être considéré comme une machine qui ne cesse de faire certains travaux. Le *comportement réactif* définit quel type de méthode sélectionner et dans quel ordre. Un *comportement actif* désigne le comportement spontané, c'est-à-dire, un travail qui est fait sans être demandé.

Policy →	BasicPolicy [';' PermPolicy]	*≪définition d'une politique≫*
BasicPolicy →	ServeMode '(' [Filter] ')'	*≪service réactif≫*
	| MethodCall	*≪service actif≫*
	| BasicPolicy ';' BasicPolicy	*≪séquence≫*
	| BasicPolicy '|' BasicPolicy	*≪choix≫*
	| BasicPolicy *'n'*	*≪répétition≫*
PermPolicy →	BasicPolicy '*'	*≪répétition infinie≫*
ServeMode →	**serveOldest** | **serveYoungest**	*≪primitif d'accès à la queue≫*
Filter →	ItfName	*≪toute méthode dans cette interface≫*
	| ItfName '.' MethodName	*≪cette méthode≫*
	| Filter ',' Filter	*≪une liste de filtres≫*
MethodCall →	ItfName '.' MethodName '(' [Expr] ')'	*≪appel de méthode distante≫*
	| MethodName '(' [Expr] ')'	*≪appel de méthode locale≫*

Figure 3.5: *Syntaxe pour la définition d'une* politique de service

3.2.2 Méthodes de Service

Une méthode de service est une abstraction d'un service exporté par un composant. Elle est définie par un sous-ensemble de déclarations Java dans lesquelles il n'y a pas d'exception et de concurrence. Cela inclut les flot de données entre les paramètres d'entrée et les résultats de la méthode, ainsi que la communication avec les services requis. La méthode de service a accès aux variables du composant mais elle ne peut pas accéder à la queue du composant.

3.3 Specifying Abstractions

Une classe est un vecteur $C =< \overrightarrow{m}, \overrightarrow{f} >$, où $\overrightarrow{m} = \{m^i(\overrightarrow{a}) : \tau^i\}$ sont les méthodes de C; $\overrightarrow{a} = \{a^j : \tau^j\}$ sont les arguments de la méthode, et $\overrightarrow{f} = \{f^k : \tau^k\}$ sont les champs.

Une abstraction de C est une classe $C_{\mathcal{A}} =< \overrightarrow{m_{\mathcal{A}}}, \overrightarrow{f_{\mathcal{A}}} >$, où chaque méthode publique $m(\{a^j : \tau^j\} : \tau)$ de C a une ou plusieurs méthode abstraite $m_{\mathcal{A}}(\overrightarrow{a_{\mathcal{A}}}) : \{\tau_{\mathcal{A}}\}$ avec $\overrightarrow{a_{\mathcal{A}}} = \{a^j_{\mathcal{A}} : \tau^j_{\mathcal{A}}\}$ les abstractions des arguments. Les domaines des arguments sont des ensembles de valeurs dans les abstractions de classes τ^i, et le résultat est une valeur abstraite dans l'abstraction de la classe τ.

3.3.1 La définition et l'utilisation d'abstractions

Une abstraction en JDC est similaire à une classe Java, avec des extensions pour traiter le non-déterminisme et l'abstraction de données. Une notion importante est que nous avons la possibilité de faire différentes abstractions pour les différentes variables du même type.

3.4 Géneration de Modèles Comportamentaux

Nous développons des modèles comportamentaux basé sur le formalisme pNets (voir le chapitre 4) qui peuvent être générés à partir de spécifications JDC. Le modèle de génération

`Abstraction` →	**abstraction** id **of** id '{'	≪*abstraction de types*≫
	`TypeDecl`*	≪*déclarations de type*≫
	`Field`*	≪*variables locales*≫
	`Constructor`*	≪*constructeurs abstraits*≫
	`Operator`*	≪*opérateurs abstraits*≫
	'}'	
`Constructor` →	`Type` '(' `args` ')'	≪*sign. du constructeur concret*≫
	[**abstracted as** `Type` '(' `args` ')'	
	'{' `Body` '}']	≪*version abstraite*≫
`Operator` →	`Type` id '(' `args` ')'	≪*sign. du operateur concret*≫
	[**abstracted as** `Type` id '(' `args` ')'	
	'{' `Body` '}']	≪*version abstraite*≫
`Field` →	`Type` id	≪*type & nom de la variable*≫
	[**abstracted as** `Type`]	≪*mapping local d'un type*≫

Figure 3.6: *Syntaxe pour la définition d'un* type abstrait

repose sur deux parties:

Tout d'abord, celui qui construit le comportement de contrôle des composants. Dans cette phase, la plupart des informations qui sont nécessaires proviennent de l'analyse de la structure des composants, et du flot de futurs. Pour le dernier, nous développons des modèles comportamentaux qui nous permettent de vérifier les problèmes liés à la synchronisation de futurs.

En plus, nous proposons un modèle pour la partie fonctionnelle des composants. Cela exige de faire une analyse statique sur la spécification afin de trouver le graphe d'appel de méthodes. Pour JDC, il est plus facile à faire que dans le cas de Java (non structuré) parce que pour les objets actifs nous avons des problèmes pour identifier précisément les appels de méthode distante. Le graphe d'appel de méthodes est ensuite utilisé dans le modèle de génération pour créer les réseaux de LTS.

En supposant que le composant C dispose de 2 sous-composants A et B comme dans la figure 3.7, cela créera un pNet comme dans la figure 3.8.

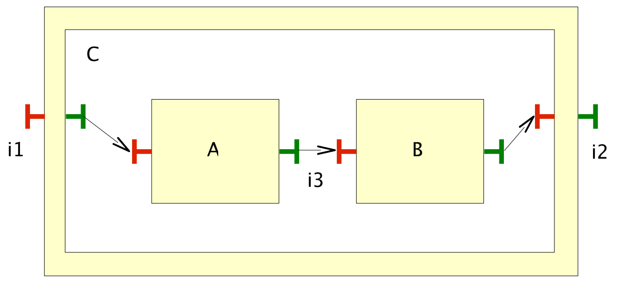


Figure 3.7: *Un composant composite*

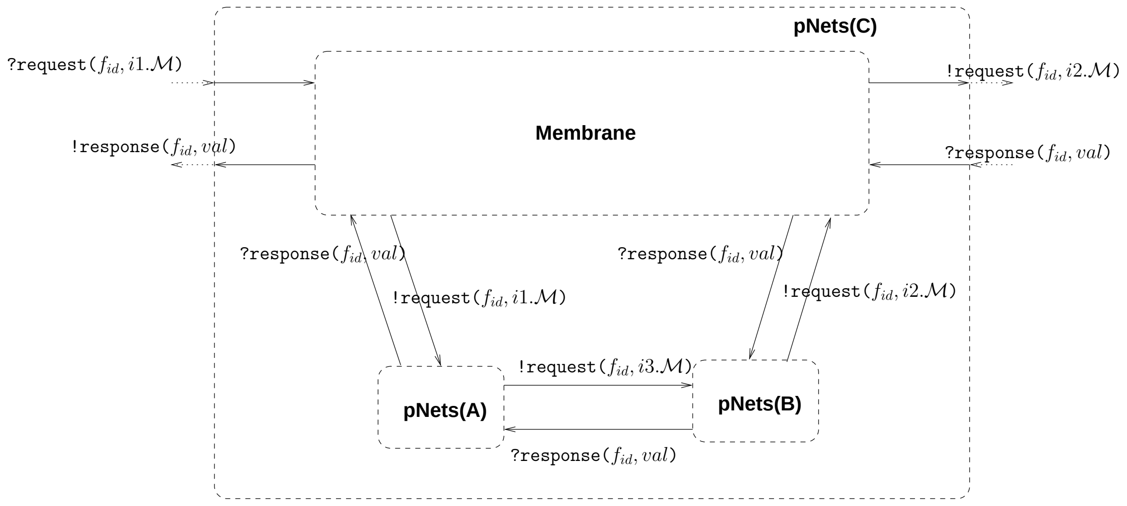

pNets(C)
?request($f_{id}, i1.\mathcal{M}$)
!response(f_{id}, val)
Membrane
!request($f_{id}, i2.\mathcal{M}$)
?response(f_{id}, val)
?response(f_{id}, val)
!request($f_{id}, i1.\mathcal{M}$)
?response(f_{id}, val)
!request($f_{id}, i2.\mathcal{M}$)
!request($f_{id}, i3.\mathcal{M}$)
pNets(A)
pNets(B)
?response(f_{id}, val)

Figure 3.8: *Modèle comportamentaux d'un composant composite*

3.5 Conclusion

Nous avons créé le langage de spécification avec deux stratégies opposées. D'une part, nous définissons le langage a un niveau d'abstraction beaucoup plus élevé que celui d'un langage de programmation, d'autre part, la partie "données" du langage est celle habituellement trouvée dans un langage de programmation. La première permet la génération de code de contrôle garanti parce que elle est simple, La dernière permet que la partie données du code généré soit directement utile étant très proche du langage de programmation.

4 Conclusions

Cette thèse a cherché dès le début à soutenir le développement de composants. Notre travail a porté à la fois sur la modélisation et la spécification des composants distribués. L'objectif de la thèse a été de réduire l'écart entre l'implementation d'un composant et sa spécification.

L'analyse d'un système peut être obtenus grâce à des modèles comportementaux qui fournissent une représentation abstraite. Par contre, les modèles comportementaux ont tendance à être de trop bas niveau pour être utilisé comme un système de spécification. Donc, afin de spécifier et d'analyser de composants distribués, ce travail a envisagé un cadre formel adapté à l'expertise de ingénieurs en logiciel. Nous avons fourni des modèles comportementaux adaptés à la vérification des systèmes de composants distribués, et avons donné un langage expressif qui permet de définir leur comportement.

4.1 Contributions

Nous allons maintenant mettre en évidence les principales contributions de cette thèse.

Formalisme Hiérarchique. Nous avons formalisé un modèle comportemental hiérarchique. La première partie de la thèse a présenté un formalisme appelé pNets. Ce formalisme est particulièrement adapté pour la modélisation de composants, car il décrit le modèle comme une hiérarchie de processus communicants, et fournit une représentation symbolique du système. Une autre contribution importante du formalisme est qu'il ne fait pas de fortes hypothèses sur le type de synchronisation utilisé. En plus, il gère aussi les données dans le processus.

Langage de Spécification de haut niveau. Nous avons fourni un langage de spécification de haut niveau pour la spécification de composants distribués adapté à l'ingénieur en logiciel. Le formalisme pNets est bien adapté aux modèles comportementaux que nous voulons exprimer. Toutefois, il est de trop bas niveau pour être directement utilisés par nos utilisateurs. Alors, nous avons défini un langage assez riche pour capturer la communication, la synchronisation, le contrôle des flot et le flot de données. Nous avons conçu le langage assez riche pour construire des modèles comportementaux, mais doté d'une syntaxe familière à nos utilisateurs.

Intégrer l'architecture et le comportement. Nous avons intégré l'architecture et le comportement en un seul langage de spécification. Bien entendu, la définition de l'architecture traite la structure des composants comme dans la plupart des ADL, mais offre aussi des primitives si jamais nous avons besoin de faire référence à la structure dans le comportement. La définition du comportement, d'autre part, prend soin de la composante de contrôle et du flot de données.

Composants comme des Services. Nous avons remarqué que les composants distribués avaient une partie de contrôle qui orchestrent les services offerts à l'environnement. Cela a été un bon point de départ pour définir le comportement. Par conséquent, nous avons défini la *politique de service* qui nous donne une définition approximative de ce que le composant fournit à l'environnement, et puis nous nous concentrons sur une définition plus détaillée de chacun de ces services.

Simplification de la Spécification Nous avons simplifié la spécification du comportement en spécifiant l'activité du composant au lieu de ses événements. Nous précisons ce que le composant fait et nous utilisons des techniques d'analyse statique afin de déduire son comportement exact (tous les événements effectués par le composant). Donc, nous sommes capables de définir le comportement de composants distribués d'une manière simple.

Générer des modèles comportementaux vérifiables. Nous avons défini et généré des modèles comportementaux vérifiable. L'analyse des spécifications donne des informations sur l'architecture, des invocations de méthode distante, des flot de futurs et des synchronisations. Cela est suffisant pour générer automatiquement des modèles comportementaux sur la base de notre formalisme pNets; par ailleurs, le comportement des modèles peut être vérifié par des techniques de model-checking.

Une représentation statique et compositionnelle du système. Nous avons défini des modèles comportementaux qui sont une représentation statique du système et peuvent être construit de façon compositionnelle. Une forte contribution est de fournir une représentation statique de futurs, en particulier lorsque les futurs peuvent être transmis à d'autres composants d'une manière non bloquante. Nous proposons une abstractions pour le futurs, puis des représentations statiques pour les différents moyens par lesquels les futurs peuvent être transmises; de plus, nous faisons cela de façon compositionnelle.

Réduire l'écart entre la spécification et l'implémentation. Nous avons fourni un mécanisme qui permet de générer des modèles comportementaux et des squelettes de code dans la même spécification. Le langage de spécification est suffisant pour générer un squelette du composant avec des garanties solides sur son comportement.

Utiliser des classes utilisateur dans la spécification. Nous avons inclus des classes utilisateur et des versions abstraites de ces classes dans le langage de spécification. Cela garantit que le code généré utilise du code correct pour les données. De plus, on a la garanti que la génération de modèle comportementaux utilise des données compatibles avec notre formalisme pNets.

Part II

Thesis

1 Introduction

Contents

1.1 Motivation

Software engineering is still an immature discipline. The problem seems to be that we do not know how develop software in a consistent way. It is impressive to see how often software is plagued with bugs and non-deterministic behaviour.

The question that arises is, why can engineers build rock-solid buildings but fail when it comes to software? A key aspect is that in most areas it is more or less easy, or at least well known, how to design a system in such a way people can independently build parts of the system, and later assemble the final product. Moreover, the final product does not emerge from scratch but from consolidated previous solutions. In computer science, on the contrary, engineers tend to rebuild the same functionality over and over, which is error-prone and costly.

So, why is it that we cannot transfer years of consolidated engineering expertise towards software life-cycle? The problem with software is that it is very abstract, so it is difficult to predict its behaviour and required resources. At a first glance, however, software looks like a mathematical model of ones and zeros, but reality is quite different. Software development has so many axis that it ends up relying on the creativity of software engineers. Since every engineer can think of brilliant, though completely opposite solutions to the same problem, fitting different pieces of software to work together becomes a real challenge.

To reuse software, it is crucial to imagine a software as a collection of modules. However, one must also consider the influences of the environment, hence it is useless to think of a software module that works absolutely decoupled from its/the environment. Moreover, the environment may as well change over time. There are two major ideas here: (i) we need to define, in an abstract way, what is expected from the environment and what is provided to the environment; and (ii) we need to conceive software in such a way that the assumptions made on the environment are the only ones needed to build the software. This would allow us to build software that works in any context that matches our assumptions.

There is a great example that shows us that the assumptions about the environment are often wrong. The European Space Agency (ESA) launched a rocket in 1996 that exploded due to a software bug [Ben-Ari 01] in the inertial navigation system (INS). The module had been previously used in Ariane 4 successfully, though when it was used in Ariane 5 it crashed. The input specifications of Ariane 5 were different from those in Ariane 4, which caused an overflow when converting a 64-bit number to a 16-bit number.

Nobody noticed this until the rocket exploded.

Software engineers have been addressing similar targets since the very beginning of software development. At first, they were called procedure calls and libraries, on the one hand, and formal or semi-formal specifications on the other hand. In fact these are foundations that gave rise to object-oriented programming in the 60s and UML (for Unified Modeling Language) [OMG 04] in the 90s.

Nowadays we are dealing with much larger projects than we used to. If we take the development of operating systems for example, we can see that the growth in the number of source code lines is huge. In Figure 1.1 we show some approximate values [Wikipedia 08] on the number of code lines needed to create different versions of Microsoft Windows. [*]

Year	*Operating System*	*Source lines of code*	*Estimated Number of Bugs*
1993	Windows NT 3.1	6 million	120 thousand
1994	Windows NT 3.5	10 million	200 thousand
1996	Windows NT 4.0	16 million	320 thousand
2000	Windows 2000	29 million	580 thousand
2002	Windows XP	40 million	800 thousand
2007	Windows Vista	50 million	1 million

Figure 1.1: *Source code lines for different Microsoft Windows versions*

Even if we do not suppose that the ratio of bugs is constant, what is that clear is that when dealing with projects of 50 million code lines quality in the software development is a major asset.

The only way to build high quality software is through a sound specification of the system. Moreover, to deal with projects of this size one must seek for compositional reasoning and for some kind of automatic software verification that gives us guarantees on the behavioural properties of our program.

[*]The estimation of the number of bugs are just illustrative. They are based on Carnegie Mellon University's CyLab Sustainable Computing Consortium estimation of 20 to 30 bugs every 1,000 lines of code.

1.1.1 Component Based Software Engineering

Recently there has been a major interest in paradigms that bring a better structuring of software, among them, Component Based Software Engineering (CBSE) [Szyperski 02]. What differentiates CBSE is composition and the stress on software reuse. Composition here means building more complex pieces of software from other software. These pieces of software are called *components* because they can be composed together, yielding a new, more complex, component.

A common problem in CBSE is that there is no concensus of what a component really is. Various definitions have been proposed in the literature, though not a single one is considered to be *the* definition of a software component. However, among the most accepted ones, we quote [Szyperski 02]:

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

What we can extract from this definition is that components establish contracts with the environment, and that components are somehow independent modules that can be deployed by third-parties. The contract with the environment sets well-defined frontiers that we call *interfaces*. A key difference with the object-oriented paradigm is that in CBSE we not only define *provided* interfaces, but also *required* interfaces. Each component has a set of interfaces, and components can be composed and assembled by applying composition rules on interfaces. When two components are bound there must be a match of the services required by one component and those provided by the other, at least concerning the interfaces we want to connect. The nesting of components takes place by wrapping components and delegating services coming in and going out the component. The "wrapped" components are called *subcomponents*, and the "wrapper" the *parent* component.

Defining the compatibility rules to bind interfaces depends on the degree of detail we want to observe at the interfaces. On one hand, interfaces must represent a high-level abstraction of what happens within the component, on the other hand they must be precise enough so that we can be sure that there will not be surprises once the system is deployed. Beyond classical interface typing, there is a large variety of requirements to set in the interfaces, starting from the protocol of messages exchanged between components, to physical requirements such as response time, infrastructure dependency, encoding of messages, and so on. Moveover, communication between components can be of many sorts, for example synchronous or asynchronous messages, method calls, data-streaming, MPI calls, and so on.

Interfaces can also provide a high-level abstraction of how components behave. This aims to establish a contract between components that provides just the information required in order to use the component. At the same time, the interface also defines an implementation guideline on the services provided and required by components. What this contract seeks is to allow third-parties to use components in their own projects with minimal effort and with a guaranteed behaviour. Moreover, the contracts say what another software developer should implement independently of our own system. This matches what software development has longed for, a compositional paradigm where it is even possible to buy components and plug them into our systems.

Nevertheless, how are these contracts/interfaces defined? A simple enumeration of the services provided and required by components is not enough as a definition of behaviour is also necessary. For example, a component may require some initialisation before allowing clients to query some data; clients must close their connection or the system will eventually run out of resources.

1.1.2 Towards a Specification of the Component Behaviour

The simplest way to specify component behaviour is by an informal documentation. They are usually well suited for simple behaviours where the designer can predict what is the global effect of a composition. When the system scales up, however, this is rapidly inappropriate. If specifications are informal, they eventually miss some behaviour, or worse, specifications are subject to different interpretations. Moreover, as the specification is informal, it is hardly possible to provide computer-aided verification that checks or infers the global behaviour.

The field of program verification and formal methods can be dated back to Floyd's early works [Floyd 67]. Formal methods use mathematics to specify and then reason about computing systems by revealing ambiguity, incompleteness, and inconsistency from specifications and designs. Because of that they have been recognised as of special interest within safety-critical systems [Barroca 92] and why not to CBSE.

Depending on the kind of formalism, it is possible to check for compatibility errors using a large range of verification techniques such as theorem provers and model-checking. Model-checking has been long used by the hardware community. Intel for example adopted model-checking since the famous Pentium bug which costed the company US$475M back in 1994 [Gerth 01]. Within the software community, however, success of model-checking has been limited. Typically software is so complex that even small systems can fall into the state explosion problem [Valmari 98]. One way of facing this issue is by abstracting the system into a simpler one that holds the desired property, and then to verify the simpler model. If the abstraction is a sound approximation of the original system, then the properties proved in the abstracted model are also true in the original system.

When it comes to component specification, many classic theories can be well adapted to support the formal description of the component behaviour. For example, process algebras, such as Milner's CCS [Milner 90, Milner 89]; Hoares's CSP [Hoare 85, Brookes 84] fits in because a component has well defined frontiers of communication. The communication are the external observable actions, and the internal behaviour is given by internal communication.

Formal methods can address most of the issues discussed above, though adoption is still limited in CBSE. Among the main reasons, we can highlight the gap between the expertise required by formal methods and the ones found in the background of software engineers. **What we require is to support the designer with tools that can hide the complexity of formal methods.**

1.1.3 Focusing on Distributed Components

The semantics of components is dependent on the chosen component model. Since there is no accepted definition of what a component is, we can find software components that vary from considering each object in an object-oriented environment as a component, to very coarse grain definitions of components where a whole application is a component. We can also find components that are deployed in a single machine or distributed in thousands of machines.

One interesting property of CBSE is that components can be used to define units of distribution and to deploy applications over thousands of machines. This is the case of the *Grid Component Model* [CoreGRID 06], which targets components running in computer Grids [Kesselman 98]. Grids are networks of highly heterogeneous machines that provide a low-cost alternative to High Performance Computing by using resources spread over the Internet instead of using expensive dedicated clusters.

When components are distributed and deployed in Grids, an immediate performance requirement is that components should hopefully be loosely-coupled. In other words, the design criteria is biased towards a system with few synchronisations in order to take benefit of parallelism. Therefore, synchronisations must be minimised. One of the classic ways of minimising synchronisations is by adopting some kind of asynchronous communication, provided by component model. Therefore, distributed components have complex interleaving of events due to parallelism, and complex synchronisation due to asynchrony.

Inferring what is the result of a composition of components like this is intrinsically difficult. Unfortunately, there is little or no support from specification languages that address these kind of components. The development of tools and theories has focused on technical services for deployment distributed components, but not enough to support the designer.

1.1.4 Contribution

Overall, this thesis is interested in the design and analysis of distributed component systems, and the generation of components with guaranteed behaviour. We are interested in an approach that does not require the target user to be expert in formal methods, but still takes advantage of a sound specification and verification of the system.

Specification Language. The main contribution is the definition of a specification language that allows software engineers to express complex synchronisations, later to analyse its system and finally generate guaranteed source code.

The specification language allows the designers to express the behaviour visible at the interfaces of the distributed component. It is formal enough in order to generate behavioural models that interface with verification tools, and it is complete enough in order to check against a large range of user requirements. This way, we provide a per-component verification (open system) and verification of a system (closed system). Many of these verifications can be done automatically, so, no advanced knowledge in formal methods is needed.

Formalism. Besides the specification language, we also work on formalisms and behavioural models that allows us to verify specifications of distributed components.

A strong aspect of the formalism is that it is powerful enough to deal with distributed systems in general. For that we have used a generalised parallel operator that allows one to encode many of the parallel operators used in distributed systems. Many process algebras can be mapped into this formalism, and thus it can be seen as an intermediate format between the high-level specification language and the verification tools. The goal is that we can specify the behaviour of our system in this formalism and then decide, depending on the user requirements, which is the best engine to verify the system.

The formalism is based on a symbolic representation of the behaviour that deals explicitly with data. This allows for a compact represention of communication and synchronisation that set the behaviour visible at the interfaces of the components.

Generation of Behavioural Models. Based on the formalism we present, we show how to automatically generate behavioural models from instances of the specification language presented above. The behavioural models allows one to verify the user requirements using existing state-of-the-art verification tools.

Particularly, they allow one to study the effect of the asynchronous semantics of the component model, and to infer the resulting behaviour of the component system, both when develop independently, or when put in an environment. Moreover, we are also able to study the effects of reconfiguration, both of the structure and the environment.

Generation of Components with Strong Guarantees. The specification language that we propose allows one to create safe-by-construction components. Safe in the sense that any property proved in the specification language is guaranteed to hold in the implementation as well. Therefore, the source code of the component does not need to be analysed because we know it is correct. Again, we seek here to hide difficulty commonly found in formal methods.

Verification Platform. Finally, in order to bring these techniques to the software engineers, we provide a set of tools that assist the component design and verification. We build tools that allows one to define the system in a graphical language, and to automatically generate the behavioural models described above.

We do not expect to create a verification platform from scratch, but to use existing verification engines. We do this by providing additional tools that interface our models with the verification tools.

1.2 Thesis Structure

The thesis is structured as follows:

Chapter 2 reviews the state-of-the-art of frameworks, formalisms, specification languages, and tools for dealing with component models.

Chapter 3 provides the context of this thesis; we present the main component model of interest, the Grid Component Model (GCM), and its reference implementation.

Next we start with the contributions of the work.
Chapter 4 presents a parametric formalism that supports the definition of behavioural models of distributed components. This formalism provides us with a compact and expressive representation of the system, as well as the possibility to model-check the models.

Chapter 5 presents the core of the thesis; a specification language adapted to distributed components. The specification language is endowed with enough formality to allow a constructive approach. Namely, the generation of behavioural models based on the formalism, and the generation of code skeletons with the control code of components.

Chapter 6 presents the generation of behavioural models from instances of the specification language. The behavioural models are based on the formalism described in Chapter 4.

Chapter 7 presents our verification platform. It presents the tools developed in this thesis to aid the software engineer in designing and verifying component-based systems, as well as two case-studies in which we applied our tools.

Chapter 8 discusses the perspectives of our work. We show how we pretend to generate safe components starting from our specification language, as well as possible extensions to the specification language.

Chapter 9 summarises the main contributions of our work.

As appendix, we provide an in-depth description of one of the case-studies.

2
State of the Art

Contents

Abstract

This chapter reviews the state-of-the-art of frameworks, formalisms, specification languages, and tools for dealing with component models.

We start by an overview of the main industrial component models and academic component models. This will set the basic vocabulary for this thesis.

Next, we review some formalisms used in component specifications; these provide formal definitions of components and are used to interface with design and verification tools.

Then, we review specification languages that have been used in the context of component specification.

We also overview tools to design and verify components.

Finally, we review some applications of formal methods in component development.

2.1 Component Models

Component models, or better component frameworks provide the means to define, program, deploy, and execute a component system. They can be separated in two large groups: the ones used in industrial applications, and those motivated by academic research. Component models are usually characterised by either those seeking simple designs, which is dominated by industrial component models, and those seeking sophisticated features such as runtime reconfiguration and separation of concerns between functional and non-functional aspects.

There is no widely accepted definition of what a software component is [Szyperski 02]. Nevertheless, we could expect a component to:

- have well-defined interfaces to interact with the environment; we will accept as a component only the pieces of software where both provisions and requirements are explicitly defined.
- allow for some sort of independent deployment; a component must be a closed, complete piece of software. Even if it has external dependencies, it should still be possible to think of a component as a black-box entity that provides and requires services.
- and allow for composition; a component must allow for some kind of composition. In other words, we would like to think of components as building blocks for more complex pieces of software. In here we do not distinguish how the components can be composed, though.

In the following, we review the basic concepts of software components, e.g. whether a component model is flat or hierarchical. This will set-up the common vocabulary of the thesis.

Composition. One of the main aspects of components is composition. There are two kinds of compositions: *horizontal* and *vertical* composition. The *horizontal* composition stands for a composition of two components at the same level of hierarchy (sibling components). The *vertical* composition stands for a composition of two components at different levels of hierarchy. This is, a component that wraps other components. The wrapper component is often called the *parent component*, and the wrapped component its *subcomponent*.

When a component model supports *vertical* composition, we say that the component model is hierarchical because we can nest components to build more complex components. These nested components are usually called *composite components*, whereas the most simple form of components (not nested) are called *primitive components*.

Similarly, when a component model does not support *vertical* composition, we call it a flat component model. In this case, only *horizontal* composition is allowed, thus the component topology is plain.

Ports and Interfaces. Another important aspect of components is that they communicate with the environment through well-defined interaction points. These set the external dependencies of each component. The interaction points receive different names depending on the component model of choice. Vocabulary varies from ports (as in ArchJava [Aldrich 02]) to interfaces (as in CCA [Armstrong 06], Fractal [Bruneton 02], SOFA [Plášil 98]), though less commonly they are called sink and sources (as in CCM [OMG 02]). Yet there are component models that support both ports and interfaces (as in UML [OMG 04], Java/A [Baumeister 05]).

Usually a *port* is considered as an entry/exit point of communication with the environment. A port defines a name which is used by the internals of the component to reference the environment. Communication on ports is often directional, and the allowed messages are signed by a set of interfaces (as in UML 2 [OMG 04], Java/A [Baumeister 05]).

An *interface*, on the other hand, usually imposes some kind of data flow. Interfaces can be *provided* (also called server, or provides), and *required* (also called client, or uses). Provided interfaces export a service to the environment, whereas required interfaces depend on a service from the environment.

In this thesis, we usually speak of *interfaces* for communication with the environment in general. This applies to both ports and interfaces.

Connectors and Bindings. The link between two or more components (as in Reo [Arbab 04]) are connectors, and explicitly defines the flow of communications happenning at both ends of the link. Connections between ports are usually called *connectors*, and between interfaces are usually called *bindings*.

In this thesis, we usually speak of *bindings* for connections in general. This applies to both connectors and bindings.

Bindings can connect a component to another component or subcomponent, and so on. However, the general rule is that a binding between components can never cross the component's frontier[*]. The different kinds of bindings are:

Connection of sibling components. This is a connection between components at the same level of hierarchy, and is usually called a "normal binding".

Delegation of a service. This is a binding between a component and one of its subcomponents. It says that a service provided by the component is implemented by one of its subcomponents, or symmetrically, a service required by the component is required by one of its subcomponents.

Business and Non-Functional Behaviour. When talking about what a component does, we are usually talking about its *business* (or functional) behaviour. This is the behaviour the component addresses, like the services it offers and requires.

There is also an orthogonal behaviour called *non-functional* behaviour. The exact definition of what is considered non-functional behaviour varies greatly on the component model used. Examples of non-functional concerns go from Quality of Service (QoS), to life-cycle management of components. We shall clarify, when convenient, the exact meaning of non-functional; however, in general in this thesis we will be interested in the latter: those related to life-cycle, such as dynamic reconfiguration.

[*]Except for some special kinds of bindings dealing with shared components

2.1.1 Industrial Component Models

In general, industrial component models aim at standardisation through the use of simple models. We provide here examples of flat component models such as CCA [Armstrong 06] and CCM [OMG 02], and hierarchical component models such as SCA [BEA Systems 05].

The component models chosen here are those most related with Grids and distributed environments. Both CCA and CCM are used in Grids, thus are equipped with some kind of collective communication, though they are flat. We will focus on collective communication later in the thesis because they provide both performance optimisation and a better structuring of the system.

SCA, on the other hand, provides a good example of a hierarchical component model used in the context of Web Services [W3C 04]. It is not primarily focused on Grids, however it does show that components can be though of services. This idea will be later used in the thesis in order to define the component behaviour.

2.1.1.1 CCA

Within the Grid community, one of most popular component models is *Common Component Architecture* (CCA) [Armstrong 06]. CCA, held by the CCA Forum, is a minimalistic component model that addresses high-performance computing.

CCA aims at facilitating software integration. It considers that it is convenient to isolate pieces of software as modules that exhibit well-defined interfaces. These interfaces are defined by normalised ports, using the *Scientific Interface Description Language* (SIDL).

SIDL was conceived as an IDL for scientific computing. It represents abstractions and data types common in scientific computing at the interface level. Amongst them, dynamic multidimensional arrays and complex numbers. SIDL is programming-language independent, being the common platform for cross-language frameworks. However, SIDL is bound to specifying types of the interfaces only, thus it is does not take into account the component behaviour.

CCA builds on core concepts, defining a *component* (the software entity), a *framework* (the container), and *ports* (the access towards the environment). CCA also proposes a configuration API and a repository API for reusing components in different contexts.

A limitation in CCA is that components are flat. In other words, CCA components can be composed (connected) horizontally but not vertically. This limitation, however, is also partially responsible for the simplicity of CCA, and thus is considered a major asset. The target users (scientists) are fond of having a simple framework for composing their applications.

One of the innovations in CCA was the introduction of abstractions for parallelism. CCA considers parallelism and distribution of data by adopting, among others, *collective ports*. The idea is to let designers focus on a higher-level abstraction and let distribution concerns at a different level of the design. These abstractions also address MxN communications, which allow communications in Grids to be optimised.

2.1.1.2 CCM

The CORBA Component Model (CCM) [OMG 02, OMG 05] is an industrial component model defined by the Object Management Group. CCM supports distributed components, and has an API that allows for platform independent designs.

Components can be seen as black boxes that provide access points, though they are flat, i.e. there is no hierarchical composition. Some of the major benefits of CCM is that is provides a clear separation between functional and non-functional concerns, in which actors of the process are clearly identified as designer, implementer, packager, deployer, end-user. Yet CCM is based on previous Corba specifications [Obj 04], therefore it inherits the notion of distributed objects.

GridCCM [Denis 04] is a grid-oriented version of CCM. It defines parallel components that handle SPMD code. In fact, GridCCM considers parallelism to be a non-functional property that must be handled separately. It is therefore defined in a separate configuration file that sets the distribution.

2.1.1.3 SCA

Service Component Architecture (SCA) [BEA Systems 05] is an industrial, hierarchical component model. Major software vendors are involved in the project, which include: BEA, IBM, Oracle, and SAP.

A SCA component is a generalisation of a Web service [W3C 04], being based on the concept of *Service Oriented Architecture* (SOA) [Erl 05]. SCA components can be implemented with different languages such as Java, BPEL, and state machines. Each component provides standardised interfaces to the outside, and the component can require other components to implement certain interfaces – called references.

A SCA module orchestrates a set of given components by wiring them together. Each wire connects a reference of one component with an interface of another component. A SCA module itself might provide interfaces to the outside – called exports or entry points, and it might require interfaces to be implemented somewhere else – called imports or external services. Thus, a SCA module can act as a component within a larger assembly, i.e, it is a hierarchical component model.

2.1.2 Academic Component Models

Within academia, there are several component models being defined. They are characterised by supporting state-of-the-art features, such as hierarchical components, dynamic reconfiguration, advanced communication styles. Some also allow the designer to formally specify the component behaviour.

The first class of academic components chosen here is the one with ArchJava [Aldrich 02] and Java/A [Baumeister 05]. They provide an approach that inspire part of this thesis, by extending Java with architectural primitives, and including formal specifications of the component behaviour. To summarise, they provide a Java-like syntax that mixes architecture and behaviour that keep the specification and implementation close. This is one of the goals of this thesis.

Then, we include a depth review of Fractal [Bruneton 02, Bruneton 04, Bruneton 06] as we believe it is a good representative of sophisticated component models that put emphasis in separation of concerns between functional and non-functional aspects. It is extensible by design which allows for variants with extensions towards Grids (GCM [CoreGRID 06]); the latter takes care of collective communications and particularly asynchronous communications. Fractal also provides structuring of non-functional aspects (AOKell [Seinturier 06] and GCM [CoreGRID 06]), and deals with dynamic reconfiguration. In this thesis we are particularly interested in providing a behavioural specification for one of Fractal variants, the GCM.

We also present the SOFA component model [Plášil 98]. SOFA is a good example of component models that have been designed aiming at sound component specification. It does not apply directly to Grids, however it provides an interesting approach for dealing with compositional reasoning and verification. Moreover, in its latest version (SOFA 2.0 [Bureš 06]) it puts emphasis in structuring non-functional aspects and in dynamic reconfiguration.

2.1.2.1 ArchJava and Java/A

ArchJava [Aldrich 02] and Java/A [Baumeister 05] provide similar approaches. Their main objective is to deal with *architectural erosion* [Perry 92], and the common approach is to introduce language primitives into Java particularly for defining architectural patterns. Concretely, they extend Java with architectural information typically found in Architecture Definition Languages (ADL) (components, ports, interfaces and bindings).

Both approaches also take into account the behaviour specification of components. In ArchJava, this is done using an extended π-calculus [Milner 93, Milner 92] formalism. The formalism allows the definition of component migration, data- and control-flow specification, among others. However, due to the expressive power of π-calculus there are few tools that can verify the behavioural consistency.

On the other hand, Java/A opts for a simpler language. The behaviour is defined by means of Labelled Transition Systems (an Interface Automata [de Alfaro 01]), defining the behaviour of the components, and of the ports. The latter is one of the main originalities of this work. The *port* has a contract of communication that must be obeyed in order to bind the port, in the sense of Behavioural Contracts [Carrez 03] that guarantee absence of deadlocks. Verification is performed by a custom Java compiler, verifying both architectural and behavioural consistency using model-checking techniques.

2.1.2.2 Fractal

Fractal [Bruneton 02, Bruneton 04, Bruneton 06] is a modular and extensible component model proposed by INRIA and France Telecom, that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms or graphical user interfaces.

The Fractal model is somewhat inspired from biological cells, where exchanges between the content of a cell and the environment are controlled by the membrane. By analogy, a Fractal component (as represented in Figure 2.1) is a runtime entity, which offers server and client functional interfaces, as well as non-functional interfaces implemented as controller objects in the membrane. All interactions with the component are interactions with the membrane of the component, and this allows interception and intercession using interception objects positioned within the membrane. Moreover, all non-functional aspects are dealt within the membrane of a component, thus enforcing separation of concerns between functional and non-functional features.

Fractal is hierarchical in the sense that components can be nested in arbitrary levels of hierarchy. The components are connected through bindings, that can either connect two subcomponents, or connect a subcomponent to the parent component (see Figure 2.1).

A Fractal component may be a *composite component*, in which case it usually contains other components, or it may be a *primitive component*, which is an atomic component implementing the business logic.

However, there are special kinds of components called *shared components*. Shared components break the topology in the sense that shared components are owned by different hierarchies of components. If we represent shared components graphically, the bindings would cross the frontiers of the component's parent. Shared components are useful when the system architecture provides common resources that must be globally accessed, for example a database. In this thesis, however, we will not be particularly interested in shared components.

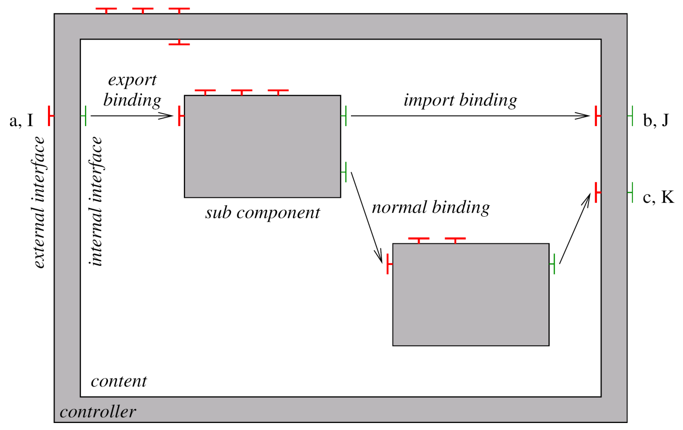


Figure 2.1: *A composite component as defined in Fractal Specification*

The Fractal model is an open component model, and in that sense it allows for arbitrary classes of controllers and interceptor objects, including user-defined ones. Fractal is meant to be extensible; in this sense it leaves unspecified how communication takes place between components, how components are specified, and what is its implementation; even bindings can be components. Non-functional features of the component can also be customised.

There are different levels of conformance, being the most simple one considering an object as a component, high conformance where components must implement a basic set of predefined controllers. For example, setting attributes (*AttributeController*), binding components (*Binding-Controller*), adding components within composite components (*ContentController*) or controlling the lifecycle of the component (*LifeCycleController*).

Fractal is based on the following definitions:

Content: one of the two parts of a component, the other one being its membrane. The content is an abstract entity controlled by a controller. The content of a component is (recursively) made of sub components and bindings.

Membrane: one of the two parts of a component, the other one being its content. The membrane is an abstract entity that embodies the control behaviour associated with a particular component. The membrane is composed by controllers.

Controller: a controller exercises an arbitrary control over the content of the component it is part of (intercept incoming and outgoing operation invocations for instance).

Server interface: a component interface that receives invocations (e.g. (a,I) in Figure 2.1).

Client interface: a component interface that emits invocations (e.g. (b,J) in Figure 2.1).

Functional interface: a component interface that corresponds to a provided or required functionality of a component, as opposed to a non-functional interface. In Figure 2.1 they are depicted horizontally, directed towards left and right for server and client interfaces resp.

Non-functional interface: a component interface that manages a "non functional aspect" of a component, such as introspection, configuration or reconfiguration, and so on. These are also called control interfaces. In Figure 2.1 they are depicted vertically, directed towards top and down for server and client interfaces resp.

Fractal usually proposes a *white-view* of the component's non-functional aspect. The non-functional aspects are globally accessible through introspection, hence most diagrams do not represent non-functional bindings.

Interfaces are defined by a *name*, a *role* (client or server), a *cardinality* (singleton or collection), a *contingency* (mandatory or optional) and a *signature* (in Java, the fully qualified name of a Java interface). In the type system proposed by Fractal, the set of functional interfaces defines the type of a component. This can be further extended to take into account the non-functional interfaces as well. Moreover, there are external and internal interfaces. How these two relate is undefined in Fractal. This leaves freedom to define *interceptors*.

Fractal components are defined by an ADL by means of an XML file. The structure can be validated by a DTD definition. In Section 3.1.1 we give more details about the Fractal ADL.

Julia. Julia [Bruneton 04] is the reference implementation of the Fractal component model. It is fully compliant with the Fractal specification, and is meant to be a lightweight implementation. One of its main goals is to minimise overhead w.r.t. plain Java.

Communication is synchronous, and the control flow is set by threads invoking the component body. That is, the component is a runtime object that can be invoked. All invocations go through a membrane that allows for control of the component. For instance, there is a counter that is incremented each time a thread enters the component, and is decremented when it leaves. This way, a component can be stopped when the counter reaches zero.

The component membrane is implemented by a mechanism of class mixin (optimised object oriented version), as well as a recent implementation based on non-functional components (version 2.5). The latter considers a membrane as a composite component that is in charge of the control part of a component. Therefore, the membrane inherits all the structuring benefits from a component oriented approach.

AOKell. Another implementation of Fractal is AOKell [Seinturier 06]. Here, the goal is akin with Julia in the sense that it privileges performance, however AOKell takes a different approach by providing an aspect-oriented approach using AspectJ [Kiczales 01].

One of the originalities of this work is to rely on aspects to deal with the control of components. The membrane is organised as an architecture of controllers, where each controller is an aspect. Moreover, the approach allows the designer to clearly define the interaction between these controllers, which are later injected into the component. This structuring of the membrane will inspire other component models such as the GCM (detailed in Chapter 3).

The Figure 2.2 [†] shows an AOKell component. Components are supervised by controllers implemented as aspects.

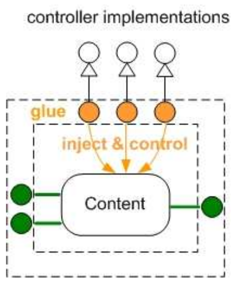


Figure 2.2: *Structure of an* AOKell *component*

Grid Component Model (GCM). The GCM [CoreGRID 06] is a novel component model defined by the european Network of Excellence CoreGrid and implemented by the EU project GridCOMP. The GCM is based on Fractal, and extends it to address Grid concerns. We shall detail this extension of Fractal in Chapter 3 as it represents the main component model of interest for this thesis.

2.1.2.3 SOFA

SOFA [Plášil 98] is a distributed component model and implementation proposed by Charles University in Prague; it stands for SOFtware Appliances. In the SOFA component model, an application is viewed as a hierarchy of nested software components. Similar to Fractal, a component is either primitive or composite. A composite component is built of other components, while a primitive one contains no subcomponents.

[†]Figure extracted from `http://fractal.objectweb.org/tutorials/aokell/index.html`

Each component in SOFA has a *template* T; this can be seen as a *component type*. T is a pair $\langle F, A\rangle$ where F is a *frame* and A is an *architecture*. The frame is a black-box view of the component, and the architecture is a grey-box view of the component.

The black-box view F defines the set of individual interfaces any component has. In F, an interface is either a provides-interface or a requires-interface.

The grey-box view A exposes the interaction visible by one level of the nested hierarchy. An architecture A gives details about the component implementation: it declares and instantiates direct subcomponents of A; and specifies the subcomponents' interconnections via interface ties. One or more architectures can be associated to a frame, creating an analogy to object orientation between interface/class and frame/architecture.

There are four kinds of interfaces ties within a template $T = \langle F, A\rangle$:

1. *binding* a requires-interface to a provides-interface between two subcomponents,
2. *delegating* a provides-interface of F to a subcomponent's provides-interfaces,
3. *subsuming* a subcomponent's requires-interface to a requires-interface of F, and
4. *exempting* an interface of a subcomponent from any ties (the interface is not employed in A).

In the case an architecture is specified as *primitive*, there are no subcomponents and its structure/implementation will be provided in an underlying implementation language.

SOFA 2.0

SOFA 2.0 [Bureš 06] is a recent project to extend SOFA, being the result of several years of experience in working on both SOFA and Fractal component models. In [Bureš 06], the main limitations of SOFA are identified as:

- having a limited support for dynamic reconfigurations,
- lacking of a structure for the control part of a component,
- and having an unbalanced support for multiple communication styles.

Dynamic Reconfiguration. SOFA 2.0 supports dynamic reconfiguration (i.e., adding and removing components at runtime, passing references to components, etc). It proposes *reconfiguration patterns* in order to avoid uncontrolled reconfigurations which lead to runtime errors. For the moment, they allow three reconfiguration patterns: (i) nested factory, (ii) component removal, and (iii) utility interface. In pattern (i) a component creates a subcomponent when called in a specific interface. Pattern (ii) is analogous to (i) for removing a component. In pattern (iii) there is a special kind of interface that is orthogonal to the component architecture, being possible to "cross" the component architecture; this is similar to the approach taken in Fractal/Julia on shared components.

Structure of the Control Part. For structuring the control part of a component, in SOFA 2.0 there are *microcomponents* and control interfaces. Microcomponents are minimalist components:

they are flat (there are no nested microcomponents); do not have no connectors; are not distributed. The parallel to Fractal would be Fractal's controllers. Control interfaces are orthogonal to business interfaces in the sense they focus on non-functional features of components. These interfaces are in direct relation to the non-functional interfaces found in Fractal.

It is interesting that the authors claim that the code implementing the business code (functional components) should be aware of the control part; moreover, they state that the business code should have access to the control part of the component. This could be seen as a cross cutting concern that hopefully should be avoided according to the Fractal community.

Multiple Communication Styles. Communication style stands for the way components interact. Typically, a component model supports a limited set of ways components may interfact, the most common ones being message passing and remote method invocations. Depending on the component model, one (or more) of these are supported, but each of them is more adequate for a particular system architecture.

In SOFA 2.0, multiple communication styles are supported thanks to classes of connectors. There are three families of connectors: delegation, subsumption, and connector. Then, the connector is specialised to match the runtime, i.e. if we are in a Unix system we can use pipes; if we are in a LAN we can use TCP/IP, etc.

2.2 Formalisms

In this section we concentrate on formalisms that have been used for distributed system verification. We start with process algebras [Bergstra 01] that have set the theoretical basis of distributed systems.

Then we present a generalisation of the synchronisation product of transition systems [Arnold 94]. We also present works on symbolic extensions of process algebras [Hennessy 95]. Both are of interest for our own pNets formalism described in Chapter 4.

Then we present a formalism more adapted to components, CI-Automata [Brim 06]. This formalism is part of the family of formalisms [Lynch 87, de Alfaro 01] that express behaviour at the interface of components in the spirit of Labelled Transition Systems.

Finally, we present an object calculus called ASP [Henrio 03] that is relevent to the understanding of this thesis. The results of determinism of ASP-calculus are later used in the behavioural models of Chapter 6.

2.2.1 Process Algebras

Process algebras [Milner 82, Bergstra 01] are the most classical way of specifying the behaviour of a system. They consider a system as *processes* that interact, usually sending/receiving messages and synchronising. They are supported by a strong mathematical basis, using operators within an algebraic theory, endowed with several types of semantics.

Among the most widely known process algebras, one can identify Milner's CCS [Milner 90, Milner 89]; Hoares's CSP [Hoare 85, Brookes 84]; and Milner's π-calculus [Milner 93, Milner 92].

CCS defines a small language whose constructors reflect simple operational ideas. The core of CCS is a congruence relation between closed *terms*, where the terms represent *processes*. A semantic process is understood to be a congruent class of terms. An equality of processes can be defined as being indistinguishable in any experiment based upon observation. An important part of the definition of CCS is that the operational semantics is presented as *Labelled Transition Systems* (LTS), in which the derivation tree are the transitions or actions which may be performed by a given process. Due to the simplicity of CCS, it is possible to develop very efficient verification tools.

The π-calculus is a more expressive process algebra that extends CCS with mobility, and some kind of reconfiguration of processes. Processes are mobile and the configuration of communications links may dynamically change. One of the tradeoffs of its expressivity is that in general it is not possible to build efficient model-checkers without additional constraints on the system.

2.2.2 Synchronised Products of Transition Systems

Synchronised Products of Transition Systems [Arnold 94] is a generalisation of the interaction between transition system. It introduced the idea of a synchronisation vector that denote how the different processes in the hierarchy synchronise on actions. Synchronisation vectors can also be used as a generalised hidding and relabelling operator.

A synchronisation vector expresses, in a hierarchical way, which are the allowed synchronisations of processes. The synchronisations take place on ports of the processes by means of expressing the valid actions. It has a distinguished external action that is the (visible) result of the synchronisation. There is a special marker used to identify which processes can progress freely; the other process that synchronise must perform the transition synchronously.

2.2.3 Symbolic Transition Graphs

Symbolic Transition Graphs (STG) [Hennessy 95], introduced by Hennessy and Lin, provides a symbolic representation of transition systems. It is particularly inspired by regular value-passing CCS [Milner 89], being in the very beginning, an extension of CCS that provided a more abstract description of processes in terms of *symbolic actions*. Another variant of STG is Symbolic Transition System (STS) [Ingólfsdóttir 01, Fernandes 07].

STG are parameterized on a number of syntactic categories. The first two are a countable set of *variables*, $Var = \{x_0, x_1, \ldots\}$, and a set of values V. $Eval$, ranged over by ρ, represents the set of *evaluations*, i.e. the set of total functions from *Var* to V. A substitution is a partial injective mapping from *Var* to *Var* whose domain is finite. Sub represents the set of substitutions and this set is ranged over by σ. They also presume a set of *expressions*, Exp, ranged over by e, which includes *Var* and V. Each e has an associated set of free variables, $fv(e)$, and it is assumed that both evaluations and substitutions behave in a reasonable manner when applied to expressions.

Formalisation. A STG is a directed graph in which each node n is labelled by a set of variables $fv(n)$ and every branch is labelled by a guarded action such that if a branch labelled by (b, α) goes from node m to n, which we write as $m \xrightarrow{b,\alpha} n$, then $fv(b) \cup fv(\alpha) \subseteq fv(m)$, and $fv(n) \subseteq fv(m) \cup bv(\alpha)$.

Another variant introduced by Lin [Lin 96] STG by allowing *assignments* to be carried in transitions. An edge now takes the form $n \xrightarrow{b,\bar{x}:=\bar{e},\alpha} n'$, where, besides a boolean condition b and an abstract action α, there is also an assignment $\bar{x} := \bar{e}$. Roughly it means if b is evaluated to *true* at node n then the action α can be fired, and, after the transition, the free variable $\bar{x}$ at node n' will have the values of $\bar{e}$ *evaluated at n*.

The benefits of symbolic representations is that the models are more compact and more expressive. They represent families of models, that are usually closer to what a system looks like. Hennessy and Lin have also defined *symbolic bisimulation equivalences* [Hennessy 95], and providing as well algorithms for both late and early symbolic bisimulations.

2.2.4 Component-Interaction Automata

The *Component-Interaction Automata* (CI Automata) [Brim 06], is a formalism for specifying the interactions in component systems. It is part of the family of *I/O Automata* [Lynch 87], and *Interface Automata* [de Alfaro 01]. These works are extensions to LTS, making explicit references to interfaces and flow of communication.

The authors aim at providing a general purpose platform for verifying component systems. The key idea is that the actions of the language, and the composition operators can be set to match a specific domain. For example, one could want to model synchronous or asynchronous communications, and by changing the composition operator this can be achieved.

Each component in a system is associated with a CI automaton. Components are denoted by natural numbers $n \in \mathbb{N}$. The components communicate through interfaces, with actions being messages, method calls, and any relevant information needed in the specific domain.

Formalisation. A component-interaction automaton is a tuple $C = (Q, Act, \delta, I, \mathcal{S})$, where:

- Q is a finite set of *states*,
- Act is a finite set of *actions*, $\Sigma = ((X \cup \{-\}) \times Act \times (X \cup \{-\})) \setminus (\{-\} \times Act \times \{-\})$ where $X = \{n \mid n \in \mathbb{N}, n \text{ occurs in } \mathcal{S}\}$, is a set of *symbols* called an *alphabet*,
- $\delta \subseteq Q \times \Sigma \times Q$ is a finite set of *labelled transitions*,
- $I \subseteq Q$ is a nonempty set of *initial states*, and
- $\mathcal{S}$ is a tuple with the subcomponents of C.

Symbols $(-, a, B), (A, a, -), (A, a, B) \in \Sigma$ are called *input*, *output* and *internal* symbols of the alphabet Σ, respectively.

- $(-, a, B)$ means that the component B receives an action a as an input.
- $(A, a, -)$ means that the component A sends an action a as an output.
- (A, a, B) means that the component A sends an action a as an output, and synchronously the component B receives the action a as an input.

The composition operator $\otimes^{\mathcal{F}}\mathcal{S}$ is parameterized by $\mathcal{F}$, where $\mathcal{F}$ is a set of transitions. The composition operator produces the cartesian product of components in $\mathcal{S}$, allowing only transitions appearing in $\mathcal{F}$. The goal in here is to allow different communication strategies to take place. Another way of seeing $\mathcal{F}$ is that it represents the binding between components, by expressing which are the allowed communications between them.

For instance, one would like the composite automaton C composing components C_1 and C_2 to: (i) allow any interleaving of actions of C_1 and C_2, or (ii) allow only those actions in which both components synchronise. Each one of these would represent different composition operator with two different set of transitions $\mathcal{F}^{(i)}$ and $\mathcal{F}^{(ii)}$ resp.

Nevertheless, the formalism in CI automata is not symbolic, meaning here that the automata are not parameterized. This basically means that it is limited to finite systems, and that specifications may not scale well in some architectures.

Another limitation is that the synchronisations between components must be static; in other words encoding rebinding is hard. In Chapter 4 we will show our formalism in which we are able to represent synchronisation vectors [Arnold 94] that could be used with this goal.

2.2.5 ASP

ASP [Henrio 03, Caromel 05b] is an object calculus that allows one to write parallel and distributed applications, particularly on wide range networks, while ensuring good properties.

ASP is based on a sequential object calculus *à la* Abadi-Cardelli [Abadi 96] to which it adds parallelism primitives. The main characteristics of ASP are: asynchronous communications, futures, and a sequential execution within each process. ASP presents strong confluence and determinism properties.

A first design decision is the absence of sharing: objects live in disjoint activities. An activity is a set of objects managed by a unique process and a unique active object. Active objects are accessible through global/distant references. They communicate through asynchronous method calls with futures. A future is a global reference representing a result not yet computed. The main result consists in a confluence property and its application to the identification of a set of programs behaving deterministically. These results can be summarised by the following simple assertions:

- *The execution is insensitive to the moment when futures are updated.*
- *The execution is only characterised by the ordered list of activities that have sent requests to a given one.*
- Several approximations can be performed in order to characterise programs behaving deterministically. *For example, every program communicating over a tree behaves deterministically.*

The sequential part of ASP (all primitives except $Active$ and $Serve$) is very similar to **impς**-calculus [Abadi 96, Gordon 97]: ASP calculus only differs in the fact that in addition to the self argument of methods (noted x_j, usually called self), an argument representing a parameter object can be sent to the method (y_j in abstract syntax). Distinguishing method arguments is necessary because of the copy semantics associated to them in the case of an asynchronous method call.

The *Active* operator creates a new *activity* from the object a. The operator $Serve$ allows to specify which *request* (distant method call) should be served.

From a practical point of view, ASP can also be considered as a model of the ProActive library [Caromel 06a]. This library provides tools for developing parallel and distributed applications in Java, and will be cited extensively during this thesis. ProActive is the basis for the reference implementation of the Grid Component Model (GCM) [CoreGRID 06], on which most of this work is based on. This is detailed in Chapter 3.

Syntax of ASP. The abstract syntax of the ASP calculus is the following (l_i are fields names, m_j are methods names):

$a, b \in L ::= x$	variable,
$\mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m}$	object definition,
$\mid a.l_i$	field access,
$\mid a.l_i := b$	field update,
$\mid a.m_j(b)$	method call,
$\mid clone(a)$	superficial copy,
$\mid Active(a, m_j)$	activates object: m_j is the service method
$\mid Serve(M)$	serves a request among a set of method labels,

M is a set of method labels specifying which request has to be served ($k > 0$):

$$M = m_1, \ldots, m_k$$

Example: Fibonacci Numbers in ASP

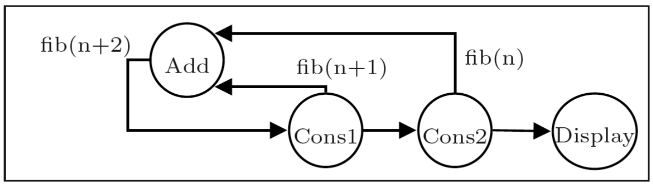
Consider the Process Network that computes the Fibonacci numbers in [Parks 03]. Let us write an equivalent program in ASP (in Figure 2.3). *Repeat* performs an infinite loop, ";" expresses sequential composition. *Repeat*, ";" and the mutually recursive definition of activities *let rec . . . and . . .* can be built from core ASP terms.

Display receives the Fibonacci numbers. Initialization consists in sending 0 ($fib(0)$) and 1 ($fib(1)$) from $Cons2$ and $Cons1$ respectively.

ASP and Component Models. In [Caromel 06b], the authors define a component model based on ASP-calculus. It aims at distribution, featuring asynchronous remote method invocations, and futures as generalized references passing through components.

Primitive components are defined as a set of *Server Interfaces* (SI) and *client interfaces* (CI), together with an ASP term representing the component behavior. Intuitively, each SI corresponds to a set of methods (very much like an interface in Java), each CI to a field (very much like a member field in Java). Composite components are recursively made of primitives and composites, with a partial binding between SIs and CIs.

Two translational semantics are proposed: one where components completely disappear in the generated term, and the other converts each components boundary into an extra active object.



$$
\begin{aligned}
&\textit{let rec Add} = \textit{Active}([n1 = 0, n2 = 0;\\
&\qquad\qquad service = \varsigma(s, \_)\\
&\qquad\qquad\qquad Repeat(Serve(set1); Serve(set2); Cons1.send(s.n1 + s.n2)),\\
&\qquad\qquad set1 = \varsigma(s, n)s.n1 := n, set2 = \varsigma(s, n)s.n2 := n], service)\\
&\textit{and } Cons1 = Active([\emptyset;\\
&\qquad\qquad service = \varsigma(s, \_)\,(Add.set1(1); Cons2.send(1); Repeat(Serve(send)))\\
&\qquad\qquad send = \varsigma(s, n)(Add.set1(n); Cons2.send(n))], service)\\
&\textit{and } Cons2 = Active([\emptyset;\\
&\qquad\qquad service = \varsigma(s, \_)\,(Add.set2(0); Display.send(0); Repeat(Serve(send)))\\
&\qquad\qquad send = \varsigma(s, n)(Add.set2(n); Display.send(n))], service)
\end{aligned}
$$

Figure 2.3: *Fibonacci numbers in ASP)*

Primitive deterministic components are defined by imposing that each set of interfering requests belong to the same server interface. Finally, a deterministic composite (DCC) avoids potential interferences by imposing at most a single binding towards a server interface. Consequently, together with asynchronism, the component model provides a good abstraction for verifying determinism properties.

Example: A Fibonacci Component

Figure 2.4 represents a component version of Fibonacci example of 2.2.5. A primitive component *Add* can be built up from active object *Add*. *Cons1* and *Cons2* have been merged in a composite component. A controller component *Cont* has been added. It exports a server interface ($ComputeFib(k)$) taking an integer k and forwarding $k - 1$ times its input to CI_c. According to the program of 2.2.5, $Cons2$ sends *send* requests to the exported client interface, thus FIB produces $Fib(1) \ldots Fib(k)$. Each primitive component activity can be easily adapted from the ASP example as shown in the case of the *Add* component (Add_{Act}).

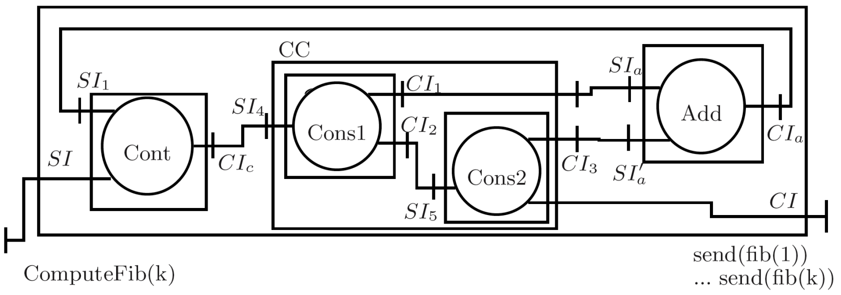


Figure 2.4: *A composite component (based on Fibonacci example)*

2.3 Specification Languages

In this section we review specification languages targetting component models, though we also include two classic specification languages that target the specification of distributed systems in general.

We first present the most simple kind of specification languages; *Architecture Description Languages* (ADLs). They focus on architectural factors and are the most widely used in component-based engineering. Almost every component model supports specification through ADLs, however they rarely address behaviour. We will base the architecture definition of our work on an ADL-like language, augmented with behavioural definition.

Then we present *Behavior Protocols* [Plášil 02] that provides a trace-semantics for describing the behaviour of components. This work is inspiring specially because the authors provide a broad application of formal methods in components; particularly they have applied the techniques in Fractal [Kofroň 06, Bulej 08]. In the last section of this chapter we will show some of these applications.

We include Sensoria [Sensoria 05] as an example of similar approach to Behavior Protocols for Web Services and SCA components. However, Sensoria describes the behaviour based on π-calculus.

STSLib [Fernandes 07] provides a formal approach similar to ours. Their formalism is based on extensions to symbolic transition systems, and they aim at generating safe-by-construction distributed components.

Next, we highlight UML 2 [OMG 04] as being one of the most used semi-formal specification languages. It provides means for specifying components and their behaviour, though without a precise semantics. Nevertheless, if UML is specialised to include a domain-specific semantics, it could be used as a graphical specification language for formal definitions of components.

Finally, we include other two examples of specification languages for distributed systems in general. Both LOTOS [ISO 89] and Promela [Holzmann 03, Gerth 97] provide sound specifications that could be used for specifying components as well. Moreover, they provide direct integration towards state-of-the-art verification engines.

2.3.1 Architecture Description Languages

The strengths of ADLs relies on its simplicity. They are light-weight languages mainly targeted at static representations of the system topology. In ADLs, we usually find the definition of components, their interfaces, subcomponents and bindings. Most of the component models use ADLs to define the components at some sort of detail. The one used in Fractal will be described in Section 3.1.1.

For defining the interfaces, ADLs are often equipped with *Interface Description Languages* (IDLs). The IDLs sign the interfaces by specifying the messages, methods, and types allowed within an interface.

More sophisticated versions of ADLs include behavioural descriptions, however we do not consider them as ADLs. Examples of these are ArchJava [Aldrich 02] (that we have already described) and AEmilia [Balsamo 02]. AEmilia [Balsamo 02] is an ADL based on Stochastic Process Algebras. It allows for behavioural and functional verification, as well as performance modelling and analysis.

Acme. Acme [Garlan 00] is a representative of this class of specification languages. It aims to define a generic hierarchical ADL with common foundations for ADLs, in such a way that different analysis tools can be integrated; for example, in [Abi-Antoun 05] Acme is integrated to work together with ArchJava (see Section 2.1.2.1).

Acme considers key elements such as: *components*, *connectors*, *systems*, *ports*, *roles*, *representations*, and *rep-maps*. A *component* is a computation unit. A *connector* is an interaction among components (a binding). A *system* is a closed component model, represented by a graph in which nodes are components and arcs are connectors. A *port* is a point of interaction between the component and the environment (an interface). A *role* is the definition of a participant in a connection, for example caller or callee in RPC.

A *representation* is a more detailed description of a component; this is the way hierarchical descriptions are described in Acme. A *representation* gives details of a component, and a *rep-map* maps a component and the representation. Multiple views of a component is possible by defining multiple rep-maps.

The ports can represent simple one-to-one communications, or more complex ones. It is possible, for instance, to specify that procedures must be called in a specified order. Furthermore, it is also possible to define multicast interfaces.

2.3.2 Behavior Protocols

The SOFA project provides a trace-semantics language called *Behavior Protocols* [Plášil 02] [‡] for defining the behaviour of components. The components are specified using *Component Definition Language* (CDL), which describes interfaces, frames, and architectures. It is based on the OMG IDL [Obj 04], and extends the features found in the IDL to allow specification of software components. Although *Behavior Protocols* were firstly designed to work together with SOFA components, it can be extended to other component models such as Fractal – this is shown in [Kofroň 06, Bulej 08].

An example of a SOFA specification in CDL is shown in Figure 2.5. The full CDL syntax can be found at the project's webpage [SOFA 98].

The behaviour of SOFA components is modelled via event sequences (traces) on the component's interfaces (connections). The event sequences are approximated and represented by regular expressions called *Behavior Protocols*.

Events are written as ⟨prefix⟩⟨interface⟩.⟨name⟩⟨suffix⟩ . The prefix (!, ? or τ) expresses whether

[‡]The spelling of *Behavior* is kept in American English to maintain the style of the authors

```
interface IDBServer {
    void Insert(in string key, in string data);
    void Delete(in string key);
    void Query(in string key, out string data);
};

frame DatabaseBody {
    provides:
        IDBServer d;
        ICfgDatabase ds;
    requires:
        IDatabaseAccess da;
        ILogging lg;
        ITransaction tr;
};
```

```
frame Database {
    provides:
        IDBServer dbSrv;
    requires:
        IDatabaseAccess dbAcc;
        ILogging dbLog;
};

architecture Database version v2 {
    inst TransactionManager Transm;
    inst DatabaseBody Local;
    bind Local:tr to Transm:trans;
    exempt Local:ds
    subsume Local:lg to dbLog;
    subsume Local:da to dbAcc;
    delegate dbSrv to Local:d;
};
```

Figure 2.5: *Example of a SOFA specification in CDL*

an event is emitted (requirement), absorbed (provides) or is an internal event. The event suffix expresses whether an event is a *request* (↑) or a *response* (↓) to an event request.

The Protocols

The *Frame* protocol of a component specifies the behaviour acceptable at the interfaces. It defines the interplay of method invocations on the provides-interfaces and reactions on the requires-interfaces of the frame. The frame protocol is given by the system's designer in CDL as the example shown in Figure 2.6.

```
frame Database {
    provides:
        IDBServer dbSrv;
    requires:
        IDatabaseAccess dbAcc;
        ILogging dbLog;
    protocol:
        !dbAcc.Open ;
            ( ?dbSrv.Insert { ( !dbAcc.Insert ;
                !dbLog.LogEvent )* }
            +
            ?dbSrv.Delete { ( !dbAcc.Delete ;
                !dbLog.LogEvent )* }
            +
            ?dbSrv.Query { !dbAcc.Query* }
        )*;
        !dbAcc.Close
    };
```

Figure 2.6: *Example of frame in Behavior protocols*

The *Architecture* protocol specifies the interplay on the method invocations on the interfaces of F and the outermost interfaces of the subcomponents in A. The architecture protocol is generated automatically combining the frame protocols of the subcomponents via a composition operator Π_X.

The *Interface* protocol defines the acceptable order of method invocations on an interface. It is intended to simplify a component design as it represents the behaviour of a component on a single interface only. The interface protocol is given in the CDL specification. Although the behaviour on the interface is also in the frame, it helps to check the correctness of the interface ties, though only partially.

Extended Behavior Protocols

The original specification of *Behavior Protocols* aimed at simplicity rather than expressivity. This is, the language was limited to traces given by regular expressions of events. Within the language, there were no parameters of any kind; moreover, support for control structures was hardly possible.

These issues lead to imprecise and hard to read specifications (under complex systems). Due to the lack of data, the *Behavior Protocols* must specify a superset of the potential implemented component behaviour. This means that the specification must admit impossible scenarios; the tools may detect errors that will never occur in any implementation, and the tools may not detect errors in some scenarios.

Moreover, *Behavior Protocols* lack of *multi-synchronisation*. This is, all synchronisations must be one-to-one, and the result of such synchronisation is a τ (invisible) action. Therefore, once synchronised, it is no longer possible to further synchronise on the same action.

These issues were addressed by an extension to *Behavior Protocols* called *Extended Behavior Protocols* (EBP) [Kofroň 07]. The EBP language incorporates basic parameters, control structures, and multi-synchronisations. Concretely, it includes local variables in the component specification parameters of method call requests, *switch* and *while* statements controlled by variables, and synchronisation of events from more than two EBP at a time. The variables are limited to finite enumeration types.

The structure of an EBP specification can be seen in Figure 2.7.

```
component componentName {
    types {
        types definition
    }
    vars {
        variable definition
    }
    behavior {
        behaviour definition
    }
}
```

Figure 2.7: *Structure of the EBP specification*

The semantics of EBP are defined in a formalism called *Nondeterministic Finite Automata with Assignment* [Kofroň 07], which is a formalism similar to *Symbolic Transition Graphs with Assignment* [Lin 96]. The drawback of EBP compared to *Behavior Protocols* is that it requires: either (i) a symbolic model-checker, or (ii) an instantiation of the parameters [§].

2.3.3 Sensoria

Sensoria [Sensoria 05] is another project which provides a mathematical framework for component interaction [Fiadeiro 06]. It targets Service Oriented Architectures (SOA) such as Web Services and SCA (Service Component Architecture [BEA Systems 05]). Their approach is akin with *Behavior Protocols*, specifying the allowed interaction within the system.

The language they provide has several operators defining how the interaction between components takes place. It is possible to define whether the communication is synchronous or asynchronous, in which instant the components are ready to initiate the interaction, and transactional communication. The kind of behavioural properties they seek for is in the domain of branching time logic with linear past [Goldblatt 87].

Sensoria is also involved with more powerful specification languages. The work on SCC (Service Centered Calculus) [Boreale 06] provides a formalism in the scope of π-calculus, extending the latter with higher-level primitives.

2.3.4 STSLib

STSLib [Fernandes 07] provides a formal component framework that synthesises components from symbolic protocols in terms of Symbolic Transition Systems (STS). STSLib can be seen as a formalism (over STS), as a component model, and as a formal framework for analysis.

STS relies on Algebraic Data Types (ADT) which are expressive formal data types. Communication of components is dealt with synchronisation vectors [Arnold 94] which allow one to encode a large set of synchronisation primitives. Moreover, they define a symbolic product of systems, which allow analysing the system without need of instantiation.

The communication in STS components is rather low-level for a component model; they exchange messages in which both emitter and receiver must agree to communicate, although there is no clear notion of required nor provided services.

STSLib also features a code generator for creating executable components. The ADTs are transformed into Java classes, and a coordinator class implements the STS protocol.

One difficulty in this approach is that the designer must specify their datatypes in ADT, i.e. in abstract formal formulas. Although they are very expressive, it is not aligned with the expertise of software engineers. Moreover, it is unclear how the generated classes (from the ADTs) can be modified while preserving the behaviour.

[§]In [Kofroň 07], the author refers the instantiation of parameters as *parameter unwinding*

2.3.5 UML 2

The *Unified Modeling Language* 2 (UML 2) [OMG 04] is one of the most widely used modelling language. UML tries to achieve compatibility with every possible implementation language. With this goal, much of the semantics of UML is undefined; it is possible, however, to associate a formalism and give a precise interpretation of the semantics.

UML provides thirteen types of diagrams, organised in three groups:

- *Structure diagrams*; these emphasise what things must be in the system being modelled.
- *Behaviour diagrams*; these emphasise what must happen in the system being modelled.
- *Interaction diagrams*; these emphasise the flow of control and data among the things in the system being modelled. They are a subset of the *behaviour diagrams*.

In particular, for this thesis we are mostly interested in the *Component diagrams* and the *Activity and State Machine diagrams*.

Component diagram. The *Component diagram* allows the system architect to give a high-level architecture of the system. It shows the relationship between software components, their dependencies and communication.

UML component diagrams feature both black box and white box views of hierarchical component systems. UML components communicate and synchronise through well-defined (provided and required) interfaces and connectors. The white-box view is used to specify the implementation of a component, in term of subcomponents and bindings.

As for all UML concepts, component diagrams are defined in a high-level and underspecified semantics (with semantics variation points), in order to deal with several component models. Also, UML 2 does not provide any methodology for using UML components, and adapting them to more concrete models.

Activity and State Machine diagram. The *State Machine diagram* models the behaviour of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.

In the black-box view, UML specifies that a protocol state machine can be attached to each component. This state machine defines the acceptable external behaviours of the component, and can be used to check the behavioural correctness of component assemblies. However, this requires to set a proper semantics for the component structures.

2.3.6 Other Non-Component Specification Languages

We describe here two languages widely used in the specification of distributed systems: LOTOS [ISO 89] and Promela [Gerth 97]. These languages provide standard means for defining formal processes that can be used as input in verification tools; CADP [Garavel 07] for LOTOS, and SPIN [Holzmann 03, Holzmann 97] for Promela for example. These languages are not meant to define components, but processes in general – in particular protocols.

We include LOTOS and Promela as reference to classic distributed system specification. LOTOS in particular provides a high-level description of processes that can be used as entry to model-checkers and can generate executable code.

2.3.6.1 LOTOS

The Language of Temporal Ordering Specification (LOTOS) is an ISO (International Standards Organisation) specification language for the formal specification of open distributed systems. It is based on the definition of the observable behaviour of a system, using definitions from process algebras.

There are different flavours of LOTOS, among the most used one *Basic LOTOS* and *Full LOTOS*.

Basic LOTOS. LOTOS is useful for defining the behaviour of concurrent systems. Each part of the system is a process that can perform internal observable actions, internal unobservable actions, and communicate / synchronise with the environment through gates. A communication on a gate is an action, that is trigerred if all processes synchronised on the gate of interest can perform the same action synchronously. In other words, if there are N processes synchronising on a gate, then all N processes must synchronise or none does. Synchronisation on a particular action includes agreeing on predicates (guards) on the action data parameters.

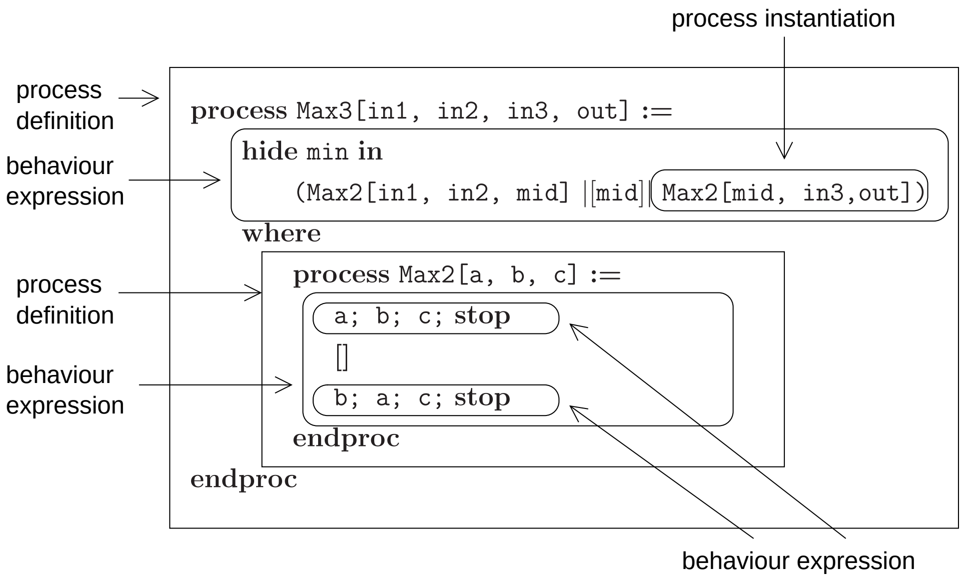


Figure 2.8: *Example of a process definition in LOTOS*

Example of LOTOS

An example of a process definition in LOTOS can be seen in Figure 2.8. A process `Max3` defines 4 gates (`in1`, `in2`, `in3`, `out`). The process `Max3` is composed of two instances of the same process `Max2`. We can see the instantiations of process `Max2` rename the gates (`a`, `b`, `c`), and both synchronise in the common gate `mid`. This gate is not seen outside `Max3` as the specification specifies that the gate is *hidden*. `Max2` is a process that performs either the sequence `a,b,c` or the sequence `b,a,c` and stops (the execution is halted).

The complete list of basic-LOTOS behaviour expressions is given in Figure 2.9, which includes all basic-LOTOS operators. Symbols `B`, `B1`, `B2` in the table stand for any behaviour expression. Any behaviour expression must match one of the formats listed in column **Syntax**.

Name	**Syntax**
inaction	**stop**
action prefix	
- unobservable (internal)	**i;**`B`
- observable	`g;B`
choice	`B1 [] B2`
parallel composition	
- general case	`B1` $\|[g_1, \ldots, g_2]\|$ `B2`
- pure interleaving	`B1` $\|\|\|$ `B2`
- full synchronisation	`B1` $\|\|$ `B2`
hiding	**hide** $g_1, \ldots, g_n$ **in** `B`
process instantiation	`p` $[g_1, \ldots, g_n]$
successful termination	**exit**
sequential composition (enabling)	`B1 >> B2`
disabling	`B1 [> B2`

Figure 2.9: *Syntax of behaviour expressions in LOTOS*

We describe here the main operators of LOTOS.

The choice operators can be deterministic or non-deterministic depending on the number of actions that can be triggered.

A process may be constituted of a hierarchy of processes; the global behaviour of the process is given by every possible interleaving of the internal processes. Synchronisation of two processes can be set as either visible or invisible actions using *hiding* operators.

Processes can stop (terminate), or not. Upon termination, a process may start or stop another process. The latter is the *disabling operator*, which can be used to interrupt a process. This operator is very useful for representing exceptions in programming languages: a process runs continuously in its "usual" behaviour but may be interrupted abruptly by some event.

Full LOTOS. In *Full LOTOS*, abstract data types (ADT) allows one to represent values, value expressions and data structures. ADT is derived from ACT ONE [Ehrig 85]; although ADTs give LOTOS a huge expressive power, they are too complex to be widely adopted by software engineers. The ADTs only define the essential properties of data and operations that any correct implementation (concrete data type) satisfies, thus one has to write formal equations defining the algebra for the data types.

Communication including data is somehow similar to value-passing, in which an emitter process performs an action with variables evaluated by expressions, and a receiver process performs actions offered with unbound variables. After the synchronisation, the variables in the receiver are bound to the values offered by the emitter. Strictly speaking, there are no emitter and receiver; instead, communication is set by an offer-agreement between processes. Moreover, this is done in such a way that different aspects of the system specification can be developed independently, and can be later synchronised by the intersection of the predicates over actions. Therefore, LOTOS provides a high-level representation of communication that can still be implemented by the LOTOS compiler.

Example of Full-LOTOS

An example of a Full LOTOS specification is found in Figure 2.10. Instead of a *process*, `Max3` is defined as a *specification*. However, it is still valid to talk about a process.

The example shows the ADT definition of "natural". The designer has to specify its sort, the operations, and then formal equations defining the abstract type. When the system size scale up, the specifications of these ADTs become one of the main drawbacks of Full LOTOS. The burdon of defining these ADTs are not akin with the goals of software engineers.

As far as pure synchronisation is concerned, this process has exactly the same behaviour as its basic LOTOS version (see Figure 2.8). However, subprocess `Max2` is now able to accept any pair of natural numbers at (formal) gates `a` and `b` , and offer the largest between them at gate `c`. Consequently, process `Max3` will accept three natural numbers at gates `in1`, `in2`, `in3`, in any order, and offer the largest of them at gate out.

```
specification Max3 [in1, in2, in3, out]:noexit
(* Defines a 4-gate process that accepts three natural numbers at three input gates,
in any temporal order, and then offers the largest of them at an output gate *)

type natural is
    sorts     nat
    opns      zero: → nat
              succ: nat → nat
              largest: nat, nat → nat
    eqns      ofsort nat
              forall x:nat
                  largest(zero, x) = x
                  largest(x, y) = largest(y, x)
                  largest(succ(x), succ(y)) = succ(largest(x, y))
endtype (* natural *)

behaviour
    hide mid in
              (Max2[in1, in2, mid] |[mid]| Max2[mid, in3, out])
where
    process Max2[a, b, c] : noexit :=
              a ?x:nat; b ?y:nat; c !largest(x,y); stop
              []
              b ?y:nat; a ?x:nat; c !largest(x,y); stop
    endproc (* Max2 *)

endspec (* Max3 *)
```

Figure 2.10: *Example of a specification in Full LOTOS*

LOTOS in Components. On the other hand, in this thesis we use LOTOS in a very different way. We encode method calls (offer / reception of values) within LOTOS through synchronisation. This way, the behaviour of components can be mapped as a special encoding into LOTOS. Supposing that primitive components are monothreaded, a simple encoding is as follows.

- A component is associated to a LOTOS process.
- The interfaces are communication points, so they can be mapped into gates.
- Remote method calls are expressed as synchronisation on gates. They are usually encoded as two messages, one calling the remote method, and another with the response value.
- A composite component is a process in which its external interfaces are gates seen by an external observer, subcomponents are subprocesses, and method calls on interfaces are communication / synchronisation between these components.
- A primitive component is a process without inner processes.

The approach that is taken in this thesis is in this vein. We shall provide the designer a high-level language that can be finally encoded in a formal language such as LOTOS.

2.3.6.2 Promela

Promela (for Process Meta Language) [Holzmann 03, Gerth 97] is another formal specification language. It is designed to describe distributed systems, using a C-like syntax and CSP notation.

Processes denote the concurrent activities within the system and can be created at any point. They communicate via message channels that implement synchronous or asynchronous communications. An asynchronous communication is held by buffers that store messages in a FIFO queue, whereas a synchronous communication channel is a rendez-vous between parties.

Statements from different processes are interleaved if they can be executed independently. It is also possible to define blocks of statements that are atomic.

Variables must be declared and their type defined. The types vary from bit, boolean, byte and interval of integers, arrays and records. New types can be defined based on these basic ones.

Statements are executable or blocked, meaning that the process will wait on a statement until it can be executed. Variables can be local or global, the global ones simulating shared-memory. These determine the synchronisations.

The models specified in Promela are non-deterministic finite state machines, therefore they can be efficiently model-checked [Holzmann 03, Holzmann 97].

2.4 Tools

We now review some tools that can be used to design and verify distributed systems. We start by describing explicit state model-checkers tools, then symbolic model-checkers.

Finally, we present some tools to design and verify distributed components. We will stress on tools related to the Fractal component model.

2.4.1 Finite Explicit-State Model-Checkers

CADP. The CADP [Garavel 07] is a toolbox for the design of communication protocols and distributed systems. It provides compilers for LOTOS, both basic and full LOTOS. The compilation generates C code that can be used for simulation, verification, and testing purposes.

The processes in LOTOS can also be transformed into explicit representations in CADP's internal formal of Labelled Transition Systems (LTS), called BCG.

CADP can also verify networks of processes, that are synchronised through synchronisation vectors [Arnold 94] which in CADP are expressed in EXP format [Lang 05]. This allows one to define many synchronisation operators, including all those in LOTOS. CADP supports comparison and reduction of LTSs, through various kinds of equivalences (such as strong bisimulation, observational equivalence, delay bisimulation, or τ^*a bisimulation, branching bisimulation, and safety equivalence) and preorder relations (such as simulation preorder and safety preorder).

The temporal logic used is called regular alternation-free μ-calculus, which is an extension of the alternation-free fragment of the modal μ-calculus [Kozen 85, Emerson 86] with action predicates and regular expressions over action sequences. It allows direct encodings of CTL [Clarke 99], ACTL [Nicola 90], and PDL [Fischer 79]. Moreover, it has an efficient model checking algorithm, linear in the size of the formula and the size of the LTS model.

SPIN. SPIN [Holzmann 03, Holzmann 97] (for Simple ProMeLa INterpreter) is a model-checker for the Promela [Gerth 97] language. Given a set of correctness claims and a system description, SPIN verifies whether or not those claims hold in the system. The system is specified in Promela, and claims are given in Linear Temporal Logic (LTL) [Pnueli 77, Manna 92].

Processes are translated into finite automata, and the global behaviour of the system is defined as the asynchronous interleaving product of all automata which is again an automaton. LTL formulas can be converted into Büchi automaton (as shown in [Vardi 86, Vardi 94]). Verification is then performed by computing the product between the global system behaviour and the Büchi automaton. If the language accepted by the resulting product is empty.

The tool supports on-the-fly verification for reachability properties in order to avoid constructing the full state space, and provides partial order reduction techniques. Besides being able to be used as an exhaustive verifier, SPIN can also be used to simulate systems. This allows designers to early prototype their systems, or to check some scenarios when the state space is too big to be verified by an exhaustive check.

DiVinE. DiVinE (Distributed Verification Environment [Barnat 06]) is the finite model-checker used by *Component-Interaction Automata*. DiVinE supports verification of properties specified in Linear Temporal Logic (LTL) [Manna 92]), or as processes expressing undesired behaviour (negative claims). It also supports distributed state-space generation, and algorithms for distributed model-checking of LTL formulas.

Verification is done with algorithms similar to those employed by the SPIN model-checker, however DiVinE provides distributed versions of the algorithms. DiVinE can also verify SPIN-compatible [Barnat 05] specifications written in Promela. Thus, as DiVinE allows for distributed verification, the authors [Barnat 05] claim that DiVinE is useful when SPIN runs out of resources.

2.4.2 Bounded Model-Checkers

Bounded model-checkers [Burch 90] unroll the finite state machines for a fixed number of steps and check whether a property is violated or not. The steps can be progressively increased, until the formula can be proved. In verification, the specifications can be verified using on-the-fly model-checking. This is supported by CADP in reachability formulas, and is the only example of bounded model-checking in CADP.

Bounded model-checking can be related to solving the satisfiability (SAT) problem [Cimatti 02], dated back to Davis and Putnam's earlier work in the 60s [Davis 60]. SAT is the problem of deciding whether the variables of a propositional formula can be assigned in such a way that the formula evaluates to true.

2.4.3 Infinite Systems Model-Checkers

There are model-checking tools that can work on symbolic representations of systems, and thus avoid unrolling the state machines. This allows for verification of infinite systems, where parameters are unbounded. We briefly comment on two of these tools.

FAST. Fast Acceleration of Symbolic Transition systems (FAST) [Bardin 03] performs automatic verification of systems augmented with (unbounded) integer variables. It uses acceleration techniques to compute the effect of iterating a control loop of arbitrary length, and uses heuristics to find automatically the good cycles to accelerate.

TReX. A Tool for Reachability Analysis of CompleX Systems (TReX) [Annichini 01] is able to generate the set of reachable configurations. As input, it accepts timed automata with parameters and counter variables communicating by shared variables and lossy channels. The automata communicate through unbounded lossy FIFO channels.

The tool performs reachability analysis on symbolic structures [Abdulla 98]. This allows for the representation of infinite sets of configurations, and it uses acceleration techniques for computing the (exact) effect of the iteration of control loops. Again, termination is not guaranteed, but the tool is also able to automatically find which cycles to accelerate.

2.4.4 Design and Specification Tools for Components

Fractal GUI. Fractal GUI [FractalGUI] is the first application released for designing Fractal components. Its main purpose is to provide a GUI for defining component types, component hierarchies and component bindings – basically, a GUI for handling the ADL.

Fractal GUI has built in code generation facilities. From the system specification, it generates the Fractal ADL descriptions, and Java code skeletons for primitive components. The skeletons include a rough implementation of Fractal's default non-functional controllers (life-cycle and binding controllers), and empty methods for each method defined in the component's server interfaces.

F4E. F4E (for Fractal For Eclipse) [F4E] is a recent project for designing Fractal components within Eclipse. It supports the standard Fractal DTD, and allows one to import / export standard ADL files. It provides a graphical layout of the component system, as well as a tree-based outline.

GIDE. GIDE (an IDE for the Grid) [GIDE] is currently the only graphical editor for the GCM. It supports the GCM DTD, and allows one to import / export GCM ADL files. Being based on GCM, it supports the definition of multicast and gathercast interfaces, though there is no support for designing the component's membrane. Up to version `2.10`, GIDE has adopted its own graphical notation for representing components.

GIDE also features the generation of Java skeletons. This is, however, limited to templates of the component without any control flow code as GIDE deals only with architectural specifications of components. Another interesting feature it provides is the ability to set a component repository. Moreover, GIDE provides runtime monitoring facilities of GCM components.

Wombat. Wombat [Martens 06] is an analysis tool for diagnosing SCA components. The formalism is based on Petri nets [Reisig 85]. Wombat allows for checking systems w.r.t. functional properties, in order to have an early compatibility check. It is mainly conceived for checking BPEL specifications of components.

A prototype of Wombat is available as a plugin to IBM's WebSphere Integration Developer (WID).

Topcased. Topcased [Pontisso 06] is an open-source environment primarily for the design of critical systems. It promotes model-driven engineering and formal methods as key technologies. One of the key elements is that it is fully integrated in the Eclipse Modeling Framework [Budinsky 04], and it applies model transformations to interface with various kinds of verification tools and formalisms.

Among the several tools found in Topcased, we highlight the editors for designing meta-models (AADL [SAE Standards 04], Ecore [Budinsky 04], and UML [OMG 04]); engines for model transformation (ATL [Jouault 06b]); and bridges towards model-checking engines (CADP [Garavel 07]).

Topcased can be used to design, simulate and analyse components. It allows one to use UML component diagram for an architecture specification, and endow the diagrams with the missing semantics. The behaviour can be given in different flavours, for example by activity diagrams or Petri Nets. It supports structured analysis, which is convenient to analyse nested systems including its control and data flow.

Turtle. Turtle [Apvrille 04] is a UML profile dedicated to the modelling and formal verification of real-time systems. It initially was not targeted at component systems. However, it can be extended to deal with the design of components such as we did in [Ahumada 07].

One of the strengths of this profile is its formal semantics. Indeed, all Turtle diagrams are first translated into an intermediate form called TIF - Turtle Intermediate Format – from which formal specifications in LOTOS can be generated. Moreover, the Turtle profile is supported by a (open source) toolkit named TTool, developed by the LabSoC laboratory from GET/ENST, including editors for various UML views of the systems, and code generators for interfacing with LOTOS and LOTOS-RT model-checking tools. It also supports code generation that is useful for system simulation.

2.5 Applications of Formal Methods in Components

The purpose of this section is to give the reader a general overview of some of the applications of formal methods in component development.

Detection of Errors. Detection of errors is the most classic applications of formal methods in components. One can check for deadlocks or interface behavioural compatibility [Allen 97, Magee 99, Plášil 02, Carrez 03, Reussner 03].

In [Adámek 04], SOFA extends the *Behavior Protocols* to capture faulty computations caused by a "bad" component composition. SOFA splits faulty computations in two categories: 1) At some point the computation cannot continue - *no continuation* error which includes two specific error types: *bad activity* and *no activity*; 2) A computation is infinite (*divergency* error).

Checking Equivalences. Component substitutability is an interesting application to components. The idea is to formally check whether a component can be replaced by another one. Behavioural substitutability in components has been addressed by Moisan et al [Moisan 03]. They build a formal framework and associate finite state machine as behaviour descriptions of components. Then, they formalise when a machine is a *safe extension* or the other, by defining a preorder relation $\leq$. If M and M' are finite state machines, then M' is a safe extension of M M' has a superset of the alphabet of M, and every sequence that inputs that is valid for M is also valid for M', and produces the same output once restricted to the alphabet of M.

Using CI Automata, the authors of [Černá 06] propose equivalences based on the set of observable actions. This allows the level of accuracy to be defined for each context, and then to use bisimulation to check for compatibility. The equivalence can be relaxed to check for implementations that provide more functionality than the one specified. The relaxed equivalences are then of great use to check for safe substitution of (trivially not identical) components.

Environment Construction. Another interesting aspect of component verification is how to check the component in a closed environment [Parízek 07a]. For that, one option is to manually define the test environment and then check for compatibility problems. Of course, we would like to have tools for creating such environment automatically. This is presented in [Parízek 07b]; the work is based on the construction of an environment by inverting the component's protocol, i.e. by computing all the valid sequences of calls that the environment could perform.

Refinement of the Specification. In *Behavior Protocols* (see Section 2.3.2), specifications can be refined in a top-down design. The idea is to substitute the top component by a refined version of the component, whose architecture behaviour is defined by combining the frame protocols of its subcomponents. This refinement is recursively repeated through the hierarchy until the inner-most primitive component whose architecture behaviour protocol is determined by its implementation. There is a relation called *protocol conformance* [Plášil 02, Adámek 03] that checks if an architecture protocol is compatible with the frame protocol. This relation allows for checking errors within the specifications. Moreover, it makes possible to automatically generate the architecture protocols based on the frame protocols of the subcomponents.

As a component implementation is a refinement of its specification, one can expect the implementation to provide additional services, and maybe to require others as well. Using *CI Automata* (see Section 2.2.4), [Černá 06] provides definitions to check whether an implementation is a refinement of a specification or not.

- The implementation provides (resp. requires) all the services provided (resp. required) by the specification.
- The implementation may provide (and require) services that are beyond the specification.
- When serving the services provided (and required) by the specification, the implementation respects the specification in all observable steps.

(iii) is the most interesting; it says that the implementation must require all services used in the specification in order to be a correct implementation. This is required because a third component may be expecting to be called.

Performance Prediction. Another trend of interest is performance prediction. Palladio [Becker 08] is a component model that allows the designer to analyse the system for testing designs w.r.t. response time of components. Their main focus is to deal with Quality of Service (QoS) requirements of designs. For that, Palladio uses stochastic process algebras, being useful for doing early analysis on possible bottlenecks, simulating the system, and so on.

Another example is Klaper [Grassi 08]. Klaper focuses on performance and reliability of designs, providing a performance model generator that can map results back to the specification.

Software adaptation. Even if software is well specified, it is often necessary to perform some adaption when composing components [Nierstrasz 95]. Different levels of interoperability can be defined [Canal 06]; signature, behavioural, semantic, and service level in increasing detail of compatibility. For each one of these, some adaptation may be required when composing components. The *signature level* refers to typical IDL specifications, signing the messages components may send and receive; therefore, a simple adaptation would map method names of the bound interfaces. The *behavioural level* extends the specification saying some information about the protocol of how these messages can be accepted. This requires one to define more complex criteria of what is a mismatch, the most common one being deadlock-freedom [Yellin 97]. The *semantic level* relates to what a component actually does, by taking into account not only the messages, protocols, but actually what what the functional behaviour is meant to do. The *service level* extends the previous levels by including as well all kinds of non-functional requirements a component such as QoS.

Software adaptation [Inverardi 01, Canal 06] promotes the use of adaptors to make mistmatching components work together. The adaptation can be performed at any of these levels, being automatic ones related to signature and behavioural levels [Bracciali 02, Inverardi 03, Canal 08].

3 Context

Contents

Abstract

This chapter describes the context of the thesis.

In the previous chapter, we have presented several component models, among them, the Grid Component Model (GCM). This will be the component model for the rest of the thesis. Therefore, this chapter complements the previous discussion.

We also present its reference implementation based on the ProActive middleware that provides the facilities to deploy components in Grids. Finally, we position our work w.r.t. the GCM.

3.1 The Grid Component Model

The GCM [CoreGRID 06] is a component model that extends Fractal to deal with Grids. The main characteristics the GCM benefits from Fractal are its hierarchical structure, the enforcement of separation of concerns, its extensibility, and the separation between interfaces and implementation.

The components are distributed among the Grid, therefore, they must deal with latency. Concretely, GCM components are loosly-coupled. Their granularity is somehow in the middle between small grain *Fractal* components and very coarse grain component models, like *CORBA Component Model (CCM)* [OMG 05] where a component is of a size comparable to an application.

Also, extensions to Fractal are related to communication and distribution. The GCM doesn't impose fixed communication semantics (e.g., streaming, file transfer, event-based). However, for dealing with latency, most GCM frameworks will probably prefer some kind of asynchronous communications. Communication between components always goes through its membrane, which is also in charge of the control of the component.

Collective communication is of great interest in the GCM, and in this sense, the GCM defines *many-to-one* and *one-to-many* communications. Their purpose is to optimise communications in a *large-scale* environment, and is useful for distribution and synchronisation of data. Contrary to other component models, this is tightly related to the infrastructure in which components will be deployed; Grids are constituted of thousands machines and the design of applications must take this into account.

We will structure the following of the section as follows. In Section 3.1.1 we will show can how define GCM components. In Section 3.1.2 we will show how GCM deals with collective communications. Then, in Section 3.1.3 we will show how the membrane can be structured, which will derive us with limitations in the current GCM ADL definition discussed in Section 3.1.4.

3.1.1 Architecture Description Language

The architecture in both Fractal and GCM is defined by an Architecture Description Language (ADL). It is an XML-based format, defined by a DTD, that contains both the structural definition of the system components (subcomponents, interfaces and bindings), and some deployment concerns. Deployment relies on *virtual nodes* (VN) that are an abstraction of the physical infrastructure on which the application will be deployed. The ADL only refers to an abstract architecture, and the mapping between the abstract architecture and a real one is given separately as a deployment descriptor. In this thesis, we don't take into account the infrastructure because we rely on the middleware for that. We consider that a component is a unit of distribution, and that the component may latter be mapped to the infrastructure.

The ADL allows the designers to describe component types, component implementations, component hierarchies and component bindings. Basically, the information found in an ADL allows one to describe a component by its interfaces, subcomponents, bindings, and possibly the class that implements its content. The interfaces are signed using yet another language, so-called Interface Description Language (IDL). For the implementations we are interested in, the IDLs are simply Java interfaces that sign an interface with their set of methods.

Both composite and primitive components are defined by an ADL, either in the form of a hierarchical definition with the complete system, a separate definition for each component, or a mixture of both. We briefly exemplify an ADL for the system in Figure 3.1. The ADL of a primitive component is described in Figure 3.2, and the ADL of a composite component is described in Figure 3.3. The composite component (Figure 3.3) shows 2 flavours of subcomponents definitions. The first one, *PrimitiveOnLeft*, is a reference to an external definition, whereas the second, *PrimitiveOnRight*, is defined inline in a hierarchical fashion.

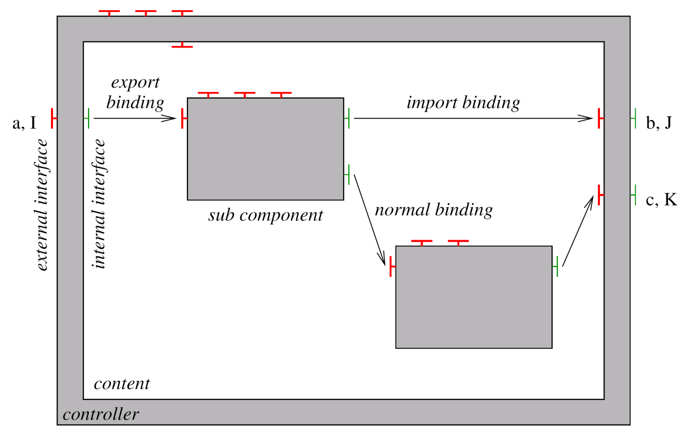


Figure 3.1: *A composite component as defined in Fractal Specification*

```
<definition name="PrimitiveOnLeft">
  <interface name="a" role="server" signature="I"/>
  <interface name="b" role="client" signature="J"/>
  <interface name="d" role="client" signature="L"/>
  <content class="PrimitiveOnLeftImpl"/>
</definition>
```

Figure 3.2: *ADL of* PrimitiveOnLeft *(a primitive component)*

```
<definition name="Composite">
  <interface name="a" role="server" signature="I"/>
  <interface name="b" role="client" signature="J"/>
  <interface name="d" role="client" signature="L"/>
  <component name="client" definition="PrimitiveOnLeft"/>
  <component name="PrimitiveOnRight">
    <interface name="c" role="client" signature="K"/>
    <interface name="d" role="server" signature="L"/>
    <content class="PrimitiveOnRightImpl"/>
  </component>
  <binding client="this.a" server="PrimitiveOnLeft.a"/>
  <binding client="PrimitiveOnLeft.b" server="this.b"/>
  <binding client="PrimitiveOnLeft.d" server="PrimitiveOnRight.d"/>
  <binding client="PrimitiveOnRight.c" server="this.c"/>
</definition>
```

Figure 3.3: *ADL of* Composite *(a composite component)*

3.1.2 Supporting M to N Communications

To meet the specific requirements and conditions of Grid computing for multiway communications, *Multicast* and *gathercast* interfaces give the possibility to *manage a group of interfaces as a single entity*, and *expose* the collective nature of a given interface. Multicast interfaces allow method invocation and their parameters to be distributed to a group of destinations, whereas, symmetrically, gathercast allow method invocations toward the same destination to be synchronised.

Multicast Interfaces: 1 to N Communications

Multicast interfaces provide abstractions for one-to-many communication. Multicast interfaces can either be used internally to a component to dispatch an invocation received by the components to several of its sub-entities, or externally to dispatch invocations emitted by the component to several clients.

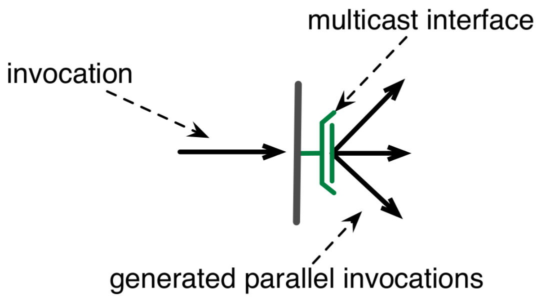


Figure 3.4: *Multicast interfaces*

A single invocation on a multicast interface is transformed into a set of invocations. These invocations are forwarded to a set of connected server interfaces (Figure 3.4). The semantics concerning the propagation of the invocation and the distribution of parameters are customisable. The result of an invocation on a multicast interface - if there is a result - is a list of results. Invocations on the connected server interfaces may occur in parallel, which is one of the main reasons for defining this kind of interface: it enables *parallel invocations*.

Typical examples of multicast are:

- *broadcast*: send the same message with the same parameters to each of the *N* server interfaces;
- *select*: send a single message to one of the connected server interfaces;
- *round-robin*: the argument is split, and one call is made with each piece of the argument (those messages are distributed in a cyclic way);
- *scatter*: same as round-robin except that the argument is necessarily split into *N* pieces and thus each server interface receives a single message.

Gathercast interfaces: M to 1 Communications

Gathercast interfaces provide abstractions for many-to-one communications. Both gathercast and multicast interface definitions and behaviours are symmetrical [Badrinath 00].

Gathercast interfaces can either be used internally to a component to gather the results of several computations performed by several sub-entities of the component, or externally to gather and synchronise several invocations made toward the component.

Gathercast interfaces gather invocations from multiple source components, and invoke a single call on the bound interface (Figure 3.5). A gathercast interface coordinates incoming invocations before continuing the invocation flow. The default behaviour is to synchronisation barriers, gather incoming data, and redistributed the return values to the invoking components. However, just as in multicast interfaces, this behaviour can be customised.

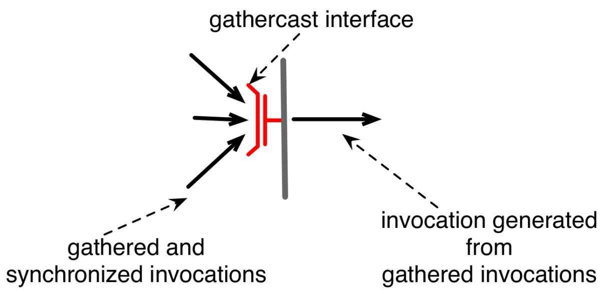

gathercast interface
gathered and synchronized invocations
invocation generated from gathered invocations

Figure 3.5: *Gathercast interface*

3.1.3 Structuring the Membrane

Components running in dynamic environments need to adapt in order to cope with the needs. In Fractal and GCM (Grid Component Model) component models, adaptation mechanisms are triggered by the non-functional (NF) [*] part of the components. The NF part, called the *membrane*, is composed of *controllers* that implement the NF concerns. Instead of an arbitrary implementation of the membrane, it is possible to apply the same structuring techniques inherited from component-oriented designs. Such a structure is presented in [Baude 07] for the GCM, and is highly inspired by the work done in AOKell [Seinturier 06]. In GCM, however, the NF components are considered as full-fledged components, having an active object of its own.

Example

Figure 3.6 considers a naïve solution for securing communications. The secure communications are implemented by three components inside the membrane: *Interceptor*, *Decrypt*, and *Alert*. First, the incoming messages are intercepted by the *Interceptor* component. It forwards all the intercepted communications to *Decrypt*, which can be an off-the-shelf component (written by cryptography specialists) implementing a specific decryption algorithm. The *Decrypt* component receives a decryption key through the non-functional server interface of the composite (interface denoted by number 1 in Figure 3.6). If it succeeds to decrypt the message, the *Decrypt* component sends the message to the internal functional components, using the functional internal client interface (interface denoted by number 2 in Figure 3.6). If a problem during decryption occurs, the *Decrypt* component sends a message to the *Alert* component. The *Alert* component deals with the decryption failures, for example, it can demand the sender to resend the message. This is done using the non-functional client interface (interface denoted by number 3 in the Figure 3.6).

3.1.4 Limitations of the GCM ADL

In the current state of the GCM ADL definition, there are some limitations of what can be expressed. The language is not powerful enough to define the full set of features found in the GCM. The ADL can certainly be extended to cope with most of the limitations listed here,

[*]In other contexts, NF refers to other kinds of concerns, like security, QoS, performance. Some of these could be addressed here, but in the GCM the usual controllers are those dealing with the component life-cycle, the reconfiguration of the content, and by extension all "autonomic" features (self-healing, self-optimising, etc).

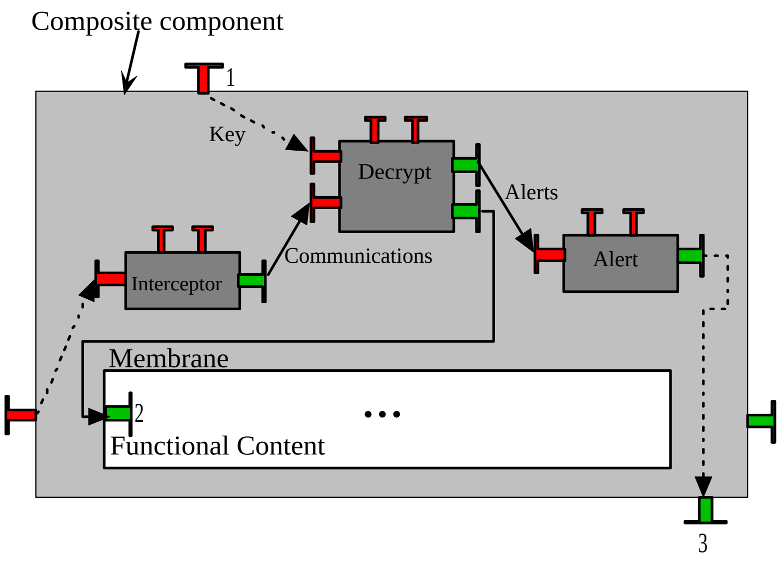


Figure 3.6: *A composite membrane implemented by components*

however this would require modifying at least the ADL parser within the middleware which is not the interest of this thesis. We enumerate the limitations concerning our model.

1. There are no component types and instances. A component definition in the ADL is at the same time a definition and a runtime instance. In other words, there is no such thing as a component type and instance of a type, though it can be somehow "simulated": a component type is defined in a file, and instances of this type reference the file within the "`definition`" attribute.

2. It is not possible to define internal interfaces. All interfaces in the ADL are considered as external interfaces. This is a major flaw in the GCM ADL as it is not possible to define *interceptors* explicitly for the same reason.

3. Non-functional components in the membrane are not standard.

Even if in [Baude 07] the authors proposed an extension to the ADL that handles a broader set of features, it is not yet standardised. Moreover, the proposition [Baude 07] is still unable to express internal interfaces, and interceptors; therefore, it will certainly be modified before reaching a standard.

3.2 GCM/ProActive

The reference implementation of the GCM is built upon ProActive as presented in [Baduel 06]. We review first how ProActive works (in the frame of active objects), and then relate to the implementation of the GCM.

3.2.1 ProActive

ProActive [Caromel 06a] is a pure Java implementation of distributed active objects with asynchronous remote method calls and replies by means of future references. A distributed application built using *ProActive* is composed of several activities, each one having a distinguished entry point, the active object, accessible from anywhere. All the other objects of an activity (called *passive objects*) can not be referenced directly from outside. Each activity owns its own and unique service thread and the programmer decides the order in which requests are served (or not). Each activity has a pending queue where the incoming requests are dropped. Requests are asynchronous method calls addressed to the active object, and should be served by the service thread. The requests are sent using a *rendez-vous* phase so there is a guarantee of delivery and of order between incoming calls. During the rendez-vous a future (reference to the future result) is created on the sender side thus allowing asynchrony. The responses are always asynchronous; their synchronisation is done by a mechanism called *wait-by-necessity*.

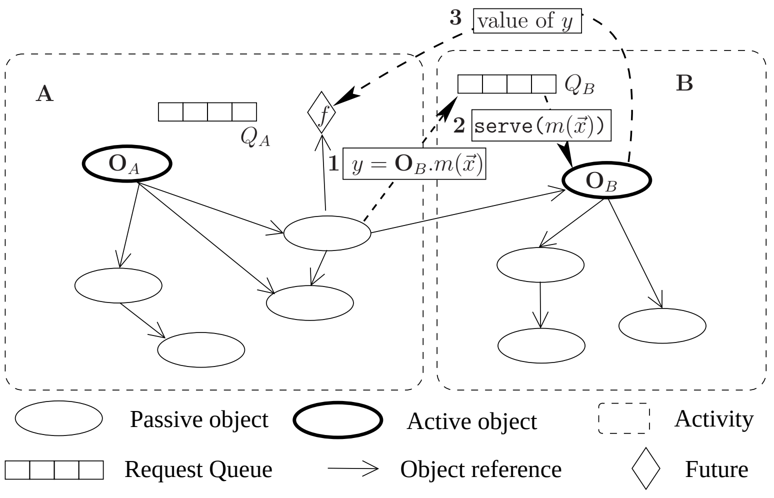


Figure 3.7: *Example of two activities communicating in* ProActive

Figure 3.7 shows an example consisting of two activities, each one having a single active object (entry points O_A and O_B) and a set of passive objects.

The method calls to active objects behave as follow:

1. When an object makes a method call to an active object ($y = \mathbf{O}_B.m(\vec{x})$), the call is stored in the request queue of the called object (Q_B) and a future reference is created and returned (y references f). A future reference encodes the promised return value, i.e. the result not yet available of a method call to an active object.

2. At some point of the execution, the called activity decides to serve the method call (`serve(`$m(\vec{x})$`)`). The request is taken from the queue and the method is executed.

3. Once the method finishes, its result is updated, i.e. the future reference (f) is replaced with the concrete method result (`value of` y).

When a thread tries to access a future reference before it has been updated with the concrete real value, it is blocked until the update takes place (*wait-by-necessity* mechanism).

A developer can specify the policy on how to chose the requests to serve from the queue. In practice this is done by implementing the `runActivity` method which is executed as soon as the activity starts. The *ProActive* API provides several versions of `serve` methods (such as blocking/unblocking serve, FIFO/LIFO order or based on queue filters among others). When `runActive` is not provided by the user, the *ProActive* middleware implements a default FIFO policy.

The ASP-calculus [Henrio 03, Caromel 05b] has been defined to provide a model for the *ProActive* library and the aspects presented above. The most notable results that are used in this thesis are the ones related to the future update. Concretely, in it proved in ASP that the different update strategies have equivalent behaviours. In other words, ASP proves confluence of future update. Therefore, the different future update strategies have only effects in the system performance. Another important aspect of the implementation is that there is no shared memory. This is central to maintain the confluence properties on future updates coming from ASP-calculus.

3.2.2 ProActive's implementation of the GCM

ProActive provides the reference implementation of the GCM. Components become active in the same way than ProActive's active objects: their membrane has a single non-preemptive control thread which serves, based on different serving policies, method requests from its unique pending queue. In fact, a composite component is only a dispatcher of services: requests from the environment to its external server interfaces (including control requests) are dispatched to inner components; similarly, calls coming from the content through its internal client interfaces are dropped to the component's request queue, and will be later dispatched towards the environment.

A graphical view of a composite is shown in Figure 3.8.

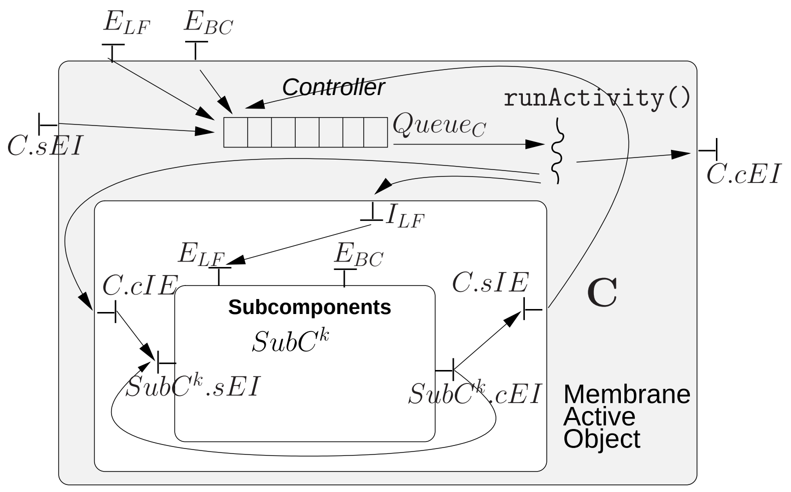


Figure 3.8: *Implementation of a composite component in* ProActive

In the case of a primitive component, there is a unique active object that serves the requests. A primitive component can be seen as an active object in which the communication with the environment is defined through interfaces. Moreover, a primitive component is monothreaded.

In the case of a composite component, there is a unique active object in the component membrane that serves the requests, and dispatches the requests to the bound interface related to the service. The policy of how to serve requests is, however, always FIFO for composite components. The idea is to minimise the complexity of the component by leaving composite components as composing agents.

To sum up, requests to other components are method calls on client interfaces, implemented via a rendez-vous protocol. Therefore, there is a delivery guarantee, and a order conservation of incoming calls. The responses (when relevant) are always asynchronous with replies by means of future references; their synchronisation is done by a wait-by-necessity mechanism.

3.2.3 Discussion on GCM Components and Futures

Futures are not in components only for convenience. In fact they play a major role in allowing for hierarchical composition. The primitive components in GCM are monothreaded, meaning there is a single thread that processes calls in the component's request queue. Because of that, without futures the system would systematically deadlock if the topology of request calls is not a DAG (Directed Acyclic Graph). Concretely, a request call could not trigger an outgoing request, i.e. if-ever a request call enters and leaves a component, the system will block. Futures also allow further concurrency without introducing explicit synchronisation primitives.

If communications occurring over the bindings are synchronous, i.e. the caller component will immediately block until the request call is treated, then the interfaces can be accessed as usual objects, having methods with parameters and a return type. When components are connected asynchronously, one must find a way to create a channel for the objects returned by the components. Futures can be used as identifier of the asynchronous invocations over components. Indeed, futures provide some kind of transparent channels that correspond to the original bindings, but taken in the opposite direction: from the server to the client.

Components as an Abstraction for Distribution. Components relieve us from a difficult analysis task: in a distributed object-oriented language with implicit futures, it is difficult to identify the communication and the creation points of futures. Indeed, asynchronous method calls are syntactically similar to local ones, and distinguishing one from the other can only be the result of a static analysis step which is by nature imprecise, consequently identifying the points where futures are created is also difficult. In a distributed component model like the GCM, however, the only method invocations that are asynchronous are the ones performed on interfaces. The topology of distribution and communication is directly given by the component structure.

Unfortunately, although the component model provides a good abstraction for distribution and specifies which calls are asynchronous, the flow of futures is still hard to approximate. In other words, the component abstraction tells us where futures are created but not where they can go.

The dynamic and transparent nature of futures implies that each result and each parameter of an invocation may contain a future; thus the only safe assumption for parameters and results is that any object received can be a future, and every field of this object can itself be a future. This leads to a very imprecise approximation of the synchronisation in the system; this over-approximation can always be improved by static analysis (when the system is closed), or by specification, as we will propose in Section 5.5.

3.3 Evaluation of Specification Languages

Although some of the specification languages found in the literature address the formal specification of components, they are not adequate to the specification of GCM components. Most specification languages describe the component behaviour as the events performed by the components. For instance this is the case for Behavior Protocols, Java/A and ArchJava. If we try to apply similar techniques to GCM components, the specification will end up being a complex mesh of events. Now we detail some of the issues in current specification languages when in comes to the GCM.

Architecture Specification. The architecture in GCM extends from Fractal, and thus has to deal with non-functional aspects. We do not see many difficulties in adapting current approaches to the GCM architecture specification, though we do have to provide new operators for dealing with the non-functional components and collective communications (one-to-many and many-to-one).

Moreover, we find a good idea to use the same language to define the architecture and the behaviour as done in Java/A and ArchJava. Having architectural primitives in the language is the basis for allowing the behaviour to reconfigure the application (in autonomic computing for instance), or to express user-given reconfiguration on-demand which are not available in Java/A though partially in ArchJava.

Performing remote method calls. In Behavior Protocols we can easily express remote method calls as an event, however, care must be taken with the future result. There will be an event for receiving the result value and another (local) event accessing the future. As the result can arrive at any time, this event would be interleaved with many other events that may be performed by the component, and therefore it is complex.

If we take the work on Java/A and ArchJava, similar problems could be cited. Communications are not meant to be asynchronous, and there is no high-level mechanism to deal with the reception of the value.

Non-component specification languages such as LOTOS and Promela would even aggravate these issues. They do not provide language primitives to refer to interfaces and method calls. These require a manual encoding given by the designer, which is error-prone and complex.

Data-driven synchronisation model. The GCM reference implementation, GCM / ProActive, is based on a data-driven synchronisation model. Moreover, as futures can be transmitted, components may have to synchronise with components to which they are not bound (but share a computation path). We found no specification language that could be used in this aspect. Again, we would require a manual encoding of this behaviour that is not aligned with our goals.

High-level view of the component behaviour . Distributed components such as the ones in GCM are usually loosely coupled and their behaviour is characterised by services offered to the environment. It is convenient, therefore, to have a specification language that provides one the with an abstraction of the services offered by the component and afterwards, once the designer has decided to use one of the services, the details of this service of interest.

We could not find such approach in any of the specification languages in the literature. The closest is the use of sub statemachines in UML where the behaviour can be detailed separately. Similar approaches can be taken in LOTOS and Promela, though these must be handled by the designer as there is no given structure organising the behaviour specification.

Syntax and Semantics. Another important aspect is that usually specification languages require some expertise in formal methods. In Behavior Protocols, as the language is based on regular expressions we can expect the software engineer to rapidly adopt the formalism. On the contrary, the Algebraic Data Types found in STSLib are too complex and will hardly be known beforehand by the software engineer.

The solution provided in Java/A and ArchJava seems promising. The use of an augmented Java compiler does provide the software engineer with "natural" syntax. However, on one hand Java/A is limited to the expressive power of LTSs which we believe is insufficient to easily describe the behaviour of distributed components (because of the problems related to asynchronous communications described above). On the other hand, being based on the π-calculus, ArchJava is complex and exposes formalisms that are not necessarily mastered by our target users.

The non-component specification languages LOTOS and Promela are both high-level formalism, but cannot be considered specification languages for components. They are highly influenced by process algebras and thus are quite distant from usual programming languages.

Model and Code Generation. We believe that a good strategy for developing components and hidding the complexity of formal methods is through synthesis of safe-by-construction components. That is, to first define their behaviour, to verify it against the user requirements, and then to generate code with guaranteed behaviour. These steps require one to generate the behavioural models for the specification and afterwards to generate runtime code.

This approach is close to the approach used in STSLib though they do require software engineers to master formal methods. On the contrary, Behaviour protocols does not envision the generation of safe-by-construction code. Instead, they rather check whether an implementation conforms to its specification. In Java/A and ArchJava the approach is to use a modified Java compiler to use the same sources for both verification and runtime, though only a small part of the code is used for verification.

3.4 Positioning

Most difficulties in the specification of GCM components come when specifying the interaction between futures, synchronisations, and the request queue. The transparent futures alleviate the programmer from synchronisation difficulties from a programming language point of view. Nevertheless, specifying and/or inferring about the synchronisations is complex. The use of traditional specification languages would require a manual encoding of the behaviour of these, meaning that the designer would need to specify:

- w.r.t. the services provided by a component:
  - how are the services provided;
  - what does a service do, i.e. how the service affects the control and data flow of the component system.
- w.r.t. futures:
  - which variables are futures;
  - when these futures are updated (by explicit encoding of an update event);
  - when access to a future is blocking or non-blocking (by tracking the first access to the future);
  - which method arguments contain futures, and how they affect the callee;
- w.r.t. request queue:
  - how the component synchronises with its request queue.

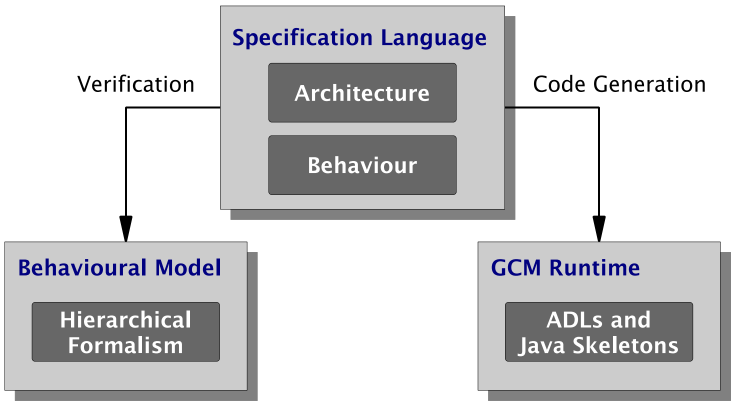


Figure 3.9: *Positioning of the Framework*

The goal of this thesis is to develop a framework on which designers may rely for specifying, verifying, and prototyping the system (see Figure 3.9) For specifying the system, we will develop a novel specification language, designed in such a way that we will still be able to verify the system, and to prototype the system. Because of these objectives, we will work on a formalism that allows us to define behavioural models that can be created from the specification of GCM components. Finally, for prototyping the system, we will work on code generators that will provide skeleton code for GCM components, starting from their specification.

4 Theoretical Model

Contents

Abstract

This chapter presents the formalism used in this thesis. It is a formal and parametric behavioural model called *pNets*. This formalism is an extension of synchronisation vectors [Arnold 94] to deal with value-passing, families of processes. The pNets formalism was previously presented in the theses of Boulifa [Boulifa 04] and Barros [Barros 05]. In this thesis, however, we formalise pNets, and use this later in Chapter 6 during the behavioural model generation of GCM components.

Motivation

In this chapter we give the formal definition of our intermediate language that we call *parameterized Networks of Synchronised Automata (pNets)*. This language is not a new calculus in the tradition of theoretical computer science that gave birth to λ-calculus, π-calculus, or σ-calculus, on which we would build new theories or new languages; nor is it a new process algebra endowed with syntax, semantics, and equivalences, that could be used to study new constructs for distributed computing. Rather, pNets give an intermediate and generic formalism intended to specify and synchronise the behaviour of a set of automata. We built this model with two goals: give a formal foundation to the model generation principles that we developed for various families of (distributed) component frameworks, and build a model that would be more machine-oriented and serve as a versatile internal format for software tools. This meanss that it must be both expressive (from the universality of synchronised LTSs) and compact (from the conciseness of symbolic graphs).

The synchronisation product introduced by Arnold & Nivat [Arnold 94] is both simple and powerful, because it directly addresses the core of the problem. One of the main advantages of using its high abstraction level is that almost all parallel operators (or interaction mechanisms) encountered so far in the process algebra literature become particular cases of a very general concept: synchronisation vectors. We structure the synchronisation vectors as parts of a *synchronisation network*. Contrary to synchronisation constraints, the network allows dynamic reconfigurations between different sets of synchronisation vectors through a *transducer* LTS. Our definition of the synchronisation product is semantically equivalent to the one given by Arnold & Nivat.

At a next step, we use Lin's [Lin 96] approach for adding parameters in the communications events of both transition systems and synchronisation networks. These communication events can be guarded with conditions on their parameters. Our agents can also be parameterized to encode sets of equivalent agents running in parallel. This leads us to the definition of pNets, that will later appear as a natural model of software systems. Indeed they correspond to the way developers specify or program these systems: the system structure is parameterized and described in a finite way (the code is finite), but a specific instance is determined at each execution, or even varies dynamically.

The results of this chapter were published in [Barros 08].

4.1 Networks of Synchronised Automata

We now give the formal definitions of the model in two steps. In order to uniform notations, we first define LTSs, Nets, and synchronisation product; these definitions are equivalent to those found in the literature. Then we give the definitions of our parameterized structures (pLTS and pNet), and of their instantiations; their semantics are in terms of standard (infinite) LTS.

Notations

In the following definitions, we extensively use indexed structures (maps or vectors) over some countable indexed sets. The indexes will usually be integers, bounded or not. When this is not ambiguous, we shall use abusive vocabulary and notations for sets, and typically write "indexed set over J" when formally we should speak of multisets, and still better write "mapping from J to the power set of $\mathcal{A}$".

We use uppercase letters A,B,I,J,... to range over sets, and lowercase letters a,b,i,j,... to range over elements of the sets. We write $\tilde{A}_J$ for an indexed multiset of sets ($\tilde{A}_J = \langle A_j \rangle_{j \in J}$), and $\tilde{a}_J$ for an indexed multiset of elements ($\tilde{a}_J = \langle a_j \rangle_{j \in J}$), where J can possibly be infinite. For indexed sets of elements or sets, we say $\tilde{a}_J = \tilde{b}_I \Leftrightarrow J = I \wedge \forall j \in J, a_j = b_j$ (element-wise equality). We write $\langle a.\tilde{a}_J \rangle$ for the concatenation of an element a at the beginning of an indexed set, $\tilde{x}_J = \tilde{e}_J$ for an indexed set of equations ($\langle x_j = e_j \rangle_{j \in J}$), $e\{\tilde{x}_J \leftarrow \tilde{e}_J\}$ for the parallel substitution of variables $\tilde{x}_J$ by expressions $\tilde{e}_J$ within expression e.

As part of our abusive notation, we extensively, and sometimes implicitly, use the following definition for indexed set membership: $\tilde{a}_J \in \tilde{A}_J \Leftrightarrow \forall j \in J, a_j \in A_j$. Cartesian product is naturally extended to indexed sets so that the following is verified: $a_0 \in A_0 \wedge \tilde{a}_J \in \tilde{A}_J \Rightarrow \langle a_0.\tilde{a}_J \rangle \in \prod_{j \in \{0\} \cup J} A_j$

We use the usual notions from (typed) term algebras: *operators*, *free variables*, *closed* and *open terms*, etc. Term algebras are endowed with a type system, that include at least a distinguished *Boolean* type and an *Action* type.

We write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \rightarrow$.

4.1.1 Labelled Transition System

We model the behaviour of a process as a Labelled Transition System (LTS) in a classical way [Milner 89]. The LTS transitions encode the actions that a process can perform in a given state.

DEFINITION 5 (LTS)
A **LTS** *is a tuple* $(S, s_0, L, \rightarrow)$, *where*

- S *(possibly infinite) is the set of states,*
- $s_0 \in S$ *is the initial state,*
- L *is the set of labels,*
- $\rightarrow$ *is the set of transitions:* $\rightarrow \subseteq S x L x S$.

4.1.2 Synchronisation Network

We define **Nets** in a form inspired by [Arnold 94], that are used to synchronise a (potentially infinite) number of processes.

DEFINITION 6 (NETWORK OF LTSS)
Let Act be an action set. A **Net** *is a tuple* $\langle A_G, J, \tilde{O}_J, T\rangle$, *where*

- $A_G \subseteq Act$ *is a set of global actions,*
- J *is a countable set of argument indexes,*
  - *each index* $j \in J$ *is called a* hole *and is associated with a* sort $O_j \subset Act$
- T *is the transducer LTS* $(S_T, s_{0T}, L_T, \rightarrow_T)$, *and*
  - $L_T = \{\overrightarrow{v} = \langle a_g.\tilde{\alpha}_I\rangle.\ a_g \in A_G,\ I \subseteq J \wedge \forall i \in I, \alpha_i \in O_i\}$

Explanations

Nets describe dynamic configurations of processes, in which the possible synchronisations change with the state of the Net. We call *transducer* the additional process in the Net in charge of controlling and synchronising the others, in a sense similar to the Lotomaton expressions [Lakas 96, Najm 92]. Lotomaton has been used to give the operational semantics of (open) LOTOS expressions.

A transducer in the Net is encoded as a LTS which labels are synchronisation vectors ($\overrightarrow{v}$), each describing one particular synchronisation between the actions (α_I) of different argument processes, generating a global action a_g. Each state of the transducer T corresponds to a given configuration of the network in which a given set of synchronisations is possible. Some of those synchronisations can trigger a change of state in the transducer leading to a new configuration of the network; that is, it encodes a dynamic change on the configuration of the system.

We say that a Net is *static* when its transducer contains only one state. Note that each synchronisation vector can define a synchronisation between one, two or more actions from different arguments of the Net. When the synchronisation vector involves only one argument, its action can occur freely.

DEFINITION 7 (SYSTEM)
A **System** *is a tree-like structure which nodes are* **Nets**, *and leaves are* **LTSs**. *At each node a partial function maps holes to corresponding* **subsystems**. *A system is* **closed** *if all holes are mapped, and* **open** *otherwise.*

DEFINITION 8 (SORT)
The **Sort** *of a system is the set of actions that can be observed from outside the system. It is determined directly by its top-level structure:*

- *L for a LTS: $Sort(S, s_0, L, \rightarrow) = L$,*
- *A_G for a Net: $Sort(\langle A_G, J, \tilde{O}_J, T\rangle) = A_G$.*

As this is often the case in process algebras, sorts here are determined statically as an explicit component of the structure, and are upper approximations of the set of actions that the system can effectively perform. The precision of this approximation depends naturally on the specific model generation procedure, but in most cases an exact computation is not possible.

4.1.3 Building hierarchical Nets

A Net is a generalised parallel operator. Complex systems are built by combining LTSs in a hierarchical manner using Nets at each level. There is a natural typing compatibility constraint for this construction, in term of the sorts of the formal and actual parameters. The standard compatibility relation is Sort inclusion: a system *Sys* can be used as an actual argument of a Net at position j only if it agrees with the sort of the hole O_j ($Sort(Sys) \subseteq O_j$). Here also, the compatibility relation may depend on the language or formalism that is modelled; for example if actions represent Java-like method calls, the compatibility could take into account sub-typing.

"Adaptation" of the Sort could also be easily implemented as an intermediate Net, containing only synchronisation vectors with a single active argument. This can express any kind of mapping between Sorts, and especially any combination of relabelling and hiding.

Our behavioural objects being LTSs, and Nets being operators over LTSs, it is natural to give their semantics in terms of products over LTSs. The definition of the *synchronisation product* below conversely, defines the LTS representing any closed Net expression, computed in a bottom-up manner. It is also possible to define a *symbolic* product over Nets that would reduce any *open* Net expression to a single Net, in the spirit of [Lakas 96], but this is not necessary for our goals.

DEFINITION 9 (SYNCHRONISATION PRODUCT)
Given an indexed set $\tilde{P}_J$ of LTSs
$\tilde{P}_J = (\tilde{S}_J, \tilde{s}_{0J}, \tilde{L}_J, \tilde{\rightarrow}_J)$, and a Net $\langle A_G, J, \tilde{O}_J, T = (S_T, s_{0_T}, L_T, \rightarrow_T)\rangle$, such that $\forall j \in J,\ L_j \subseteq O_j$, we construct the ***product LTS*** *$(S,\ s_0,\ L,\ \rightarrow)$, where*

- *$S = \prod_{j \in \{T\} \cup J} S_j$,*
- *$s_0 = \langle s_{0_T}.\tilde{s}_{0_J}\rangle$,*
- *$L \subseteq A_G$,*
- *and the transition relation $\rightarrow$ is defined as:*

$$s \xrightarrow{l_t} s' \Leftrightarrow \begin{pmatrix} s = \langle s_t.\tilde{s}_J\rangle \wedge\ s' = \langle s'_t.\tilde{s}'_J\rangle \wedge \\ \exists s_t \xrightarrow{\langle l_t.\tilde{\alpha}_I\rangle} s'_t\ \in \rightarrow_T,\ \exists I \subseteq J,\ \forall i \in I, s_i \xrightarrow{\alpha_i} s'_i \in \rightarrow_i\ \wedge \forall j \in J \backslash I, s_j = s'_j \end{pmatrix}$$

Explanations

The result of the product is also a LTS, built upon the cartesian product of the LTSs used as arguments and the transducer $\{T\} \cup J$. The transitions of the product are those described by the synchronisation vectors of the transducer. Moreover, as the producer is a LTS, it can in turn be synchronised with other LTSs in a Net. This property enables us to have different levels of synchronisations, i.e. a hierarchical definition for a system.

4.2 Parameterized Networks of Synchronised Automata

Next we enrich the above definitions with parameters in the spirit of Symbolic Transition Graphs [Lin 96]. We start by giving the notion of parameterized actions. We leave unspecified here the constructors and operators of the action algebra, they will be defined together with the mapping of some specific formalism to pNets.

4.2.1 Parameterized Labelled Transition Systems

DEFINITION 10 (PARAMETERIZED ACTIONS)
Let V be a set of names, $\mathcal{L}_{A,V}$ a term algebra built over V, including the constant action τ. We call:

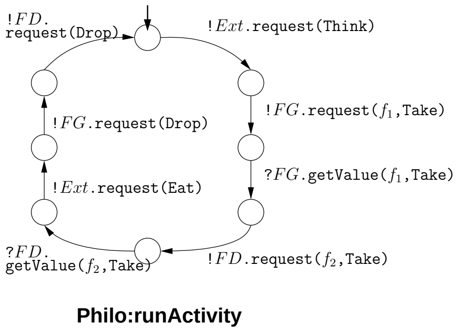
- *$v \in V$ a parameter,*
- *$a \in \mathcal{L}_{A,V}$ a parameterized action,*
- *$\mathcal{B}_{A,V}$ the set of boolean expressions (guards) over $\mathcal{L}_{A,V}$.*

Example

In Milner's *Value-passing CCS* [Milner 89] the action algebra has constructors "`tau`", "`a`" for input actions, "`'a`" for output actions, "`a(x)`" for parameterized actions. Then "`'out(3)`" is a closed output action term, "`a(x,y)`" an open input action term with parameters `x` and `y`, and "x+y=3" a guard.

DEFINITION 11 (PARAMETERIZED LTS)
A parameterized Labelled Transition System (pLTS) is a tuple $(V, S, s_0, L, \rightarrow)$, where:

- *V is a finite set of parameters, from which we construct the term algebra $\mathcal{L}_{A,V}$,*
- *S is a set of states; each state $s \in S$ is associated a finite indexed set of free variables $fv(s) = \tilde{x}_{J_s} \subseteq V$,*
- *$s_0 \in S$ is the initial state,*
- *L is the set of labels, $\rightarrow$ the transition relation $\rightarrow \subset S \times L \times S$*
- *Labels have the form $l = \langle \alpha,\ e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}} \rangle$ such that if $s \xrightarrow{l} s'$, then:*
  - *α is a parameterized action.*
    - *the action defines input $iv(\alpha)$, possibly defining new variables $iv(\alpha) \subseteq V$;*
    - *the action defines output $oe(\alpha)$ using action expressions.*
  - *$e_b \in \mathcal{B}_{A,V}$ is the* optional *guard,*
  - *the variables $\tilde{x}_{J_{s'}}$ are assigned during the transition by the* optional *expressions $\tilde{e}_{J_{s'}}$ with the constraints:*
    - *$fv(oe(\alpha)) \subseteq iv(\alpha) \cup \tilde{x}_{J_s}$*
    - *$fv(e_b) \cup fv(\tilde{e}_{J_{s'}}) \subseteq iv(\alpha) \cup \tilde{x}_{J_s} \cup \tilde{x}_{J_{s'}}$*



PhiloRunActivityLTS $= \langle V, S, s_0, L, \rightarrow \rangle$
with:
$V = \{f_1, f_2\}$
$S = \{s_i\}$, `i ∈ [0:7]`
L=\{ `!`Ext`.request(Think)`, `!`Ext`.request(Eat)`,
`!`FG`.request(`f_1`,Take)`, `?`FG`.getValue(`f_1`,Take)`,
`!`FD`.request(`f_2`,Take)`, `?`FD`.getValue(`f_2`,Take)`,
`!`FG`.request(Drop)`, `!`FD`.request(Drop)` \}
$\rightarrow$ such that:
s_0 : `!`Ext`.request(Think)` $\rightarrow$ s_1,
s_1 : `!`FG`.request(`f_1`,Take)` $\rightarrow$ s_2
...

Figure 4.1: *Example of a pLTS*

Example

Figure 4.1 is based on an implementation of the philosophers problem in ProActive. It represents the pLTS for the body behaviour of a Philo component (we will see in Chapter 6 how we generate behaviour models for these kind of components). The action alphabet used here reflects the communication schema: each remote request sent by the body has the form " !$dest$`.request`$(f, \mathcal{M}(\tilde{arg}))$ ", where $dest$ is the remote reference, $\mathcal{M}$ is the method name, with parameters $\tilde{arg}$, and f is a future reference. More precisely, f is the identifier of the future proxy instance. Requests that do not require a response do not use a future proxy.

4.2.2 Parameterized Synchronisation Networks

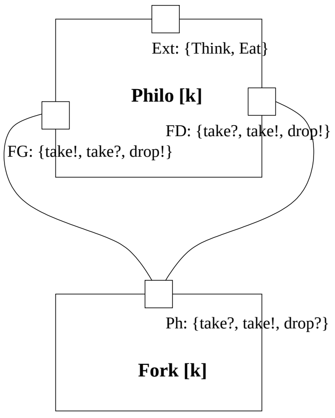
DEFINITION 12 (PARAMETERIZED NETWORKS)
*A Parameterized Network (**pNet**) is a tuple* $\langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T\rangle$*, where:*

- *V is a set of parameters,*
- *$pA_G \subset \mathcal{L}_{A,V}$ is a set of (parameterized) external actions,*
- *J is a finite set of holes, each hole j being associated with (at most) a parameter $p_j \in V$ and with a sort $O_j \subset \mathcal{L}_{A,V}$.*
- *T is the transducer LTS (S_T, s_{0T}, L_T, T_T), which transition labels ($\overrightarrow{v} \in L_T$) are synchronisation vectors of the form: $\overrightarrow{v} = \langle a_g, \{\alpha_t\}_{i \in I, t \in B_i}\rangle$ such that:*
  - *$I \subseteq J$*
  - *$B_i \subseteq \mathcal{D}om(p_i)$*
  - *$\alpha_i \in O_i$*
  - *$fv(\alpha_i) \subseteq V$*

Explanations

Each hole in the pNet has a parameter p_j, expressing that this "parameterized hole" corresponds to as many actual arguments as necessary in a given instantiation of its parameter (we could have, without changing the expressivity, several parameters per hole). In other words, the parameterized holes express *parameterized topologies* of processes synchronised by a given Net. Each parameterized synchronisation vector in the transducer expresses a synchronisation between some instances ($\{t\}_{t \in B_i}$) of some of the pNet holes ($I \subseteq J$). The hole parameters being part of the variables of the action algebra, they can be used in communication and synchronisation between the processes.

A *static* pNet has a unique state, but it has state variables that encode some notion of internal memory that can influence the synchronisation. Static pNets have the nice property that they can be easily represented graphically. Such graphics are described in previous publications to represent parameterized processes in the Autograph editor [Madelaine 92].


Ext: {Think, Eat}
Philo [k]
FD: {take?, take!, drop!}
FG: {take!, take?, drop!}
Ph: {take?, take!, drop?}
Fork [k]

$PhiloNet = \langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T\rangle$ with:

$V = \{k, f_1, f_2\}$

$pA_G = \{$`Think(k), Eat(k), !TakeG(k), !TakeD(k), ?TakeG(k), ?TakeD(k), DropG!k, DropD!k`$\}$

$J = \{$`Philo, Fork`$\}$

$p_{Philo} = k, p_{Fork} = k$

$O_{Philo} = \{$`!`Ext`.request(Think), !`Ext`.request(Eat), !`FG`.request(`f_1`,Take), !`FD`.request(`f_2`,Take), ?`FG`.getValue(`f_1`,Take), ?`FD`.getValue(`f_2`,Take), !`FG`.request(Drop), !`FD`.request(Drop)`$\}$

$O_{Fork} = \{$`?`Ph`.request(`f_1`,Take), ?`Ph`.request(`f_2`,Take), !`Ph`.getValue(`f_1`,Take), !`Ph`.getValue(`f_2`,Take), ?`Ph`.request(Drop)`$\}$

This pNet is static, T has a unique state, and transitions with the following labels:
$L_T = \{$
$\langle$`Think(k), !Philo[k].`Ext`.request(Think)`$\rangle$
$\langle$`Eat(k), !Philo[k].`Ext`.request(Eat)`$\rangle$
$\langle$`!TakeG(k), !Philo[k].`FG`.request(`f_1`,Take), ?Fork[k].`Ph`.request(`f_1`,Take)`$\rangle$
$\langle$`!TakeD(k), !Philo[k].`FD`.request(`f_2`,Take), ?Fork[k+1].`Ph`.request(`f_2`,Take)`$\rangle$
$\langle$`?TakeG(k), ?Philo[k].`FG`.getValue(`f_1`,Take), !Fork[k].`Ph`.getValue(`f_1`,Take)`$\rangle$
$\langle$`?TakeD(k), ?Philo[k].`FD`.getValue(`f_2`,Take), !Fork[k+1].`Ph`.getValue(`f_2`,Take)`$\rangle$
$\langle$`DropG(k), !Philo[k].`FG`.request(Drop), ?Fork[k].`Ph`.request(Drop)`$\rangle$
$\langle$`DropD(k), !Philo[k].`FD`.request(Drop), ?Fork[k+1].`Ph`.request(Drop)`$\rangle$ $\}$

Figure 4.2: *Example of a pNet*

Example

The drawing in Figure 4.2 shows a (static) pNet representing the classical philosophers problem, with 2 parameterized holes (indexed by the same variable k) for philosophers and forks. On the right hand side are the corresponding elements of the formal pNet, in which we explicitly list the sort of the holes (O_{philo} and O_{Fork}), and synchronisation vectors parameterized over the index k and the future ids f_1 and f_2.

The sorts of our parameterized structures are sets of parameterized actions. This definition extends the *sorts* from Definition 8:

DEFINITION 13 (PARAMETERIZED SORTS)

- *The sort of a pLTS: Sort*$(V, S, s_0, L, \to) = \left\{\alpha \mid \exists l \in L.\ l = \langle \alpha,\ e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}}\rangle\right\} = pA_G$
- *The sort of a pNet: Sort*$\langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T\rangle = pA_G$

4.2.3 Building hierarchical pNets

Except from the occurrence of parameters in the structure of labels, the rest of the construction of complex systems as hierarchical pNet expressions is similar to the previous section, with the additional parameterization of arguments: an actual (parameterized) argument of a pNet at position j is a pair $\langle Sys, \mathcal{D}\rangle$, where Sys is a pNet (or pLTS) that agrees with the sort of the hole

($Sort(Sys) \subset O_j$), and $\mathcal{D}$ is the actual domain for the hole parameter p_j, i.e. denotes the set of similar arguments inserted in this hole.

We do not define a synchronisation product for pLTS that would give some kind of "early" or "symbolic" semantics of our generalised pNets. Instead, we define instantiations of the parameterized LTS and Nets, based on a (possibly infinite) domain for each variable.

Given a hierarchical pNet expression, and instantiation domains for all parameters in this expression, the definitions below allow us to construct a (non parameterized) Net expression, by applying instantiation separately on each pLTS and each pNet in the expression. This can be performed both for closed or open pNet expressions, the result being, respectively, closed or open Net expressions. In the former, closed Net expressions can then be reduced to a single LTS (expressing the global behaviour) using the synchronous products in a bottom-up way.

Definition 14 (pLTS Instantiation)
Given a pLTS $P_p = \langle V, S_p, s_{0_p}, L_p, \to_p \rangle$, with $V = \tilde{x}_V$ and given a countable domain for each variable $\mathcal{D}_V = \{\mathcal{D}(x)\}_{x \in V}$, and an initial assignment ρ_0 for the variables of the initial state s_{0_p}, the instantiation $\Phi(P_p, \mathcal{D}_V)$ is a LTS $P = \langle S, s_0, L, \to \rangle$, such that:

- $S = \bigcup_{s_p \in S_p} \left\{ s_p\{\tilde{x}_V \leftarrow \tilde{e}_V\} \mid \forall x \in V,\ \forall e_V \in \mathcal{D}(x) \right\}$,
- $s_0 = s_{0_p}\{fv(s_{0_p}) \leftarrow \rho_0(fv(s_0))\}$,
- *L is the set of ground actions (i.e. closed terms) of the action algebra $\mathcal{L}_{A,V}$,*
- *$\to\ (\subseteq S x L x S) = \bigcup_{t \in \to_p} \Phi(t)$ is the union of instantiations of the parameterized transitions, built in the following way:*
  let $t = s \xrightarrow{l_p = \langle \alpha, e_b,\ \tilde{x}_{J_{s'}} := \tilde{e}_{J_{s'}} \rangle} s'_p$ be a transition,
  let $V_t = fv(s) \cup fv(\alpha) \cup fv(s')$ the free variables of t, and $\mathcal{D}_{V_t}$ their instantiation domains, then

$$\Phi(t) = \bigcup_{\tilde{e}_{V_t} \in \mathcal{D}_{V_t}} \left\{ \begin{array}{l} \mathit{if}\ (e_b\{\tilde{x}_{V_t} \leftarrow \tilde{e}_{V_t}\} = \mathit{False})\ \mathit{then}\ \emptyset \\ \mathit{otherwise} \\ \quad \mathit{let}\ \psi = \{\tilde{x}_{V_t} \leftarrow \tilde{e}_{V_t}\} \\ \quad \mathit{in}\ \left\{ \psi(s) \xrightarrow{\psi(\alpha)} s' \left\{ \mathit{if}\ \left(\exists j \in J_{s'}, x = x_j\right)\ \mathit{then}\ x \leftarrow \psi(e_j)^{\star}\ \mathit{else}\ x \leftarrow e_x \right\} \right\} \end{array} \right\}$$

Apart from the proliferation of indexes, this definition is quite natural and straightforward; only the case when variables of the target state are assigned during the transition needs care (see $\star$ in the equation), because the assigned open expressions $\tilde{e}_{J_{s'}}$ need themselves to be instantiated.

This operation has an upper-bound complexity that is exponential in the cardinality of the instantiation domains, in number of states and transitions.

Definition 15 (pNet Instantiation)
Given a pNet $N_p = \langle V, pA_G, J, \tilde{p}_J, \tilde{O}_J, T \rangle$, with the transducer $T = (S_T, s_{0T}, L_T, T_T)$, and given domains $\mathcal{D}_V$ for variables in V, the instantiation $\Phi(N_p, \mathcal{D}_V)$ is a Net $N = \langle A'_G, J', \tilde{O}'_{J'}, T' \rangle$, with $T' = \langle S_{T'}, s_{0T'}, L_{T'}, T_{T'} \rangle$ constructed in the following way:

1) *expand the parameterized holes: $J' = \Phi(J) = \uplus_{j \in J} \mathcal{D}(p_j)$ where $\uplus$ is a disjoint union (or concatenation) of sets; let $J'_j \subset J'$ be the part of J' corresponding to the expansion of hole number j;*

2) *instantiate the sort of holes and the global sort:*

```
    for $i \in J'_{j'}$, build $\tilde{O}'_i = \bigcup_{a \in O_j} \Phi(a)$
    $A'_G = \bigcup_{a \in pA_G} \Phi(a)$

3) instantiate the transducer:
    $S_{T'} = S_T$
    $s_{0T'} = s_{0T}$
    $L_{T'} = \bigcup_{\vec{v} \in L_T} \{\Phi(\vec{v})\}$ the expansion of the synchronisation vectors:
    . for each $\vec{v} = \langle a_g, \{\alpha_{i,t}\}_{i \in I, t \in B_i} \rangle$ let $V = fv(\vec{v})$, $\mathcal{D}_V$ their instantiation domains,
    .   for each possible valuation $\tilde{e}_V$ of the variables in V,
    .     let $\phi = \{\tilde{x}_V \leftarrow \tilde{e}_V\}$ the corresponding instantiation function,
    .     expand each parameterized action by $\Phi(\alpha_{j,t})$ = if $j \notin I$ then $\langle *, ..., * \rangle$
    .       else $\langle x_1, ..., x_{|J'_i|} \rangle$, with $x_k = *$ if $k \notin B_i$, $\phi(\alpha_{j,t})$ otherwise,
    .     build $\Phi(\phi, \vec{v})$ as a vector of cardinality $|J'|$ as the concatenation of subvectors
    .     $x \in \Phi(\alpha_{j,t})$ for each hole $j \in J$,
    . $\Phi(\vec{v}) = \{\Phi(\phi, \vec{v})\}_\phi$
    $T_{T'} = \bigcup_{(s, \vec{v}, s') \in T_T} \{(s, a, s'), a \in \Phi(\vec{v})\}$
```

Naturally, even if the above definition does not suppose finiteness of the parameter domains, it will be used in practice with finite instantiation domains, and finite vectors.

Example

We give here a small instantiation of the philosopher system from Figure 4.2.

$\Phi\,(PhiloNet, \mathcal{D}(k) = \{1, 2\}, \mathcal{D}(f_1) = \{1\}, \mathcal{D}(f_2) = \{2\}) = \langle A'_G, J', \tilde{O}'_{J'}, T' \rangle$ with:

```
$A'_G$ = {Think(1), Think(2), Eat(1), !TakeG(1),...}
$J'$ = {Philo, Philo, Fork, Fork}
$O'_{Philo^{(1)}}$ = {!Ext.request(Think), !Ext.request(Eat), !FG.request(1,Take), ...}
$O'_{Philo^{(2)}}$ = {!Ext.request(Think), !Ext.request(Eat), !FG.request(1,Take), ...}
...
$L_{T'}$ = {
⟨ Think(1), !Ext.request(Think), *, *, * ⟩
⟨ Think(2), *, !Ext.request(Think), *, * ⟩
...
⟨ !takeG(1), !FG.request(1,Take), *, ?Ph.request(1,Take), * ⟩
⟨ !takeD(1), !FD.request(2,Take), *, *, ?Ph.request(2,Take) ⟩
... }
```

Expressivity

In [Barros 04], the authors gave examples of pNets representing various kinds of recursive functions: the "data flow" within an index family of pLTSs is expressed by an adequate indexing within the synchronisation vectors. But one should note that this expressivity is gained from the properties of the indexes domains (here integers with standard arithmetic): the pNets formal definition is (on purpose) separated from the data domain definition, and does not allow us, by itself, to provide any formal expressivity result.

Another aspect of expressivity is the representation of classical patterns of distributed systems. We claim that pNets, used with simple (first order) parameter domains, provide powerful and easy representations for our needs, including two-way or multi-way synchronisation, dynamic composition operators, or dynamic creation/activation/orchestration of indexed families of processes.

The first application of pNets published [Barros 04] was for ProActive distributed applications, based on active objects, before the introduction of components. In [Barros 04, Boulifa 04], pNets was used as the formalism for a methodology for generating behavioural models for ProActive, based on static analysis of the Java/ProActive code. The pNets model fits well in this context, and allows one to build compact models, with a natural relation to the code structure: a hierarchical pNet is associated to each active object of the application, and a synchronisation network represents the communication between them.

Going from active objects to distributed and hierarchical components allows us to gain precision in the generated models. The most significant difference is that required interfaces are explicitly declared, and active objects are statically identified by components, so we always know whether a method call is local or remote. Moreover, the pNets's formalism expresses naturally the hierarchical structure of components. This was presented in [Barros 05, Barros 08].

4.3 Data Abstraction

The main interest of the instantiation mechanism defined so far is the ability to build specific domain instantiations with specific properties. In particular, if the instantiation domains are finite, and are built in such a way that they constitute abstract interpretations of the initial parameter domains, then the instantiated Net is finite. Moreover if parameters were only used as value-passing variables in the original pNet (by contrast with parameters of the system topology), then we can apply a result from Cleaveland and Riely [Cleaveland 94] to justify the use of finite model-checking on our instantiated model:

PROPERTY 1
Let Sys be a closed pNet system, with parameters in V, (concrete) parameter domains $\mathcal{D}_V$, and abstract parameter domains $\mathcal{A}_V$, with the following hypothesis:

- *each $\mathcal{A}_v$ is an abstract interpretation[a] of the corresponding concrete domain $\mathcal{D}_v$;*
- *the domains of pNet holes parameters in Sys are unchanged by the abstraction;*

then the abstraction preserves the specification preorder.

[a][Cleaveland 94] was using a slightly relaxed condition called "galois insertions"

The *specification preorder* [Cleaveland 94], or the better known *testing preorder* [Cleaveland 93] are closely related to safety and liveness properties. Given a system and a specification (a set of properties in temporal logics), one can build a "most abstract" (finite) value interpretation relatively to the specification, and try to establish its satisfaction. If this succeeds, the result is valid also for the concrete (potentially infinite) system; if it fails, one can select a more concrete (= more values) interpretation and repeat the analysis.

Example

Unfortunately, the examples presented earlier are too simple for giving a significant example of abstraction. Rather let us use an example extracted from a previous case-study. It consists in modelling the chilean electronic tax systems [Attali 04]. There we were manipulating invoice documents that could typically be described as structures $doct = \langle vendorid, invoiceid, date, content\rangle$, that would be check by government services against $\langle vendorid, invoiceid\rangle$ records. In the case-study, the abstract domain was $doct = \langle vendorid \in [0..2], invoiceid \in [0..2]\rangle$; this is an abstract interpretation preserving all safety properties involving at most 2 invoice documents.

The previous result does not apply when the instantiated variables are parameters of the system topology. But the same procedure can be used to build a finite model for one or more finite abstractions of the value domains. Even if this does not provide a proof of validity on the original system, it is still a valuable debugging tool. As an example, one could check safety properties involving Philo[1] and Fork[2] in the philosopher system, using an abstract domain for indexes defined as $\{\{1\}, \{2\}, \{others\}\}$. But this will not prove that such a property holds for a system with an arbitrary number of philosophers.

4.4 Conclusion

This chapter defines the pNets formalism, a powerful extension of labelled transition systems, that features a better structure in terms of hierarchical synchronisation networks, and more expressivity through the use of parameters at both LTS and Networks levels. This formalism is used for representing the behavioural semantics of distributed systems.

This kind of semantic-level model is widely used in analysis and verification toolsets, because it provides a compact and well-defined intermediate format for connecting code analysers or code generators with model-checking or equivalence engines. When dealing with concurrent or distributed systems, intermediate models often make strong hypothesis on the kind of synchronisation and communication mechanisms addressed, for example LOTOS-like parallelism in CADP, channels in Promela, or Petri nets in other cases. Our choice with the pNet model is to have low-level primitives (LTS + synchronisation vectors) that are able to represent many possible mechanisms. Another important trade-off is between parameterized representations (close to developers code) and lower-level explicit-state encodings that are required by the model-checkers.

We argue that the pNets model allows for finite and compact representation of systems, expressive enough to capture a large family of behavioural properties of both synchronous and asynchronous applications. We will show in Chapter 6 how the behaviour of GCM components can be expressed using pNets.

Symbolic Product. In this chapter we have not defined a symbolic product for pNets. We have chosen to only define the product of instantiated Nets. Nevertheless, it should be possible to define it, similarly as done in STSLib [Fernandes 07]. This would allow for a late instantiation which would avoid unnecessary intermediate state expansion. The resulting product should, however, be equivalent once instantiated with the same parameter domains. We have not included the symbolic product because we have not proven that both approaches are indeed equivalent.

5 A Specification Language for Distributed Components

Contents

Abstract

In this chapter we propose a textual specification language for distributed components. In Chapter 4 we have presented a formalism called pNets for describing the behaviour of processes. Nevertheless, the formalism is too low-level to be used as a specification language, and lacks of the high-level concepts particular to the different contexts in which we want to use it.

In this chapter we present a novel specification language called *Java Distributed Components* (JDC for short) to be used at design phase of distributed components. The language is endowed with enough formality so it allows a constructive approach, namely the generation of behavioural models which can be model-checked, and the generation of code skeletons with the control flow of components.

Globally, this approach aims at generating components with strong guarantees w.r.t. their behaviour.

Motivation

After evaluating the specification languages found in the literature (see Section 3.3), we found that none fits well in the context of distributed components. In the GCM, most difficulties come when specifying the synchronisations. From a practical point of view, we focus on GCM/ProActive presented in Section 3.2.

The transparent futures found in GCM/ProActive alleviate the programmer from synchronisation difficulties, allow for separation of concerns (the source code can be really independent from the physical infrastructure), and give optimisation opportunities at the middleware level. On the other hand, specifying and/or inferring about synchronisations becomes more complex. To our knowledge, no specification language has been proposed within this context.

Instead of proving that legacy code is safe, in this chapter we seek a constructive approach similar to [Coglio 05, Fernandes 07]. The idea is find a suitable specification language that can be verified, and that can be used to generate safe by construction code. The language is called *Java Distributed Components* (JDC for short). pNets is left as the underlying formalism that defines the behaviour semantics of JDC specifications, and that interfaces with model-checkers.

In parallel, we have also been defining a graphical version of JDC to be used within VCE (for Vercors Component Editor). The tool will be described in Chapter 7. Moreover, we will also validate our approach by specifying a large case study called CoCoME [Rausch 08] within Appendix A.

The results of this chapter were published in [Cansado 08c, Cansado 08b].

5.1 Foundations of the Specification Language

Distributed components tend to be coarse grain units of composition, and are often loosely-coupled. In the following we present a specification language in the form of an extension of a subset of Java for specifying these components. The language includes both the architecture and the behaviour definitions, and is endowed with enough formality and control-flow information so that we are able to:

- on one hand *check the correctness of the system* (Chapter 6): we build a behaviour model that can be model-checked against temporal formulas;
- on the other hand *generate safe components* (Chapter 8): we want to generate the control code of components that is guaranteed to respect the specification.

We opt for a Java-like language for several reasons: (i) it is close to the target expertise of engineers, using common syntax such as method calls and data classes; (ii) it allows part of the specification to be embedded within the code skeletons; (iii) it uses the same datatypes as in the implementation, guaranteeing that operations on the datatypes are directly useful without modification.

5.1.1 Architecture

The architecture definition is closely related to classic ADLs. We will provide a Java-like syntax for defining the architecture, which is used to define the component type, and expose one level of the component implementation. In other words, we will define the component's subcomponents, and bindings.

One of the ideas is to leave room for further extensions that are not yet considered in the language. Moreover, as the architecture is part of the language, defining reconfiguration primitives that relate directly to the architecture is easier.

Among the extensions that we plan, we want to address parameterized topologies of component systems. This will be done by including parameters in the component definition, and then using expressions on the definitions to specify families of components. This is of particular interest in Grids where the component system will be deployed on several clusters, with different number of machines in each cluster. Another extension to classic ADLs is that we expose both external and internal interfaces of the component, which allows us to define interceptors *à la GCM* between calls coming in and going out the component.

5.1.2 Decomposing the Behaviour into Services

We propose to specify the component behaviour as a black-box specification. The specification is given by a set of concurrent services, each one being an independent activity, or service. For each service, we define a policy and then a detail of its behaviour.

The first part of a service is called the *service policy*; it defines how a component selects requests depending on its internal state, and any behaviour the component triggers by its own. This is a rough specification of the component protocol, however, it gives the user a good idea of how the component should be used. For instance, the specification may define that a component must serve requests in a particular order.

The second part of the service specifies what each external service exposed at the service policy actually does. This behaviour is defined by a Java-like language that can be closely be mapped to the Java programming model, using ProActive as the basis.

In the specification, we include an abstraction of the control and data flow, remote method calls performed within the service method, and access to data. Although static analysis is required in JDC to infer the system behaviour, it is easier than in standard Java. Remote calls are easily identified by calls on the component's client interfaces; future creation points are identified as the results of these calls; there is no concurrency within the service method; and there is no exception handling (for the sake of asynchrony).

5.1.3 Futures

Using transparent futures in the specification language brings the same advantages as in the programming language: the system designer doesn't have to wonder if a variable might contain a future; or more precisely, no explicit synchronisation mechanism is needed for variables that may sometimes contain a future. This extends reusability of specifications as they may fit several contexts, where values are remotely computed, or come from local instances. It is even possible that the component that receives a future and the component that computes the future result don't share a direct binding. These two components must however be indirectly linked by a communication path. We call these *implicit communication channels*.

A drawback of transparency of futures is non-determinism; it is in general not statically decidable whether a variable is a future or not at a given point of the program. However, additional synchronisation can be specified, ensuring that, after synchronisation upon a variable, this variable is known to be value, or a future with a filled value.

5.1.4 Datatypes and Abstraction

The datatypes used in JDC are standard Java classes. This way the code-skeletons obtained by our generation tools will be directly usable. On the one hand, arbitrary datatypes often have large (possibly infinite) domains which can't be model-checked directly. On the other hand, the kind of behavioural properties we seek only require an abstraction of these datatypes. Therefore, whenever verification is desired, the specification includes as well an abstraction of the user types. This way, we are able to derive an abstract specification that contains only variables of "simple" domains.

The abstraction keeps solely data influencing the control-flow and the synchronisations, however, it must preserve the behavioural properties in the sense of Cousot's abstract interpretations [Cousot 01]. If abstractions are finite and constitute abstract interpretations of the initial parameter domains, then the model is finite. Following [Cleaveland 94], we build an abstract interpretation of the system behaviour, from abstractions of the domains of the program variables; this construction can be used for finite model-checking as it preserves safety and liveness properties.

The abstractions are mappings of variables of user types to predefined first order datatypes (*simple types* for now on). Simple types themselves are provided as Java classes, and as a particular case, can be used in JDC programs. They are: point (or singleton), booleans, enumerated types, integers, intervals of integers, strings, records of simple types, and arrays of simple types.

In our work we decompose the abstraction in two steps: the first maps concrete types to potentially infinite *simple types* allowing us to generate parameterized *pNets* models. From pNets, we can apply many different proof methods, including inductive theorem proving techniques, that can address a large family of properties. The second step is based on finite partitions of parameter domains that depend on each set of properties to prove. In this case, the abstraction produces finite pNets on which we can use explicit-state model-checkers.

Finally, our abstractions must consider futures. Even if a variable has insignificant values, access to the variable may still trigger synchronisation. This makes the choice of a good abstraction tricky, and some variables are only kept within the abstraction in order to signal eventual access on them. In other words, these variables have an abstract domain with 2 values *filled* or *non-filled*.

5.1.5 Discussion

In this language we have taken 2 opposite approaches. On the one hand, the behaviour is defined using a high-level language that is much more abstract than a programming language. Here, we do not want to give details about the implementation, but we still want to be able to derive partial implementations (code skeletons). This approach goes from being abstract to being concrete.

On the other hand, the data part of the language is very close to Java. In fact, we require the designer to define its user-classes, and then to define mappings for the domains of variables of type “user-class” towards our simplified simple types. This approach goes in the opposite direction, from being concrete to abstract. The general idea is that as the specification relies on datatypes from a programming language, we can expect the data part of the code skeletons to be directly useful. Similarly, the abstraction of the data part will allow us to map the specification to an abstract specification. This is a specification in which all variables are of simple types, and therefore we know how to interface with verification engines.

What is common in both approaches is the goal: to generate components with guaranteed behaviour. Generating code from a structured behaviour specification is feasible, and would probably consits in generating code for simulating a state-machine. However, generating code for the data part of the language is much more difficult, though possible. In STSLib [Fernandes 07] and in CADP’s Full LOTOS compiler [Garavel 89] for example, the authors provide automatic implementations of Abstract Data Types. These abstract types are incomplete by essence (most of the business code is missing), thus we expect the generated code to require major changes in the implementation. Hence, the difficulty would be how to guarantee that the new refinement preserves the system behaviour. This is why we rather ask the designer to provide the abstractions and the user classes, which we think are more aligned with the expertise of the final user.

5.2 Architecture Specification

In the next sections, we present elements of the abstract and concrete syntax of JDC. Each box defines a piece of JDC syntax, using:

- keywords in bold (e.g. **component**);
- terminal symbols written between simple quotes (e.g. '{');
- non-terminal symbols in monospace (e.g. `Services`);
- optional expressions with square-brackets (e.g. [`expr`]);
- choices with | (e.g. `expr1` | `expr2`);
- concatenations of zero (resp. one) or more expressions with * and $^+$ (e.g. `expr`*, `expr`$^+$);
- identifiers as id .

5.2.1 Defining a Component

The definition of a component type comprises its external interfaces with both provisions and requirements, and a specification of its behaviour. The behaviour is either given by a black-box specification in the form of a set of *Services* (Section 5.3), or by a composition of components, also called *Architecture* (Section 5.2.2), or even by both. This is shown in Figure 5.1.

`Component` →	**component** id '{'	*≪component definition≫*
	external interfaces	
	`Interface`*	*≪set of interfaces≫*
	[`Services`]	*≪black-box description≫*
	[`Architecture`]	*≪content description≫*
	'}'	
`Interface` →	**server** | **client**	*≪interface role≫*
	interface `InterfaceType` id ';'	*≪type and name≫*

Figure 5.1: *Syntax for defining a* component

Each interface in a component has a role (either server or client), a type (a Java interface as in most IDLs), and a name [*]. The interfaces defined within the context of the component definition are *external interfaces* and can be bound to the environment. Interfaces determine both provided and required services of a component; provided services are defined by server interfaces, and required services are defined by client interfaces.

5.2.2 Composing Components

The composition of components is done within the *architecture*, see Figure 5.2. It exposes the content of a component by means of its subcomponents, its internal interfaces, and the bindings. The subcomponents are named and typed, the type being given by either an external component definition, or by an inline definition. The bindings connect two interfaces among the component's internal interfaces and the subcomponents' external interfaces.

[*]In GCM, an interface also has a contingency (optional or mandatory), and cardinality (singleton, collection, or multiple). These have not been included in JDC for the moment.

`Architecture` →	**architecture**	
	contents	
	`Subcomponent`*	≪*set of subcomponents*≫
	internal interfaces	
	`Interface`*	≪*set of interfaces*≫
	bindings	
	`Binding`*	≪*set of bindings*≫
`Subcomponent` →	**component** `ComponentType` <u>id</u> ';'	≪*named subcomponent*≫
	| `Component`	≪*inline definition*≫
`ComponentType` →	<u>id</u>	≪*reference to a type*≫
`Binding` →	**bind** '(' `SourceItf` ','	
	`TargetItf` ')' ';'	≪*binds a pair of interfaces*≫

Figure 5.2: *Syntax for defining an* architecture

In the GCM, the relation between an internal interface and an external interface of a component is arbitrary: *interceptors* can transform or intercept any incoming invocation. For simplicity, in here we assume that there is an exact match for each pair of external-internal interfaces (interfaces that have the same type and name, but with opposite roles); and that invocations on an external (resp. internal) server interface is directly forwarded to the corresponding internal (resp.external) client interface.

Example

The CoCoME example [Rausch 08] was implemented using GCM / ProActive, and detailed in Appendix A. It is a Point-Of-Sale system, in which the cash desk deals with the sales. The cash desk and its hardware controllers are implemented as components. In Figure 5.3 we show an extract of the example, where an application component has a single peripheral called Scanner; the latter is controlled by a component.

```
component CashDesk {
  external interfaces
    server interface ApplicationIf appIf;
    client interface ScannerIf scannerIf;
    // ... external interfaces
  architecture
    contents
      component Application application;
      component Scanner scanner;
    internal interfaces
        server interface ApplicationIf appIf;
        // ... internal interfaces
    bindings
      bind(this.appIf, application.appIf);
      // ... bindings
}
```

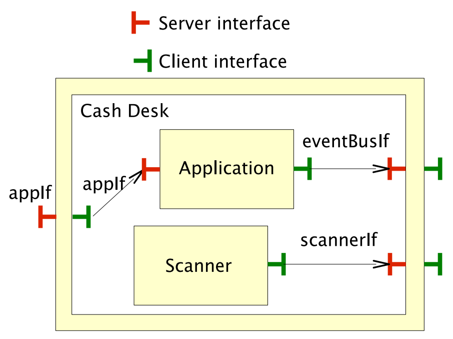


Figure 5.3: *Example of an* architecture *specification*

5.3 Behaviour Specification

When designing a system, the designer would like to adopt a top-down approach: specifying first the behaviour of a component before going down into its architecture. To this aim, we propose to specify directly the behaviour acceptable by the interfaces; this is called a *black-box* behaviour of a component. Of course, different *architecture* definitions can match the same component *black-box*. We leave the equivalence (or preorder) between a component *black-box*, and its implementation (*architecture*) unspecified. Many existing work can apply, starting with all notions of simulations and bisimulations inherited from process algebras. They have to be adapted to our component model though, e.g. in a way similar to the component substitutability relations of Component-Interaction Automata [Černá 06].

In GCM there are two kinds of components, *primitives* that are atomic components, and *composites* that are components composed of other components. Primitives are monothreaded, and concurrency is introduced by composites. The concurrency in JDC is specified by a set of concurrent services within the *Services* block (see Figure 5.4). Each *service* denotes a sequential process with its own set of local variables. A sequential process is split into the *service policy* that defines the high-level protocol of the service, and a set of *service methods* that details the behaviour of the methods exported by the component. Finally, a service also provides a set of local methods that are not accessible directly by the environment.

```
Services →  services
              Service+                ≪one or more concurrent services≫
 Service →  service '{'
              LocalVariableDecl*      ≪variables of the component≫
              policy '{' Policy '}'   ≪service policy≫
              ServiceMethodDecl*      ≪service methods (exported)≫
              LocalMethodDecl*        ≪local methods (not exported)≫
            '}'
```

Figure 5.4: *Syntax for defining a* service

The behaviour specification of the component is an abstraction of the control-flow, some elements of data-flow, and access to data. Concretely, for the distributed components we deal with, we want to focus on:

Access to the component's request queue, the access to the queue is a blocking operation, therefore it is essential to infer the system's synchronisations.

Remote method calls, these represent communication between components. A remote call is always asynchronous; it creates a request in request queue of the callee component, and creates a future in the caller for dealing with the promised result. Remote calls are identified by calls on client interfaces.

Future flow, these represent the creation of implicit communication channels for the responses of futures between the component that computes the value of the future, and the component that receives the reference to the future. The future flow can be identified by tracking future objects in parameters and results of remote method calls.

Data-access, these trigger synchronisations between components. They are identified using static analysis, or given explicitly within the specification.

5.3.1 Service Policy

The service policy defines how incoming requests are selected from the queue depending on the internal state of the component, and any behaviour triggered internally. In Figure 5.5, we present its syntax. It basically is a (non-deterministic) state-machine, expressed by regular expressions. The actions can express *reactive* or *active* behaviour.

```
     Policy →  BasicPolicy [ ';' PermPolicy ]               ≪policy definition≫
BasicPolicy →  ServeMode '(' [ Filter ] ')'                 ≪reactive service≫
               | MethodCall                                 ≪active service≫
               | BasicPolicy ';' BasicPolicy                ≪sequence≫
               | BasicPolicy '|' BasicPolicy                ≪choice≫
               | BasicPolicy 'n'                            ≪n repetition≫
 PermPolicy →  BasicPolicy '*'                              ≪infinite repetition≫
  ServeMode →  serveOldest | serveYoungest                  ≪request queue access prim.≫
     Filter →  ItfName                                      ≪any method in this interface≫
               | ItfName '.' MethodName                     ≪this method≫
               | Filter ',' Filter                          ≪a list of filters≫
 MethodCall →  ItfName '.' MethodName '(' [ Expr ] ')'      ≪remote method call≫
               | MethodName '(' [ Expr ] ')'                ≪local method call≫
```

Figure 5.5: *Syntax for defining a* service policy

The component behaviour can be seen as a machine that is constantly doing some work. This is reflected by a policy defined within an infinite loop. We can define this in JDC with a policy that ends in a permanent policy `PermPolicy`. Moreover, the component may be stopped in between requests.

The *reactive* behaviour defines which kind of methods to select, and in which order to pick them from the queue. This represents work that depends on the requests in the component's request queue. As an example, `serveOldest(itf.m1, itf.m2)` selects from the queue the oldest request on the interface `itf` matching the method names `m1` or `m2`; if none of them is in the queue, the service blocks until one of them arrives. Then, the request is served, i.e., the control is delegated to the service method representing the request.

Additionally, an *active* behaviour denotes spontaneous behaviour, i.e., some work that is done without being requested. This can be either a remote method call, or a local method call; both are affected by an expression on the component's variables. In our example, a component in charge of the scanner sends signals to the application component whenever a product is scanned. The signals take the form of method calls on the application components. For the scanner component, this behaviour is spontaneous as the interaction with the physical scanner is abstracted away.

The service policy is the only block authorised to access the queue. Basically, this ensures that the code generated for the service policy will be complete w.r.t. how the component provides services. Moreover, the state-machines are precise enough to ensure that the code generated will be the final implementation of the `runActivity()` method of a GCM/ProActive component, and no other method will access the component's request queue. This is essential for guaranteeing that the application fulfils its behavioural specification. More details are given in Chapter 8.

Example

An example of a *Service* definition is found in Figure 5.6, which is part of the behaviour of the cash desk application from Figure 5.3.

The component has a single service (the component is indeed monothreaded), and its behaviour is mainly reactive. Concretely, the component performs a local method call `init()`, and then starts serving requests in FIFO order. If there are no requests in its queue, it blocks until one arrives.

If the component is set to stop, then after it has finished processing a request, it will stop, even if there are still pending requests in its request queue.

```
services
  service {
    // variables of simple types
    Bool expressMode;
    public enum CashState{
      IDLE, STARTED, PAYING
    }
    CashState cashState;
    // ... other variables of simple types and user types

    // initialises the system with some RPC and then treats calls in FIFO order
    policy {
      init(); // local method
      serveOldest(applicationIf)*
    }
   // ... the service methods
 }
```

Figure 5.6: *Example of a* service *definition*

5.3.2 Concurrent Behaviour

A primitive component is typically specified by a single *Service*. However, a component with a single *Service* could also specify a composite component with a pipeline of subcomponents inside as in Figure 5.7. In both the primitive and the composite with the pipeline, two request calls are treated sequentially.

However, a single *Service* cannot express concurrency as it represents a sequential activity. Instead, concurrency of requests is defined by multiple services within a component. Each service is an independent activity serving requests in parallel, with its own set of local variables and provided services.

A drawback of this approach is that it is not possible to define interference among the services directly. That is, we must rely on an *architecture* definition that composes independent components in order to express interference. Other alternatives would have introduced more complexity to the language; moreover, the generation of the control code would have been difficult as the programming model doesn't have explicit concurrency.

```
component Pipeline {
  external interfaces
    server interface Compute itf;
  services // only one
    service {
      policy { serveOldest()* }
      Result itf.compute(Data d) { ... }
  }
}
```

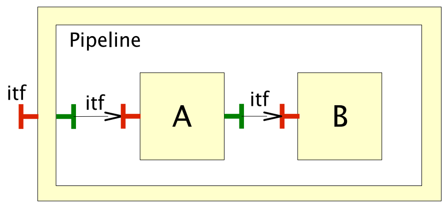


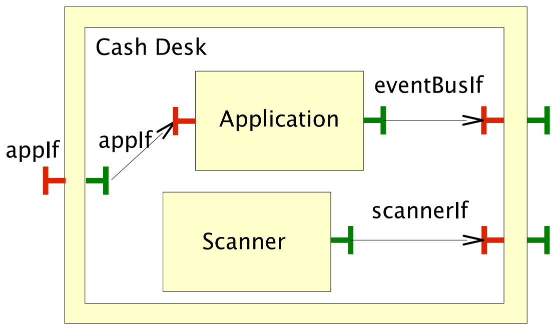
Figure 5.7: *Example of a pipeline component*

Example

In Figure 5.8 we show different architectures; on the left, the same cash example from Figure 5.3; and on the right, a component in which the two of its subcomponents interfere.

For the cash desk, we may provide a blackbox definition with multiple services, one for each of its subcomponents.

However, for the component system on the right, the subcomponent `A` may change the behaviour of component `B`. Seeing this system from outside, we would observe two processes that implement some kind of shared memory. Therefore, we cannot partition both components as two independent processes.



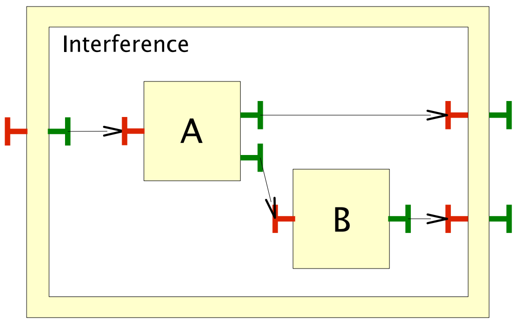


Figure 5.8: *Example of concurrent components*

5.3.3 Service Methods

A service method is an abstraction of a service exported by a component. It is defined by means of a subset of Java statements in which there is no exception handling, and no concurrency. This includes the relevant dataflow between input parameters and results of the method, as well as communication with required services. The service method has access to the component's variables, however, it doesn't access the component's request queue.

The Java syntax is extended to declare explicitly server methods. This is done by prefixing the name of the server interface in which it is defined on the method name.

Client interfaces are accessed as usual objects but they *cannot be transmitted to other components*. This last requirement is very important to ensure that all the interactions between components are realised through the client interfaces and bindings. However, it is possible to create alias of interfaces, meaning that we will need static analysis to detect if they are transmitted to other components.

```
void applicationIf.barcodeScanned(Barcode barcode) {
  switch (cashState) {
    case IDLE:
    case PAYING:
      break; // ignore signal
    case STARTED:
      Product product = cashDeskIf.getProduct(barcode);
      if (product == null) {
        eventBusIf.productBarcodeNotValid();
        break;
      }
      if (expressMode && products.isFull())
        // the specification has been violated
        __ERROR("ExceededNumberOfProducts");
      else {
        products.add(product);
        runningTotal.add(product.getPurchasePrice());
        eventBusIf.runningTotalChanged(runningTotal, product);
      }
  }
}
```

Figure 5.9: *Example of a* service method

Example

An example of a service method is depicted in Figure 5.9. The behaviour focuses on a use-case where the cash desk may provide an express mode for dealing with short-sales (sales with a limited amount of products). When the barcode of a product is scanned, the component reacts according to its internal state. Its usual behaviour is to get the product information by performing a remote method call (`getProduct(barcode)`), add the product to a list of `products`, and update some information regarding the current sale (`runningTotal`). The specification is quite close to Java, notably the operations on the variable `product` are the ones that would be expected in a real implementation.

The `__ERROR()` annotator. In a specification, it is useful to annotate undesired behaviours. We provide this feature by including a special annotation `__ERROR()`. Such an annotation will be encoded as a special label `__ERROR()` within the behavioural models, hence we can automatically verify them using reachability check.

In the ProActive runtime, the middleware does not provide support for this behaviour. We can, of course, simulate it by connecting to the ProActive logger, or forcing the application to throw an exception.

5.4 Specifying Abstractions

This section shows how to define and use abstractions of user types in JDC. One particularity is that a class may have more than one abstraction defined, each one focusing on the significant behaviour of a variable.

The abstractions ensure that we are able to generate behavioural models based on pNets. The pNets format allows us to interface with several verification tools; for the moment we focus on finite-state model-checkers, but using pNets we can potentially interface with infinite-state model-checkers and theorem provers as well.

5.4.1 Formalisation of an Abstraction

A class is a tuple $C =< \overrightarrow{m}, \overrightarrow{f} >$, where $\overrightarrow{m} = \{m^i(\overrightarrow{a}) : \tau^i\}$ are the methods of C; $\overrightarrow{a} = \{a^j : \tau^j\}$ are the method arguments; and $\overrightarrow{f} = \{f^k : \tau^k\}$ the fields.

An abstraction of C is a class $C_{\mathcal{A}} =< \overrightarrow{m_{\mathcal{A}}}, \overrightarrow{f_{\mathcal{A}}} >$, where each public method $m(\{a^j : \tau^j\} : \tau)$ of C has one or more abstract method $m_{\mathcal{A}}(\overrightarrow{a_{\mathcal{A}}}) : \{\tau_{\mathcal{A}}\}$ with $\overrightarrow{a_{\mathcal{A}}} = \{a^j_{\mathcal{A}} : \tau^j_{\mathcal{A}}\}$ the abstract arguments. The domains of the abstract arguments are sets of values in the abstractions of classes τ^i, and the result is an abstract value in the abstraction of class τ.

For defining what is a good abstraction of the domains of the variables in the specification, we need to identify:

- where in the specification are the *variables of interest* – those used in the properties to be proved;
- what are the significant values of these *variables of interest* – these will determine their abstract domain;
- which other variables in the program influence (through control-flow and data-flow) the *variables of interest* – these other variables will also have a non-empty abstract domain.

For each *variable of interest* of type C, we attach an abstract domain in the following manner:

- for each public method m of C, and each possible abstract type of arguments and results of m, we provide an abstract version $m_{\mathcal{A}}$ that captures the accesses on the class variables, accesses on the variables passed as arguments, and relevant results of them.
- the fields of the concrete variable that are of interest are included as a record. The domains of these fields are such that they are precise enough w.r.t. the set of properties to prove.
- for each field of the variable of abstract type, map variables to abstract types, or ignore the variables if they are not *variables of interest*. This is done recursively through the class.

5.4.2 Defining and Using Abstractions

In this subsection, we will describe how the designer defines and uses abstractions within the JDC specification. The abstractions of variables of user types are part of the specification, and must be given by the designer.

An abstraction in JDC is similar to a Java class, with extensions to deal with non-determinism and data abstraction. An important notion is that we may have to use different abstractions for different variables of the same concrete type, within a given program. This means that in the abstract program, we may need different versions of the abstract operators, depending on the abstract types of the arguments.

Example

For example, consider a concrete program with variables:

$$\mathtt{x} : \mathtt{Int},\ \ \mathtt{y} : \mathtt{Int}$$

then the abstract program may have variables of abstract types:

$$\mathtt{x} : \mathtt{Sign},\ \ \mathtt{y} : [\mathtt{0..3}]$$

The operator + may need to be defined for arguments in `Sign` and [`0..3`].

We solve this problem in two phases: there is a library of abstract classes (here `Sign` and `interval` as abstractions of `Int`, with standard abstract operators in each, such that these libraries can be defined in a generic way, and be reused easily.

$$\textit{e.g.}\ \ + : \mathtt{Sign} * \mathtt{Sign} \rightarrow \mathtt{Sign}$$

Then for a specific program, the designer defines abstract classes that inherit the required abstract classes from the library. Afterwards, the designer defines additional abstract operators depending on the specific abstraction of variables, and of the occurrences of the operators found in the code.

$$\textit{e.g.}\ \ + : \mathtt{Sign} * [\mathtt{0..3}] \rightarrow \mathtt{Sign}$$

`Abstraction` →	**abstraction** <u>id</u> **of** <u>id</u> '{'	*≪datatype abstraction≫*
	`TypeDecl`*	*≪type declarations≫*
	`Field`*	*≪local variables≫*
	`Constructor`*	*≪abstract constructors≫*
	`Operator`*	*≪abstract operators≫*
	'}'	
`Constructor` →	`Type` '(' `args` ')'	*≪sign. of concrete constructor≫*
	[**abstracted as** `Type` '(' `args` ')'	
	'{' `Body` '}']	*≪abstract version≫*
`Operator` →	`Type` <u>id</u> '(' `args` ')'	*≪sign. of concrete operator≫*
	[**abstracted as** `Type` <u>id</u> '(' `args` ')'	
	'{' `Body` '}']	*≪abstract version≫*
`Field` →	`Type` <u>id</u>	*≪type & name of variable≫*
	[**abstracted as** `Type`]	*≪local mapping of a type≫*

Figure 5.10: *Syntax for defining* abstract types

The abstract type. Figure 5.10 shows the syntax for defining an abstract type. The abstract type relates to a concrete type saying it is "an abstraction of the concrete type X". It is possible to declare new types, fields, constructors and operators that relate to the concrete type.

Both constructors and operators are abstractions of concrete constructors and operators resp. As mentioned in Section 5.4.1, their arguments and results are abstract versions of those in the concrete constructors and operators. The fields within an abstraction are variables of type *simple type*, or of type *usertype* provided with the definition of its abstraction.

Mapping. A concrete type can be globally mapped to an abstract type, meaning that all variables of that concrete type will have the same abstract domain. Complementary, one can selectively determine the abstraction for each variable. The latter is done by setting a local mapping of a variable to an abstract type.

It is up to the designer to specify these abstractions and to provide the mappings for each variable. However, we could offer the designer tools to support this mapping. Such a procedure was used in the Bandera toolset [Dwyer 01]. There are a couple of differences: first, Bandera starts from a Java program and generates a static approximation of it. This procedure is complex and the result is imprecise by nature. We start from a specification of the system behaviour; this is an abstraction with more information (given by the user) than what can be obtained by static analysis of source code.

Second, the abstraction in Bandera maps variables of user types directly to finite abstract types. In our work, the abstraction maps variables of concrete types to potentially infinite simple types; this allows us to generate parameterized (and possibly infinite) models in pNets. In theory this allows us to apply many different proof methods, including inductive theorem proving techniques, and to address a larger family of properties. Currently, our platform uses explicit-state model-checkers. In this case we define a second abstraction based on finite partitions of parameter domains. The partitions are specific for each set of properties to prove, and produce finite pNets.

Underspecification. It is often useful (or required) to underspecify what are the results of an expression, possibly as the result is a set of abstract values. The language includes for that two non-deterministic operators.

- `ANY`, non-deterministically returns any element of the abstract domain.
- `ANYELEMENT`, non-deterministically selects an element from a list.

Moreover, it is often not possible to statically decide whether a variable refers to a value or to a future. The safe assumption is to consider such variable as *possibly future*. Here, we exploit the fact that a *non-future* variable is semantically equivalent to a *future* variable with filled value. Nevertheless, the user must keep in mind that some traces in the specification may never occur in a concrete implementation. A solution can be then to make the specification more precise by enforcing more synchronisation on a variable (by means of `touch()`). After the synchronisation, the variable is known to be *non-future*.

There is an additional use for the primitive `touch()`. It synchronises on the variable without describing which operations are applied. This allows details of the implementation to be filled-in later without changing the synchronisations occurring in the system. Concretely, when the `touch()` primitive is used together with a variable mapped to a Singleton domain, the programmer will be able to read the variable as many times as she/he wants, and assign it arbitrary (concrete) values. The component will continue to behave similarly, as the variable does not contain significant abstract values (it is a Singleton domain), and there is guarantee that no further synchronisation on the variable is possible.

Example

Figure 5.11 illustrates an abstraction of a variable of interest. This variable has data that influences the control-flow by triggering synchronisations, and changing the data-flow.

In our cash desk example, there are "normal-sales", as well as "short-sales". A short-sale must not exceed a maximum number of products, but there is no constraint on the type of products. The abstraction of the product list must, therefore, be precise enough to take into account whether the maximum has been exceeded or not, but may abstract away the details about the product information.

The abstraction we chose for the product list does not count the number of products. Instead, it focuses on the states the list can have: the list can be either `EMPTY`, in a useful state `OK`, or `FULL`. This abstraction is imprecise w.r.t. the number of products it has, so actions on the list are non-deterministic. Adding a product from an `EMPTY` state never reaches the limit for a short-sale, however, from an `OK` state it may (the transition between an `OK` state and a `FULL` state is non-deterministic). Note that the context guarantees that we never call `add()` when the list is `FULL`.

The abstraction for the `product` is such that we are able to signal access upon the variable. This is necessary as the `product` may be a future; indeed, in Figure 5.9 `product` is the return of a remote method call and thus can be a future. Therefore, the `product` is abstracted as a Singleton domain (`Product_A`) such that the access is signalled by `touch`. Nevertheless, the operations on `product` are not specified, and the behaviour is "shielded" because the `touch` primitive hides the internal details.

```
abstraction
ListProducts_A of ListProducts {
  enum ListState { EMPTY, OK, FULL }
  List<Product> products abstracted as
  ListState;

  ListProducts() abstracted as
  ListProducts_A() {
    products = EMPTY;
  }

  Product get() abstracted as
  Product_A get() {
    switch(products) {
      case EMPTY:
        return null;
      case OK:
        if (Bool.ANY())
          products = EMPTY;
        return Product_A.ANY();
      case FULL:
        products = OK;
        return Product_A.ANY();
  } }

  Bool isFull() {
    return (products == FULL);
  }

  void add(Product product) abstracted as
  void add(Product_A product) {
    product.touch();
    switch(products) {
      case EMPTY:
        products = OK;
        break;
      case OK:
        if (Bool.ANY())
          products = FULL;
        break;
      case FULL:
        break;
} } }
```

Figure 5.11: *Example of a* datatype abstraction

5.5 Extending the Interface Definition

As we already mentioned, by switching from an object-oriented to a component-oriented design, we make the application topology and dependencies explicit because: (i) every component contains a single thread; (ii) all method invocations are restricted to calls on client interfaces; and (iii) all future creation points are restricted to results of these method calls on client interfaces. This removes some of the imprecision of the static analysis. Nevertheless, in open environments it is still not possible to know whether a parameter (or any subfield) received in a method call is a future or a value. This is due to transparency of first-class futures.

This section suggests an extension to the Interface Description Language (IDL) to improve the precision of analysis and specification; we also explain how this extension prevents the occurrence of some deadlocks.

5.5.1 Principles

In order to be safe, the behavioural model must be an over-approximation of the implementation, including a proxy not only for futures, but also for variables or parameters which *may* be futures. Such imprecision is due to the undecidable nature of static analysis, and to the transparent nature of futures.

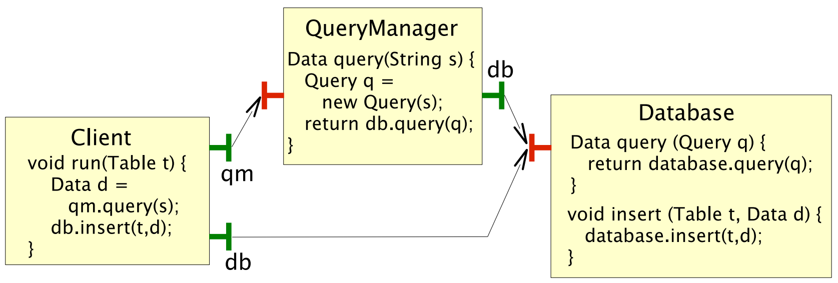


Figure 5.12: *Race condition in GCM / ProActive*

We will start by considering the example of Figure 5.12. It consists of a *Database*, wrapped by a component, that is accessed by a *Client*, yet another component. We have simplified the model such that every component serves requests in FIFO order.

The *Client* component queries for some data, identified by the method call `qm.query(`s`)` which creates a future d. The data s is properly formatted by the *QueryManager* component and then forwarded to the *Database* component. Once the *Client* creates the future in the variable d, it inserts a new entry into the table t with data from d; this is a method call performed directly towards the *Database*.

The system may deadlock, though, due to a race condition on access to the *Database*. If the *Client* accesses the *Database* before the *QueryManager* does, the *Database* will access the future d – thus block –, but d will never be updated because the *Database* itself must update this future. In the example, the deadlock is avoided if one enforces further synchronisation on the *Client* side in order to guarantee that the *Database* always receives a value instead of a future.

In other words, we would like to specify that some variables that are transmitted cannot be futures. Up to now, the only way is by putting within the caller's code a synchronisation for ensuring that the future is filled with the result value. In our example, this would take the form of a statement d`.touch()` in the *Client* component before calling the database. Nevertheless, even under this scenario the *Database* doesn't know that d will certainly contain a value. Thus, the behavioural model for the *Database* still considers that d may be a future.

We have not yet detailed how the interface is signed (by an IDL) in JDC. The GCM IDL specifies the interface signatures, but is insufficient to deal with transparent first-class futures. Based on the interface signature, one does not know whether method parameters are futures or not. Moreover, there is no way of controlling which parameters cannot be futures. Typing futures would solve the issue, however, we would lose all the good properties shown in Section 3.2.3. A better solution is to specify, within the IDL, which parameters (or fields) cannot be futures (i.e. marking them as *strict value*); the other parameters are allowed to be futures or not. This is less restrictive than typing because some parameters can still be either a value or a future.

In an open system this information cannot be inferred by static analysis. It is a contract on futures that affects both client and server: client interfaces must ensure that method parameters match the interface specification; server interfaces assume – and may test – that method parameters agree with the interface specification. The contract also decreases the non-determinism in the behaviour of the server side of the communication.

It is true that by the use of strict parameters there is less concurrency; components have to enforce further synchronisations before performing remote invocations. On the other hand, behavioural models are more precise and closer to real executions; the programmer can specify parameters that are known to be non-futures.

5.5.2 Interface Specification

The difficulty is finding, statically, a proper abstraction for the parameter structure. In theory, every subfield of every parameter may be a future. Therefore, a static representation of arbitrary types is impractical. Here we suggest a relatively precise approximation; marking a field as *strict value*, means that, recursively, all its subfields (known at runtime during serialisation) are *strict values* as well. Similarly, not marking a field implicitly means that, recursively, all its subfields (except the marked ones) *may be futures*. This applies for the IDLs used in both JDC and GCM.

In the example of Figure 5.12, a solution to the deadlock mentioned before is to force value-passing of d. Based on Java 1.5 Annotations the specification of the interface `DB` looks like:

```
interface DB {
   Data query(Query q);
   void insert(Table t, @StrictValue Data d);
}
```

On the practical side, if d is still a *non-filled* future by the time the method `insert(`t`,` d`)` is invoked, the invocation is halted until the future is updated. This way, the system is guaranteed to be deadlock-free.

An implementation in ProActive

The main modification in ProActive would be in the *Meta-Object-Protocol* (MOP). Concretely, the MOP must:

on the client side: during serialisation, any parameter marked as *strict value* will enforce an explicit synchronisation on the related object; the overhead is payed only for methods with annotated futures.

on the server side: during deserialisation, any parameter marked as *strict value* can be checked not to be a future; to avoid overhead, one may assume that the sender respects the contract because it was previously checked during serialisation. Moreover, the affected parameters will never block because they are guaranteed to be concrete values.

5.6 Conclusion

Our contribution in this chapter is to provide a high level specification language for distributed software components, called JDC. Our approach is to define the architecture, the behaviour, and an abstraction of data within the specification language.

The two main design criteria for this language are: (i) the specification is formal enough in order to generate behavioural models that can be model-checked, and (ii) the specification allows for the generation of code skeletons that include the control code of components.

For accomplishing these goals we have mixed two opposed strategies. On the one hand, we define the language in a much higher abstraction level than a programming language; on the other hand, the data part of the language is the one typically found in a programming language. The former, being simple, allows for the generation of guaranteed control code. The latter, being close to the programming language, allows the data part of the generated code to be directly useful.

In order to allow for the generation of behavioural models, the data part is provided with abstractions for the domains. These abstractions map variables of *user-defined types*, to our simple types. The latter are the datatypes supported by our pNets formalism, and thus are able to interface with model-checkers.

The architecture definition is closely related to classic ADLs, defining the component type, its subcomponents, and bindings. More interesting is our approach for defining the component behaviour, in which we give a rough definition of the services provided by the component (service policy), and afterwards a precise definition of the behaviour of each service.

Moreover, we also provide extensions to traditional interface description languages (IDLs) in order to avoid deadlocks. Precisely, we tackle deadlocks due to non-determinism whether a variable is a future or a value. We do this by annotating which arguments in a method call cannot be futures, allowing the unannotated ones to be futures or not. The result is that we lift some synchronisation from the behaviour up to the interface level, which yields more precise behavioural models and avoids some deadlocks.

Limitations. This work builds on the GCM, however, at the moment only the core of it is addressed. We plan to extend the language to cope with other interesting features, such as group communications and non-functional aspects (dynamic reconfiguration).

The abstractions of user classes must guarantee that they are abstract interpretations [Cousot 01] of the user classes given a set of properties. This is not something we would like to give a software engineer without a tool support. Moreover, the abstract version of a user class should be able to deal with operations on sets of abstract domains. In the current definition of the language, we only consider basic operators that select any value within the type domain, or enumerate all of them.

6 Building Behavioural Models

Contents

Abstract

This chapter defines models and algorithms that allow us to generate behavioural models of components specified with JDC. We base the model generation on:

First, the generation of the control part of components; this is based on the analysis of the system architecture, which defines the communication between communication but leaves undefined the functional behaviour of the components. This defines the control behaviour of components.

Second, the generation of pNets encoding the functional behaviour of a component that complements the control model. This is done through a static analysis step on the JDC behaviour specification.

Motivation

In this chapter, we develop behavioural models based on the pNets formalism (see Chapter 4) that can be generated from JDC specifications (see Chapter 5). We show that the design of JDC is akin with the generation of behavioural models.

Architecture: the architecture definition of JDC gives us the topology of the system, by defining the communication links and the component hierarchy. We will use the communication links to create synchronisation vectors that encode the synchronisations and data-flow. The component hierarchy will be used to define the hierarchy of pNets in such a way that it represents a mapping of components into processes.

Abstractions of user types: the abstractions allow us to map variables of user-types to variables of simple-types, the latter being the only supported types in pNets.

Communication strictly through interfaces: the use of strict communications allow us to compute the *Sort* of the pLTSs, and identify the creation of futures.

Decomposition of the behaviour as services: the structure allows us to build the functional behaviour of the component following a pattern described in this section. Roughly, it constitutes of a control part orchestrated by the service policy, and of a functional part driven by the service and local methods.

First, in Section 6.1 we generate behavioural models for the control part of components; this step leaves undefined the Service behaviour. These models allows us to set the communication links between components and map components to processes. This allows us to define the *Sort* of the functional behaviour of components, effectively decoupling the control from the functional behaviour. Extending this approach, we will show in Section 6.1.4 how to include some controllers in the model in order to statically verify properties of some basic reconfiguration and deployment.

Second, as part of the communication links between components, in Section 6.2 we will describe a static representation of futures. In this step, it is important to track which variables hold futures, in order to infer which are the blocking operations, and where we shall put proxies for futures.

Third, in Section 6.3 we generate behavioural models for the Service behaviour. We reuse the structuring of the Service definition in JDC (*service policy + methods*) to fill the pNets of the Service behaviour.

Finally, in Section 6.4 we present two use cases in which our models can be applied.

This chapter is greatly inspired by the work done by Boulifa [Boulifa 04] and Barros [Barros 05]. In [Boulifa 04], the author presented algorithms for automatically generating behavioural models for ProActive (Java) active objects. In [Barros 05], the author presented behavioural models for GCM/ProActive components. Here we adapt and extend these works to components defined with JDC; our contributions are explicitly stressed.

6.1 Control Structure

In this section, we show how the control structure of components defined by JDC can be automatically generated.

Section 6.1.1 defines the component model; this will identify the external actions of the component.

Then, Sections 6.1.2 and 6.1.3 give a structure for filling the component Sort. Section 6.1.2 gives the structure for a component given by a Service definition, and Section 6.1.3 gives the structure for a component given by an Architecture definition.

Finally, Section 6.1.4 extends the previous models when we are interested in checking some basic reconfiguration and deployment.

6.1.1 Sort for the Functional Behaviour

The first thing to do is to map the actions that are observable from the exterior of a component. We know that these will be given by the both client and server interfaces. We compute the Sort for the component, see Figure 6.1.

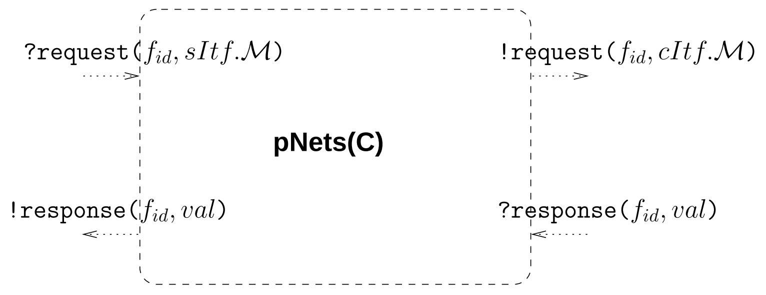

?request($f_{id}, sItf.\mathcal{M}$)
!request($f_{id}, cItf.\mathcal{M}$)
pNets(C)
!response(f_{id}, val)
?response(f_{id}, val)

Figure 6.1: *Sort of the pNet of a component*

For each server interface *sItf*, an incoming remote method call is of the form: `?request(`f_{id}, *sItf*.$\mathcal{M}$`)` that represents a request for a method call $\mathcal{M}$, which result should update the future f_{id} – we shall implicitly denote f for f_{id} when it is convenient. The response (update of the future) is an action `!response(`f_{id},*val*`)`; it updates the future f_{id} with the value *val*. In the case the result type of the method call is `void`, there is no return action.

For each client interface *cItf*, an outgoing remote method call is of the form: `!request(`f_{id}, *cItf*.$\mathcal{M}$`)` that represents a request for a method call $\mathcal{M}$, which result should update the future f_{id}. The response is an action `?response(`f_{id},*val*`)`, if any.

Now, depending whether a component provides us with an architecture describing its subcomponents or with a service definition, we will create two models that fill in his pNet.

6.1.2 Components defined by a Service Definition

When the component is defined by a Service definition, the behavioural model consists in a hierarchy of pNets as shown in Figure 6.2. Basically, requests arrive at the component's request queue, and the service (here called Body) serves requests from queue or performs work of its own. Moreover, this is a model as well for a primitive component in GCM/ProActive.

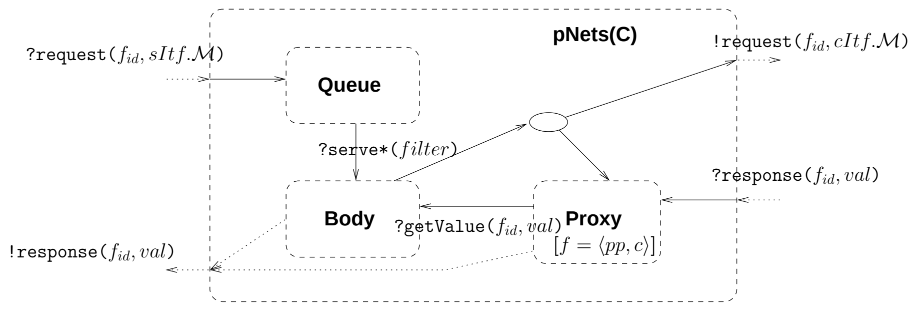


Figure 6.2: *Behavioural Model of a component*

Request Queue. The request queue is modelled by a pNet that en-queues all external request calls, and allows one to select en-queued requests.

The queue can be modelled in different ways, the most straight-forward (and naïve) being a pLTS encoding the behaviour of the queue. This is a structure that has external actions for en-queueing a request (`?request` (f_{id}, *method*)), and for dequeueing a request with a specific policy (`?serve`* (*filter*)). `serve`* is the policy (either `serveOldest` or `serveYoungest`). However, care must be taken when creating the queue, because in general the state-space is exponential in the queue length and in the number of possible requests. Remark that the number of possible requests depend on the number of different parameters once instantiated with finite domains.

The queue in GCM components is meant to be infinite. Therefore, abstracting the component queue into a finite-state representation requires the designer to find, if any, the maximum number of simultaneous requests in the queue. In the ProActive implementation, if ever the component queue is full, the callee component will be blocked trying to complete the rendez-vous protocol (see Section 3.2.1) which is not the normal behaviour of GCM components.

Proxies for Futures. The remote calls on client interfaces c*Itf* are actions performed by the Body, though synchronised by the future proxy. The proxy is a pNet that encodes the asynchronous behaviour of components. Once the call is performed, the proxy is ready to receive the result value. The Body can synchronise with the Proxy with the action `?getValue` (f_{id},*val*) to access the content of the future.

We will detail the behaviour of proxies in Section 6.2, however for the sake of this section we introduce the main concepts here. The Proxy is given by a pLTS created for each remote method call (method call on a client interface). Futures are tagged with the program point *pp*, and with a counter *c*. The counter is required in order to allow families of futures to be created; more precisely, in a loop, two successive remote method calls at the same program point create two different futures at the same program point. This is a finite parameterized approximation considering that the loop is finite.

In the behavioural model, this means that there is a proxy for each one of these futures, i.e. the proxy is parameterized by $f = \langle pp, c \rangle$. There are different models for proxies depending whether the future is transmitted or not. Moreover, we will see in Section 6.2 that if a future is transmitted as a return value, then the proxy must be in charge of sending the response action `!response(`f_{id},val`)`. Otherwise, it is up to the Body to return this value as a usual method call.

Body. The component Body is a pNet that implements the component (user) functional behaviour. In JDC, this is given by the Service definition in JDC through the *service policy* and the *service and local methods*. We create a pNet that fills the body from this information, which will be detailed in Section 6.3.

6.1.3 Components defined by an Architecture Definition

An Architecture specification in JDC is the specification of a composite component. We model the composite component as a pNet, and each of its subcomponents as a pNet as well. This is a hierarchical construction of the behavioural model.

In GCM, composite components have a membrane that interacts with the environment. The membrane dispatches incoming method calls to internal subcomponents, and dispatches method calls going out the component to the environment. This can be seen as a primitive component that serves requests from the queue and performs (dispatches) the request to the bound interfaces. Therefore, we model the membrane as a pNet similar to the component from Figure 6.2.

Structure. The composite is modelled by a pNet with a control part (the membrane), and pNets for each subcomponent. The structural information is found within the *Architecture* definition in JDC, or more generally through ADLs in the case of Fractal or GCM.

Supposing that the component C has 2 subcomponents A and B as in Figure 6.3, this will create a pNet as in Figure 6.4.

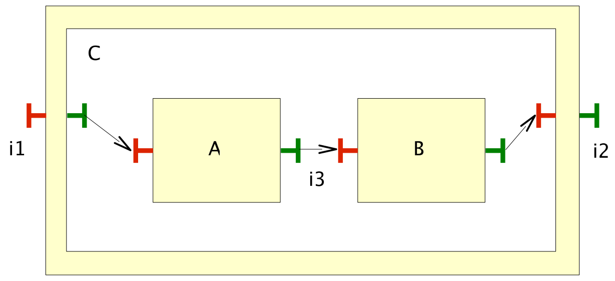


Figure 6.3: *A composite component exposing its architecture*

Note that the internal communication between components A and B do not go through the

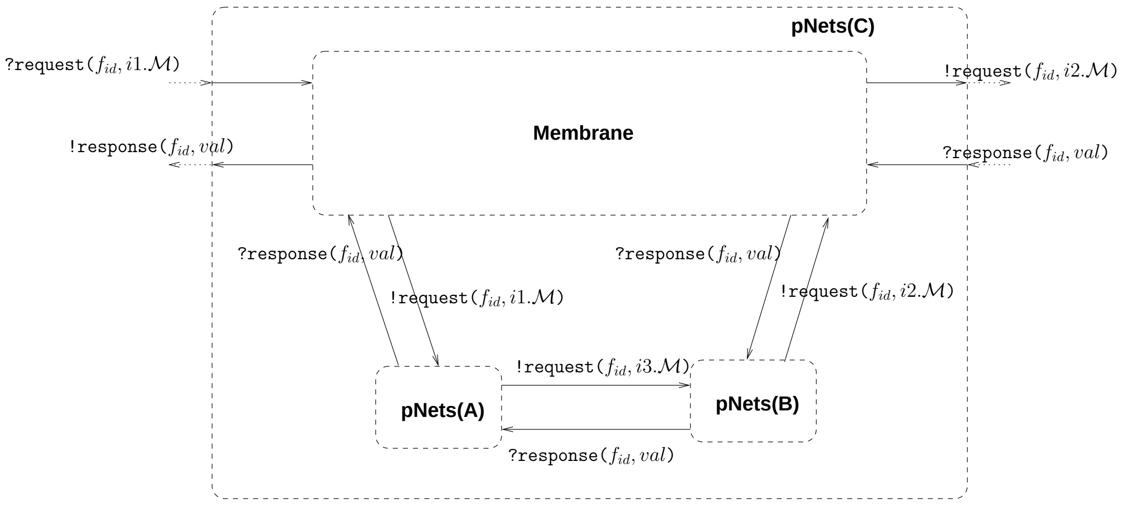

pNets(C)
?request($f_{id}, i1.\mathcal{M}$)
!response(f_{id}, val)
Membrane
!request($f_{id}, i2.\mathcal{M}$)
?response(f_{id}, val)
?response(f_{id}, val)
!request($f_{id}, i1.\mathcal{M}$)
?response(f_{id}, val)
!request($f_{id}, i2.\mathcal{M}$)
!request($f_{id}, i3.\mathcal{M}$)
pNets(A)
pNets(B)
?response(f_{id}, val)

Figure 6.4: *Behavioural model of a composite component*

membrane, however all external communication does. The pNets of A and B have well-defined Sorts, so they can later be filled in independently.

Membrane. As the membrane is a dispatcher of services, we know exactly its functional behaviour. In Figure 6.5 we depict the membrane behaviour. It is a pNet as follows:

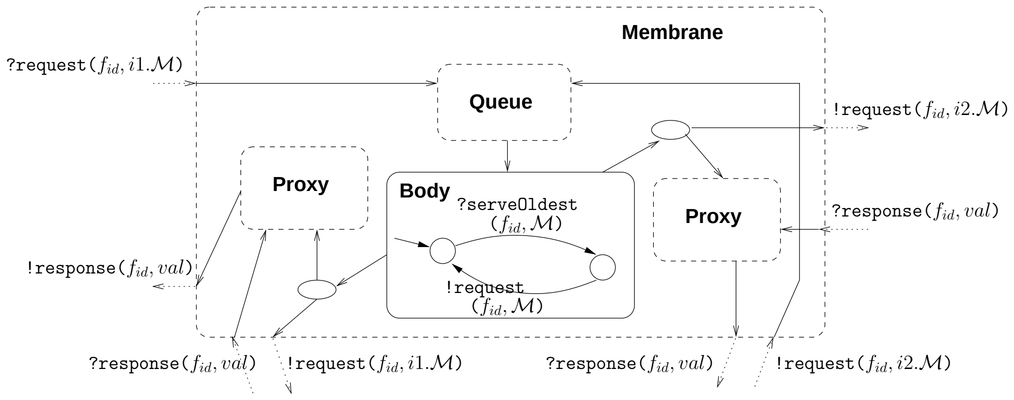

Membrane
?request($f_{id}, i1.\mathcal{M}$)
Queue
!request($f_{id}, i2.\mathcal{M}$)
Proxy
Body
?serveOldest ($f_{id}, \mathcal{M}$)
Proxy
?response(f_{id}, val)
!response(f_{id}, val)
!request ($f_{id}, \mathcal{M}$)
?response(f_{id}, val)
!request($f_{id}, i1.\mathcal{M}$)
?response(f_{id}, val)
!request($f_{id}, i2.\mathcal{M}$)

Figure 6.5: *Behavioural model of a component membrane*

- it has a queue that represents the composite component's request queue; both requests coming into the component and going out of the component are en-queued in this queue.
- it has a Body that dispatches method calls to the bound interfaces; the Body serves requests and its only action is to perform a new request on the bound interface affected by the request. If the method call is not `void`, then there will the a response. However, the response is not handled by the Body, but by the proxies. This way the Body is free to process the next request.
- proxies for retransmitting the response values to fill futures. In Figure 6.5 we show two proxies, however they are identical. They only help us understand the membrane behaviour. The **proxy on the left** is in charge of communication due to the component's *server* interfaces; it will forward the response values coming from the inner subcomponent

A to the environment. The **proxy on the right** is in charge of communication due to the component's *client* interfaces; it will forward the response values coming from the environment to the inner subcomponent B.

6.1.4 Extending the Model with Controllers

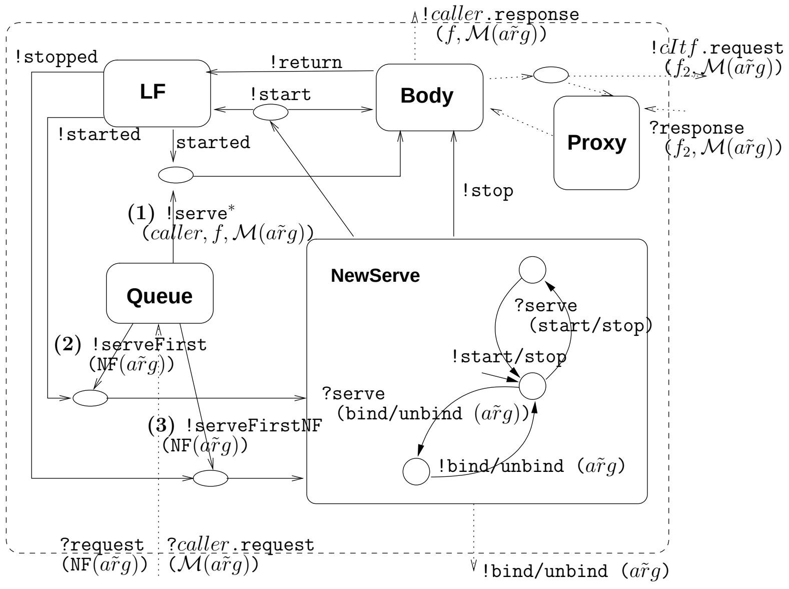
The non-functional control of components was previously given by Barros [Barros 05], and more formalised in [Barros 08].

We have changed the communication between the Body, Proxy, and the environment; this avoids a causal ordering problem in the model shown in [Barros 05]. We have also defined how the Binding Controller interacts with the environment.

The previous models were suitable for representing the functional behaviour of components. Even if in JDC we still do not have any reconfiguration primitives, it is still interesting to included these controllers. For instance, check for the correct system deployment, or model the behaviour of some basic reconfiguration. For that, we can automatically generate controllers *à la Fractal/GCM* from the structural information.

6.1.4.1 Primitive Components

For the moment we do not describe how the components deal with futures (see Section 6.2). In Figure 6.6, we first study some control processes, namely a controller for dealing with the component's life-cycle (**LF**), and a controller for serving functional and non-functional requests (**NewServe**).



!$caller$.response ($f, \mathcal{M}(\tilde{arg})$)
!stopped
LF
!return
!start
Body
!$cItf$.request ($f_2, \mathcal{M}(\tilde{arg})$)
Proxy
?response ($f_2, \mathcal{M}(\tilde{arg})$)
!started
started
!stop
(1) !serve* ($caller, f, \mathcal{M}(\tilde{arg})$)
Queue
NewServe
?serve (start/stop)
!start/stop
(2) !serveFirst (NF($\tilde{arg}$))
?serve (bind/unbind ($\tilde{arg}$))
(3) !serveFirstNF (NF($\tilde{arg}$))
!bind/unbind ($\tilde{arg}$)
?request (NF($\tilde{arg}$))
?$caller$.request ($\mathcal{M}(\tilde{arg})$)
!bind/unbind ($\tilde{arg}$)

Figure 6.6: *Behaviour model for a GCM/ProActive Primitive*

The component's life cycle is controlled by a pLTS called **LF**. Its behaviour is described in

Figure 6.7, which is simply a machine aware of the component's state, either started or stopped, and that signals this state to the rest of the component parts.

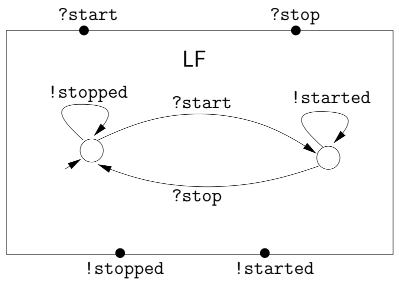


Figure 6.7: *pLTS for the Life Cycle (LF) controller*

NewServe implements the treatment of control requests. The action "start" fires the process representing the method `runActivity()` in the **Body**. "stop" triggers the `!stop` synchronisation with **Body** (Figure 6.6). This synchronisation should eventually lead to the termination of the `runActivity()` method (`!return` synchronisation).

In JDC, we suppose that the component is immediately set to start; however, a trivial extension is to add an initialisation phase, the "normal" activity and then a finalisation phase. These correspond to the `initActivity`, `runActivity`, and `endActivity` proposed in Figure 7.5 in Page 155. The end of the initialisation would wait first wait for a signal to start activity, and before serving each new request, the component checks whether it has been set to stop by consulting the life-cycle controller.

The **Queue** pNet can always perform any of these three actions:

1. serve the first functional method corresponding to the `serve` API primitive used in the body; this is reflected by either `serveOldest` or `serveYoungest` primitives of JDC.
2. serve a control method only at the head of the queue; and
3. serve only control methods in FIFO order, bypassing the functional ones.

Which of these actions is performed will depend on the life-cycle controller. The latter constraints the possible actions by synchronising on either `!started` or `!stopped` actions. More precisely, the synchronisation vectors are affected by the actions `!started` and `!stopped` coming from the **LF** pLTS. This virtually controls which actions will be allowed within the component's pNets.

It is relevant to say that these three behaviours are exactly those implemented by the ProActive middleware, supposing no advanced features are used (i.e. overridding the default component behaviour with a customised one). Stopping a component in the GCM means that its functional activity is suspended, while NF calls are still processed; the latter allow reconfiguring the component. In our model, reconfiguration actions will only be allowed when the life-cycle is ready to emit the `!stopped` action.

6.1.4.2 Composites Components

The implementation of the GCM composite component (Figure 3.8 in Page 78) is modelled by Figures 6.8. Figure 6.9 is the component membrane that fits in the *interceptor* of Figure 6.8.

In Figure 6.8, we show the behaviour for composite components. The behaviour of subcomponents is represented by the box named *SubC*$^{\mathbf{k}}$. For each interface, there is a Binding Controller with the pLTS; *cII* and *sII* for internal interfaces, and *cEI* and *sEI* for external interfaces. Primitive components have a similar automaton without subcomponents and internal interfaces. The interceptor of this drawing represents the behaviour of the component's membrane.

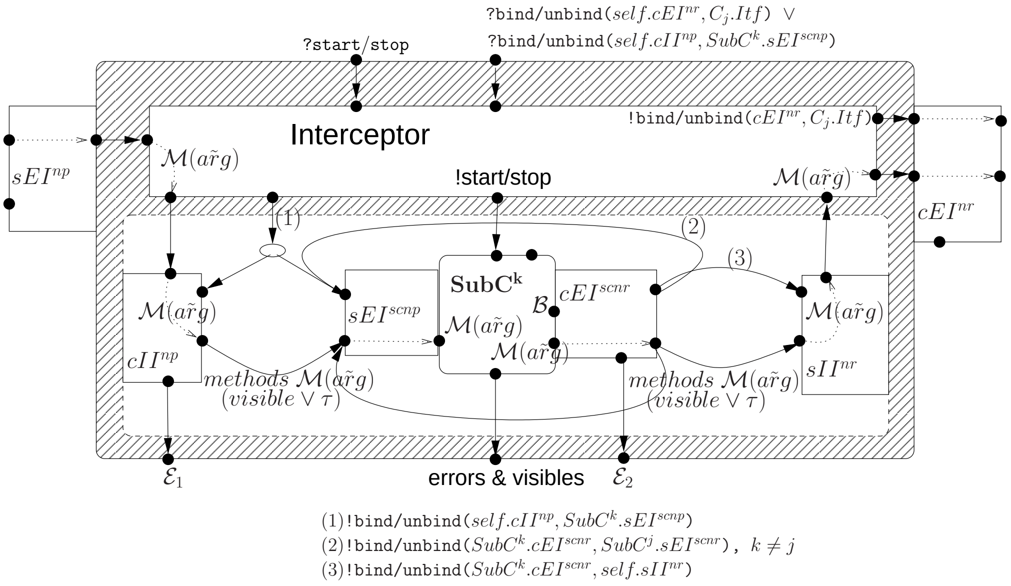

?start/stop
?bind/unbind(self.cEI^nr, C_j.Itf) ∨
?bind/unbind(self.cII^np, SubC^k.sEI^scnp)
Interceptor
!bind/unbind(cEI^nr, C_j.Itf)
sEI^np
M(arg)
!start/stop
cEI^nr
(1)
(2)
(3)
SubC^k
B
cEI^scnr
sEI^scnp
M(arg)
cII^np
sII^nr
methods M(arg) (visible ∨ τ)
E_1
errors & visibles
E_2
(1)!bind/unbind(self.cII^np, SubC^k.sEI^scnp)
(2)!bind/unbind(SubC^k.cEI^scnr, SubC^j.sEI^scnr), k ≠ j
(3)!bind/unbind(SubC^k.cEI^scnr, self.sII^nr)

Figure 6.8: *Synchronisation pNet for a GCM Composite Component*

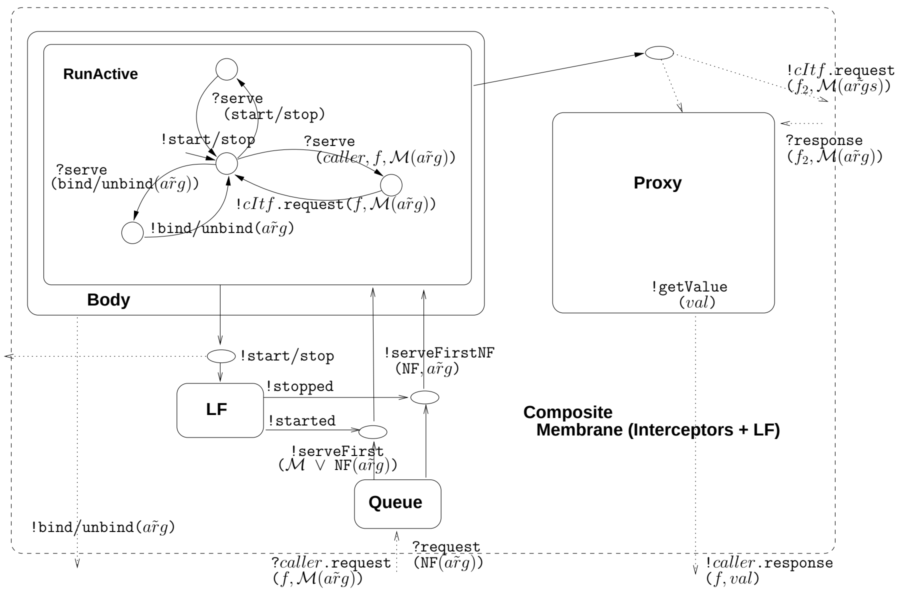


Figure 6.9: *Behaviour of a composite membrane*

Membrane. The membrane of a composite component in GCM/ProActive is an active object. When started, it serves functional or control methods in FIFO order, forwarding method calls between internal and external functional interfaces. When stopped it serves only control requests. The membrane with controllers is depicted in Figure 6.9. This is an extended version of the model from Figure 6.5.

The Queue and LF (Life-Cycle Controller) in Figure 6.9 are the same as presented in Section 6.1.4.1.

Binding Controller. The interfaces of a component abstract the component from its environment. Moreover, interfaces are in charge of routing calls coming in and going out the component. For the same reason we need to include in our pNet models controllers that simulate the behaviour of the interfaces. We call these Binding Controllers (**BC**). In Figure 6.10 we show such a controller for an interface Itf_1 of component C.

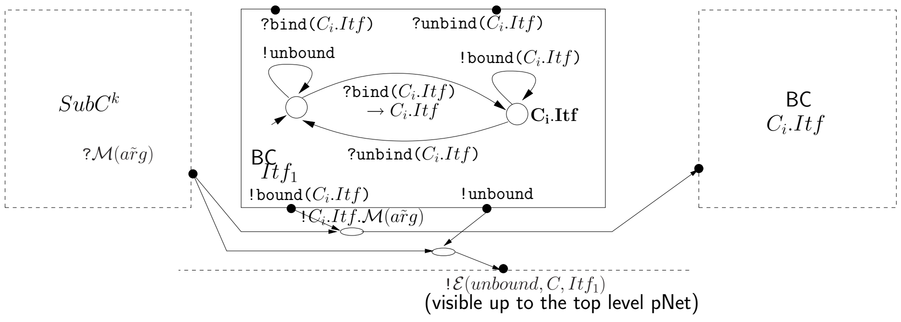

$SubC^k$
$?\mathcal{M}(\tilde{arg})$
?bind($C_i.Itf$)
?unbind($C_i.Itf$)
!unbound
!bound($C_i.Itf$)
?bind($C_i.Itf$) $\rightarrow C_i.Itf$
$\mathbf{C_i.Itf}$
?unbind($C_i.Itf$)
BC Itf_1
!bound($C_i.Itf$)
!unbound
$!C_i.Itf.\mathcal{M}(\tilde{arg})$
BC $C_i.Itf$
$!\mathcal{E}(unbound, C, Itf_1)$
(visible up to the top level pNet)

Figure 6.10: *pLTS for the Binding Controllers*

The **BC** is an automaton that memorises the attached interface. Interfaces may be bound and unbound. The expected behaviour is to trigger an error if a method call is performed over an unbound interface, or to redirect the call if it is bound.

An unbound error is a distinguished error action $!\mathcal{E}(unbound, C, Itf_1)$, visible at the higher level of hierarchy. When the interface is bound, the call will be forwarded (synchronously) to the environment.

6.2 Transparent First-Class Futures

The results of this subsection were published in [Cansado 08b].

Up to this point we have described how to automatically generate the control part of components. Missing parts are: how the futures can be modelled, detailed in this section; and how to create a pNets for the functional behaviour, detailed in Section 6.3.

The objective of this section is to give a behavioural model for transparent first-class futures. We assume that the accesses to the component interfaces and the creation point of futures are given in the functional behaviour of the component (Body). We call *future update* the operation consisting in replacing a reference to a future by the value that has been calculated for it.

6.2.1 Static Representation of a Future

The representation of a future must allow the contained object to be accessed, i.e. to synchronise futures. We call `waitFor` the primitive allowing the update of a future to be awaited (this primitive has also been named `touch` or `get` [Flanagan 99]). When futures are transparent, this waiting operation is automatically performed upon an access to the content of the future. For the moment, we consider that futures cannot be passed between remote entities, and thus the future is necessarily accessed by the same entity that created it (at another point of the execution).

The objective is to be able to provide a model for the following piece of code:

```
f=itf.foo();      // creation of a future
if (Bool)
   f.bar1();      // wait-by-necessity if bool is true
f.bar2();         // wait-by-necessity only if bool is false
```

In this piece of code, if `f.bar1()` is executed, then `f` must be filled; in this case `f.bar2()` will be necessarily non-blocking. Otherwise, `f.bar2()` may or may not be blocking depending if the future `f` is already filled by the time the call is performed. Note that it is much simpler in frameworks in which futures are explicit, i.e. if futures are typed.

The previous example shows that futures are filled transparently at any time. Thus, whenever it is not statically decidable whether an object is a future or a value, it must be assumed as a future. This is an over-approximation that will, at least, include all possible synchronisations a variable may trigger. Therefore, static analysis of a program with futures requires the set of abstract values to be multiplied by two.

Indeed, statically each variable is either known to contain a value which *is not a future*, or, equivalently, a filled future, ranging in the domain of the usual static domain for values; or the variable *may be a future*, and when the future will be filled its value will range in the domain of the usual static domain for values. Note that an object that is not a future is semantically equivalent to a filled future. In abstract interpretation [Cousot 77] it would be easy to construct a lattice for this new abstract domain: suppose without futures, the abstract domain is a lattice $(D, \prec)$, then the new abstract domain taking futures into account is the *lattice* $D' = D \cup \{fut(a) | a \in D\}$ equipped with the order $\prec'$ built such that if $a \prec b$, then $a \prec' b$, $a \prec' fut(b)$, and $fut(a) \prec' fut(b)$. Indeed the abstract value a gives more information than $fut(a)$. To summarise, statically, the value for our objects are either "filled" or "potentially non-filled"; these abstract values are composed with the usual abstract values required for the analysis.

In the ProActive middleware, the example above creates a *proxy* in the first line, and all calls to the future stored in `f` would go through the proxy object, leading, if necessary, to a wait-by-necessity. For our model, the idea is the same: the variables have the "classical" static abstract domain, and the augmented lattice is taken into account by an additional automaton with the behaviour of a proxy. Initially, the proxy is in an empty state where the object can only be filled with a value, so any access to the variable will be blocking. In general, two instances of the same method call have two different futures, so the proxies are parameterized by the instance of method call.

6.2.2 Locally-used Futures

In Figure 6.11, we show a first model on how two components communicate. This corresponds to the pNets model of the communication between two activities in ProActive depicted in Figure 3.7 in Page 77.

The action `request(`f`,`$\mathcal{M}$`(args))` puts the request in the server's queue, and initialises the local proxy. The call contains the *identifier* of the future to be updated, f. Once computed, the value of f is updated (`response(`f, val`)`). The pair $\langle pp, c\rangle$ is the future identifier f.

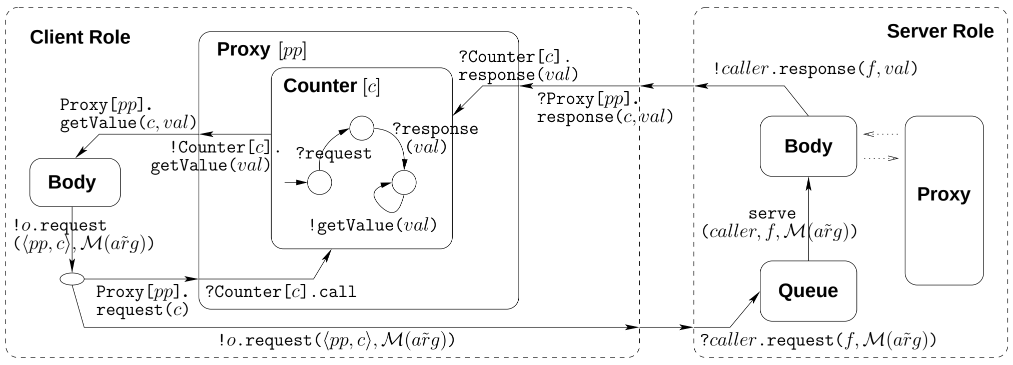

Client Role
Proxy [pp]
Counter [c]
?Counter[c].response(val)
?Proxy[pp].response(c, val)
Proxy[pp].getValue(c, val)
!Counter[c].getValue(val)
?request
?response (val)
Body
!getValue(val)
!o.request(⟨pp, c⟩, M(arg))
Proxy[pp].request(c)
?Counter[c].call
!o.request(⟨pp, c⟩, M(arg))
Server Role
!caller.response(f, val)
Body
Proxy
serve (caller, f, M(arg))
Queue
?caller.request(f, M(arg))

Figure 6.11: *Model of a communication between two components*

The model shown in Figure 6.11 is similar to the ones presented by Boulifa [Boulifa 04] and Barros [Barros 05]. In this work, we slightly changed the encoding of the rendez-vous (see Section 3.2.1); the original proposal did not guarantee causal ordering. Moreover, we decided to give the behaviour of the proxy explicitly as a hierarchy of pNets, one parameterized with the program point (pp), and another parameterized with a counter (c).

6.2.3 Transmission of Futures

The models presented by Boulifa [Boulifa 04] and Barros [Barros 05] did not allow futures to be transmitted. This subsection extends the models into this direction.

Futures can be transmitted in the parameters of a method call, or in the return value of a method call; in all of these cases, the futures are transmitted in a non-blocking manner. Because of that, a future in an activity may have been created locally or by a third-party. In both cases, the activity is aware of the future *identifier*. The references of futures known by a component are local. Only when synchronising the components the references must match the data-flow. In pNets, the matching of the references is done using synchronisation vectors; they allow us to synchronise different labels which we use to link the future references. This technique allows us to create the behavioural model for each component independently. The proxies will therefore include an action `!forward(`*val*`)` forwarding value *val*.

On the practical side, different future update strategies can be designed for propagating the values that should replace future objects. Despite having differences in performance, the update policies have equivalent behaviour, proved using ASP in [Caromel 05b]. This leaves freedom to choose any update policy. Indeed, the behavioural models proposed behave similarly to a concrete implementation, but have different update strategies. We will first show some properties our models.

Let $\sigma_{\mathtt{C}}$ be a valid execution on component `C`, `pNets(C)` the behavioural model of `C`, and f_{id} a future, then the model is built in such a way that:

PROPERTY 2
if `getValue(`f_{id}, val`)` *in* $\sigma_{\mathtt{C}}$*, then* `Proxy(`f_{id}`)` *is in* `pNets(C)`

Comment

As a consequence of this property, the model has a proxy dealing with every future a component may receive. Due to imprecision of the abstraction, the component may even have proxies for futures that would never exist at run-time. However, any potential synchronisation is considered within the model.

PROPERTY 3
if the value of f_{id} is computed, then all proxies of f_{id} are updated eventually

Comment

The property is true even for proxies that don't exist at run-time. The property is guaranteed by construction: (i) *the proxy that creates f_{id}* initially synchronises with the remote method call. The proxy then waits for the result (value of f_{id}). When the value of f_{id} is updated, the proxy forwards the value of f_{id} to all components to which the local component has sent the reference f_{id}. (ii) *all other proxies of f_{id}* are initially in a state in which they are ready to receive the value of f_{id}; this guarantees they will also be able to be updated. When the proxy is updated, it forwards the value of f_{id} to all components to which the local component sent the reference f_{id}. In fact, a proxy forwards the value of a future to all components it has sent the reference to synchronously. Therefore, each proxy only needs one port "`forward`" for each future, independently of the number of components to which it sent the reference.

PROPERTY 4
the update of proxies that do not exist at run-time has no influence in the behavioural model

Comment

Depending on the data-flow, some components will receive the value of f_{id}, though the reference f_{id} was not transmitted. In this case, the reference f_{id} is also unknown to the given component, and thus the content of the future is innaccessible, i.e. the business part of the component is not affected.

We will provide a model for each scenario. When:

1. a component sends a future created locally as a method call parameter,
2. a component sends a future created by a third-party as a method call parameter,
3. and a component returns a future within the return value of a method call.

Sending a future created locally as a method call parameter. In Figure 6.12, the `Client` performs a method call $\mathcal{M}_1$ on `Server-A`, and creates a `Proxy(`f`)` for dealing with the result. Then the `Client` sends the future to a third activity (`Server-B`) in the parameter of the method $\mathcal{M}_2(f)$. From `Server-B`'s point of view, there is no way of knowing if a parameter is (or contains) a future, so every parameter in a method call must be considered as a potential future. `Server-B` includes, therefore, a proxy for dealing with the parameter f of the method call $\mathcal{M}_2$. For the sake of comprehension, however, in the figure the identifiers for futures already match the data-flow.

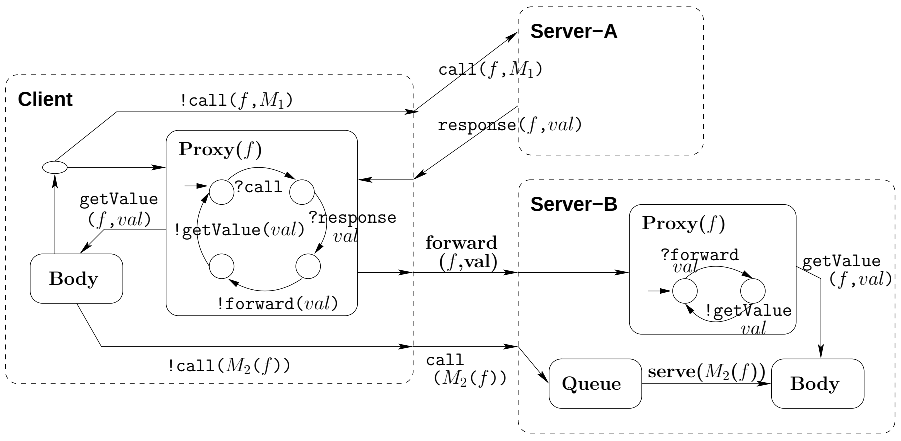


Figure 6.12: *Model for sending a future created locally as a method call parameter*

Sending a future created by a third-party as a method call parameter. The previous example is extended such that `Server-B` transmits the future f to `Server-C`. This is partially depicted in Figure 6.13. The proxy in `Server-B`, after receiving the value of the future (`?forward(`*val*`)`), forwards the value to the components it has sent the future reference.

Returning a future. In Figure 6.14 an activity (`Server-B`) creates a future f_2 and then transmits f_2 to the `Client` within the result of the method call $\mathcal{M}_1(args)$. The behavioural model

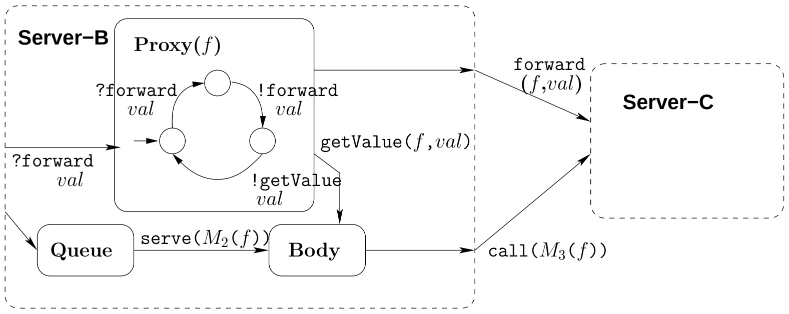


Figure 6.13: *Model for sending a future created by a third-party as a method call parameter*

is slightly different from the one in ASP: instead of returning a future, there is a proxy on the server in charge of forwarding the concrete value once it is known; no value or future is sent to the `Client` in the meanwhile. Using this mechanism, the behavioural model of the `Client` is the same no matter whether `Server-B` returns a value or a future. Moreover, `Client` remains as usual; the result of the method call $M_1(args)$ has a proxy `Proxy(`f_1`)` dealing with the result. It is up to the proxies of the `Client` and the `Server-B` to synchronise in order to match the expected behaviour. Concretely, the action with the response to the `Client` (`response(`f_1, val`)`) is synchronised with the forward action (`forward(`f_2, val`)`) of the `Proxy(`f_2`)`; it will then update the `Proxy(`f_1`)`. If the `Client` accesses the future, then it synchronises with its local proxy, `Proxy(`f_1`)`.

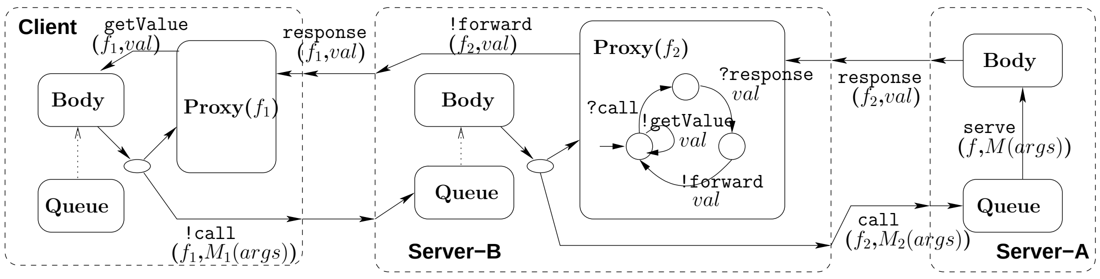


Figure 6.14: *Model for returning a future*

6.2.4 Summary: How to Build a Future Proxy?

We showed in this section that it is possible to specify the behaviour of proxies for futures providing a good approximation of the future flow is given. To summarise:

- Each proxy finishes with a `!getValue` transition to allow the access to the future value.
- At the future creation point (i.e. on the caller side of a remote method invocation), a proxy starts by two transitions: `?call` for synchronising with the remote call, and `?response` for synchronising with the response.
- In the other activities that can *receive* the future, the proxy starts a single transition `?forward` for receiving the forwarded future value
- If the activity may send the future to another one, then the `!getValue` transition is preceded by a `!forward` one.
- Proxies that are only used for transmitting a future reference as the value of another future

do not synchronise on the action `!getValue` because they simply forward the value they receive as the value for *another* future. Assigning a future reference to another future is directly ensured at a higher-level, that is to say by the composition itself. This ensures that the behavioural model is still compositional as no name of an externally created future exist in the proxy.

6.3 Functional Behaviour

In this section we generate the functional behaviour of the component. This is the behaviour that is specified within the *black-box* view of the JDC specification.

The behaviour will fill-in the *Body* pNet of the Service structure (Section 6.1.2). For this, we will suppose that the control part of the component is already built and we will provide a suitable pNet with the missing behaviour, i.e. a *closed* pNet for the Body which call the *Service* pNet.

To create the functional behaviour, we have supposed that the JDC specification has been provided with abstractions for variables of user classes (see Section 5.4).

6.3.1 Model for a Component Service

The *Service* pNet is built from the JDC behaviour specification of a component. It is a pNet that refines the Body pNet, by defining subnets for the service policy, for the service methods, and for the local methods. In Figure 6.15 we sketch the behavioural model for the Service.

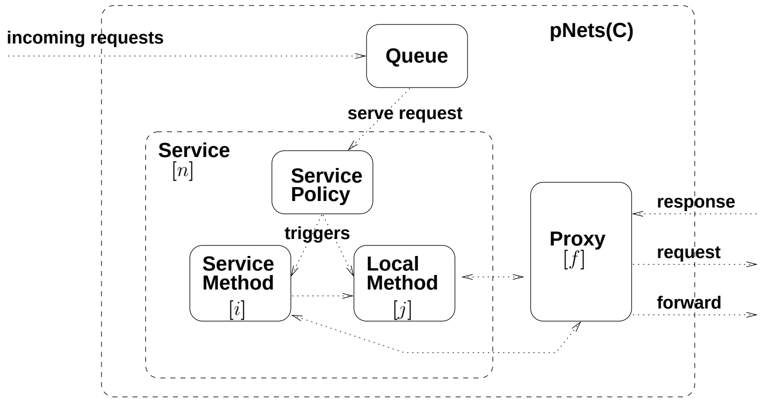


Figure 6.15: *Model of a component with multiple services*

We also need a model for dealing with multiple services. As each service in JDC is an independent sequential program, we define a pNet with a unique entry point. All the entry points are triggered by the component's request queue that feeds all services.

Component Variables. Care must be taken with the component's variables. They are shared between the component's methods. On the contrary, there is no shared memory in pNets.

The shared variables are set as parameters of the top pNet of each component. Inner pNets have their own parameters, and for all parameters that represent shared variables, they must be renamed in order to match the name of variable on the top.

Implementation of shared memory

The implementation of this shared memory mechanism must rely on the instantiation tool. One way is to define shared variables as parameters present in all actions; this literally means the component state is pushed into each process activation, and pulled back when returning to the caller process. In the thesis of Barros [Barros 05], he proposes a tool named `FC2Instantiate` that provides such functionality. In the tool, all variables are global, meaning that assignments of variables are performed synchronously among all processes.

6.3.2 Model for the Service Policy

In JDC, the service policy is described by a regular expression (see Section 5.3.1). Hence, the transformation to pNets is straightforward.

The *reactive* behaviour is transformed into two actions, one synchronised with the queue (`!serve(`$Itf.\mathcal{M}(\tilde{arg})$`)`), and another that fires the affected service method (`!call(`$Itf.\mathcal{M}(\tilde{arg})$`)`). The serve will be an instance of either `serveOldest` or `serveYoungest` and the behaviour is handled by the Queue. When the service method is fired, the service policy is blocked until the called method returns with an action `?return`.

Similarly, the *active* behaviour is transformed into an action that fires the method directly. It will block the service policy until it returns with an action `?return`.

A concatenation (`Policy` ';' `Policy`) of two policies is done by a deterministic internal transition that links both partitions of the pLTS.

A choice (`Policy` '|' `Policy`) of two policies is done by a non-deterministic internal transition.

The repetition (`Policy` 'n') uses a counter variable for simulating a loop.

The permanent policy (`PermPolicy` '*') is a non-deterministic choice between repeating the same policy, or finishing the component activity. We can also provide a variation in which it only leaves the loop if the component's life-cycle controller is set to stop.

Example

We use the classic Dinning Philosophers example in this section. The system architecture is depicted in Figure 6.16.

Figures 6.17 and 6.18 provide the service policies and the pNets generated from the service policies of the Philosopher component and the Fork component respectively.

The Philosopher is an "active" component in the sense that it continuously performs remote method calls without synchronising with its request queue. On the other hand, the Fork is a "reactive" component as it responds to requests.

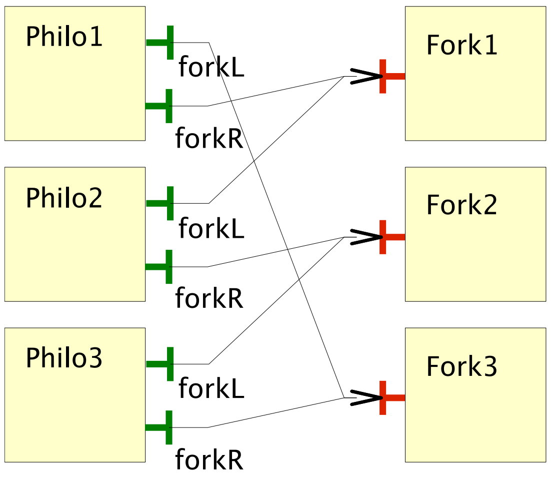
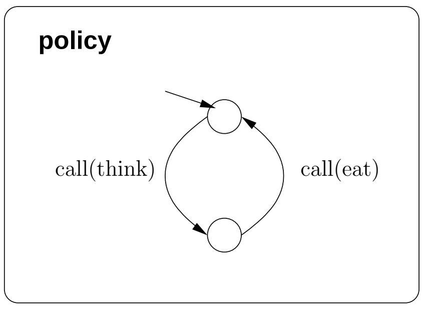
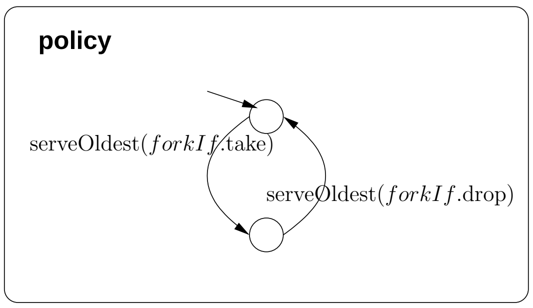


Figure 6.16: *Philosopher System Architecture*



```
policy {
   (
     think();
     eat()
   ) *
}
```

Figure 6.17: *pNets generated for the Philosopher's service policy*



```
policy {
   (
     serveOldest("forkIf.take");
     serveOldest("forkIf.drop")
   ) *
}
```

Figure 6.18: *pNets generated for the Fork's service policy*

6.3.3 Model for the Service and Local Methods

For building the pNets with the behaviour of the methods in JDC, we need to perform some static analysis on the specification. It is not the goal of this work to define a static analysis tool. On the contrary, we shall define here what can be expected from a static analysis tool, and use these results later in the model generation. Then, we create a pLTS for each service and local method.

An advantage of working on JDC is that the analysis is easier that when using Java active objects as in [Boulifa 04]. User-classes are replaced by abstract versions that will not contain any remote method invocations, the access to futures is identified, and business code is not present. In other words, the Java code is abstracted (in the sense of data types and domains), sliced (no business code), and simplified (access to data and remote method calls).

6.3.3.1 eXtended Method Call Graph

> *In JDC, the initial behaviour of a component is necessarily given by its* service policy*. Therefore, we do not need to analyse which is the entry point of the XMCG. Moreover, we are absolutely sure which are the components, whereas in the previous work [Boulifa 04], the active objects were dynamically assigned and instantiated. In other words, determining if an object was active or not was not always statically decidable.*
>
> *The analysis on active objects by Boulifa [Boulifa 04] was imprecise w.r.t. whether calls were local or remote, thus the analysis required to include an additional edge with an non-deterministic choice between local and remote method calls.*
>
> *In JDC, this is fixed because we know that calls on client interfaces are remote, otherwise they are local.*

Each component will be statically analysed in order to define its eXtended Method Call Graph (XMCG). The XMCG [Boulifa 04] is a structure containing: the results of usual class and control-flow analysis; sequential code encoding the data-flow; and constructs relative to JDC, namely futures, remote method calls (and responses), and accesses to the component's request queue.

An XMCG is a tuple $\langle \mathcal{M}, V, \xrightarrow{calls}_C, \xrightarrow{succs}_T \rangle$, where:

- $\mathcal{M}$ are methods,
- V is a set of nodes,
- $\xrightarrow{calls}_C$ is an inter-procedural (method calls) transfer relation,
- and $\xrightarrow{succs}_T$ is an intra-procedural (sequential control) transfer relations.

The nodes in V are typed as:

- *ent* $(c, m, args)$ the entry node of method $m \in M$, called by object c,
- *call* $(calls)$ encoding method calls (local or remote),
- *pp* (lab) encoding an arbitrary program point with label *lab*,
- *ret* (val) encoding the return node of a method with result value *val*,
- *use* (f, val) encoding the access point of future f with value *val*.

call nodes have a set of non-deterministic outgoing method call edges, *calls*(n), with $\forall c$ in *calls*(n), $\exists n'.\langle n, c, n'\rangle \in \xrightarrow{calls}_C$, each call being either:

- *Remote* ($itf.m, args, var, f$) for a call to method m of a client interface *itf* through the proxy f,
- *Local* ($o.m, args, var$) for a call to method m of a local object o.

Sequential Programs. In order to provide concise behavioural models, the goal is to group code that encodes the data-flow as an atomic (local) action. We look for the largest set of statements in which there are no communication nor synchronisation. This is depicted in the XMCG by letting all paths within a sequential program to have at most one communication, synchronisation or local call (the frontiers).

The outgoing transfer edge is a tuple $\langle succs(n) = MT, N\rangle$, with $\langle n, MT, N\rangle \in \xrightarrow{succs}_T$. MT is a meta-transition; it represents a sequential program with a non-empty set of resulting states N. The resulting states are therefore the exit points of the meta-transition.

Frontiers of the Meta-Transitions. Communication, synchronisation, and local method calls are frontiers of the meta-transitions. Therefore, they must be identified in the static analysis step. They correspond to nodes of type *call* (*calls*) and *use* (f, *val*) in the XMCG.

Communication is restricted to calls performed on client interfaces. They will always have the form *itf.m*(*args*), where *itf* is a client interface, *m* is a method, and *args* are the arguments of the method call.

Synchronisation are of two kinds: access to a future's content (f`.getValue()`), and access to component's request queue (`serve`*`(`*filter*`)`).

Local method calls are those calls performed on local objects. They will activate the process corresponding to the local method call with the variables it needs (component's local variables and method arguments). The calling process is blocked until the callee finishes with some result (if any).

Comment

Each communication/synchronisation will have an action in the resulting behavioural model. This will let the model to be built bottom-up (as the model may be hierarchically synchronised), and to verify formulas involving communication.

Example

In Figure 6.19 we depict the JDC specification of the local methods used in the Philosopher component. The corresponding XMCG call graph is found in Figure 6.20. *think* and *eat* are the local methods of the component, and are the ones responsible of performing the remote calls and synchronising on futures. The call graph is a precise representation of the component activity as there is no non-determinism, and all method calls are identified as either local or remote. If we perform the analysis on the service policy as well, we would have the complete call graph of the component, seen in Figure 6.21. The latter, however, is not necessary in JDC because the pNets for the service policy can be directly derived without computing the XMCG, though it can be useful for providing complementary information to the designer. Previous work by Boulifa [Boulifa 04] require this complete call graph.

```
void think() {                              void eat() {
  int f1 = forkL.take();                      forkL.drop();
  f1.waitFor();                               forkR.drop();
  int f2 = forkR.take();                    }
  f2.waitFor();
}
```

Figure 6.19: *Local Methods of the Philosopher component*

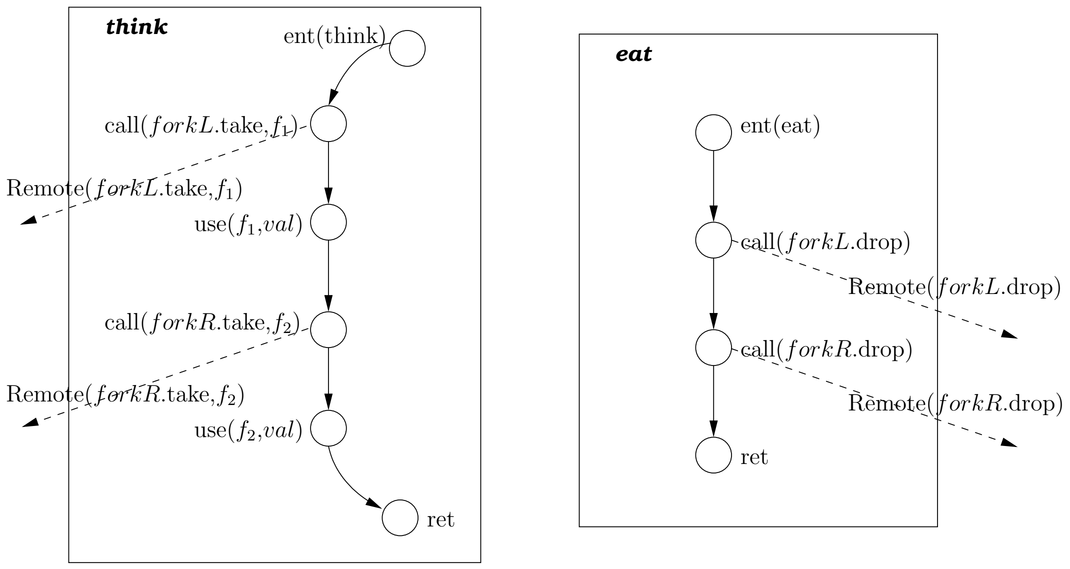


Figure 6.20: *XMCG of the Philosopher component (necessary analysis)*

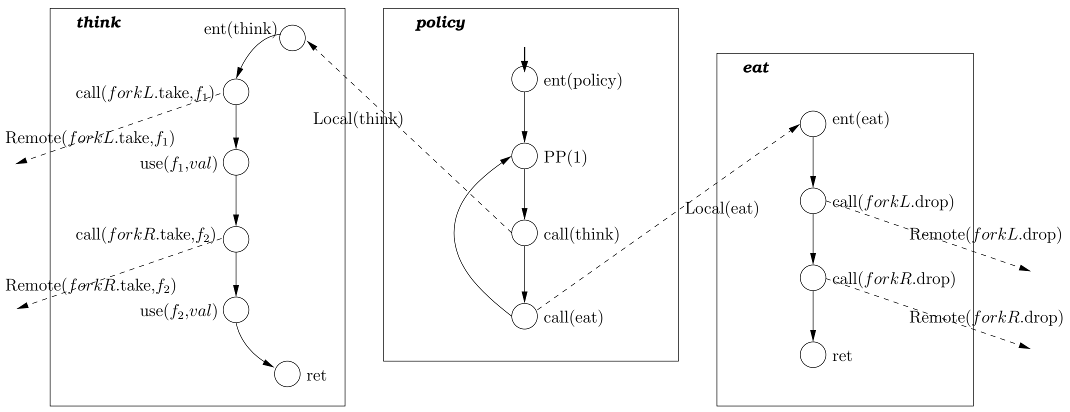


Figure 6.21: *XMCG of the Philosopher component (complete analysis)*

6.3.3.2 Building the pNets

Once we have built the XMCG for a component, we need to create the pNets model that fills-in the Service pNets. For each method *m*, we use the Procedure `Method-Behav (m, n, XMCG)`, where *n* is the entry node of *m* in the XMCG, to compute the corresponding pLTS.

In Figure 6.22 we write the rules for creating pLTSs from the XMCG. These rules have been extracted from previous work done by Boulifa [Boulifa 04], thus we do not comment on the details of the rules. Nevertheless, some changes have been realised. We removed the rule for accessing the request queue because neither the service methods nor local methods are allowed to access the queue. We also updated the rule DO-CALL to reflect that remote method calls are well identified due to the component model.

```
Method-Behav ($m$, $n$, $<M$, $V$, $\xrightarrow{calls}_C$, $\xrightarrow{succs}_T>$) :
  Aut.init = {$fresh\ s_0$}; Map = $\emptyset$; $Caller$ = $\emptyset$; ToDo = {$<n,\ s_0>$}
  while ToDo $\neq$ $\emptyset$
    ToDo.choose $<n,\ s>$
    if Map($n$) then DO-LOOP-JOIN
    else
      select $n$ in
        ent(c,m,args)                : DO-ENTRY
        call($calls(n)$)             : DO-CALL
        pp(lab)                      : DO-PP
        use(f,val)                   : DO-FUTURE
        ret(val)                     : DO-RETURN
    unless $n$=Ret
       let $MT, N$ = $succs(n)$ in
       foreach $n_i$ in $N$ do
          fresh $s_i$; ToDo.Add $<n_i,\ s_i>$
       Aut.add $s_1$ $\xrightarrow{MT)}$ $\mathcal{S} = \{s_i\}_i$

DO-ENTRY (c, m, args) =
   $fresh\ s_1$; $Caller$ = $c$, Map = Map $\cup \{n \mapsto s_1\}$
   Aut.add $s$ $\xrightarrow{?Call\_m(c,args)}$ $s_1$

DO-PP (lab) =
   if $observable$(lab) then Aut.add $s$ $\xrightarrow{Obs(lab)}$ ($fresh\ s_1$), Map = Map $\cup \{n \mapsto s_1\}$
   else $s_1$ = $s$

DO-CALL ($calls(n)$) =
  fresh $s_1$, Map = Map $\cup \{n \mapsto s_1\}$
  foreach $call$ in $calls(n)$
    match $call$ with
      "Remote(itf.m,args,var,f)":   Aut.add $s$ $\xrightarrow{!f.Q\_m(itf,args)}$ $s_1$
      "Local(o.m,args,var)":        Aut.add $s$ $\xrightarrow{!o.Call\_m(args)}$ ($fresh\ s_2$)

DO-RETURN (val) =
   Aut.add $s$ $\xrightarrow{!Caller.Ret\_m(val)}$ ($fresh\ s_1$)

DO-JOIN-LOOP () =
    $s_1$ = Map($n$)
    $Aut.replace(s, s_1)$

DO-FUTURE (fut,val) =
  Aut.add $s$ $\xrightarrow{?U\_m(fut,val)}$ ($fresh\ s_1$)
  Map = Map $\cup \{n \mapsto s_1\}$
```

Figure 6.22: *Rules for creating pLTSs from the XMCG*

Example

Figure 6.23 provides the pNets model generated for the Philosopher component. It is the synchronisation of the service policy of Figure 6.17 and the XMCG call graph of Figure 6.20.

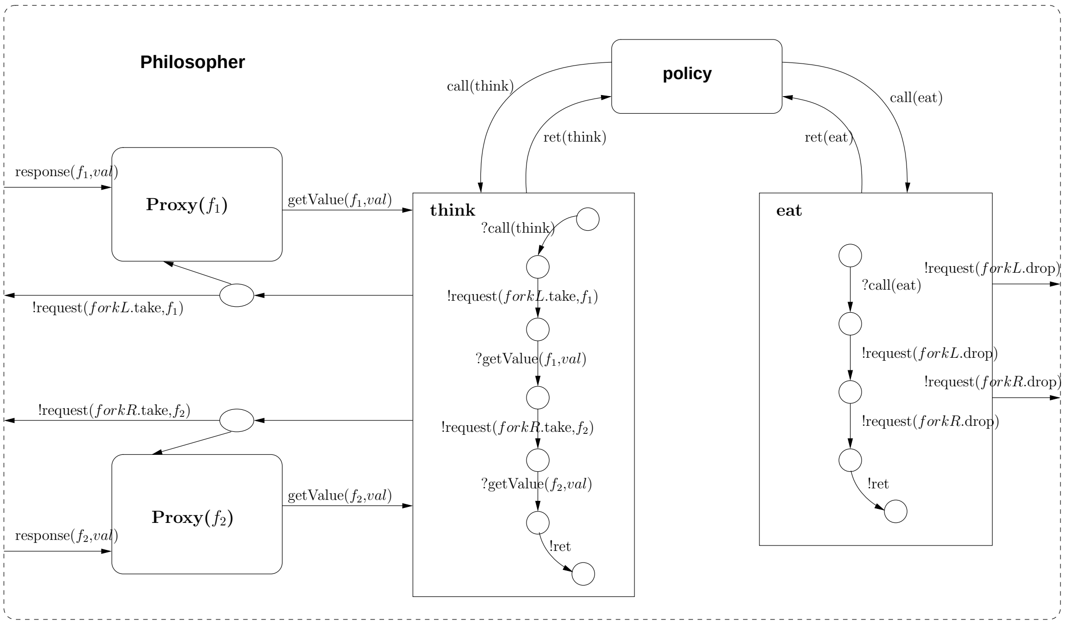


Figure 6.23: *pNets generated for the Philosopher component*

6.4 Applications

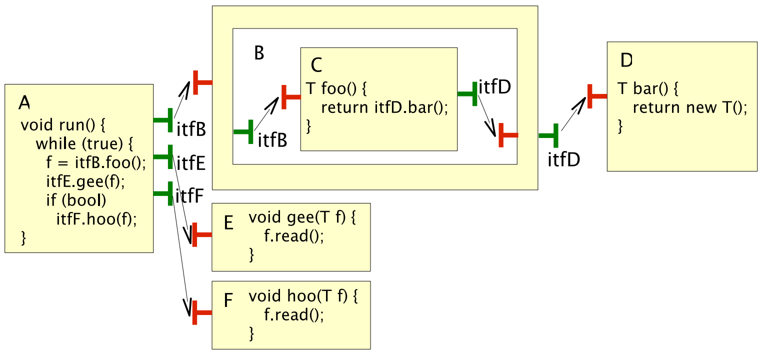
In this section we provide two applications of the behavioural models defined in this chapter.

In Section 6.4.1 we show a component system that stresses on transmission of futures. We show the behavioural model that we create, and some properties of the model.

In Section 6.4.2 we show how the behavioural models can be used to detect components blocked due to access to a future that will never be updated.

6.4.1 Transmission of Futures

In this example, we do not take into account any reconfiguration nor deployment. The system is composed like in Figure 6.24. It contains a component `A` that requests some services of `B`, and stores the return value in a variable f. Component `A` doesn't access the return value f immediately; instead, it forwards f to the component `E`, and possibly forwards f to the component `F`. Finally, `A` accesses f. Component `B` is a composite component that wraps a primitive component named `C`. The component `C`, when serving the method `foo()`, requests a service to `D` by means of its wrapper, `B`, and returns.



```
void run() {
  while (true) {
    f = itfB.foo();
    itfE.gee(f);
    if (bool)
      itfF.hoo(f);
}
```

```
T foo() {
  return itfD.bar();
}
```

```
T bar() {
  return new T();
}
```

```
void gee(T f) {
  f.read();
}
```

```
void hoo(T f) {
  f.read();
}
```

Figure 6.24: *Component system with multiple transmission of futures*

In GCM/ProActive, this would instantiate 6 active objects; one per primitive component (`A`, `C`, `D`, `E`, `F`), plus one per composite component (`B`). The active object for `B` mediates services: requests coming from the composite's server interfaces are dispatched to a subcomponent, requests coming from its subcomponents client interfaces are dispatched towards an external component. For that it makes extensive use of first-class futures; it serves a request, performs a remote method call, creates a future for holding the result, and then sends back the future to the caller. In other words, `B` *delegates* the requests it receives to components `C` and `D`, returning the future corresponding to the delegated method call.

6.4.1.1 Behavioural Model

Figure 6.25 shows the top level structure of the model created for the system above. Components `E` and `F` have similar behaviours. Components `B` and `C` are detailed in the pNets model `BC` depicted in Figure 6.26 (which is as well a model for a composite component). In this example, we index each future by the name of the component that created it.

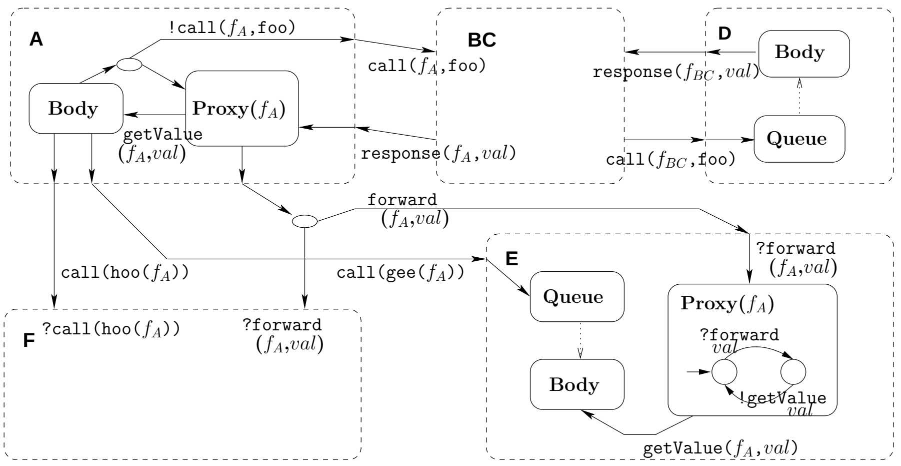


Figure 6.25: *pNets model of the component system from Figure 6.24*

In the pNets model of `A`, futures are forwarded to several activities; a future is sent as parameter of the method calls to `E` and `F` in `request(gee(`f_A`))` and `request(hoo(`f_A`))` resp. A proxy is created in each callee with the identifier (f_A) matching the proxy of the caller, i.e. `Proxy(`f_A`)`. `Proxy(`f_A`)` in the net of `A`, after receiving the concrete value, will forward the value to both activities `E` and `F`. This is seen as an action `forward(`f_A, val`)`. As a remark, the update of `Proxy(`f_A`)` in `F` is done no matter if the component is called or not, however if the call is never performed the proxy is unreachable (its identifier is unknown). Moreover, the reader may have noticed that the model for component `A` is missing the component's request queue. This is an optimisation considering that the component does not access the queue.

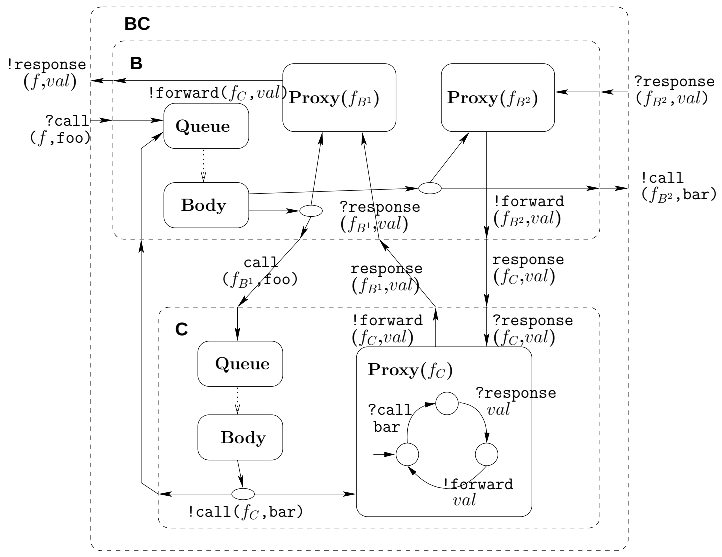


Figure 6.26: *pNets model of components* `B` *and* `C`

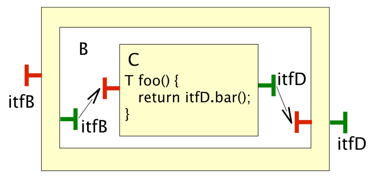


Figure 6.27: *Components* `B` *and* `C`

Figure 6.26 shows the behavioural model for components B and C. Component B creates the proxies Proxy(f_{B^1}) and Proxy(f_{B^2}) for the calls foo and bar resp. B doesn't access the proxies, so the responses of the calls are forwarded directly by the proxies. The same models applies for component C. It creates a proxy Proxy(f_C) when calling bar. C returns the future f_C, so Proxy(f_C) is the one forwarding the value it receives as a response to B.

6.4.1.2 Properties

In terms of behaviour, the value of f has no impact on the control flow, thus it is abstracted to a single abstract representative *dot*. It is the proxy that takes care of the abstract values *filled* and *non-filled*, meaning that we only care if the future has been filled or not and when it is accessed. We use the CADP [Garavel 07] toolbox for generating the state-space and for the verification; the complete LTS for the system has: 12 labels, 575 states and 1451 transitions; when minimised using branching bisimulation 52 states and 83 transitions remain. Some properties have been proved using alternation-free μ-calculus formulas [Mateescu 00]:

PROPERTY 5
System is deadlock-free

Comment

As the program loops (it restarts to an initial state given by the while loop) we proved in CADP that, on the global-state space, every state has at least one successor. Moreover, that the initial state is always accessible, i.e. the component model can be "reset" to an initial state.

PROPERTY 6
All futures are necessarily updated

Comment

This is proved by stating that the call on itfB.foo() in component A will update all futures in a finite number of actions. In pNets, this is (see Figure 6.28): starting in a state where request(f_A, foo) is performed, all leading traces will perform the future updates along the transmitting chain. More precisely, as no future is returned until a real value is known, when D computes the value, the components of the chain (D, B, and C) reply. Those response messages follow all the chain leading to A. Finally, A forwards the value to E and F (forward(f_A, val)).

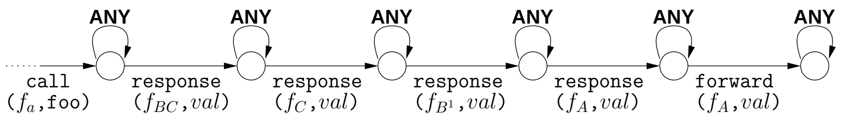


Figure 6.28: *Automaton representing the traces where futures are updated*

PROPERTY 7

System deadlocks if the composite doesn't support first-class futures

Comment

This property shows one of the design criteria of composite components: they were implemented in GCM/ProActive using first-class futures of ProActive.

Suppose that the programming language doesn't support the transmission of futures, which implies that a method call must return a value (if any). Figure 6.29 shows a modified version of the composite `B` with this behaviour. When the component `B` receives a request `?request(`f`, foo)`, the `Body` of `B` should: call the component `C` (action `!request(`f_{B^1}`, foo)`), access the return value (action `getValue(`f_{B^1}, val`)`), and then return the value of f_{B^1} (action `!response(`f, val`)`). The value of f_{B^1} is computed by component `C` on a service that must go through component `B`. Therefore, this value will never get computed as component `B` is blocked synchronising on `getValue(`f_{B^1}, val`)`. Such a scenario systematically results in deadlocks.

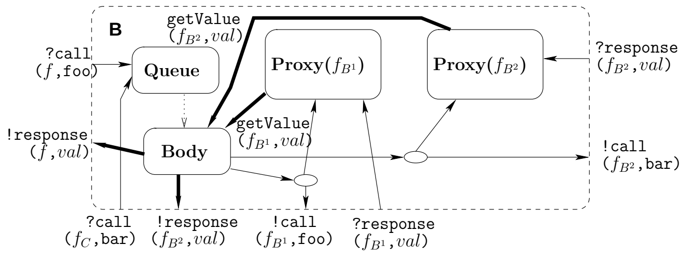


Figure 6.29: *pNets model of a composite without first-class futures*

PROPERTY 8

System deadlocks if `itfB.foo()` *is synchronous*

Comment

In this scenario, the system deadlocks in a similar way than before; if `foo()` is synchronous, then this call blocks component `B` until the result is known. What it means is that a synchronous call cannot trigger a flow that goes through a composite twice. This is a common pitfall for inexperienced programmers with GCM/ProActive that we can fortunately detect within our models.

6.4.2 Detecting Blocked Components

We now investigate how can we detect components blocked indefinitely due to an access to a future that will never be filled. For this, we will reuse the database example previously shown in Page 111, depicted in Figure 6.30.

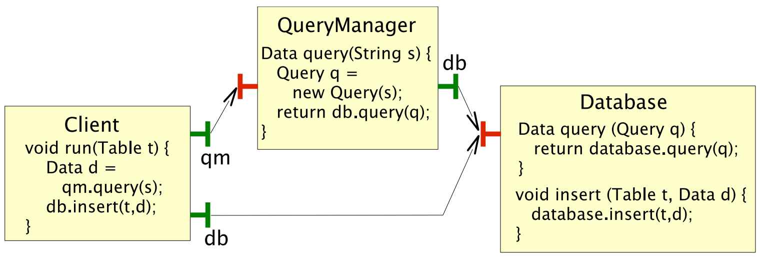


Figure 6.30: *Race condition in GCM / ProActive*

The system may deadlock due to the race condition on on access to the *Database*. Using our behavioural models, we can detect such error by means of deadlock search, i.e. checking if every state has at least one successor.

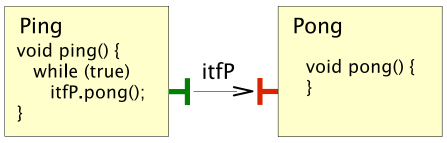


Figure 6.31: *Two components running continuously*

Now, suppose the database example is put in parallel with two components that run continuously as in Figure 6.31. Using the same analysis over the complete system, no deadlock is found: indeed, some part of the system is constantly doing some work, i.e., in the global state-space every state has at least one transition. What we need is a finner grain definition of a blocked component.

In the ASP-calculus, synchronisations happen upon access to a future and when serving a request from the request queue. In the following we consider components that serve requests in a FIFO order, and thus no synchronisation on a request is made. Therefore, all deadlocks in a system must be related to access to a future. More precisely, there must be at least a future that is accessed and that is never updated. This gives us a starting point for defining what is a (non)-blocking future.

We will slightly change the model in such a way that everytime the component accesses a future, there is first a visible, non-synchronised action `waitFor(`f`)`. After that, the component must perform the usual synchronised action `getValue(`f, val`)` where the content of the future is retrieved.

Unfortunately, due to an unfair scheduler, a subsystem (*e.g.* the Ping-Pong component system from Figure 6.31) could interact indefinitely while some components never progress. This is reflected in the synchronous product of the system; once the action `waitFor(`f`)` is performed, the action `getValue(`f, val`)` is reachable but not inevitable.

Therefore, we impose some kind of fairness in traces. We use the definition of *fair reachability of predicates* as given by Queille and Sifakis [Queille 83].

DEFINITION 16 (FAIR SEQUENCE)
A sequence is fair iff it does not infinitely often enable the reachability of a certain state without infinitely often reaching it.

Then, within the global state-space, we are interested in *fair sequences* of the future access. A component system obeys the fair sequences hypothesis if all possible sequences of its global state space are fair.

DEFINITION 17 (FAIR FUTURE ACCESS)
Given a (closed) component system, a future f is fair iff, under fair sequences, if each time the action `waitFor(`f`)` *is performed, then the action* `getValue(`f, val`)` *is eventually reached.*

An equivalent μ-calculus formula in the syntax of CADP model-checker is:

$$[true^{*}.\texttt{waitFor}(f).(\neg\texttt{getValue}(f, val))^{*}]$$
$$\langle(\neg\texttt{getValue}(f, val))^{*}\ .\texttt{getValue}(f, val)\rangle\ true$$

Now, we can define what a *non-blocking component* is:

DEFINITION 18 (NON-BLOCKING COMPONENT)
A distributed component is non-blocking *iff every future it accesses is* fair.

When synchronisations are only due to future access[*], an interesting consequence from this definition is that a *non-blocking* distributed component system is deadlock-free. In other words, if the system deadlocks, then there is at least one component blocked waiting for a future.

Therefore, for proving that each component accesses to all of its futures eventually, we can list all futures it accesses (by listing all actions of type `getValue`), and express it as a conjunction of the formula presented for *fair futures*. The main advantage of our approach is that this can be encoded in a model-checker, and thus we can ensure that every needed future reference is updated; in other words the program will have the expected behaviour: *every access to objects in the program will occur*.

[*]In GCM/ProActive for example, if components implement a FIFO service policy, than we can consider that the only synchronisations are due to future access.

6.5 Conclusion

In this chapter we have shown how JDC specifications can derive behavioural models. The model generation relies on two parts:

First, one that builds the control behaviour of components. In this phase most of the information that is required comes from the analysis of the component structure, and from the flow of futures. For the latter we develop behavioural models that allow us to check for problems related to future synchronisation.

Second, we propose a model for the functional part of components. This requires us to apply static analysis on the specification in order to infer a method call graph. This is easier in JDC than using (non-structured) active objects because we can identify precisely the remote method calls, and thus the creation of futures. The method call graph is later used in the model generation to create parameterized labelled transition systems encoding the behaviour.

6.5.1 Perspectives

Optimising the Queue. An interesting optimisation for the queue can be exploited by analysing the service policy. When the serving policy is not FIFO, it is possible to build a pLTS for the queue that factorises the accesses in a smart way. For example, if the component serves only the method m_1 in a state, and only the method m_2 in another state, then it would be possible to create two pNets processes. Each one of these processes will deal with a queue that accepts only m_1 or m_2, effectively eliminating unnecessary interleaving. Some proofs of this factorisation have been given by Boulifa [Boulifa 04], however the service policy language gives room for further optimisation.

All queues used in our models are represented by pLTSs encoding the usual behaviour of a queue. However, an interesting open question is whether including queues as first-class language constructs in the pNet model would provide us more efficient representations to feed the verification tools, in our case to explicit-state model-checkers. A promising alternative is to use infinite-state model-checking (see Section 2.4.3). They are able to use acceleration techniques in order to compute the exact effect of iterating a control loop of arbitrary length, using automata with counters and unbounded FIFO queues. The contents of the queue must be described using regular expressions, though, and to our knowledge there is limited work on mixing traditional model-checking techniques with infinite-state model-checking.

Static approximation of the topology under reconfiguration. In the current state of JDC, the system topology is static (there is no reconfiguration). Even if reconfiguration is allowed, it is still possible to build a finite (parameterized) approximation of the topology. In this case, the topology of a distributed component system is dynamic and (possibly) unbounded, because components can be created dynamically. Finite approximation[†] would still be possible due to: (i) there is a finite set of component definitions, (ii) and there is a finite number of lines that

[†]A proof of a static approximation was presented by Boulifa [Boulifa 04]; however, the proof refers to the dynamic creation of active objects only.

instantiate components. Therefore, one can expect to build a finite number of parameterized network of processes.

7

A Platform for the Design and Verification of GCM Components

Contents

Abstract

This chapter presents the architecture of our verification platform. We present the tools developed to aid the software engineer in designing and verifying component-based systems.

While the full JDC language has no tool support yet, we have defined a graphical language that is a subset of JDC. We present our graphical editor for this language, and tools that allow us to interface with state-of-the-art verification engines.

We include as well two case-studies modelled and analysed using early versions of our tools.

Motivation

The goal of this section is to describe the platform we are creating for the design and verification of GCM components. The general approach is to provide tools that will assist the designer from the early states of design up to a useful prototype of the system. One important aspect is that we want software engineers to analyse their designs without being experts in formal methods.

Rather than creating a specific verification engine (such as a model-checker), we implement tools that generate the behavioural models described in Chapter 6. These models are based on the pNets formalism (see Chapter 4), thus we provide tools that allows us to efficiently interface models in pNets with existing state-of-the-art model-checkers.

7.1 Platform Overview

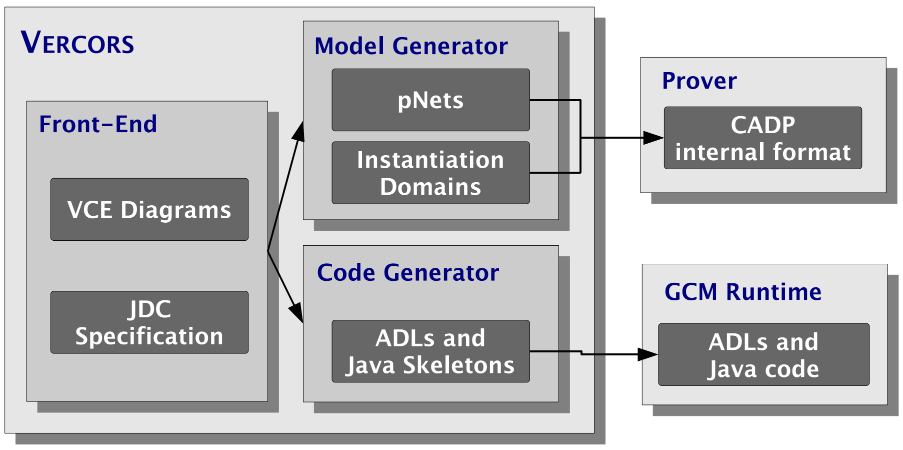


Figure 7.1: *The Vercors architecture*

In Figure 7.1 we sketch the architecture overview of the specification and analysis platform that we are building, so-called Vercors.

Front-End. We provide a graphical tool called VCE (for Vercors Component Editor) for designing components, detailed in Section 7.2. VCE is built as an Eclipse plug-in using code generated using the Topcased environment. VCE supports a graphical language for defining components in the same spirit than JDC. A JDC specification can be edited directly in a text editor, or could be generated from the diagrams of VCE.

For the moment, JDC has been conceptually defined, but does not have any tool support. An Eclipse plug-in would ease the JDC development, and give an integrated interface to the various analysis and generation functions of the platform.

The platform has two goals. One dealing with verification of designs, and the other with code generation.

Model Generator. The kernel of our platform is the generation of behavioural models. For that, we take as input the specifications given by VCE diagrams or JDC specifications. The first step is to deal with data abstraction: data types in a JDC specification are standard, user-defined Java classes, but they must be mapped to Simple Types before generating the behavioural models and running the verification tools. The abstraction is part of the JDC syntax, but the tool will offer guidance on the definition of correct mappings (correct abstract interpretations). This phase ends-up with a “JDC Abstracted Specification” in which all data are Simple Types, that are provided as a predefined library.

Such an Abstract Specification will then be given as input to our model generator ADL2N (see Section 7.3), that implements the behavioural semantics of the language, and builds a model in terms of pNets, including all necessary controllers for non-functional and asynchronous capabilities of the components. In particular it will ensure that the abstraction is compatible with the data flow within the components. Potentially this step could be automatically derived from the syntax of a formula.

Additionally, ADL2N implements a second step of abstraction, that uses finite partitions of the Simple Types to build finite models for classical (explicit-state) model-checkers. In any case the pNets objects are hierarchical and very compact: the datatypes in pNets are kept parameterized – not instantiated; and families of components are families of processes in pNets – this is particularly interesting in Grids, e.g. a set of workers in a master-worker pattern.

Prover. Our Vercors platform is using the CADP toolset [Garavel 07] for state-space generation, hierarchical minimisation, (on-the-fly) model-checking, and equivalence checking (strong/weak bisimulation). The properties to prove are fed into CADP in the form of regular μ-calculus formula.

In the future, we would like to specify these properties within JDC, which would be be subject to the same abstractions, and finally be translated into regular μ-calculus formula. We also plan to use other state-of-the-art provers, and in particular apply so-called “infinite system” provers to deal directly with certain types of parameterized systems.

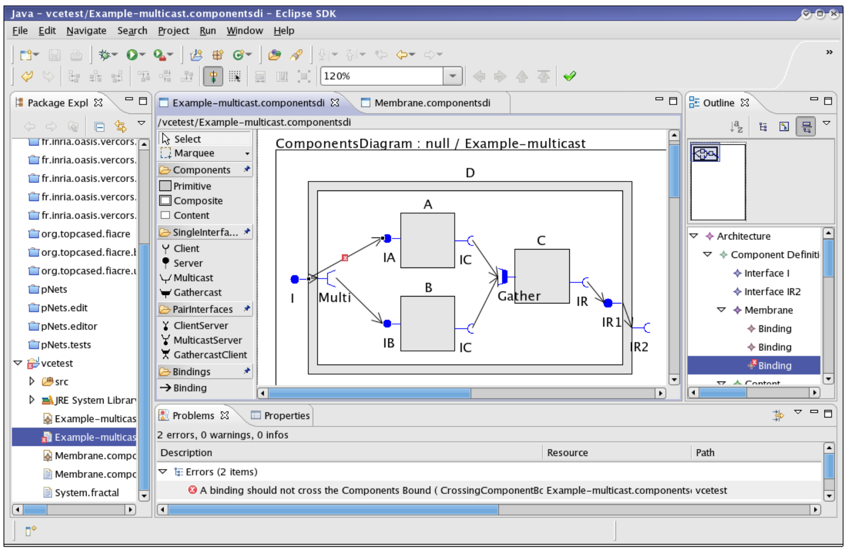
Code Generator. Another central part of the platform will be the code generator that is not (yet) currently developed. Its main purpose is to generate code with skeletons of GCM components. These skeletons include the control flow, data flow, and the synchronisation code in such a way that it is guaranteed to behave similarly to its specification – see Section 8.1.

GCM Runtime. The reference implementation of the GCM is GCM / ProActive, and will provide the technical services for deployment and executing components created from our code generator. However, we will require the programmer to refine the implementation by adding the missing business code in such a way that the behaviour is the same as specified – see Section 8.1.

7.2 Vercors Component Editor

VCE (for Vercors Component Editor) is our graphical component editor for designing components. It provides diagrams for defining the component architecture (see Section 7.2.1), and diagrams for defining the component behaviour (see Section 7.2.3).

The tool, seen in Figure 7.2, is implemented using Model-Driven-Architecture tools, with a Topcased Ecore meta-model at its kernel.



2 errors, 0 warnings, 0 infos

Errors (2 items)

A binding should not cross the Components Bound (CrossingComponentBc Example-multicast.components vcetest

Figure 7.2: *Screenshot of VCE*

For every primitive component (meaning here a component with no subcomponents), it is mandatory to give a statemachine diagram with its behaviour, whereas for composites it is optional when the component is provided with its architecture.

7.2.1 Diagrams for Architecture Specification

The architectural part of VCE was presented as a tool paper in [Cansado 08d].

In this section we present the architectural definition of our component editor. The editor is based on concepts from the Fractal and GCM models, but we have significantly changed some of the graphical notations. One important change is the representation of components. Fractal sets the role of an interface based on the orientation (left/right, top/down). While these conventions make interpretation of small diagrams easier, the diagrams do not scale well. We prefer to use a more classical notation with no orientation constraints, and interface types distinguished by icons and/or colours.

7.2.2 Graphical Language

In general, we can consider our meta-model as a *Domain-Specific-Language* or as a UML profile. We have reused part of the look & feel found in UML, particularly the icons for depicting interfaces.

Interfaces. The interface drawings are taken from UML component diagrams when possible. The server interfaces are shown as filled circles (e.g. interfaces I, IA, IB, IR1 in Figure 7.3), and client interfaces as semi-circles (e.g. interfaces IC, IR, IR2 in Figure 7.3).

Within our meta-model we also distinguish between *external* and *internal* interfaces. External interfaces are accessible by the environment; and internal interfaces accessible by the component's subcomponents.

There are also *collective* interfaces. These are *Multicast* and *gathercast* interfaces, for which we have created new custom icons. Collective interfaces are not considered in UML, and hence it was not possible to reuse existing ones. In Figure 7.3, we show the icons `Multi` and `Gather` that represent these interfaces respectively. The interface `Multi` broadcasts incoming messages to components `A` and `B`, and the interface `Gather` gathers and synchronises calls coming from interfaces `IC` towards the component `C`.

Content. The *content* of a component is represented as a white rectangle inside the component, and the *membrane* is the grey area that surrounds the content. In the case of a primitive component, we do not include its content, therefore it is depicted by a grey-filled rectangle.

Binding. A *binding* between a pair of interfaces is presented as an arrow from a client interface to a server interface. We have no support for composite bindings found in Fractal. We stick to GCM in this matter by delegating this kind of behaviour to interfaces and components.

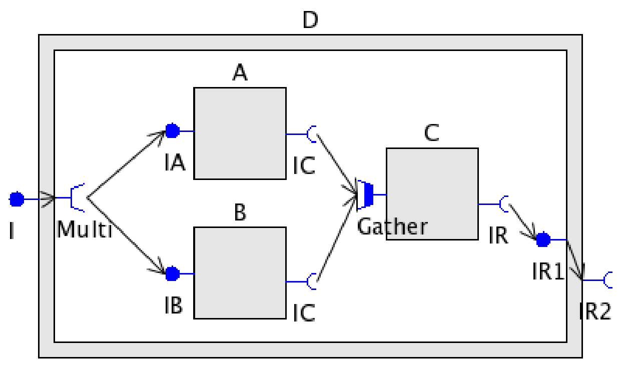


Figure 7.3: *Example of composite component exposing its content*

Membranes and Non-Functional Components. By exposing the component's membrane it is possible to control non-functional (NF) aspects. We depict the membrane with a grey area; in composites it surrounds the content, and in primitives it fills the whole figure.

The access rights of each interface are defined by marking each interface either as functional or NF. Examples of NF interfaces are `I_NF_Control` and `I_NF` in Figure 7.4. These interfaces are connected to NF components that handle the component's life-cycle.

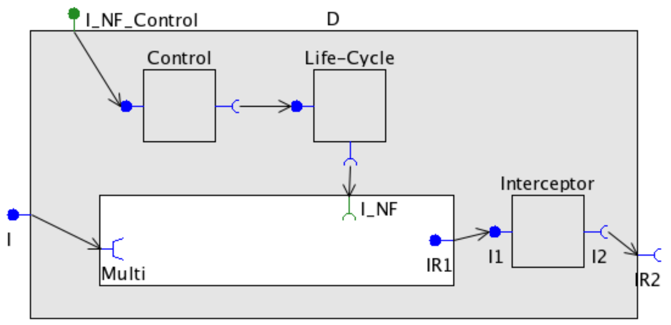


Figure 7.4: *Example of a component exposing its membrane*

VCE allows the designer to intercept functional calls entering or leaving a component. The interception takes place as NF components that are connected to the component's external functional interface, and to component's internal functional interface; they are called *interceptors*. This is a convenient way of implementing security aspects, or to check and adapt the component protocol.

In Figure 7.3, the binding ⟨IR1, IR2⟩ forwards the calls from the internal interface to the external interface. In Figure 7.4, however, the calls are intercepted and sent to a NF component called `Interceptor`.

7.2.2.1 Model Validation

To ensure the integrity of the user model, we define a minimum set of invariants that every model must hold. These invariants are defined with OCL (Object Constraint Language) [Obj 03], and complement the meta-model by expressing constraints that were left undefined. By defining the rules using OCL, we let our meta-model open. This allows us, in theory, to define different set of rules depending on particular implementations of the GCM.

However, checking for interface compatibility is not feasible using OCL. This would require us to define, within the meta-models, the full interface compatibility of Java. Instead, interface compatibility will be checked independently by our tool using hand-made Java code.

The errors are mapped back to the user diagrams. Back in Figure 7.2, the designer connected the external interface with a subcomponent's interface. This error was detected, and reported as a red cross on the object that violated the constraint in the *Problems* tab, in the *Outline* view, and in the *diagram*.

7.2.2.2 Interfacing with ADLs

We are able to generate ADLs from these diagrams useful within GCM. In its current state, we only generate GCM ADL files with the definition of the content. The membrane is still not taken into account because the ADL for dealing with non-functional aspects is still being defined.

Moreover, we also allow the designer to import ADLs in XML. The layout, however, is manual meaning that the user needs to manually place the components in the diagram.

7.2.2.3 Differences between JDC and VCE

The architecture definition in JDC is very close to that found in VCE. The main differences are those related to NF specification of components. We have added NF interfaces and NF components in the graphical language. Moreover, the implicit bindings found in JDC between external and internal interfaces are put visible in VCE. This allows us to control on how the routing between external and internal interfaces takes place.

We have also added support for GCM's collective interfaces (multicast and gathercast) which are not supported by JDC.

7.2.3 Diagrams for Behaviour Specification

The behaviour specification presented in this section is ongoing work. We have still not set the concrete graphical language, nor it is implemented. This work is, however, an extension of our previous work [Ahumada 07] on synchronous specifications of Fractal components using UML 2.

For the behavioural modelling we chose to use a variant of UML 2 State Machine Diagrams. Indeed, these diagrams specify sequences of events managed by various components. The diagrams provide a behavioural structure that matches those of JDC: the component has a service policy, and then a detailed specification of each of its service methods. There are variables in the components that are visible (and shared) by all submachines a machine may have, and are initialised with an additional state machine called `InitActivity`.

The request queue of the components are implicit in the semantics of the language. Therefore, it is not necessary to define them in state machines. The same is true for proxies implementing the asynchronous communications of futures in JDC. The designer uses futures as special variables, and the model generation will take care of adding the necessary proxies in the behavioural model. For the moment, we have chosen to have explicit futures in this variant of the specification language. This way the static analysis is simpler, and we can explicitly have an access to a future as a synchronisation primitive.

The state machines diagrams allow developers to define states, to perform method calls, to synthesise behaviour with submachines (submachine operator), to define complex transitions (meta-transitions, Section 6.3.3.1) between states (with choice operators, guards, and mixing of data and control flow), to specify actions over variables, and to abstract away implementation choices (non-deterministic choice operator). Moreover, there are distinguished start and stop operators for defining initial and final machine states. The stop operator returns execution to its parent machine; if the machine is a root, the machine halts. A local method call is also defined by a submachine.

The meta-transitions code a sequential behaviour with a unique communication or synchronisation. For that, they have a unique entry point but multiple exit points. Each one of the exit points denote a possible execution that ends up with a communication, synchronisation or a local method call.

We consider method calls to be composed of two separate events: the call itself, and the access to its return value (for non-void calls, which is stored in a local variable). A remote method call is of the form: `<var>::=<itf.methodName>!<expr>` with `<itf.methodName>` the client interface and method call invoked, `<var>` the name of the variable that will store the result value (future), and `<expr>` an expression for evaluating the method call's arguments. We have still not defined how futures can be accessed, however it will require an explicit synchronisation primitive on the variable.

Service Policy. The service policy is defined as a state-machine named `runActivity`. By default, the service policy is FIFO, though the designer can define a custom policy as in JDC. This is done by using JDC's queue access primitives (see Section 5.3.1).

Service Methods and Local Methods. The behaviours of the service and local methods are defined by state machines. There must be one such state machine for each public method offered on a provided interface of the component, and it appears in the component behaviour with a header starting with the keyword `service`. This header has a specific syntax, taking care of the bindings of arguments coming from the method call, and the bindings of the parameters define at the interface. When the interface is parameterized, these parameters represent control-flow affecting a precise interface within the component.

Collective Interfaces. We have still to define how data distribution is to be given in VCE. We will probably provide the designer with default libraries, and leave room, if needed, for giving a state-machine with custom behaviour for the interfaces.

Example

An example of a behavioural definition of a component is shown in Figure 7.5. It is based on a Buffer component from the classical Producer-Consumer problem.

The first two machines (from left to right) represent an initialisation machine, and the `runActivity` state-machine. The `runActivity` submachine is using predefined service policy methods from JDC, named `Serve*`. These methods allow for various policies of selection and execution of requests in the queue. In the example depending on some internal variable `buff`, the component will serve `Put` or `Get` methods from its request queue.

Next, we provide samples of the service methods `Put` and `Get`. `Get` is a state machine that returns a value to the caller component. `Put` is another state machine that stores a value in the component's local variable `buff`.

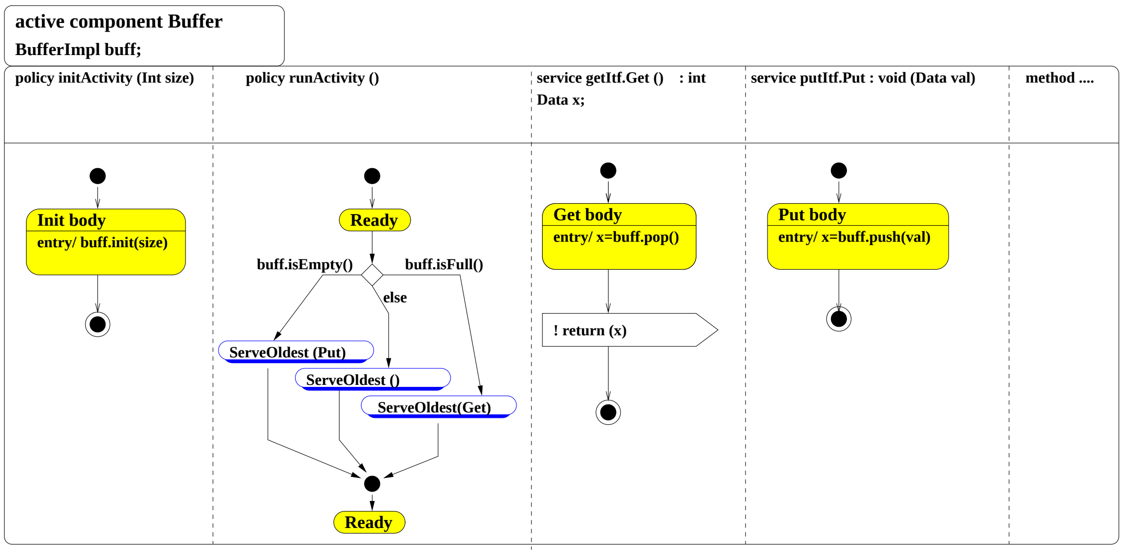
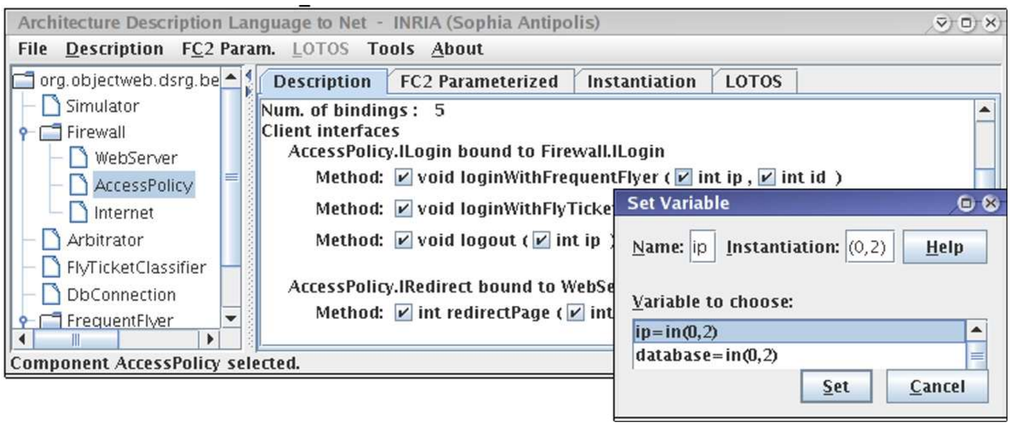


Figure 7.5: *Example of State machines for a Buffer component*

7.3 A Tool for Creating the Control Network

ADL2N (Figure 7.6) is a tool written in Java that (quasi-)automatically builds the behavioural models in pNets seen in Chapter 6. Currently, its input are the system's ADLs and IDLs.


Architecture Description Language to Net - INRIA (Sophia Antipolis)
File Description FC2 Param. LOTOS Tools About
org.objectweb.dsrg.be
Simulator
Firewall
WebServer
AccessPolicy
Internet
Arbitrator
FlyTicketClassifier
DbConnection
FrequentFlyer
Description FC2 Parameterized Instantiation LOTOS
Set Variable
Help
Set Cancel

```
Num. of bindings :  5
Client interfaces
    AccessPolicy.ILogin bound to Firewall.ILogin
        Method: ☑ void loginWithFrequentFlyer ( ☑ int ip , ☑ int id )
        Method: ☑ void loginWithFlyTicke
        Method: ☑ void logout ( ☑ int ip )
    AccessPolicy.IRedirect bound to WebSe
        Method: ☑ int redirectPage ( ☑ int
Component AccessPolicy selected.
```

Name: ip Instantiation: (0,2)

Variable to choose:
ip=in(0,2)
database=in(0,2)

Figure 7.6: *Screenshot of* ADL2N

Basically, ADL2N analyses the ADLs of the system and builds the control part of the components (see Section 6.1). ADL2N does not take as input the behavioural definition for the moment, therefore the models it creates are "open" pNets. Concretely, ADL2N will only set the Sort of the Body (functional behaviour of the component).

In the mid-term we will integrate ADL2N within VCE. This will allow us to use the behavioural specification of components together with the generated control network; this yields the complete system behaviour.

In practice the user of ADL2N uses a GUI to specify at the same time the methods that will be visible, the arguments that are significant for the proofs, and finite domains for these arguments.

The visibility of methods and the abstraction depend on the formulas to be checked. Although it should be possible to infer safe abstractions given a set of formulas, for the moment it is up to the user to provide finite abstractions of the data domains. The finite abstract domains are defined in ADL2N in order to interface with explicit state model-checkers.

Output. The pNets formalism does not have a concrete format. Therefore, we rely on the FC2Parameterized [Barros 05] format that provides us similar expressivity.

ADL2N creates two files: one file with the parameterized network in FC2Parameterized format; the other file with the finite instantiations for the parameter domains defined by the user.

7.3.1 Interfacing with Model-Checkers

The models created by ADL2N are parameterized. We provide two tools part of our framework, namely FC2INSTANTIATE and FC2EXP [*] that create finite instantiations of the models, and transform the files into the input formats of the model-checkers.

FC2Instantiate. Given a system of communicating LTSs with parameters and the domain of its unbound parameters, FC2INSTANTIATE is a Java tool that generates a finite system of communicating automata by translating each of the parameters to all the values in its domain. The formal definition of the algorithm used in the instantiation was presented in Section 4.2.3. The input is a parameterized network in FC2Parameterized and finite instantiations of the parameter domains. The output is a finite network in FC2 format[†].

We use FC2INSTANTIATE to create the finite networks by feeding the finite abstractions and the parameterized network, both given by ADL2N.

FC2Exp. Finally, in order to use the CADP verification tool, the FC2 format must be transformed into the input formats of CADP. This is done by a tool called FC2EXP; it translates network of labelled transition systems in FC2 format to the input format of CADP. The sets of synchronisation vectors are translated into the hierarchical EXP format [Lang 05], and the labelled transition systems into BCG format [Lang 05].

Using the Model-Checker. We use several engines from the CADP toolbox in our verification process. Being based on process algebras theory, the CADP toolbox provides among many others: a compiler for high-level protocol descriptions written in the ISO language LOTOS, on-the-fly capabilities, distributed space-generation and several diagnostics. All these features, although not originally intended at component verification, fit nicely within our platform for such purposes.

Concretely, we use the Evaluator model-checker from the CADP toolset, that features a very efficient check of branching-time logics, together with on-the-fly generation, cluster-based distributed state-generation, tau-confluence reduction, etc.

[*]Both tools were previously presented by Barros [Barros 05]

[†]The FC2 format definition can be found in the FC2Tool user manual [Ressouche 94]

7.4 Case-Studies

We have tested earlier versions of our tools within two case-studies:

1. A prototype implementation of a WiFi Internet access within an airport [Ježek 05]; this is a case-study of a collaboration between the Charles University in Prague and France Telecom. Our results were published in [Barros 06], and summarised in Section 7.4.1.
2. A Point-Of-Sale (POS) trading system example called CoCoME [Rausch 08]; this is a common example of component systems which yielded a tutorial book for comparing software component models. In Appendix A we show our contribution to this initiative, published in [Cansado 08a]. We summarise them here in Section 7.4.2.

7.4.1 WiFi Internet Access within an Airport

For simplicity we focus only on a subset of the system. It consists in a client (identified by an IP address), who tries to connect to a wireless network providing Internet access. Any client accessing a web page should be at first authenticated; if not, the client end ups in a login web page. At the login page, the user is asked either for a valid ticket id, or for a frequent flyer id (if the user is registered in a frequent flyer programme). Once authenticated, the user is granted access to any web page he desires until his time-lease expires.

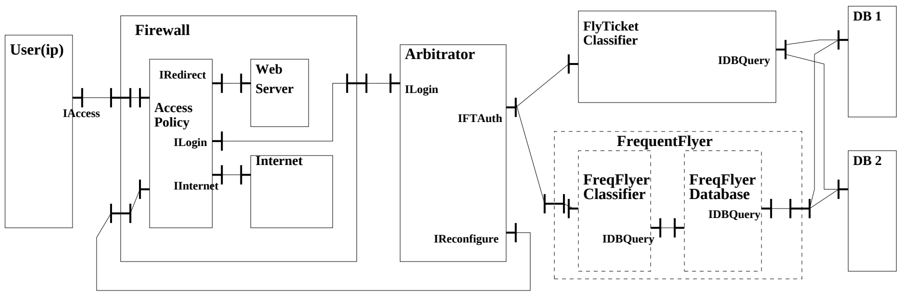


Figure 7.7: *Simplified architecture of the WiFi Internet Access example*

The architecture of interest can be seen in Figure 7.7. The main component inside the Firewall is AccessPolicy, which routes the traffic between the WebServer and the Internet based on the firewall rules. The network resulting of the ADL analysis performed by ADL2N for the component Firewall is shown in Figure 7.8.

The behaviour of primitive components was specified in LOTOS. The LOTOS specifications were compiled into LTSs using the CAESAR tool of CADP, and actions were mapped into compatible ones generated by ADL2N.

In the case-study we did not include asynchronous controllers, therefore we limited the communication to synchronous rendez-vous in which there are no queue nor proxies in components. For the other parameters, we provided ADL2N with finite abstract domains representing full partitions of the domains. There are 3 distinguished values for the possible web pages `url`: one for the login page, one for the web page requested by the user in the Internet and a third one for every other possible web page. We consider only 2 abstract ticket `ids`, one

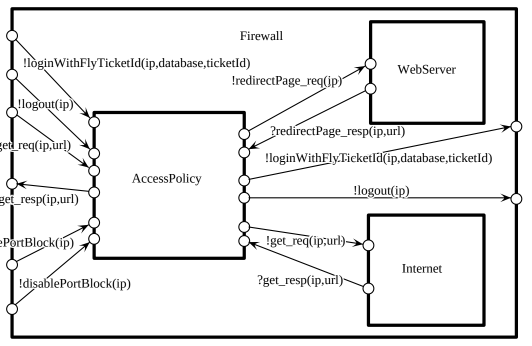


Figure 7.8: *pNet obtained by ADL analysis for the Firewall component*

encoding a distinguished ticket given by the user and the other for remaining tickets. Since only the distinguished ticket is valid, then the authentication is granted only if the system successes in finding this ticket. For the validity of the tickets, a boolean representing a valid/invalid ticket is enough as we do not take into consideration the time control. Similarly, the system has two databases, one representing the database with the ticket needed.

The full functional behaviour of the system is easily generated in a single desktop machine (Pentium 4 3GHz, 1GB RAM); this model was built with all remote method calls kept visible. Ignoring the components drawn in dash lines of the Figure 7.7, the LTS of the system has 2152 states with 6553 transitions (non-minimised) and 57 states with 114 transitions (reduced), both with 17 visible labels, while the biggest primitive component has 5266 states with 27300 transitions. One could also reduce the size of the various parts of the models, taking into account the set of actions that occur in the formulas to be proved, and using environment interfaces to reduce the intermediate construction sizes. Finally, for bigger systems, it is possible to use the on-the-fly facilities of CADP for the state space generation, or even to compute the state space in a distributed manner on a computation grid.

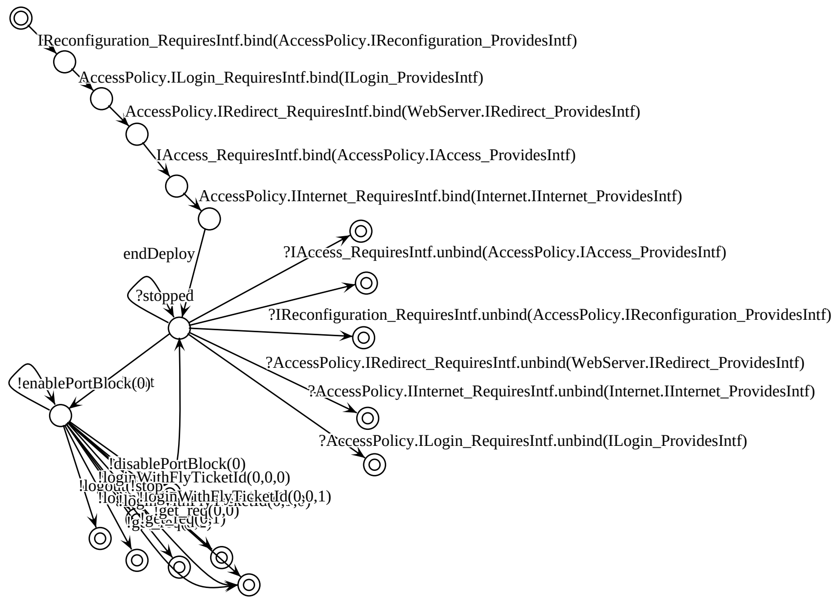

IReconfiguration_RequiresIntf.bind(AccessPolicy.IReconfiguration_ProvidesIntf)
AccessPolicy.ILogin_RequiresIntf.bind(ILogin_ProvidesIntf)
AccessPolicy.IRedirect_RequiresIntf.bind(WebServer.IRedirect_ProvidesIntf)
IAccess_RequiresIntf.bind(AccessPolicy.IAccess_ProvidesIntf)
AccessPolicy.IInternet_RequiresIntf.bind(Internet.IInternet_ProvidesIntf)
endDeploy
?IAccess_RequiresIntf.unbind(AccessPolicy.IAccess_ProvidesIntf)
?stopped
?IReconfiguration_RequiresIntf.unbind(AccessPolicy.IReconfiguration_ProvidesIntf)
?AccessPolicy.IRedirect_RequiresIntf.unbind(WebServer.IRedirect_ProvidesIntf)
!enablePortBlock(0)
?AccessPolicy.IInternet_RequiresIntf.unbind(Internet.IInternet_ProvidesIntf)
?AccessPolicy.ILogin_RequiresIntf.unbind(ILogin_ProvidesIntf)
!disablePortBlock(0)
!loginWithFlyTicketId(0,0,0)
!logout
!stop
!loginWithFlyTicketId(0,0,1)
!get_req(0,0)

Figure 7.9: *LTS for the deployed Firewall component*

We also tested the correct deployment of the Firewall component. This was done by including in the model generation the life-cycle and binding controllers. This yielded a model with 41 visible labels, 29750 states with 85254 transitions for the non-minimised model, and 327 states with 1034 transitions for the minimised model. A slice of the resulting automaton can be seen in Figure 7.9. It shows a sequential trace of binding actions, followed by a transition labelled `endDeploy`. Afterwards, there are two major blocks of actions: the actions allowed when the component is stopped, and the actions allowed when the component is started.

7.4.2 CoCoME

The CoCoME case-study consists in a Trading System as it can be observed in a supermarket handling sales. This includes the processes at a single *Cash Desk*. Typical operations are scanning products using a *Bar Code Scanner* or paying by credit card or cash. A general schema of the architecture can be seen in Figure 7.10. The subset of CoCoME we modelled consists in 16 components, 5 of them being composites. Furthermore, composite components were designed with up to 5 layers of hierarchy, stressing the need of hierarchical component models.

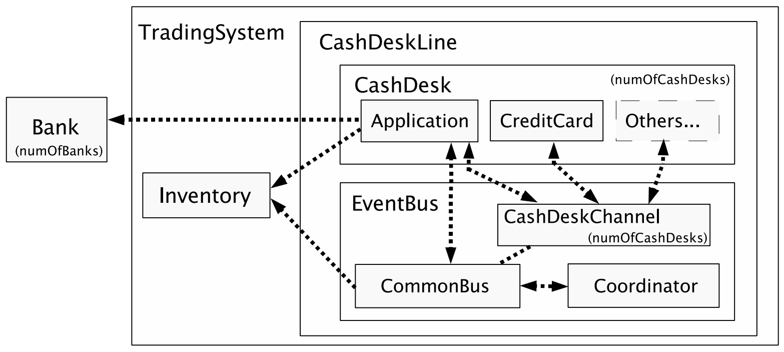


Figure 7.10: *The CoCoME overview*

The *Cash Desk* is the place where the Cashier scans the products the Customer wants to buy and where the payment (either by credit card or cash) is executed. Furthermore it is possible to switch into an express checkout mode which allows only Customer with few goods paying with cash. To manage the processes at a *Cash Desk*, a lot of hardware devices are necessary (*Card Reader, Cash Box, Cash Desk GUI, Light Display, Printer, Scanner*).

We specified the system using JDC and an old version of VCE called CTTool [Ahumada 07]. From these specifications, we created by hand a GCM/ProActive implementation showing how the control code can be generated from the JDC specification. From the same specification, completed by abstraction functions, we have shown how to generate models suitable for verification, in the form of parameterized networks of synchronised transition systems.

For a synchronous model (no queues nor proxies), and parameters with abstract domains ranged over by intervals of size 2 and 3, the synchronous product has: 81 visible labels (message instances), 1.25MM states with 3MM transitions without minimisation, and 9.800 states with 33.000 transitions after branching minimisation.

We were able to verify some use-case scenarios:

- *Main Sale Process.* We proved in the state-space generated by CADP that this trace is feasible, and that it is not possible to start a new sale before booking the former one.
- *Booking an Empty Sale.* We showed that the specification allows an empty sale to be booked, which seems to be an under-specifcation of the system.
- *Successful Sale with Insufficient Money.* We found an error in the specification. There is no verification whether the amount entered in the Cash Box is enough to cover the bill.
- *Safety of the Express Mode.* The system ends-up in an inconsistent state if an express mode signal is triggered during an ongoing sale.

More details can be found in the Appendix A.

7.5 Conclusion

In this chapter we have presented a general overview of our platform. The platform is meant to allow designers to specify their systems, to verify behavioural properties, and finally to generate components with guaranteed behaviour.

The platform is still under development. The front-end of the platform is partially implemented and allows the designers to define their components using a graphical language. There are also tools for creating the control part of components, with the possibility to individually select which actions (method calls and arguments) can be hidden as internal actions.

We also provide tools for interfacing with current state-of-the-art model-checkers. These are based on abstraction onto finite models. The designer provides finite abstractions for the domains of variables and we create finite models that can be used within model-checkers (such as CADP).

Finally, there is still no support for code generation. The required tool should be based on the work presented in Section 8.1 and would provide programmers with a prototype implementation that serves as starting point for deploying safe components.

8 Perspectives

Contents

Abstract

In this chapter we present perspectives to JDC. On the one hand, we present the ideas for algorithms that generate, from the JDC specification, code skeletons for distributed components. The generated code relies on the reference implementation of the GCM, and the result is code guaranteed to conform by construction with the specification.

On the other hand, we discuss several language extensions that have not been taken into account. These deal with a broader range of GCM features such as many-to-one and one-to-many communications, as well as advanced features specific to ProActive.

8.1 Building Safe-By-Construction Components

In the previous chapters we presented a low level model for the behavioural semantics of distributed components, a specification language for distributed components, and procedures to build the behavioural models of components from their specifications. This allows us, in theory, to prove whether the specifications are safe in some sense. However, does the implementation of a (component) system conform to its specification? There are two classic ways of answering this question: either one builds a model of the implementation and proves that the implementation is a refinement of the specification (or that the implementation does not violate the specification), or one uses a constructive approach [Coglio 05, Fernandes 07] in which the implementation satisfies the specification by construction. In this section we focus on the latter approach.

Roughly speaking, we will show how to generate the control code of the distributed component, and we set rules stating what the user can do in the remaining code. This control code includes every possible communication and synchronisation the component may perform with its environment. The generated code is based on the definition of the component architecture, and is formed by an ADL definition, plus GCM/ProActive code skeletons from the specification.

8.1.1 Runtime Requirements

A first step in generating code for distributed components is to set an adequate runtime support (library or middleware). For this, it is useful to summarise the features that the component model of choice should support.

We basically need a component model and middleware supporting:

- *Hierarchical distributed components*. The specifications of components in JDC are hierarchical, and components are distributed. We believe these are good features that must be reflected in the runtime.
- *Asynchronous method calls*. Distributed components require asynchronous communications to deal with latency. Moreover, we have supposed in JDC that all communications were asynchronous, therefore we need a runtime that implements the same behaviour.
- *Transparent futures*. The specifications in JDC do not declare futures explicitly. However, we could choose to generate code with explicit futures as the analysis of components allow us to identify (in most cases) which variables are futures. The drawback of generating code with explicit futures is that we must statically decide whether a variable will be a future or not. However, this may vary depending on the set of components (the environment). Therefore, we would have to generate code that matches a particular environment. On the contrary, if futures are transparent the same code is valid within a wider context.
- *First-class futures*. The futures in JDC can be transmitted in a non-blocking manner. This property must be supported by the runtime of choice due to the same reasons stated above.
- *Customisation of the service policy*. The runtime must allow one to select which requests to serve and in which order.
- *Abstraction from the physical infrastructure*. The JDC specification does not include any detail about the physical infrastructure. This property is definitively to be preserved. We seek a middleware that provides hooks to later map components to physical resources.

Moreover, we require the runtime to have only 2 strict operations:

- access to the request queue; and
- access to the content of a future.

Of course, these requirements are akin with GCM/ProActive as JDC was mainly conceived as a specification language for this kind of components.

Alternative. Another candidate is Creol [Johnsen 06]. Creol is a formal framework that supports active objects that communicate through asynchronous method calls similar to those found in ProActive. Futures are explicit, however, which would force us to type futures in the generated code.

On the other hand, in Creol active objects are multithreaded. It could be interesting to study the relation with JDC's concurrent Services.

There is also a component version of Creol [Owe 07], but components are not hierarchical. They also provide an interesting approach of defining *deontic contracts*, where two (or more) components must agree on what is obliged, permitted, and forbidden.

8.1.2 Implementation Overview of a GCM/ProActive Component

We briefly review now how a GCM/ProActive code looks like. A component in GCM is typed by the set of its (external) interfaces. This is preferably defined within a separate ADL file for each component definition.

Both primitive and composite components have ADL definitions that define the component type. For primitive components, the ADL includes the signature of the class that implements the component behaviour. For composite components, the ADL includes the architecture that the component implements but no behaviour other than the default one provided by the middleware (dispatch incoming calls to the bound interfaces). In Subsection 8.1.3 we show details on the ADLs.

Once defined by the ADL, a primitive component is implemented by a Java class. This corresponds to a object-oriented implementation of a component. Figure 8.1 shows a template implementation of a primitive component in GCM/ProActive.

```
// usual header, which includes the package and imports

public class ComponentName implements
       RunActive,              // denotes a GCM component
       BindingController,      // GCM non-functional controller
       MyServerInterface, ... // list of server interfaces {
   // client interfaces, which include type and name
   private MyClientInterface itf;
   // local variables
   private ...
   // local methods
   private void localMethod() { ... }

   // service method (methods defined within the server interfaces)
   public ...

   // GCM controllers
   public String[] listFc () { ... } // and others

   public ComponentName () { } // empty constructor required by ProActive

   // service policy
   public void runActivity(Body body) {
      Service service = new Service(body);
      localMethod();    // call to a local method
      while (body.isActive()) {
        serveOldest(); // serve in FIFO order indefinitely
      }
   }
}
```

Figure 8.1: *Code structure of a primitive component implementation in GCM/ProActive*

The component class implements all *server interfaces*. This is, for each method defined in the component's server interfaces, the class implementation has a method with the same name giving its behaviour. Note that this pattern is prone to collision of method signatures; if two or more interfaces define a method with the same signature, only one implementation is possible for this method.

The *client interfaces* are fields of the component class. This allows any method within the component to access the environment through the name of the interface.

As the component is active, i.e. that is has its own thread of control, the GCM component implements the `RunActive` interface. Then, there is a distinguished method called `runActivity` that has access to the component's request queue. By default, ProActive gives a FIFO implementation of this method, but in general the method implements an application-dependent policy. In the example, the component calls a local method (line 30), and then serves methods from its request queue in a FIFO order (line 32) until the component is stopped (variable `isActive` set to `FALSE`).

Other interfaces like the `BindingController` can be implemented to allow reconfiguration. These interfaces are used by the middleware to change the component's bindings when stopped.

8.1.3 Generating the ADLs

There are several strategies that can be used to generate the components' ADLs. For instance, one may generate an ADL file with every component definition, or an ADL file for each component, or a mixture of both.

Our approach is to set an ADL file for each JDC component definition; this file takes the role of a component type, named `componentName.fractal`. We find this strategy better than using a single hierarchical file because it provides a grey-view of the component architecture at each layer of the hierarchy, meaning that at each level the ADL only exposes how sibling components interact to implement the parent component. Moreover, this approach is the only way of representing a component type in GCM and, therefore, the only way of defining several instances of the same component type.

Example: *Several instances of the same type*

An example of 2 subcomponents of the same type can be seen in Figure 8.2. Subcomponents `A` and `B`, defined within a component named `Composite`, reference the component type `C`. The type of `C` is defined by a second file `C.fractal`. Both subcomponents will be initialised identically and will have the same behaviour. Nevertheless, GCM components are statefull, meaning that `A` and `B` will have their own (non-shared) local memory.

For each *Architecture* specification of a component, the compiler generates a composite component. The composite architecture is expressed with the GCM ADL (Architecture Description Language). It includes the component's external interfaces, subcomponents, and bindings at this level of hierarchy. For the moment, we do not consider the internal interfaces because the current GCM ADL definition has no support for them.

```
<!-- File Composite.fractal -->
<definition name="Composite">
  <component name="A" definition="C"/> <!-- Subcomponent of type C -->
  <component name="B" definition="C"/> <!-- Subcomponent of type C -->
  ...
</definition>

<!-- File C.fractal -->
<definition name="C"> <!-- Defines a type -->
  ...
</definition>
```

Figure 8.2: *Composite component defining a component type, and two instances of this component*

For components that are defined directly through a single *Service*, we generate an ADL defining its type (interfaces), and a reference to a Java class that contains the code skeleton. This class will be described in Section 8.1.4.

We also generate an additional GCM component for dealing with components defined with multiple services. Each service will be implemented by a GCM primitive component; another GCM composite component, that relates to the JDC specification, is created that wraps all these subcomponents. For this, each service must have a disjoint set of server interfaces – there are no particular constraints to client interfaces. The result is a container component that delegates services as in Figure 8.3.

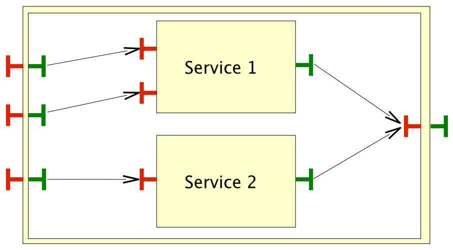


Figure 8.3: *Architecture for Multiple Services*

Finally, the physical infrastructure is abstracted by relating only to virtual nodes on which the component will eventually be deployed. These are specified as tags in the ADL. The GCM deployment, which is defined elsewhere, is then in charge of mapping a virtual node to a machine in the Grid.

```
  <virtual-node name="primitive-node" cardinality="single" />
```

8.1.4 Generating the Java code skeletons

In this section we study how to generate code skeletons for primitive components. We base the generation on the behavioural definition given by a service in JDC.

The code generator will create code skeletons that contain the control flow of the component. It is important to remember that a primitive component has its own (unique) thread of control, independent from the queue, that will be responsible for accessing the queue and performing some work. The component is only able to serve another request once the previous request has been processed (the control returns to the `runActivity` method).

Queue. The queue is implemented automatically by ProActive by means of declaring the component as an "active object". Therefore, the component has an independent thread which is permanently ready to en-queue a new request, and provides access primitives to the rest of the component.

Futures. The futures are also handled automatically by the middleware. Variables are not typed as futures or not, and the middleware decides at runtime whether a call on a variable is a blocking operation or not.

Main class. The component implementation consists in a Java class with a specific pattern (Figure 8.1). This class implements all server interfaces that will set all methods accessible from outside (besides some control methods needed by the middleware, detailed in Section 8.1.4.1). We rely on the strong functional behaviour encapsulation of GCM for this matter. We add some constrains, though: all server interfaces must have different names, and methods defined in these server interfaces must have unique signatures.

The service policy is implemented by ProActive's `runActivity()` method as we will detail in Section 8.1.4.2.

For each service method, we generate templates with the expected control and data flow; more in Section 8.1.4.3.

Finally, we add into the component class the service's variables. These members are private; their types are those defined in the JDC specification. In this part of the code generation we take party of having real user-classes within the specification, as the programmer will not need to modify code that references these variables.

8.1.4.1 GCM controllers

The GCM controllers we create allow basic reconfiguration operations. We automatically generate them using the information from the JDC's definition of a primitive component; the specification of the behaviour is not needed here.

Figure 8.4 depicts the automatically generated code. It shows an implementation of the `BindingController` interface from Fractal and GCM which sets technical services for rebinding components. We only provide the simplest behaviour for the `BindingController` interface; to list interfaces and to rebind interfaces[*]. An example of such code can be seen below, with `CASHBOXEVENTIF` being the name of a client interface of the component.

For starting and stopping components, ProActive already implements a life-cycle controller; this is enough for performing basic reconfiguration. By basic reconfiguration we mean

[*]This is similar to the code generated by FractalGUI [FractalGUI] and Fraclet [Fraclet]

```
  // Implementation of the Controller interfaces
  public String[] listFc () {return new String[] { CASHBOXEVENTIF };}

  public Object lookupFc (final String clientItfName) {
     if (CASHBOXEVENTIF.equals(clientItfName))
        return cashBoxEventIf;
     return null;
  }
  public void bindFc (final String clientItfName,final Object serverItf){
     if (CASHBOXEVENTIF.equals(clientItfName))
        cashBoxEventIf = (CashBoxEventIf)serverItf;
  }
  public void unbindFc (final String clientItfName){
     if (CASHBOXEVENTIF.equals(clientItfName))
        cashBoxEventIf = null;
  }
```

Figure 8.4: *Automatically generated GCM controllers*

reconfiguration that does not require customisation of the life-cycle controller. By default, when a stop signal is received, the life-cycle controller broadcasts the signal to inner subcomponents. However, if the application has a dependency cycle, then this strategy may cause a deadlock (we can check this using our behavioural models). Though, a custom implementation of the life-cycle controller may wait for a stable component state before proceeding with the reconfiguration.

8.1.4.2 Service Policy

The service policy is implemented by the `runActivity` method in ProActive. It receives a reference to the active object's body as parameter, which provides access to the component's request queue.

Generating a skeleton for the `runActivity` is simple considering that the service policy is a regular expression. The service policy could also be specified using a state machine as in VCE (see Section 7.2.3). The latter provides more expressive power than that found in JDC because one can specify expressions on the component's variables that is not possible in JDC right now. Nevertheless, in the following we only give details of the simpler subset found in JDC.

The generated code is based on the ProActive API as follows:

Queue Access Primitives. The two request queue access primitives of JDC (`serveOldest` and `serveYoungest`) can be implemented using ProActive's `blockingServeOldest()` and `blockingServeYoungest()` methods. Both access primitives may be called without parameters (any method will be matched), or with a filter (only methods matching the filter will be affected). For that case, the API provides the interface `RequestFilter` for implementing filters on the component's request queue. We implement this interface with the class `RequestFilterJDC`; this implementation allows us to select requests matching an interface and/or a method name.

`ServeMode` →	**serveOldest** | **serveYoungest**	≪*request queue access prim.*≫

Remote Method Call. A remote method call is a statement in Java identical to the one in JDC. The asynchronous semantics are taken care by ProActive.

$\boxed{\texttt{MethodCall} \rightarrow \quad \texttt{ItfName}\ '.'\ \texttt{MethodName}\ '('\ [\ \texttt{Expr}\ ]\ ')' \quad \ll\textit{remote method call}\gg}$

Local Method Call. A local method call from the service policy will simply delegate the thread of control to the related method.

$\boxed{\texttt{MethodCall} \rightarrow \quad \texttt{MethodName}\ '('\ [\ \texttt{Expr}\ ]\ ')' \quad \ll\textit{local method call}\gg}$

Sequence. The sequence operator is implemented by sequencing two blocks of Java code.

$\boxed{\texttt{BasicPolicy} \rightarrow \quad \texttt{BasicPolicy}\ ';'\ \texttt{BasicPolicy} \quad \ll\textit{sequence}\gg}$

Choice. The choice operator is implemented by a non-deterministic choice between the two policies.

$\boxed{\texttt{BasicPolicy} \rightarrow \quad \texttt{BasicPolicy}\ '|'\ \texttt{BasicPolicy} \quad \ll\textit{choice}\gg}$

'n' Repetition. Repetition is implemented by a while loop with a counter.

$\boxed{\texttt{BasicPolicy} \rightarrow \quad \texttt{BasicPolicy}\ 'n' \quad \ll\textit{n repetition}\gg}$

Permanent Policy. The permanent policy is implemented by a while loop that is repeated until the component is stopped. This is represented by the statement `while (body.isActive())`.

$\boxed{\texttt{PermPolicy} \rightarrow \quad \texttt{BasicPolicy}\ '*' \quad \ll\textit{infinite repetition}\gg}$

Example

In Figure 8.6 we show an example of code generated from the JDC service policy specification from Figure 8.5.

The component will perform two remote method calls `saleStarted()` and `saleFinished()` on the client interface `cashBoxEventIf`, and then will non-deterministically behave as two separate scenarios which will join again when the iteration in the policy ends.

The non-deterministic behaviour is simulated in Java by a random choice in `(new AnyBool()).isTrue()`. Note that a concrete implementation will probably modify this random choice accordingly.

Continuing with the example, the component also accesses its request queue, selecting the request matching the method name `changeAmountCalculated`. Finally, the component activity loops indefinitely, which is captured by the main `while` loop.

More details of the example can be found in Appendix A.

```
policy {
  ( cashBoxEventIf.saleStarted();
    cashBoxEventIf.saleFinished();
    (
      cashMode();
      cashAmount();
      serveOldest(controlIf.changeAmountCalculated);
      cashBoxEventIf.cashBoxClosed()
    )
    |
    ( creditCardMode() )
  )*
}
```

Figure 8.5: *Example of a JDC service policy*

```
public void runActivity(Body body) {
  Service service = new Service(body);
  while (body.isActive()) {
    cashBoxEventIf.saleStarted();
    cashBoxEventIf.saleFinished();
    if ((new AnyBool()).isTrue()) {
      cashMode();
      cashAmount();

      // serveOldest(controlIf.changeAmountCalculated)
      RequestFilterJDC filter = new RequestFilterJDC();
      filter.addInterfaceMethod("controlIf", "changeAmountCalculated");
      service.blockingServeOldest(filter);

      cashBoxEventIf.cashBoxClosed();
    } else
      creditCardMode();
  }
}
```

Figure 8.6: *Example of a JDC service policy implemented in GCM/ProActive's runActivity*

8.1.4.3 Service Methods

Language definition. The exact language definition for specifying the behaviour is missing in JDC. We have left open how the service methods are defined. As a first step we expect to deal with a simplified subset of JDC that we call aJDC (for Abstract JDC).

In aJDC, datatypes are always of type *simpletypes*, and futures are typed. One can think of aJDC as a JDC specification in which of variables of user types have been replaced by their abstract versions. Moreover, a static analysis step has determined precisely whether a variable is a future or not; in the case this is not possible, the user must solve the ambiguity.

On one hand, the goal of this simplified flavour of JDC is to provide an easy starting point for the code generators. On the other hand, the goal is to obtain an easy mapping towards our graphical language in VCE (as proposed in the state-machine of Section 7.2.3).

aJDC can be defined as a Domain-Specific-Language (DSL), and rapidly implemented using TCS [Jouault 06a]. TCS uses Meta-Modelling techniques to implement support within Eclipse for textual languages in the form of DSL.

Synchronisation. The key aspects of the generated code rely on the synchronisations. To be sure that the generated code behaves exactly as defined in JDC, the synchronisations must be exactly those inferred in JDC.

In JDC, the explicit synchronisation primitive `touch()` can be expressed in ProActive with an API call `ProActive.waitFor(`*f*`)`, where *f* is the variable to perform the synchronisation. A call to this method is allowed whether *f* is a future or not; in case *f* is not a future, the middleware ignores the invocation.

In the case a variable *f* in JDC was abstracted into a Singleton abstract domain, after a synchronisation `ProActive.waitFor(`*f*`)` the variable can be freely manipulated as any ordinary Java variable. As the variable does not contain significant values, the only effect on the control flow could be an eventual synchronisation that is already performed and specified.

If *f* does not have a Singleton domain, after a synchronisation `ProActive.waitFor(`*f*`)` the variable can be read (and access any read-only method), but changing its value must be done with caution. The new concrete value must have the same abstract value in the abstraction, otherwise the user could have changed the control flow of the application.

Non-Determinism. Typically the programmer will have to refine the generated code to deal with non-determinism. In JDC, we have included an explicit enumeration of the abstract domain. We can only provide a simulation of this behaviour within the generated code, and it will be up to the programmer to fill in the exact implementation.

Annotations. An approach is to leave part of the generated code as Java annotations. This can help the programmer to identify what part of the code is related to control, and on the other hand it could be used by tools in order to assist static analysis. An example of how annotations could be used together with JDC is as follows. We annotate the generated code for the `runActivity` from Figure 8.6 in Figure 8.7.

In the example, the generated code is annotated with marks to identify remote method calls (`@RPC`), loops (`@while`), conditional choices (`@if`), external data (`@USERCLASS`), and access to the request-queue (`@service`). The programmer has decided to include a logger (`monitor`) that was not included within the JDC specification. These annotations allow us to easily endow Java with additional semantics.

Annotating the code is also of great interest in order to keep the specification and the implementation synchronised. The work on Fraclet [Fraclet] uses Java annotations in order to represent components, interfaces, bindings and Fractal controllers in a compact way. This is later translated by the compiler into plain Java, but relieves the programmer from the burdon of writting code that has been already specified.

8.1.4.4 User Classes

We do not try to generate the user classes. On the contrary, they must be provided by the designer, together with the definition of their abstractions. Therefore, we should have

```
public void runActivity(Body body) {
  monitor.addMessage("Start");
  Service service = new Service(body);
  @while (body.isActive()) {
    @RPC cashBoxEventIf.saleStarted();
    monitor.addMessage("Sale␣Started");
    @RPC cashBoxEventIf.saleFinished();
    monitor.addMessage("Sale␣Finished");
    @if (@USERCLASS(new AnyBool()).isTrue()) {
      monitor.addMessage("Cash␣Mode");
      cashMode();
      cashAmount();

      // serveOldest(controlIf.changeAmountCalculated)
      RequestFilterJDC filter = new RequestFilterJDC();
      filter.addInterfaceMethod("controlIf", "changeAmountCalculated");
      @service.blockingServeOldest(filter);

      @RPC cashBoxEventIf.cashBoxClosed();
    } else {
      monitor.addMessage("Credit-Card␣Mode");
      creditCardMode();
    }
  }
}
```

Figure 8.7: *Example of annotated code to be used together with JDC*

a support in order to assist the designer in this process, similarly as showed in the Bandera toolset [Dwyer 01].

Another issue that leave open is how can we allow refinement of these classes. For example, the signature of user classes will probably change during the implementation to include new arguments, so the code we generate should be refined as well.

8.1.5 Rules for Modifying the Generated Code

We now enumerate the rules that the programmer must follow in order to ensure that the generated code has the same behaviour as specified in JDC.

We call *safe* an implementation that completes the code skeletons by an implementation following the rules:

1. The ADLs cannot be modified by the programmer. Doing so would break the system architecture and the consistency of the generated code.

2. The request queue is not accessible by the implementation code. In other words, only the automatically generated code for the `runActivity()` method is allowed to access the queue.

3. The implementation may not perform remote method invocations other than those specified in JDC. In other words, method calls on client interfaces are only those automatically generated. Otherwise, the programmer would change the component's protocol for both caller and callee.

4. Within the user-classes, every field that has been omitted in the abstract version of the class must be strictly a value. In other words, only variables that are considered in the JDC specification may contain futures, but other variables may be defined within user-classes as long as they are strictly non-futures. The goal is to forbid new variables from enforcing further synchronisation on futures that have not been specified. For guaranteeing such behaviour, we should rely on the middleware; however, ProActive does not yet implement any mechanism for specifying which variables cannot be futures. Another option is to use static analysis to verify these properties.

5. The component's variables, given by the set of variables in the JDC specification, should always have values in the abstract domains as defined in the specification. In other words, these variables may be modified with multiple concrete values as long as they all have the *usual* abstract values in the abstract domain. Moreover, the first access to a variable declared in JDC (thus possibly a future) must be the one automatically created by the code generator. Afterwards, the implementation may freely access the variable in a read-only mode, or change the variable to other concrete values (under the constraint above). The goal is to be sure that the control flow of the component is the one defined by the specification.

6. The implementation may include any business code interleaved within the generated code. This should not have an influence on the control flow of the generated code.

Checking these properties would rely on a static analysis of the implementation. Rules 3 and 4 are checked using aliases analysis. However, verifying rule 5 is more complex. If we wish to perform this automatically, we would require an abstraction function that takes variables of concrete domains and returns its abstract values. Then, a static analysis of the implementation could check if the behaviour has changed.

8.2 Language Extensions

Among the large number of language extensions that we can think of, there are two sets that can be defined here: those related to GCM in general, and those related to ProActive.

8.2.1 Extensions dealing with the GCM

Collective communication. The main element in GCM that is not dealt with by JDC is many-to-one and one-to-many communication (also called multicast/gathercast).

It is not only a matter of defining an interface as multicast/gathercast. We need to define its behaviour for capturing, at least, the distribution of data to bound interfaces. This distribution is user-defined, therefore we can provide a library with the most common distribution policies, but we must also let the designer specify new policies. Complementarily, we need to adapt our behavioural models in order to take this into account.

Parameterized Topologies. Once collective communication is supported, another key aspect is to define parameterized topologies of components. This is particularly interesting in Grids, where the topology is, in general, parameterized by the number of clusters, and by the number machines in each cluster. In a master-worker pattern for example, we would like to define parallel and identical components for the workers. We have been working in this direction in our VCE tool and in JDC. Some prototypes are presented in the CoCoME use-case detailed in Appendix A, and in the grammar of Figure 8.8.

`Component` →	**component** id '('[`FormalParams`]')' '{'	*≪param. component≫*
`Subcomponent` →	**component**	
	`ComponentType` '('[`ActualParam`]')'	*≪partial specialisation≫*
	id [`Expr`] ';'	*≪multiple components≫*
`Interface` →	[`BoolExpr`] (**server** | **client**)	*≪guard and role≫*
	interface `InterfaceType` id [`Expr`]	*≪indexed interfaces≫*
`Binding` →	[`BoolExpr`] **bind**'('	*≪bind if guard is true≫*
	`Comp`[`Expr`]'.'`SourceItf`[`Expr`]','	*≪source interface≫*
	`Comp`[`Expr`]'.'`TargetItf`[`Expr`]')'	*≪target interface≫*

Figure 8.8: *Syntax for parameterized topologies of components*

What we need to do is to add parameters to the component architecture. That is, the component has parameters denoting a family of components. These parameters can be used to define collections of interfaces, give partial instantiations of subcomponents, and so on. For the bindings, the references will be expressions on the variables, allowing for conditional bindings.

Moreover, the parameters can have influence on the data-flow, defining routes and data distribution. This is why we should base this work on the collective interfaces [Bruneton 06].

Non-Functional Components. There has been work within the GCM community 3.1.3 to define non-functional (NF) within the membrane. For the time being, we only worked on the architecture definition of NF components in VCE (see Section 7.2). The difficulty here is to define the exact interaction between different NF components. There is still no concensus within the GCM community on what (and how) NF components are allowed to do. Therefore, we are awaiting for further research into this trend, and investigating what could be interesting to represent within the behavioural models.

A major relevance of NF components is that by structuring the membrane, the component becomes more compositional. This is in the sense that Fractal proposes a *white-box* view of the NF aspects. This means that, in theory, a NF interface could be used to change arbitrary values in the component. On the contrary, by structuring the membrane, the designer would have to say exactly what is to be changed, and who has access rights for that. Therefore, we can expect to deal with components in a much more compositional way, which heals state-explosing in model-checking.

Reconfiguration primitives. It should not be difficult to define a small set of reconfiguration primitives that address most of use cases of dynamic reconfiguration. Moreover, we already have defined models for dealing with basic reconfiguration such as stopping a component and changing its bindings. Therefore, we think that this should be the main research path in JDC as it is definitively aligned with our goals.

Similar projects, such as SOFA 2.0 (see Section 2.1.2.3), limit reconfiguration to pre-existing reconfiguration patterns. This is also another trend that must be considered, and JDC could be the language used to define these patterns in a high-level abstraction.

Limited Concurrency. Another trend for research in JDC is its limited concurrency. Concurrency can only be expressed through an architecture that exposes the interaction of services through subcomponents. This leads to designs in which a service is strongly tied to a primitive component, so refinement of subcomponents is limited.

There are two reasons why this was not yet considered in JDC. First, we do not have an equivalence relation between the black-box definition and its architecture implementation. Second, we do not want to build a complex specification language with many synchronisation primitives and parallel operators, which would be more complex than those found in the programming language.

One option would be to include access to some shared resources, either in the form of shared variables or shared methods. However, it is not clear how to generate GCM code that complies with this kind of specifications. Another approach is the one taken by LOTOS. LOTOS includes a parallel operator where one can specify parallel processes that communicate through some shared gates. This can be used to define configuration patterns, but then again this is somehow expressing an architecture where interfaces and bindings are synchronisations on gates.

8.2.2 Extensions dealing with ProActive

An important issue about JDC is that it does not take into account specificities of ProActive (the reference implementation of the GCM). It was indeed the goal of JDC to deal with GCM implementations in general. We will present in the following some extensions that could be useful for dealing specifically with GCM/ProActive.

Advanced Synchronisation Primitives. The ProActive API provides advanced control on synchronisations, which can enhance the application performance and may avoid deadlocks. Nevertheless, if rich primitives are used, the deterministic properties inherited from the ASP-calculus cannot be guaranteed any longer.

The most used one is the `ProActive.isAwaited(`f`)` method. It allows one to check whether a future f has been filled or not (in the case of a non-future variable this is trivially true). However, this makes the program immediately dependent on how futures are updated (i.e. there is no confluence on future update strategies). On the contrary, the models we provided in Chapter 6 require confluency of future update strategies. In particular, we have used a different update strategy than the one implemented in ProActive.

Similarly, the ProActive API provides a non-blocking access to the request queue. This allows for full introspection of the request queue. We will definitively not support this as it would make verification through model-checking unfeasible. However, there is still room including some queue access primitives. For example, we can imagine that a boolean predicate for filtering some methods can be applied without major impact in the state-space. Moreover, believe that adding non-blocking access to the request queue will only locally affect the model, precisely only the model of the queue.

Asynchronous Exception Handling. Yet another feature in ProActive is asynchronous exception handling [Chazarain 05, Caromel 05a]. Roughly, the idea is to allow a method call to be performed asynchronously even if the method may raise an exception. The exception is only dealt with when the `catch` of the `try/catch` block is reached, or on a strict access to the future variable (which will finally throw an exception). However, transmitting a future in the meanwhile must also be a strict operation. This is to avoid problems if-ever an exception is thrown within a future which should have never been existed in a sequential execution.

The implementation of such exception handling breaks the semantics of ASP, so properties on confluence are no longer valid. Therefore, once again the behavioural models for futures we proposed in Section 6.2 are not valid.

Immediate Services. An immediate service in ProActive is a method that is executed within the callee by the caller thread. This means that the active object "allows a remote thread" to execute and modify the state of the active object. Moreover, the synchronisation must be handled explicitly by the programmer.

We have not deal with immediate services because we consider that it should not be used by the GCM programmer. We believe that programming with immediate services is error-prone, and we are sure that it completely breaks the model inherited from ASP-calculus. It could be useful though, for programmers of the ProActive middleware. Therefore, including support for immediate services in JDC could help the development of ProActive.

8.3 Conclusion

In this chapter we presented a small set of remaining work for JDC. We have outlined how we could create safe-by-construction components based on JDC specifications, and then reviewed language extensions that are required in order to deal with a broader range of GCM features.

The generation of safe GCM components is based on code generated from a verified JDC specification. The code includes the architectural definitions of components, as well as part of the behaviour implementation. The latter, is mainly focused on the remote method calls performed by the component, and the synchronisations that may eventually happen. The generated code is not final, as it is built upon an abstract specification of the system. However, the programmer may refine the generated code with business code such that the control code is unmodified. This is still an early work on the generation of GCM components from a specification. There is no tool that implements the ideas described in this chapter, and one of the open questions is how designers can refine the generated code with tool support (or annotations) in order to guarantee that the behaviour is unmodified.

We also reviewed extensions to JDC that address a broader set of GCM paradigms. These take the form of many-to-one and one-to-many communications, parameterized topologies of components, non-functional components, and reconfiguration primitives. We discussed the need for a better handling of concurrency in JDC, that is currently limited to the architecture definition.

Complementary, we also reviewed extensions that tackle particular features found in the ProActive API. These features allow the optimisation of communications by exposing futures and synchronisations. These are mainly characteristics that we wanted to hide from the designer, but that are very useful for developers of the ProActive middleware.

9 Conclusions

What this thesis has sought from the very beginning is the support for the development of safe components. Our work has focused both on the modelling and the specification of distributed components. The thesis objective has been to narrow the gap between component implementation and component specification.

A sound analysis of a system can be obtained through behavioural models which provide an abstract representation. However, behavioural models tend to be too low-level to be used as a system specification. Therefore, in order to specify and analyse distributed components, this work has envisioned a formal framework adapted to the expertise of software engineers. We have provided behavioural models suitable for verification of distributed component systems; and provided an expressive language that allows one to define a high-level abstraction of the system behaviour.

To summarise, our work takes away complexity "inherent" to software specification from the designer, and puts it on the verification tools.

9.1 Contributions

We now highlight the main contributions of this thesis.

Hierarchical Formalism. We have formalised a hierarchical model to support the definition of behavioural models. The first part of the thesis presented a formalism called pNets. The formalism is particularly adapted for modelling components because it describes the model as a hierarchy of communicating processes, and provides a symbolic representation of the system. Another strong contribution of the formalism is that is does not make strong assumptions on the kind of synchronisation used, and it handles data within the processes. The formalism can also be thought of as an intermediate language that interfaces with state-of-the-art verification tools. *This provides a sound formalism to define various kinds of behavioural models.*

High-Level Specification Language. We have provided a high-level specification language for specifying distributed components that is suitable for software engineers. The pNets formalism is well adapted to the kind of behavioural models we want to express. However it is too low-level to be directly used by our target users. Therefore, we have defined a high-level specification language rich enough to capture the communication, synchronisation, control-flow, and data flow inside components. We have designed the specification language to be formal enough to build behavioural models, but endowed with a syntax familiar to our target users. *This allows an easier adoption of software engineers.*

Integrate Architecture with Behaviour. We have integrated the architectural and behavioural definitions into a single specification language. Components provide software composition which can be intuitively represented. With that in mind, the specification language includes not only the behavioural, but also the architectural definitions. Of course the architectural definition deals with structural definitions as in most ADLs, but also provides language primitives if we ever need to reference the structure within the behaviour. The behavioural definition, on the other hand, takes care of the component's control and data flow, but can also have concrete primitives making references to the system architecture. *This avoids architectural erosion.*

Components as Services. We have provided, first an intuitive definition of the component behaviour through the *service policy*, and then a detailed definition of each service. We noticed that distributed components had some kind of control part that orchestrated the services offered to the environment. That was a good starting point to define the behaviour. Hence, we have defined the component's *service policy* that give us a rough definition of what the component provides to the environment, and then we focus on a more detailed definition of each of these services. *This makes component behaviour easy to understand.*

Simplify the Specification. We have simplified the behaviour specification by specifying the component activity instead of the events. We specify just what the component does, and use static analysis techniques in order to infer the exact behaviour (all the events performed by the component). *This defines the complex synchronisations of distributed components in a simple way.*

Generate Verifyable Behavioural Models. We have defined and generate verifyable behavioural models from instances of the specification language. The analysis of the specifications gives us information about the architecture, remote method invocations, future flow, synchronisation on access to the content of a future, and access to the component's queue. This is enough to automatically generate behavioural models based on our pNets formalism; moreover, the behavioural models can be verified by model-checkers. *This gives a verifyable abstraction of the system's behaviour.*

A Compositional Static Representation of the System. We have defined behavioural models that are a static representation of the system and can be built in a compositional way. A strong contribution of the work is to provide a static representation of futures, particularly when futures can be transmitted to other components in a non-blocking manner. The behavioural models encode the asynchronous behaviour of components, including a request queue for buffering the requests, and proxies for futures. We propose an abstract domain for futures, and then static representations for the various ways futures can be transmitted; moreover, we do this is in a compositional way. *This gives a compositional static representation of the system that can be verified.*

Narrow the Gap between Specification and Implementation. We have provided a mechanism that can generate behavioural models and code skeletons from the same specification. The specification language is adequate to generate a skeleton of the component implementation with strong guarantees w.r.t. their specification. This is done by structuring the *service policy* such that the control of the component can be automatically generated, and defining rules for the reminder of the component behaviour. *This ensures strong guarantees on the runtime behaviour.*

Use User-Classes within the Specification. We have included user-classes and abstract versions in the specification language. This guarantees that the generated code uses correct implementations of data. Moreover, this guarantees that the behavioural model generation uses datatypes compatible with our pNets formalism, but more importantly, *this guarantees that the model is a safe abstraction of the specification.*

9.2 Comparison to Other Approaches

In this last section, we compare our approach to existing approaches. The evaluation of existing specification languages given in Section 3.3 shown us the main drawbacks when dealing with GCM components. To summarise, we highlighted that most specification languages are fit to define the architectural part of the GCM (though limited to functional concerns) but are not adapted to a data-driven synchronisation model. Particularly, this meant that futures (and more precisely transparent first-class futures) required a manual encoding of their behaviour.

Architecture Specification. In this work, most novel contributions are related to behavioural definition of components. Nevertheless, our graphical language found in VCE (see Chapter 7.2) gives a new language for defining the architectural part of the GCM. In there we have included primitives for dealing with one-to-many and many-to-one communications, but more importantly, we have given the means for structuring non-functional aspects of components.

With respect to JDC, the architectural definition is the one we classically find in existing ADL languages. Nevertheless, as it is given together with the behavioural definition, we believe it is easier to extend the language for dealing with reconfiguration. The idea of augmenting Java for this is similar to that in Java/A and ArchJava, though Java/A does not consider reconfiguration whereas in ArchJava reconfiguration is based on π-calculus.

Asynchronous Semantics. In this work we have focused on how distributed components equipped with asynchronous semantics can be specified. As the events triggered by the use of futures can be highly interleaved within the component behaviour, classic approaches such as the ones used in Behavior Protocols (describing the events through regular expressions), Java/A (describing the interface behaviour through LTSs) do not provide high-enough abstractions to the designer. We believe that our work is complementary to these works in the sense that we provide a much higher abstraction of the behaviour definition and afterwards we infer which are the events that can be triggered by the components. Therefore, the resulting behavioural model includes the component's events, matching the abstraction level of the other approaches.

Data Abstractions. With respect to the data part of the language, a novel approach is that we did not try to generate sound implementations of the user classes. We believe that the implementations of the user classes will require major modifications before being useful. Therefore, guaranteeing the behaviour of the data part of the generated code would not be possible. This is not the case for the control part of the language, in which it is much easier to separate the code dealing with the control from that of the calculation. Combining both generated control code with user-provided user classes is easier and safer.

If we compare this approach with existing ones, in STSLib the authors generate classes from Algebraic Data Types (ADTs). However, the ADTs comprise the abstract behaviour that any implementation of data types should have. This is done by defining equations that are quite distant from the expertise of software engineers.

More related to our approach would be the work on Bandera on static analysis of Java code. Nevertheless, their goal is to assist the generation of sound abstractions of the user classes. Once the abstractions are defined, they abstract the Java source code which can be finally verified (model-checked). They do not consider, yet, the abstractions of futures (and particularly access to futures). Moreover, they focus on Java source code so they need to take care of the business code whereas our approach starts from an abstraction of the control and data-flow of distributed components.

Formalism. Comparing the pNets formalism with other formalism can be made in several trends. The pNets formalism is meant to describe the behaviour of distributed systems in general so it is somehow closer to LOTOS and Promela than to ASP. It is also difficult to compare it with Behavior Protocols because Behavior Protocols is specific to components, whereas pNets solely describes some behaviour but lacks of semantics.

LOTOS is a standarised formalism that addresses distributed systems in general as well. However, in LOTOS the behaviour is given through communicating processes that must agree on actions in order to communicate. In pNets we use a generalised parallel operator (synchronisation vectors) that subsumes the parallel operator found in LOTOS. The data part found in LOTOS, on the contrary, is much more expressive than the simple types found in pNets. This makes LOTOS very expressive, though complex.

ASP is a formalism used to prove properties of distributed active objects (or components) in general whereas we can use pNets to prove the behaviour of a particular application. Additionally, in ASP we can formalise the behaviour of a program whereas in pNets we describe its behaviour. With respect to analysis, pNets has been conceived to interact mostly with automatic checkers, particularly with model-checkers. ASP, on the contrary, requires a theorem-prover in order to analyse a program; even more, as ASP is very expressive, it is highly non-deterministic and thus it is difficult to automatically generate proves.

Bibliography

[Abadi 96] Martín Abadi & Luca Cardelli. A theory of objects. Springer-Verlag, New York, 1996.

[Abdulla 98] Parosh Aziz Abdulla, Ahmed Bouajjani & Bengt Jonsson. *On-the-Fly Analysis of Systems with Unbounded, Lossy FIFO Channels*. In CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification, pages 305–318, London, UK, 1998. Springer-Verlag.

[Abi-Antoun 05] Marwan Abi-Antoun, Jonathan Aldrich, David Garlan, Bradley Schmerl, Nagi Nahas & Tony Tseng. *Software Architecture with Acme and ArchJava (Research Demonstration)*. In Proceedings of the 27th International Conference on Software Engineering, St. Louis, MS, May 2005.

[Adámek 03] Jiří Adámek. *Static analysis of component systems using behavior protocols*. In Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 116–117. ACM Press, 2003.

[Adámek 04] Jiří Adámek & František Plášil. *Component Composition Errors and Update Atomicity: Static Analysis*, 2004.

[Ahumada 07] Solange Ahumada, Ludovic Apvrille, Tomás Barros, Antonio Cansado, Eric Madelaine & Emil Salageanu. *Specifying Fractal and GCM Components With UML*. In proc. of the XXVI International Conference of the Chilean Computer Science Society (SCCC'07), Iquique, Chile, November 2007. IEEE.

[Aldrich 02] Jonathan Aldrich, Craig Chambers & David Notkin. *ArchJava: connecting software architecture to implementation*. In ICSE '02: Proceedings of the 24th International Conference on Software Engineering, pages 187–197, New York, NY, USA, 2002. ACM.

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

[Annichini 01] Aurore Annichini, Ahmed Bouajjani & Mihaela Sighireanu. *TReX: A Tool for Reachability Analysis of Complex Systems*. In CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification, pages 368–372, London, UK, 2001. Springer-Verlag.

[Apvrille 04] Ludovic Apvrille, Jean-Pierre Courtiat, Cristophe Lohr & Pierre de Saqui-Sannes. *TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit*. IEEE transactions on software Engineering, vol. 30, no. 7, July 2004.

[Arbab 04] Farhad Arbab. *Reo: a channel-based coordination model for component composition*. Mathematical. Structures in Comp. Sci., vol. 14, no. 3, pages 329–366, 2004.

[Armstrong 06] Rob Armstrong, Gary Kumfert, Lois Curfman McInnes, Steven Parker, Ben Allan, Matt Sottile, Thomas Epperly & Tamara Dahlgren. *The CCA component model for high-performance scientific computing*. Concurr. Comput. : Pract. Exper., vol. 18, no. 2, pages 215–229, 2006.

[Arnold 94] André Arnold. Finite transition systems: semantics of communicating systems. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1994.

[Attali 04] Isabelle Attali, Tomás Barros & Eric Madelaine. *Formalisation and proofs of the Chilean Electronic Invoices System*. In XXIV International Conference of the Chilean Computer Science Society (SCCC 2004), pages 14–25, Arica, Chile, 2004. IEEE Computer Society.

[Badrinath 00] B. R. Badrinath & Pradeep Sudame. *Gathercast: The design and implementation of a programmable aggregation mechanism for the Internet*. In Proceedings of IEEE International Conference on Computer Communications and Networks (ICCCN), October 2000.

[Baduel 06] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel & Romain Quilici. Grid Computing: Software Environments and Tools, chapitre Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[Balsamo 02] Simonetta Balsamo, Marco Bernardo & Marta Simeoni. *Combining Stochastic Process Algebras and Queueing Networks for Software Architecture Anaysis*. In ACM Press, editeur, Proc. of the 14th Int. Conf. on Software Engineering and Knowledge Engineering (SEKE'02), S. Angelo d'Ischia, Italy, 2002.

[Bardin 03] Sébastien Bardin, Alain Finkel, Jérôme Leroux & Laure Petrucci. *FAST: Fast Acceleration of Symbolic Transition systems*. pages 118–121. Springer, 2003.

[Barnat 05] Jiří Barnat, Vojtěch Forejt, Martin Leucker & Michael Weber. *DivSPIN – A SPIN compatible distributed model checker*. In M. Leucker & J. van de Pol, editeurs, Proceedings of 4th International Workshop on Parallel and Distributed Methods in verifiCation, pages 95–100, July 2005.

[Barnat 06] Jiří Barnat, Luboš Brim, Ivana Černá, Pavel Moravec, Petr Ročkai & Pavel Šimeček. *DiVinE – A Tool for Distributed Verification (Tool Paper)*. In Computer Aided Verification, volume 4144/2006 of *LNCS*, pages 278–281. Springer Berlin / Heidelberg, 2006.

[Barroca 92] Leonor M. Barroca & John A. Mcdermid. *Formal Methods: Use and Relevance for the Development of Safety-Critical Systems.* Comput. J., vol. 35, no. 6, pages 579–599, 1992.

[Barros 04] Tomás Barros, Rabea Boulifa & Eric Madelaine. *Parameterized Models for Distributed Java Objects*. In Forte'04 conference, volume LNCS 3235, Madrid, 2004. Spinger Verlag.

[Barros 05] Tomás Barros. *Formal Specification and verification of Distributed Component Systems*. PhD thesis, Université de Nice-Sophia Antipolis, 2005.

[Barros 06] Tomás Barros, Antonio Cansado, Eric Madelaine & Marcela Rivera. *Model Checking Distributed Components : The Vercors Platform*. In 3rd workshop on Formal Aspects of Component Systems, Prague, Czech Republic, September 2006. ENTCS.

[Barros 08] Tomás Barros, Rabéa Boulifa, Antonio Cansado, Ludovic Henrio & Eric Madelaine. *Behavioural Models for Distributed Fractal Components*. Annals of Telecommunications, accepted for publication, 2008. also Research Report INRIA RR-6491.

[Baude 07] Françoise Baude, Denis Caromel, Ludovic Henrio & Paul Naoumenko. *A Flexible Model and Implementation of Component Controllers*. In CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments, pages 12–23, june 2007. CoreGRID TR-0080 technical report.

[Baumeister 05] Hubert Baumeister, Florian Hacklinger, Rolf Hennicker, Alexander Knapp & Martin Wirsing. *A Component Model for Architectural Programming*. In Proc. 2nd Int. Wsh. Formal Aspects of Component Software (FACS'05), Elect. Notes Theo. Comp. Sci., 2005.

[BEA Systems 05] BEA Systems, IBM, IONA, Oracle, SAP AG, Siebel Systems & Sybase. *Service Component Architecture*. Whitepaper, November 2005.

[Becker 08] Steffen Becker, Heiko Koziolek & Ralf Reussner. *The Palladio Component Model for Model-Driven Performance Prediction: Extended version*. Journal of Systems and Software, 2008.

[Ben-Ari 01] Mordechai Ben-Ari. *The bug that destroyed a rocket*. SIGCSE Bull., vol. 33, no. 2, pages 58–59, 2001.

[Bergstra 01] Jan A. Bergstra, Alban Ponse & Scott A. Smolka. Handbook of process algebra. North-Holland, 2001.

[Boreale 06] Michele Boreale, Roberto Bruni, Luis Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, Antonio Ravara, Davide Sangiorgi, Vasco Vasconcelos & Gianluigi Zavattaro. *SCC: a Service Centered Calculus*. In M. Bravetti & G. Zavattaro, editeurs, Proceedings of WS-FM 2006, 3rd International Workshop on Web Services and Formal Methods, volume 4184 of *Lecture Notes in Computer Science*, pages 38–57. Springer Verlag, 2006.

[Boulifa 04] Rabéa Boulifa. *Génération de modèles comportementaux des applications réparties*. PhD thesis, Université de Nice - Sophia Antipolis – UFR Sciences, December 2004.

[Bracciali 02] Andrea Bracciali, Antonio Brogi & Carlos Canal. *Adapting Components with Mismatching Behaviours*. In CD '02: Proceedings of the IFIP/ACM Working Conference on Component Deployment, pages 185–199, London, UK, 2002. Springer-Verlag.

[Brim 06] Luboš Brim, Ivana Černá, Pavlína Vařeková & Barbora Zimmerova. *Component-interaction automata as a verification-oriented component-based system specification*. SIGSOFT Softw. Eng. Notes, vol. 31, no. 2, page 4, 2006.

[Brookes 84] Stephen D. Brookes, Charles Antony Richard Hoare & Andrew William Roscoe. *A Theory of Communicating Sequential Processes*. J. ACM, vol. 31, no. 3, pages 560–599, 1984.

[Bruneton 02] Eric Bruneton, Thierry Coupaye & Jean-Bernard Stefani. *Recursive and dynamic software composition with sharing*. Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02), 2002.

[Bruneton 04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quema & Jean-Bernard Stefani. *An Open Component Model and Its Support in Java*. In 7th Int. Symp. on Component-Based Software Engineering (CBSE-7), LNCS 3054, May 2004.

[Bruneton 06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma & Jean-Bernard Stefani. *The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems*. Softw. Pract. Exper., vol. 36, no. 11-12, pages 1257–1284, 2006.

[Budinsky 04] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick & Timothy J. Grose. Eclipse modeling framework. Addison-Wesley Professional, 2004.

[Bulej 08] Lubomír Bulej, Tomáš Bureš, Thierry Coupaye, Martin Děcky, Pavel Ježek, Pavel Parízek, František Plášil, Tomáš Poch, Nicolas Rivierre, Ondřej Šerý & Petr Tůma. CoCoME in Fractal, volume 5153 of *Lecture Notes in Computer Science*. Springer, April 2008. `http://agrausch.informatik.uni-kl.de/CoCoME`.

[Burch 90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. Mcmillan, David L. Dill & L. J. Hwang. *Symbolic model checking: 10^{20} states and beyond*. In Logic in Computer Science, LICS, Proceedings., Fifth Annual IEEE Symposium, pages 428–439, 1990.

[Bureš 06] Tomáš Bureš, Petr Hnetynka & František Plášil. *SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model*. In SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[Bureš 08] Tomáš Bureš, Martin Děcky, Petr Hnětynka, Jan Kofroň, Pavel Parízek, František Plášil, Tomáš Poch, Ondřej Šerý & Petr Tůma. *CoCoME in SOFA*, April 2008. `http://agrausch.informatik.uni-kl.de/CoCoME`.

[Canal 06] Carlos Canal, Juan Manuel Murillo & Pascal Poizat. *Software Adaptation*. In Special Issue on Coordination and Adaptation Techniques for Software Entities, pages 9–31. in L'objet, 2006.

[Canal 08] Carlos Canal, Pascal Poizat & Gwen Salaün. *Model-Based Adaptation of Behavioral Mismatching Components*. IEEE Transactions on Software Engineering, vol. 34, no. 4, pages 546–563, 2008.

[Cansado 08a] Antonio Cansado, Denis Caromel, Ludovic Henrio, Eric Madelaine, Marcela Rivera & Emil Salageanu. A Specification Language for Distributed Components implemented in GCM/ProActive, volume 5153 of *Lecture Notes in Computer Science*. Springer, 2008. `http://agrausch.informatik.uni-kl.de/CoCoME`.

[Cansado 08b] Antonio Cansado, Ludovic Henrio & Eric Madelaine. *Transparent First-class Futures and Distributed Components*. In 5th workshop on Formal Aspects of Component Systems, Malaga, Spain, Sep To be published, 2008. ENTCS.

[Cansado 08c] Antonio Cansado, Ludovic Henrio & Eric Madelaine. *Unifying Architectural and Behavioural Specifications of Distributed Components*. In 5th workshop on Formal Aspects of Component Systems, Malaga, Spain, Sep To be published, 2008. ENTCS.

[Cansado 08d] Antonio Cansado, Eric Madelaine & Pablo Valenzuela. *VCE: A Graphical Tool for Architectural Definitions of GCM Components*. In 5th workshop on Formal Aspects of Component Systems, Malaga, Spain, Sep To be published as annex, 2008. ENTCS.

[Caromel 05a] Denis Caromel & Guillaume Chazarain. *Robust Exception Handling in an Asynchronous Environment*. In A. Romanovsky, C. Dony, JL. Knudsen & A. Tripathi, editeurs, Proceedings of ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems. Tech. Report No 05-050, Dept. of Computer Science, LIRMM, Montpellier-II Univ. July. France, 2005.

[Caromel 05b] Denis Caromel & Ludovic Henrio. A theory of distributed object. Springer-Verlag, 2005.

[Caromel 06a] Denis Caromel, Christian Delbé, Alexandre di Costanzo & Mario Leyton. *ProActive: an Integrated platform for programming and running applications on Grids and P2P systems*. Computational Methods in Science and Technology, vol. 12, no. 1, pages 69–77, 2006.

[Caromel 06b] Denis Caromel & Ludovic Henrio. *Asynchonous Distributed Components: Concurrency and Determinacy*. In Proceedings of the IFIP International Conference on Theoretical Computer Science 2006 (IFIP TCS'06), Santiago, Chile, August 2006. Springer Science. 19th IFIP World Computer Congress.

[Carrez 03] Cyril Carrez, Alessandro Fantechi & Elie Najm. *Behavioural Contracts for a Sound Assembly of Components*. In Springer-Verlag, editeur, in proceedings of FORTE'03, volume LNCS 2767, 2003.

[Chazarain 05] Guillaume Chazarain. *Exceptions et Asynchronisme*. Master thesis, Master recherche STIC spécialité RSD, UNSA, 2005.

[Cimatti 02] Alessandro Cimatti, Enrico Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani & Armando Tacchella. *Integrating BDD-Based and SAT-Based Symbolic Model Checking*. pages 265–276. 2002.

[Clarke 99] Edmund M. Clarke, Orna Grumberg & Doron A. Peled. Model checking. MIT Press, Cambridge, MA, USA, 1999.

[Cleaveland 93] Rance Cleaveland & M.C.B. Hennessy. *Testing Equivalence as a Bisimulation Equivalence*. Formal Aspects of Computing, vol. 5:1-20, 1993.

[Cleaveland 94] Rance Cleaveland & James Riely. *Testing-Based Abstractions for Value-Passing Systems*. In Int. Conference on Concurrency Theory (CONCUR), volume 836 of *Lecture Notes in Computer Science*, pages 417–432. Springer, 1994.

[Coglio 05] Alessandro Coglio & Cordell Green. *A Constructive Approach to Correctness, Exemplified by a Generator for Certified Java Card Appplets*. In Proc. IFIP Working Conference on Verified Software: Tools, Techniques, and Experiments, October 2005.

[CoreGRID 06] CoreGRID. *Basic Features of the Grid Component Model (assessed)*. Rapport technique, CoreGRID, Programming Model Institute, 2006. Deliverable D.PM.04, `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf`.

[Cousot 77] Patrick Cousot & Radhia Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pages 238–252, New York, NY, USA, 1977. ACM Press.

[Cousot 01] Patrick Cousot. *Abstract Interpretation Based Formal Methods and Future Challenges, invited paper*. In R. Wilhelm, editeur, Informatics — 10 Years Back, 10 Years Ahead, volume 2000 of *Lecture Notes in Computer Science*, pages 138–156. Springer-Verlag, 2001.

[Davis 60] Martin Davis & Hilary Putnam. *A Computing Procedure for Quantification Theory*. J. ACM, vol. 7, no. 3, pages 201–215, 1960.

[de Alfaro 01] Luca de Alfaro & Tom Henzinger. *Interface automata*. In Proceedings of the. ACM Press, January 2001.

[Denis 04] Alexandre Denis, Christian Pérez, Thierry Priol & André Ribes. *Bringing High Performance to the CORBA Component Model*. In SIAM Conference on Parallel Processing for Scientific Computing, February 2004.

[Dwyer 01] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Păsăreanu & Hongjun Zheng. *Tool-supported Program Abstraction for Finite-state Verification*. In Proceedings of the 23rd International Conference on Software Engineering, 2001.

[Ehrig 85] Hartmut Ehrig & Bernd Mahr. Fundamentals of algebraic specification i. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985. ISBN 0387137181.

[Emerson 86] E. A. Emerson & C. L. Lei. *Efficient Model Checking in Fragments of the Propositional μ–Calculus*. In Symposion on Logic in Computer Science, pages 267–278, Washington, D.C., USA, 1986. IEEE Computer Society Press.

[Erl 05] Thomas Erl. Service-oriented architecture: Concepts, technology, and design. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[F4E] F4E. *F4E Homepage*. `http://fractal.objectweb.org/f4e/index.html`.

[Fernandes 07] Fabricio Fernandes & Jean-Claude Royer. *The STSLIB Project: Towards a Formal Component Model Based on STS*. In Proceedings of the Fourth International Workshop on Formal Aspects of Component Software (FACS'07), Sophia Antipolis, France, September 2007. To appear in ENTCS.

[Fiadeiro 06] José Luiz Fiadeiro, Antónia Lopes & Laura Bocchi. *A Formal Approach to Service Component Architecture*. Web Services and Formal Methods, vol. 4184, pages 193–213, 2006.

[Fischer 79] Michael Fischer & Richard Ladner. *Propositional Dynamic Logic of Regular Programs*. Journal of Computer and System Sciences, vol. 18, no. 2, pages 194–211, 1979.

[Flanagan 99] Cormac Flanagan & Matthias Felleisen. *The Semantics of future and an application*. Journal of Functional Programming, vol. 9, no. 1, pages 1–31, 1999.

[Floyd 67] Robert W. Floyd. *Assigning meanings to programs*. In J. T. Schwartz, editeur, Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19, pages 19–32, Providence, 1967. American Mathematical Society.

[Fraclet] Fraclet. *Fraclet Homepage*. `http://fractal.objectweb.org/tutorials/fraclet/index.html`.

[FractalGUI] FractalGUI. *Fractal GUI Homepage*. `http://fractal.objectweb.org/fractalgui/index.html`.

[Garavel 89] Hubert Garavel. *Compilation of LOTOS Abstract Data Types*. In Son T. Vuong, editeur, FORTE, pages 147–162. North-Holland, 1989.

[Garavel 07] Hubert Garavel, Frédéric Lang, Radu Mateescu & Wendelin Serwe. *CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes*. In CAV, 2007.

[Garlan 00] David Garlan, Robert T. Monroe & David Wile. *Acme: Architectural Description of Component-Based Systems*. In Gary T. Leavens & Murali Sitaraman, editeurs, Foundations of Component-Based Systems, pages 47–68. Cambridge University Press, 2000.

[Gerth 97] Rob Gerth. *Concise PROMELA Reference*, 1997. available at `http://spinroot.com/spin/Man/Quick.html`.

[Gerth 01] Rob Gerth. *Model Checking if Your Life Depends on It a View from Intel's Trenches*. In Matthew B. Dwyer, editeur, SPIN, volume 2057 of *Lecture Notes in Computer Science*, page 15. Springer, 2001.

[GIDE] GIDE. *GIDE Homepage*. `http://perun.hscs.wmin.ac.uk/dis/gide/wiki/index.php/GIDE:Documentation`.

[Goldblatt 87] Robert Goldblatt. Logics of time and computation. Center for the Study of Language and Information, Stanford, CA, USA, 1987.

[Gordon 97] Andrew D. Gordon, Paul D. Hankin & Søren B. Lassen. *Compilation and Equivalence of Imperative Objects*. FSTTCS: Foundations of Software Technology and Theoretical Computer Science, vol. 17, pages 74–87, 1997.

[Grassi 08] Vincenzo Grassi, Raffaela Mirandola, Enrico Randazzo & Antonino Sabetta. *KLAPER: An Intermediate Language for Model-Driven Predictive Analysis of Performance and Reliability*. In Andreas Rausch, Ralf Reussner, Raffaela Mirandola & František Plášil, editeurs, The Common Component Modeling Example: Comparing Software Component Models, volume 5153 of *Lecture Notes in Computer Science*, pages 327–356. Springer, 2008.

[Hennessy 95] Matthew Hennessy & Huimin Lin. *Symbolic bisimulations*. Theoretical Computer Science, vol. 138, no. 2, pages 353–389, 1995.

[Henrio 03] Ludovic Henrio. *Asynchronous Object Calculus: Confluence and Determinacy*. PhD thesis, Université de Nice - Sophia Antipolis – UFR Sciences, November 2003.

[Hoare 85] Charles Antony Richard Hoare. Communicating sequential processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[Holzmann 97] Gerard Holzmann. *The Model Checker SPIN*. IEEE Trans. Softw. Eng., vol. 23, no. 5, pages 279–295, 1997.

[Holzmann 03] Gerard Holzmann. The spin model checker, primer and reference manual. Addison-Wesley, 2003. ISBN 0-321-22862-6.

[Ingólfsdóttir 01] Anna Ingólfsdóttir & Huimin Lin. *A Symbolic Approach to Value-Passing Processes*, 2001.

[Inverardi 01] Paola Inverardi & Simone Scriboni. *Connectors Synthesis for Deadlock-Free Component-Based Architectures*. In ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering, page 174, Washington, DC, USA, 2001. IEEE Computer Society.

[Inverardi 03] Paola Inverardi & Massimo Tivoli. *Deadlock-free software architectures for COM/DCOM applications*. J. Syst. Softw., vol. 65, no. 3, pages 173–183, 2003.

[ISO 89] ISO. *LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*. vol. ISO 8807, 1989.

[Ježek 05] Pavel Ježek, Jan Kofroň & František Plášil. *Model Checking of Component Behavior Specification: A Real Life Experience*. In International Workshop on Formal Aspects of Component Software (FACS'05), Macao, 2005. Electronic Notes in Theoretical Computer Science (ENTCS).

[Johnsen 06] Einar Broch Johnsen, Olaf Owe & Ingrid Chieh Yu. *Creol: a type-safe object-oriented model for distributed concurrent systems*. Theor. Comput. Sci., vol. 365, no. 1, pages 23–66, 2006.

[Jouault 06a] Frédéric Jouault, Jean Bézivin & Ivan Kurtev. *TCS:: a DSL for the specification of textual concrete syntaxes in model engineering*. In GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering, pages 249–254, New York, NY, USA, 2006. ACM.

[Jouault 06b] Frédéric Jouault & Ivan Kurtev. *Transforming Models with ATL*. In Satellite Events at the MoDELS 2005 Conference, LNCS 3844, pages 128–138. Springer, 2006. ISBN=0302-9743.

[Kesselman 98] Carl Kesselman & Ian Foster. The grid: Blueprint for a new computing infrastructure. Morgan Kaufmann Publishers, November 1998.

[Kiczales 01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm & William Griswold. *Getting started with ASPECTJ*. Commun. ACM, vol. 44, no. 10, pages 59–65, 2001.

[Knapp 08] Alexander Knapp, Stephan Janisch, Rolf Hennicker, Allan Clark, Stephen Gilmore, Florian Hacklinger, Hubert Baumeister & Martin Wirsing. Modelling the CoCoME with the Java/A Component Model, volume 5153 of *Lecture Notes in Computer Science*. Springer, April 2008. `http://agrausch.informatik.uni-kl.de/CoCoME`.

[Kofroň 06] Jan Kofroň, Jiří Adámek, Tomáš Bureš, Pavel Ježek, Vladimir Mencl, Pavel Parízek & František Plášil. *Checking Fractal Component Behavior Using Behavior Protocols*. In 5th Fractal Workshop, Nantes, France, July 2006.

[Kofroň 07] Jan Kofroň. *Behavior Protocols Extensions*. PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, September 2007.

[Kozen 85] Dexter Kozen. *Results on the Propositional Mu-Calculus*. Theoretical Computer Science, vol. 40, 1985.

[Lakas 96] Abderrahmane Lakas. *Les Transformations Lotomaton : une contribution à la pré-implémentation des systèmes Lotos*. PhD thesis, Univ. Paris VI, june 1996.

[Lang 05] Fréderic Lang. *Exp.open 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods*. In Proceedings of the 5th International Conference on Integrated Formal Methods IFM'2005, Eindhoven, The Netherlands, 2005. LNCS 3771.

[Lin 96] Huimin Lin. *Symbolic Transition Graph with Assignment*. In U. Montanari & V. Sassone, editeurs, CONCUR '96, Pisa, Italy, 1996. LNCS 1119.

[Lynch 87] Nancy A. Lynch & Mark R. Tuttle. *Hierarchical correctness proofs for distributed algorithms*. In PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, pages 137–151, New York, NY, USA, 1987. ACM.

[Madelaine 92] Eric Madelaine. *Verification Tools from the CONCUR project*. EATCS Bull., vol. 47, 1992.

[Magee 99] Jeff Magee, Jeff Kramer & Dimitra Giannakopoulou. *Behaviour Analysis of Software Architectures*. In WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1), pages 35–50, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[Manna 92] Zohar Manna & Amir Pnueli. The temporal logic of reactive and concurrent systems. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[Martens 06] Axel Martens & Simon Moser. *Diagnosing SCA Components Using Wombat*. In Business Process Management, volume 4102 of *Lecture Notes in Computer Science*, pages 378–388. Springer-Verlag, 2006.

[Mateescu 00] Radu Mateescu. *Efficient Diagnostic Generation for Boolean Equation Systems*. In Tools and Algorithms for Construction and Analysis of Systems, pages 251–265, 2000.

[Milner 82] Robin Milner. A calculus of communicating systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Milner 89] Robin Milner. Communication and concurrency. Prentice Hall, 1989. ISBN 0-13-114984-9.

[Milner 90] Robin Milner. *Operational and Algebraic Semantics of Concurrent Processes*. In J. van Leeuwen, editeur, Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics, pages 1201–1242. Elsevier, Amsterdam, 1990.

[Milner 92] Robin Milner, Joachim Parrow & David Walker. *A calculus of mobile processes*. Inf. Comput., vol. 100, no. 1, pages 1–77, 1992.

[Milner 93] Robin Milner, Joachim Parrow & David Walker. *Modal logics for mobile processes*. Theoretical Computer Science, vol. 114, no. 1, pages 149–171, 1993.

[Moisan 03] Sabine Moisan, Annie Ressouche & Jean-Paul Rigault. *Behavioral Substitutability in Component Frameworks: a Formal Approach*. In In: Proceedings of the 2003 Workshop of Specification and Verification of Component Based Systems, pages 22–28, 2003.

[Najm 92] Elie Najm, Abderrahmane Lakas, A. Serouchni, Eric Madelaine & Robert de Simone. *ALTO: an interactive transformation tool for LOTOS and LOTOMATON*. In T. Bolognesi, E. Brinksma & C. Vissers, editeurs, Third Lotosphere Workshop and Seminar, Pisa, 1992.

[Nicola 90] Rocco De Nicola & Frits W. Vaandrager. *Action versus state based logics for transition systems*. In I. Guessarian, editeur, Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science, volume 469 of *LNCS*, La Roche Posay, France, 1990. Springer.

[Nierstrasz 95] Oscar Nierstrasz & Theo Dirk Meijler. *Research directions in software composition*. ACM Comput. Surv., vol. 27, no. 2, pages 262–264, 1995.

[Obj 03] Object Management Group. *UML 2.0 Object Constraint Language (OCL) Specification*, formal/03-10-14 edition, 2003. version 2.0.

[Obj 04] Object Management Group. *The Common Object Request Broker Architecture (CORBA): Core Specification*, 2004. version 3.0.3.

[OMG 02] OMG. *CORBA components, version 3*. Document formal/02-06-65, 2002.

[OMG 04] OMG. *UML 2.0 Superstructure Specification*, 2004.

[OMG 05] OMG. *CORBA Component Model, V3.0*. `http://www.omg.org/technology/documents/formal/components.htm`, 2005.

[Owe 07] Olaf Owe, Gerardo Schneider & Martin Steffen. *Components, objects, and contracts*. In SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems, pages 95–98, New York, NY, USA, 2007. ACM.

[Parks 03] Thomas Parks & David Roberts. *Distributed Process Networks in Java*. In Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS2003), Nice, France, April 2003.

[Parízek 07a] Pavel Parízek & František Plášil. *Modeling Environment for Component Model Checking from Hierarchical Architecture*. Electron. Notes Theor. Comput. Sci., vol. 182, pages 139–153, 2007.

[Parízek 07b] Pavel Parízek & František Plášil. *Partial Verification of Software Components: Heuristics for Environment Construction*. euromicro, vol. 0, pages 75–82, 2007.

[Perry 92] Dewayne E. Perry & Alexander L. Wolf. *Foundations for the study of software architecture*. SIGSOFT Softw. Eng. Notes, vol. 17, no. 4, pages 40–52, 1992.

[Plášil 98] František Plášil, Dušan Bálek & Radovan Janeček. *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*. In Proc. Fourth International Conf. Configurable Distributed Systems (ICCDS'98), pages 42–52. IEEE CS Press, May 1998.

[Plášil 02] František Plášil & Stanislav Visnovsky. *Behavior Protocols for Software Components*. IEEE Transactions on Software Engineering, vol. 28, no. 11, 2002.

[Pnueli 77] Amir Pnueli. *The Temporal Logic of Programs*. In FOCS'77, pages 46–57. IEEE, 1977.

[Pontisso 06] Nadège Pontisso & David Chemouil. *TOPCASED Combining Formal Methods with Model-Driven Engineering.* In ASE, pages 359–360. IEEE Computer Society, 2006.

[Queille 83] Jean-Pierre Queille & Joseph Sifakis. *Fairness and related properties in transition systems – A Temporal Logic to Deal with Fairness*. Acta Informatica, vol. 19, no. 3, pages 195–220, 1983.

[Rausch 08] Andreas Rausch, Ralf Reussner, Raffaela Mirandola & František Plášil. The Common Component Modeling Example: Comparing Software Component Models, volume 5153 of *Lecture Notes in Computer Science*. Springer, 2008.

[Reisig 85] Wolfgang Reisig. Petri nets: an introduction. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[Ressouche 94] Annie Ressouche, Robert de Simone, Amar Bouali & Valérie Roy. The FC2Tool user manuel. `http://www-sop.inria.fr/meije/verification/`, 1994.

[Reussner 03] Ralf H. Reussner, Iman H. Poernomo & Heinz W. Schmidt. *Reasoning on Software Architectures with Contractually Specified Components*. In A. Cechich, M. Piattini & A. Vallecillo, editeurs, Component-Based Software Quality: Methods and Techniques, numéro 2693 in LNCS, pages 287–325. Springer, 2003.

[SAE Standards 04] SAE Standards. *Architecture Analysis and Design Language (AADL), AS5506*. Rapport technique, Society of Automotive Engineers, November 2004.

[Seinturier 06] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien & Thierry Coupaye. *A Component Model Engineered with Components and Aspects*. In Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski & Kurt C. Wallnau, editeurs, CBSE, volume 4063 of *Lecture Notes in Computer Science*, pages 139–153. Springer, 2006.

[Sensoria 05] Sensoria. *Sensoria webpage*, 2005. Software Engineering for Service-Oriented Overlay Computers.

[SOFA 98] SOFA. *SOFA: Software Appliances Web SIte*, 1998. `http://nenya.ms.mff.cuni.cz/`.

[Szyperski 02] Clemens Szyperski. Component software: Beyond object-oriented programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Valmari 98] Antti Valmari. *The State Explosion Problem*. In Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, pages 429–528, London, UK, 1998. Springer-Verlag.

[Vardi 86] Moshe Y. Vardi & Pierre Wolper. *An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report)*. In LICS'86, pages 332–344. IEEE, 1986.

[Vardi 94] Moshe Y. Vardi & Pierre Wolper. *Reasoning about infinite computations*. Inf. Comput., vol. 115, no. 1, pages 1–37, 1994.

[Černá 06] Ivana Černá, Pavlína Vařeková & Barbora Zimmerova. *Component Substitutability via Equivalencies of Component-Interaction Automata*. In Proceedings of the Workshop on Formal Aspects of Component Software (FACS'06), Prague, Czech Republic, September 2006. To appear in ENTCS.

[W3C 04] W3C. *World Wide Web Consortium, Web Services Glossary*. http://www.w3.org/TR/ws-gloss/, February 2004.

[Wikipedia 08] Wikipedia. *Source lines of code*. `http://en.wikipedia.org/wiki/Source_lines_of_code`, 2008.

[Yellin 97] Daniel M. Yellin & Robert E. Strom. *Protocol specifications and component adaptors*. ACM Trans. Program. Lang. Syst., vol. 19, no. 2, pages 292–333, 1997.

Part III

Appendices

A

Case-Study

Contents

Abstract

In this appendix we validate our approach in a large case-study. This work is part of the CoCoME initiative to establish a common case-study to analyse different component models and modeling techniques.

We specify the CoCoME common example in both JDC specification language and its graphical version VCE[a].

From these specifications we generate behavioural models that allows us to analyse the system behaviour the system. We verify various use-case scenarios described in the CoCoME reference manual.

We also present a hand-made implementation of the code that will be generated from JDC.

[a]We use an earlier version of VCE called CTTool

Motivation

In this appendix apply our specification methods on a common example of component based software engineering. This example is called Common Component Modelling Example (CoCoME [Rausch 08]). As its name suggests, CoCoME is an initiative for defining a common component example, and may be used for comparing different component models. The subset of CoCoME we modelled consists in 16 components, 5 of them being composites. Furthermore, composite components were designed with up to 5 layers of hierarchy, stressing the need of hierarchical component models. In fact this system necessitates hierarchical components, component multiplicities, collective communications (for addressing a specific component and broadcasting a message), modelling of exceptions, and synchronous and asynchronous method calls.

We have chosen to include in this appendix only the specification of two components, with the required views of each, with the JDC text, and some of the CTTool[*] diagrams as illustrations. The first one is the (composite) *Cash Desk* component, for which we include and explain:

- its black-box view with the definition of external interfaces, and the specification of its visible behaviour;
- its architectural view with its subcomponents and bindings;
- its GCM view, with excerpts of the generated ADL code.

The second one is the (primitive) *Cash Box Controller* component, with:

- its black-box view with its interfaces and behaviour, with much more details on the definition of its service methods;
- no architecture as it is primitive;
- an abstraction specification of a user-defined datatype;
- pieces of generated Java/ProActive code;
- a fragment of its deployment specification.

The contents of this annex were published in [Cansado 08a]. The interested reader may find useful to compare this work with the ones provided by Java/A [Knapp 08], SOFA [Bureš 08], and Fractal [Bulej 08].

A.1 Description of the Case-Study

The example studied in this appendix consists in a Trading System as it can be observed in a supermarket handling sales. This includes the processes at a single *Cash Desk*. Typical operations are scanning products using a *Bar Code Scanner* or paying by credit card or cash. A general schema of the architecture can be seen in Figure A.1.

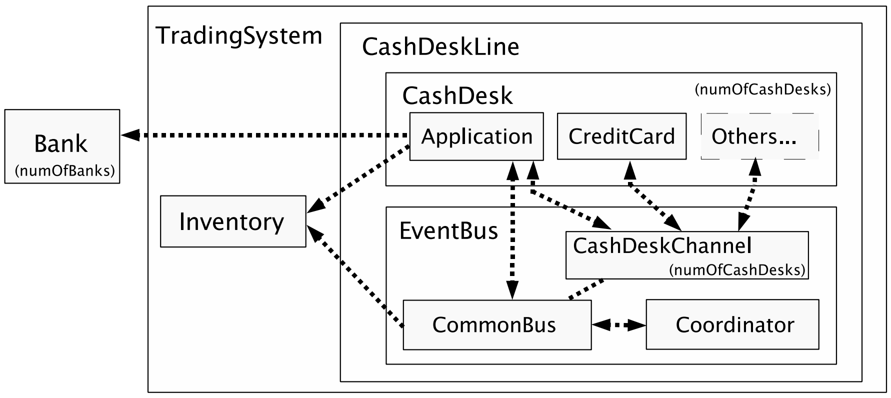


Figure A.1: *The CoCoME overview*

[*]CTTool is an old version of VCE, presented in [Ahumada 07]

The *Cash Desk* is the place where the Cashier scans the products the Customer wants to buy and where the payment (either by credit card or cash) is executed. Furthermore it is possible to switch into an express checkout mode; this mode allows only Customers with few products paying by cash.

To manage the processes at a *Cash Desk*, a lot of hardware devices are necessary (*Card Reader, Cash Box, Cash Desk GUI, Light Display, Printer, Scanner*).

Using the *Cash Box* which is available at each *Cash Desk*, a sale is started and finished. Also the cash payment is handled by the *Cash Box*.

To manage payments by credit card a *Card Reader* is used.

The Cashier uses the *Bar Code Scanner* in order to identify all the products the Customer wants to buy. At the end of the payment process a bill is produced using a *Printer*. Each *Cash Desk* is also equipped with a *Light Display* to let the Customer know if this *Cash Desk* is in the express checkout mode or not.

Besides the *Cash Desks*, the system includes an *Inventory*. This is, however, simplified in our case-study. We consider only that the *Inventory* will stock the bills and provides information on the products, such as price and availability.

Finally, there are a set of banks accessible by the system. They are external entities for which we only provide an abstract specification.

A.2 Modeling the CoCoME

Before starting with those examples of component specifications, let us define what are the different views that we use.

A.2.1 Black-box View

We call black-box view of a component its externally visible architecture and behaviour. Therefore it includes the common part of both primitive and composite component: the list of its interfaces (defining which are client and server interfaces), and the definition of these interfaces (the Java methods and their signatures). The black-box view allows the component to be used without exposing its internals.

A.2.2 Architectural View

The architectural view gives a one-level refinement of a component as a composition of subcomponents. For each one of these subcomponents, the designer must provide its black-box view. Note that, when an architecture is provided for a component, the `policy` section in the black-box definition is optional as it is implicitly defined by the architecture.

For the CoCoME model, we have specified the architecture, whenever given in the CoCoME reference, of every component except for the Inventory. The latter was given only a black-box specification to simplify the model. Within this section we show the architecture of the *Cash Desk* component.

A.2.3 GCM View

Our objective is to generate GCM components from the JDC specification. JDC is rich enough to be able to generate automatically both the ADL describing the structure of the application, and the skeleton of the primitive components in ProActive.

For the composite components, the ADL can be automatically inferred from the architectural view presented above. For the primitive components, the ADL mainly consists in the definition of interfaces and can be generated automatically. The skeleton of the Java class implementing the primitive can be generated from the JDC black-box specifying the behaviour of the component. The user then only has to write the business code, resolving all the abstractions and non-determinism present in the black-box definition.

Concerning non-functional aspects, they are not specified for now in the JDC. Thus it is also the role of the programmer to provide and compose these aspects. In general, most of those aspects are provided by the component middleware, e.g. Fractal requires basic management controllers to be implemented by each component. In most cases, dealing with non-functional aspects consists in invoking operations on these controllers. For the moment, those invocations have to be performed manually by the programmer; but on the long term basis we would like to include them in the JDC so that it is possible to study and verify the interaction between functional and non-functional concerns.

The generation of code from the JDC specification is not working yet; thus based on the JDC specification we wrote the GCM/ProActive code for the CoCoME example. This code consists of a set of composite components defined in the ADL, and a set of primitive components written in Java and using ProActive. Recall that, at deployment, a thread is created for each primitive and each composite component, and those components communicate by asynchronous method calls with transparent futures, leading to a parallel and distributed implementation of the application.

A.2.4 Deployment View

The deployment view of an application is out-of-scope for this work. However, since the modelling of distributed aspects is an important part of a specification, this section outlines the ProActive deployment scheme, and shows how the distribution nature of GCM components can be captured. Further, using the CoCoME architecture as an example, it is shown how to map the components to the physical infrastructure.

ProActive and GCM comprehensive deployment framework is based on the concept of *Virtual Node* (VN). A VN is above all an *application abstraction* that, at modelling or programming time, captures the distributed nature of a system. Typically, a given application is specified to deploy on several VNs (e.g. each component on a separate VN), each capturing a specific entity or related set of entities of the application. At deployment, each VN is mapped to one or several machines on the network, using appropriate protocols.

The number and characteristics of VNs are chosen by the application designer, providing both guidance and constraints to be used and enforced at deployment time. Both parallelism and de facto distribution can be captured by VNs. Moreover, the designer can also specify multi-threaded constraints by using a single VN for several components, capturing a forced co-allocation. When building composite components, one has the possibility to merge some inner VNs into a single one, specifying co-allocation of the corresponding inner components. One also has the possibility to maintain at the level of the composite some of the inner VNs, specifying an independent mapping of the corresponding inner components to the physical infrastructure.

A VN has several characteristics. The most important is its *cardinality*, which can be *single* or *multiple*. The former captures the fact that a single node of the infrastructure has to be used to execute the corresponding component, the later gives the possibility at deployment to map the component on several machines. This powerful possibility is to be related to multicast and gathercast interfaces: a collective interface often corresponds to a Multiple VN with the same cardinality.

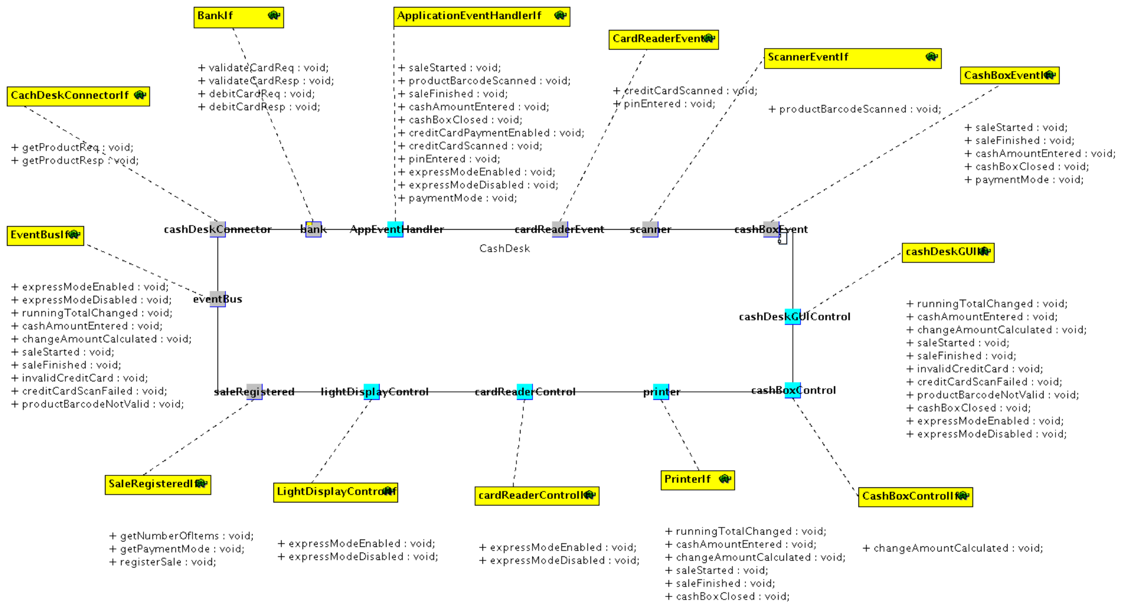
At deployment time, all the VNs of the CoCoME specification will be mapped to one or several machines of the physical infrastructure using an XML file. The ProActive implementation makes it possible to choose from many protocols to select and access the actual nodes (rsh, ssh, LSF, PBS, Globus, etc.), and to control the number of components per machine, and per process (JVM).

A.2.5 Specification of the *Cash Desk* Component

In this subsection, we specify the *Cash Desk* component. We give its black-box specification together with a matching architecture specification. Finally, we outline its implementation within the GCM/ProActive.

A.2.5.1 Black-Box View of the *Cash Desk*

The black-box of the *Cash Desk* starts by defining its server and client interfaces. In there, we see that the `bankIf` is a collection interface as it addresses multiple banks, but method calls are routed to one bank at a time.

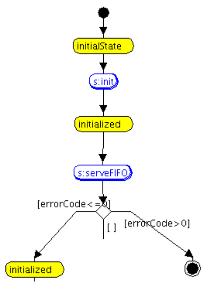

BankIf
+ validateCardReq : void;
+ validateCardResp : void;
+ debitCardReq : void;
+ debitCardResp : void;
ApplicationEventHandlerIf
+ saleStarted : void;
+ productBarcodeScanned : void;
+ saleFinished : void;
+ cashAmountEntered : void;
+ cashBoxClosed : void;
+ creditCardPaymentEnabled : void;
+ creditCardScanned : void;
+ pinEntered : void;
+ expressModeEnabled : void;
+ expressModeDisabled : void;
+ paymentMode : void;
CardReaderEventIf
+ creditCardScanned : void;
+ pinEntered : void;
ScannerEventIf
+ productBarcodeScanned : void;
CashBoxEventIf
+ saleStarted : void;
+ saleFinished : void;
+ cashAmountEntered : void;
+ cashBoxClosed : void;
+ paymentMode : void;
CachDeskConnectorIf
+ getProductReq : void;
+ getProductResp : void;
EventBusIf
+ expressModeEnabled : void;
+ expressModeDisabled : void;
+ runningTotalChanged : void;
+ cashAmountEntered : void;
+ changeAmountCalculated : void;
+ saleStarted : void;
+ saleFinished : void;
+ invalidCreditCard : void;
+ creditCardScanFailed : void;
+ productBarcodeNotValid : void;
cashDeskConnector
bank
AppEventHandler
cardReaderEvent
scanner
cashBoxEvent
CashDesk
eventBus
cashDeskGUIControl
cashDeskGUIIf
+ runningTotalChanged : void;
+ cashAmountEntered : void;
+ changeAmountCalculated : void;
+ saleStarted : void;
+ saleFinished : void;
+ invalidCreditCard : void;
+ creditCardScanFailed : void;
+ productBarcodeNotValid : void;
+ cashBoxClosed : void;
+ expressModeEnabled : void;
+ expressModeDisabled : void;
saleRegistered
lightDisplayControl
cardReaderControl
printer
cashBoxControl
SaleRegisteredIf
+ getNumberOfItems : void;
+ getPaymentMode : void;
+ registerSale : void;
LightDisplayControlIf
+ expressModeEnabled : void;
+ expressModeDisabled : void;
cardReaderControlIf
+ expressModeEnabled : void;
+ expressModeDisabled : void;
PrinterIf
+ runningTotalChanged : void;
+ cashAmountEntered : void;
+ changeAmountCalculated : void;
+ saleStarted : void;
+ saleFinished : void;
+ cashBoxClosed : void;
CashBoxControlIf
+ changeAmountCalculated : void;

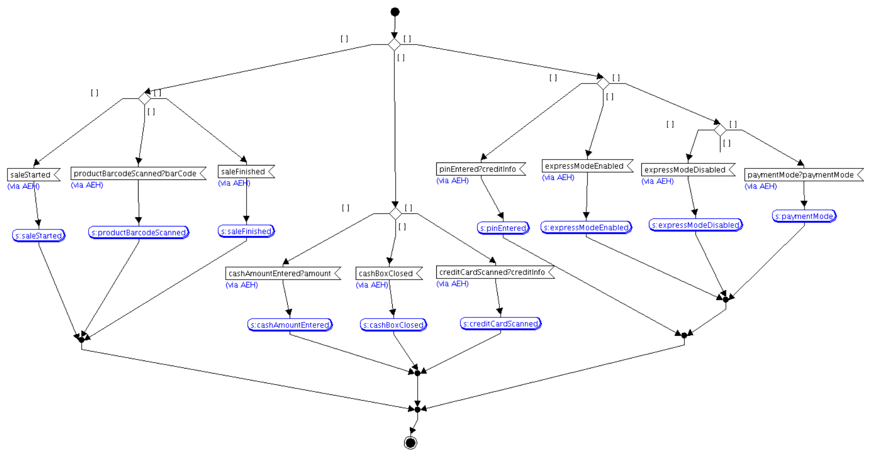
```
component CashDesk(int numOfBanks) {
  interfaces
    server interface CardReaderControlIf cardReaderControlIf;
    server interface ApplicationEventHandlerIf applicationEventHandlerIf;
    client interface BankIf banksIf[numOfBanks];
    // ... all 10 other interfaces
}
```

These interfaces must be properly defined. For example, in the code below we see the definition of the `ApplicationEventHandlerIf` interface, which exposes the full method signature using user-classes.

```
public interface ApplicationEventHandlerIf {
    void saleStarted();
    void saleFinished();
    void cashAmountEntered(CashAmount moneyAmountEntered);
    void cashBoxClosed();
    void creditCardScanned(CreditCardScanned creditCardScanned);
    void pinEntered(PIN pin);
    void paymentMode(PaymentMode paymentMode);
    void expressModeDisabled();
    void expressModeEnabled();
    void productBarcodeScanned(ProductBarcode barcode);
}
```

No matter how the component is to be implemented (either by a primitive or a composite component), in JDC it is mandatory to provide a behavioural specification, either in the form of a black-box definition, or in the form of its architectural implementation, or both. For this component, we provide a black-box specification; this starts with the service policy defined with a regular expression. In the case of the `CashDesk`, there are multiple services denoting that there are multiple processes visible from the outside. This stands for a compact representation of the interleavings admitted by the component.



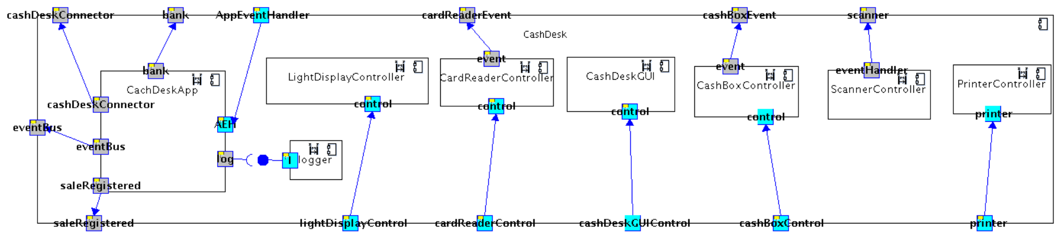


```
 services
   service { // CardReaderController
      policy {
         ((emit() | serveOldest(cardReaderControlIf.expressModeDisabled))*;
         serveOldest(cardReaderControlIf.expressModeEnabled);
         serveOldest(cardReaderControlIf.expressModeDisabled))*
      }
      void emit() {
         if (__ANY(bool)) // non-deterministic choice
            cardReaderEventIf.creditCardScanned(__ANY(CreditCard));
      }
      // ... cardReaderControlIf.expressModeEnabled
      // ... cardReaderControlIf.expressModeDisabled
   }
   service { // CashDeskApplication
      locals {
         CashState cashState;
         // ... other local variables
      }
      policy {
         init(); expressModeDisabled();
         serveOldest(applicationEventHandlerIf) *
      }
      void applicationEventHandlerIf.saleStarted() {
         switch (cashState.getState()) {
            case cashState.IDLE:
               cashState = cashState.STARTED;
               break;
            case cashState.STARTED:
            case cashState.PAYING:
               __ERROR(NotIdleException);
               break;
         }
      }
      // ... other methods
   }
}
```

Note that each service defines its own service methods and possibly its own local variables.

A.2.5.2 Architectural View of the *Cash Desk*

As the black-box specification only defines the externally visible behaviour and not how it is implemented, there are, of course, several architecture definitions that match the same component's black-box specification. A possible architecture implementing the `CashDesk` is shown in the following. Note that we added a logger just to trap and print exceptions but it does not influence the component behaviour.



The first part of the architecture defines the subcomponents which compose the component. When applies, they are provided with parameters for their correct deployment as with the `CashDeskApplication`.

```
architecture CashDesk(int numOfBanks) {
   contents
      component CashDeskApplication(numOfBanks) application;
      component CashReaderController cashReader;
      component CashDeskGUI cashDeskGUI;
      component cashBoxController cashBoxController;
      component LightDisplayController lightDisplayController;
      component PrinterController printerController;
      component ScannerController scannerController;
```

Then, we define the bindings section exposing how the functional delegation takes place, and synchronisations between components. Note that it is possible to address either a specific component or a specific collection interface.

```
   bindings
      // application
      bind(this.applicationEventHandlerIf, application.applicationEventHandlerIf);
      bind(application.cashDeskConnectorIf, this.cashDeskConnectorIf);
      bind(application.eventBusIf, this.eventBusIf);
      bind(application.saleRegisteredIf, this.saleRegisteredIf);

      // bind all bank interfaces
      for (int i: numOfBanks) {
         bind(application.banksIf[i], this.banksIf[i]);
      }

      // cashReader
      bind(this.cashReaderControlIf, cashReader.controlIf);
      bind(cashReader.eventIf, this.eventIf);

      // cashDeskGUI
      bind(this.cashDeskGUIControlIf, cashDeskGUI.controlIf);

      // cashBoxController
      bind(this.cashBoxControllerControlIf, cashBoxController.controlIf);
      bind(cashBoxController.eventIf, this.cashBoxControllerEventIf);

      // lightDisplayController
      bind(this.lightDisplayControllerControlIf, lightDisplayController.controlIf);

      // printerController
      bind(this.printerIf, printerController.printerIf);

      // scannerController
      bind(scannerController.scannerIf, this.scannerIf);
}
```

The corresponding ADL file, in XML format, will be generated from this part of the specification. It will then be used both to generate the synchronisation structures for the model-checker, and as an input to the component factory of ProActive at deployment time, but all this is left as future work for the moment.

A.2.5.3 GCM View of the *Cash Desk*

Now, we present a simplified ADL description of the *Cash Desk* component. It focuses on the *Cash Desk Application* subcomponent, the other subcomponents being similar. The ADL description starts with the definition of the external interfaces of the *Cash Desk* component, together with their roles (client or server).

```
<component name="CashDesk">
  <interface signature="CashDeskLine.if.LightDisplayControlIf" role="server" name="
      lightDisplayControlIf"/>
  <interface signature="CashDeskLine.if.CardReaderControlIf" role="server" name="
      cardReaderControlIf"/>
  <interface signature="CashDeskLine.if.CashDeskGUIIf" role="server" name="cashDeskGUIIf
      "/>
  <interface signature="CashDeskLine.if.CashBoxControlIf" role="server" name="
      cashBoxControlIf"/>
  <interface signature="CashDeskLine.if.PrinterIf" role="server" name="printerIf"/>
  <interface signature="CashDeskLine.if.ApplicationEventHandlerIf" role="server" name="
      applicationEventHandlerIf"/>
  <interface signature="if.CashDeskConnectorIf" role="client" name="cashDeskConnectorIf"
      />
  <interface signature="if.SaleRegisteredIf" role="client" name="saleRegisteredIf"/>
  <interface signature="CashDeskLine.if.CardReaderEventIf" role="client" name="
      cardReaderEventIf"/>
  <interface signature="CashDeskLine.if.CashBoxEventIf" role="client" name="
      cashBoxEventIf"/>
  <interface signature="CashDeskLine.if.ScannerEventIf" role="client" name="
      scannerEventIf"/>
  <interface signature="if.BankIf" role="client" name="bankIf"/>
  <interface signature="CashDeskLine.if.EventBusIf" role="client" name="eventBusIf"/>
```

Then the subcomponent *Cash Desk Application* is described by its external interfaces, this component is a primitive one (line 180), so the path of its implementation is given (line 179).

```
<component name="CashDeskApplication">
  <interface signature="CashDeskLine.if.ApplicationEventHandlerIf" role="server" name=
      "applicationEventHandlerIf"/>
  <interface signature="if.CashDeskConnectorIf" role="client" name="
      cashDeskConnectorIf"/>
  <interface signature="if.SaleRegisteredIf" role="client" name="saleRegisteredIf"/>
  <interface signature="CashDeskLine.if.EventBusIf" role="client" name="eventBusIf"/>
  <interface signature="if.BankIf" role="client" name="bankIf"/>
  <content class="CashDeskLine.CashDesk.CashDeskApplication"/>
  <controller desc="primitive"/>
</component>
<component name="CardReaderController"> ... </component>
<component name="LightDisplayController"> ... </component>
<component name="ScannerController"> ...  </component>
<component name="PrinterController"> ...  </component>
<component name="CashBoxController"> ...  </component>
<component name="CaskDeskGUI">    ...     </component>
```

Finally, the bindings of the *Cash Desk Application* are described, in this example only two kinds of bindings are shown: import bindings like the first one, and export bindings like the others.

```
<binding client="this.applicationEventHandlerIf" server="CashDeskApplication.
    applicationEventHandlerIf"/>
<binding client="CashDeskApplication.cashDeskConnectorIf" server="this.
    cashDeskConnectorIf"/>
<binding client="CashDeskApplication.saleRegisteredIf" server="this.saleRegisteredIf"/
    >
<binding client="CashDeskApplication.eventBusIf" server="this.eventBusIf"/>
<binding client="CashDeskApplication.bankIf" server="this.bankIf"/>
...
<controller desc="composite"/>
</component>
```

A.2.5.4 Deployment View of the *Cash Desk*

When defining the *Cash Desk Line* composite, a VN `CashDeskLineVN`, cardinality Multiple is specified as the composition of all the `CashDeskVN`. The effective cardinality of this VN is attached to the number of cashDesks (`numOfCashDesks` in the specification).

A.2.6 Specification of the *Cash Box Controller* Component

In this section, we specify the *Cash Box Controller* component. We give its black-box view, but the architectural view does not apply because we do not decompose the behaviour of the *Cash Box Controller* into subcomponents.

A.2.6.1 Black-Box View of the *Cash Box Controller*

The component has a client and a server interface, and defines a non-trivial service policy. The controller can be seen as an active component in the sense that it triggers events regarding the *Cash Box*, so it is not awaiting for any signal to be received.

```
component CashBoxController {
 interfaces
    server interface CashBoxControlIf controlIf;
    client interface CashBoxEventIf eventIf;

 services
    service {
       policy {
          ( eventIf.saleStarted(); eventIf.saleFinished();
           ( cashMode(); cashAmount(); serveOldest(controlIf.changeAmountCalculated);
               eventIf.cashBoxClosed() )
           |
           ( creditCardMode() )
          )*
       }
```

There are no state variables (variables defined within the “locals” block), nevertheless the component is not stateless; the service policy implicitly defines that the component cycles through some states, each one defining which are the actions that the *Cash Box Controller* may do. For example, the component only serves requests from the queue when a client is paying with cash; otherwise, the component is seen as a machine sending events regardless of the environment (as the environment does not take the hardware interaction into account).

The component defines local methods and service methods; the latter have their method names prefixed by the interface they belong. For the method `changeAmountCalculated(..)`, and from the behavioural point of view, we are only interested in the access to the variable sent as argument, but not what we actually do with it; so the behavioural model can block the execution until the concrete value of the variable is known.

```
    // local methods
       void cashMode() {
          eventIf.paymentMode(new PaymentMode(CASH));
       }
       void creditCardMode() {
          eventIf.paymentMode(new PaymentMode(CREDIT));
       }
       void cashAmount() {
          eventIf.cashAmount(__ANY(CashAmount));
       }
    // service methods
       void controlIf.changeAmountCalculated(CashAmount changeAmount) {
          changeAmount.waitForValue();
    }}
} // end of CashBoxController black-box definition
```

There are some non-deterministic choices, both in the events that may be sent, and in the data sent. For example, we do not know exactly which amount the user paid, so we use the "oracle" function __ANY(CashAmount) to choose any value within the variable domain. It will be up to the data abstraction to define this domain, or in the case of the implementation, to the programmer to define its real value. A mapping of this class into Simple Types is given as:

```
public class CashAmount {
   private double amount abstracted as enum {"ZERO", "NOT_ZERO" };

   public __ANY() {
      return new enum {"ZERO", "NOT_ZERO" };
   }
   public double getAmount() abstracted as {
      return amount;
   }
   public void add(CashAmount purchasePrice) abstracted as {
      if (this.amount == "ZERO" && purchasePrice.getAmount() == "ZERO")
         amount = "ZERO";
      else
         // non-determinism within the data abstraction as a negative
         // value could leave the amount in zero
         amount = __ANY(CashAmount);
}}
```

A.2.6.2 GCM View of the *Cash Box Controller*

Next, we give the implementation of the *Cash Box Controller* primitive component. This Java code is to be instantiated as an active object (in Java implementing the RunActive interface). The active object implements the CashBoxControlIf server interface and contains a field named cashBoxEventIf implementing the client interface of the component. Finally, it also implements Fractal's BindingController interface to allow dynamic binding of its interfaces.

```
public class CashBoxController implements CashBoxControlIf, BindingController,
    RunActive {
  public final static String CASHBOXEVENTIF_BINDING = "cashBoxEventIf";
  private CashBoxEventIf cashBoxEventIf;

  public CashBoxController () { } // empty constructor required by ProActive
```

Note that necessary hooks for Fractal controllers are implemented on the form of the four first methods of the object. These are generated automatically.

```
  // Implementation of the Controller interfaces
  public String[] listFc () {return new String[] { CASHBOXEVENTIF_BINDING };}

  public Object lookupFc (final String clientItfName) {
     if (CASHBOXEVENTIF_BINDING.equals(clientItfName))
        return cashBoxEventIf;
     return null;
  }
  public void bindFc (final String clientItfName,final Object serverItf){
     if (CASHBOXEVENTIF_BINDING.equals(clientItfName))
        cashBoxEventIf = (CashBoxEventIf)serverItf;
  }
  public void unbindFc (final String clientItfName){
     if (CASHBOXEVENTIF_BINDING.equals(clientItfName))
        cashBoxEventIf = null;
  }
```

Recall that changeAmountCalculated is the only method of the server interface, requests addressed to this component will be asynchronous calls to this method. It corresponds to the same method as in the black-box view above.

```
  // Implementation of the functional interfaces
  public void changeAmountCalculated(CashAmount changeAmount) {
     // the amount to be returned as change
     System.out.println(changeAmount.getAmount());
  }
```

Then, the service method in ProActive (runActivity) implements the service policy. It is the kernel of a ProActive component as it exposes the component's behaviour, and is called by the middleware when the component is started. It consists of a set of invocations on the client interface (cashBoxEventIf), together with a blocking service on the changeAmountCalculated method. This method is a direct translation of the policy section of the black-box presented above and should not be modified by the ProActive programmer.

```
  public void runActivity(Body body) {
     Service service = new Service(body);
     while (body.isActive()) {
        cashBoxEventIf.saleStarted();
        cashBoxEventIf.saleFinished();
        if ((new AnyBool()).prob(50)) {
           cashMode();
           cashAmount();
           service.blockingServeOldest("changeAmountCalculated");
           cashBoxEventIf.cashBoxClosed();
        } else
           creditCardMode();
  }}
```

By contrast the service method and the local (private) methods that are declared in the JDC (CashBoxController service declaration will contain the true functional code of the component, and are directly modifiable.

```
  private void cashMode() {
     cashBoxEventIf.paymentMode(new PaymentModeImpl(PaymentModeImpl.CASH));
  }
  private void creditCardMode() {
     cashBoxEventIf.paymentMode(new PaymentModeImpl(PaymentModeImpl.CREDIT));
  }
  private void cashAmount() {
     cashBoxEventIf.cashAmountEntered(
       new CashAmountImpl(1000)); // the client paid 1000
  }
} // end of CashBoxController implementation
```

A.2.6.3 Deployment View of the *Cash Box Controller*

If we want to model a system where a cashBoxController can be instantiated on its own processing element, then a VN, for instance named cashBoxControllerVN, cardinality single has to be specified and exported at the level of the *Cash Desk*. At the next level, when building the *Cash Desk Line* component, the middleware will be able to take care of transforming this VN into an appropriate VN at the composite level.

```
 <exportedVirtualNodes>
   <exportedVirtualNode name="cashBoxControllerVN">
    <composedFrom>
      <composingVirtualNode component="this" name="cashBoxControllerVN"/>
    </composedFrom>
   </exportedVirtualNode>
  </exportedVirtualNodes>
  <content class="CashDeskLine.CashDesk.CashBoxController"/>
  <virtual-node name="cashBoxControllerVN" cardinality="single"/>
```

A.3 Analysis

The models obtained in the previous section allow us to generate both parameterized and finite abstractions of the system behaviour, either for a single component, or for an arbitrary assembly. In principle, this allows for checking:

- simple "press-button" properties, like the absence of deadlocks, or the absence of certain types of events (predefined error events),
- more complex temporal properties expressed as temporal logic formulas, or in a formalism that can be translated into the temporal logic language understood by the model-checker,
- conformance between the implementation of a component (computed from the behaviour of its architecture) and its black-box specification, expressed as an equivalence or a preorder relation (as in [Černá 06]).

In this appendix we only consider verification performed on the finite abstraction of the model, and the verification is done by the Evaluator model-checker (from the CADP toolset). Both the parameterized case (using "infinite-state" engines), and the conformance checking are left for further work.

In the following pages, we give examples of verification of the CoCoME requirement scenarios, that we have performed with an old version of VCE (see Section 7.2) called CTTool.

A.3.1 System Verification

The JDC tool support itself being not yet available, we have conducted the analysis activities in this section using directly the capabilities of CTTool: it works by generating LOTOS code that implements a (synchronous) semantics of UML component diagrams and state-machines (including data-types), and passing this LOTOS code to the CADP verification toolset. CTTool itself includes a user interface that hides most of the verification engine complexity, and provides a number of menus for controlling the CADP functions.

There are many ways of encoding formulas. Some of them are very powerful as μ-calculus, but at the same time hardly usable by non-experts. So, having software engineer's expertise and JDC in mind, we propose to write formulas using automata. Transitions contain predicates with logic quantifiers and states can be marked as either acceptance or rejection. The automata may change to any state whose transition predicates are satisfied. If a final state is unreachable, the formula is false. Moreover, there are special predicates:

- `ANY` meaning that any label satisfies the predicate;
- `NOT(i), (i AND j), (i OR j)` meaning that any label but `i` given satisfies the expression, both `i` and `j` must satisfy the expression and either `i` or `j` may satisfy the expression respectively; and
- `ANYOTHER` meaning that all labels not satisfying other transitions from the state satisfies the predicate.

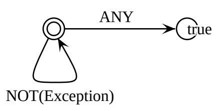
Then, formulas are readable and easy to write, and have language constructs compatible with both CTTool's state-machines and JDC's regular expressions.

A.3.1.1 Absence of Deadlocks

There are basic formulas that can be proved, the most common being the absence of deadlocks. In the case of our CTTool specification, this ends-up being trivially false because of two reasons:

- the specification of exceptions: in our specification, any time an exception is raised, we just block the system. So we may want to search for deadlocks that are not following an exception;
- the synchronous semantics of CTTool components: in CTTool, components are mono-threaded, and communications are synchronous. As a result, the system deadlocks due to race conditions over the EventBus. Concretely, events are not atomic within the EventBus, so a controller may trigger an event (and therefore block the EventBus) while the Application is running an ongoing sale. At this moment, if-ever the Application needs to access any of its controllers (through the EventBus), the system deadlocks.

To show this, we write a formula expressing that all deadlocks are the consequence of an exception, and to model-check this formula. More precisely we write the negation, i.e. that any transition is followed by some other transition as long as an exception has not been raised.

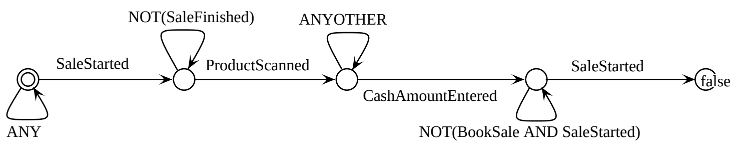


The answer we get when we evaluate the former formula is "false" (the formula does not stand). As a diagnostic we receive a trace in which the ScannerController triggers a ProductBarCodeScanEvent, blocking the EventBus. Meanwhile, the application is trying to synchronise with the EventBus for querying the price of the previously scanned product.

Note that these kind of scenarios would not be present in a real ProActive application because of the asynchronous method calls which buffers requests in the queues: we have more deadlocks in a synchronous implementation of the system than those we would have with ProActive. Nevertheless, using the CTTool specification we were able to prove some interesting scenarios, and to find some errors (or underspecifications) within the reference CoCoME specification.

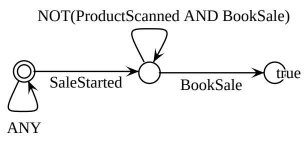
A.3.1.2 Main Sale Process

To illustrate the capabilities of our approach to verify more specific properties, we have checked (still in the synchronous model) some of the usage scenarios listed in the CoCoME specification. Our first scenario is defined in CoCoME's requirements as a trace in UML sequence diagram. We successfully verified that this trace is feasible in the state-space generated by CADP. An even more interesting scenario can be encoded as a negation of the following: a sale starts; before it finishes, valid products are scanned; the client pays with enough cash; the sale is not registered and a new sale starts. CADP's diagnostic strikes out that it is not possible to start a new sale before booking the former one.



A.3.1.3 Booking an Empty Sale

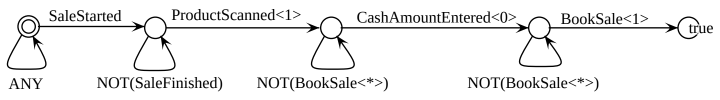
Although it may not be an error, it is strange that a system allows an empty sale to be booked. This trace was found when searching for the shortest path (by using Breadth-First Search algorithm) that books a sale. Alternatively, a software engineer may want to explicitly verify if this scenario is feasible with the automaton below. A sale is started; no products are scanned; the sale is booked.



A.3.1.4 Successful Sale with Insufficient Money

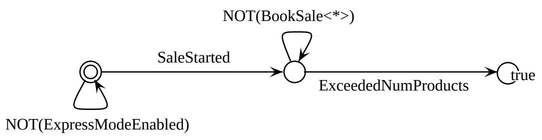
We found that it was possible to book a sale even if the client pays with insufficient money. The problem is reflected by the fact that there is no way of aborting a sale when paying with cash, and there is no verification whether the money fulfils expenses. Note that this issue was verified running the CoCoME reference implementation – which allowed the buyer to pay $0 – and going through the UML specification – which has nothing relative to it. In fact, the insufficient funds exception was only foreseen within a credit card payment. Nonetheless, in general, once a sale is started, it is not possible to abort, so the system must book the sale before proceding.

Note that for this scenario, the data abstraction plays an important role as it plays a role in the control-flow of the application. The amount that the client pays then is abstracted with two values: one with insufficient money and the other with sufficient money.



A.3.1.5 Safety of the Express Mode

A non-precised scenario was found. There is nothing within the CoCoME reference specification that states when a *Cash Desk* may switch from/to an express mode. In fact, the system ends-up in an inconsistent state if an express mode signal is triggered during an ongoing sale. This scenario can be found using the following property:



Conclusion

In this appendix we presented a large case-study for validating our approach. We aim at integrating at the level of the specification language the architecture specification and its implementation together with the black-box behaviour of the components in order to generate safe code by construction.

We specified the system using JDC and CTTool. From these specifications, we created by hand a GCM/ProActive implementation showing how the control code can be automatically generated from the JDC specification. From the same specification, completed by abstraction functions, we have shown how to generate models suitable for verification, in the form of parameterized networks of synchronised transition systems.

Finally, the CTTool specification was used to check for safety properties of the reference CoCoME specification scenarios. Because of our synchronous encoding, the EventBus component ended-up being a source of deadlocks. As a result of this verification activity, we found a number of interesting features, that we interpreted as bugs or under-specifications in the CoCoME official definition.

Limitations. There are number of features that are not considered in this approach: we do not consider any "performance" aspect of the specification (response time or other quality of service measures); neither do we try to fully specify the functional part of the code (data computation), as could be done in some proof-assistant based approaches.

Other limitations come from the early state of some of our software platform: our model generator is currently limited to the synchronous interpretation of the component systems (only the synchronous controllers are generated), so the features relying on asynchronous communication, and in particular the existence of functional deadlocks in Grid-based systems cannot be analysed. These developments are planned for the next version of the tool. Currently we have no direct support for the JDC language itself, so we rely on the existing CTTool platform to provide an alternative tool suite. We are working on an analysis platform with direct support for JDC, in which the CTTool diagrams will only be an alternative syntax for the specification. The JDC development platform will include an Eclipse plug-in, a Java code generation tool, a model generation tool, and a JDC formula compiler. With them, we hope to answer the software engineer's needs when developping distributed components.

Formal specification and verification of distributed component systems

Abstract

Components are self-contained building blocks. They communicate through well-defined interfaces, that set some kind of contract. This contract must guarantee the behavioural compatibility of bound interfaces. This is particularly true when components are distributed and communicate through asynchronous method calls.

This thesis addresses the behavioural specification of distributed components. We develop a formal framework that allows us to build behavioural models. After abstraction, these models are a suitable input for state-of-the-art verification tools. The main objective is to specify, to verify, and to generate safe distributed components.

To this aim, we develop a specification language close to Java. This language is built on top of our behavioural model, and provides a powerful high-level abstraction of the system. The benefits are twofold: (i) we can interface with verification tools, so we are able to verify various kinds of properties; and (ii), the specification is complete enough to generate code-skeletons defining the control part of the components. Finally, we validate our approach with a Point-Of-Sale case-study under the Common Component Model Example (CoCoME).

The specificities of the specification language proposed in this thesis are: to deal with hierarchical components that communicate by asynchronous method calls; to give the component behaviour as a set of services; and to provide semantics close to a programming language by dealing with abstractions of user-code.

Spécification formelle et vérification des systèmes des composants distribuées

Résumé

Les composants sont des blocs logiciels qui communiquent par des interfaces bien définies. Ces interfaces définissent un contrat avec l'environnement. Ce contrat doit garantir la compatibilité comportementale des interfaces. Cette compatibilité est en particulier importante quand des composants sont distribués et communiquent par des méthodes asynchrones.

Cette thèse se base sur les spécifications comportementales des composants distribués. Nous développons un cadre formel qui nous permet de construire des modèles comportementaux pour ces composants. Après une phase d'abstraction, ces modèles peuvent être utilisés en entrée pour des outils de vérification modernes. L'objectif principal est de spécifier, vérifier et au final de produire des composants distribués avec un comportement garanti.

Pour ce faire, nous développons une langage de spécifications proche de Java. Ce langage est établi sur notre modèle comportemental, et fournit une abstraction puissante de haut niveau du système. Les avantages sont les suivants: (i) nous pouvons nous connecter avec des outils de vérification: ainsi nous sommes capables de vérifier plusieurs sortes de propriétés ; et (ii), les spécifications sont assez complètes pour produire des squelettes de code de contrôle des composants. Finalement, nous validons notre approche avec un cas d'étude à l'aide d'un exemple commun de système à composants ("Common Component Model Example (CoCoME)").

Les particularités du langage proposé sont : traiter des composants hiérarchiques qui communiquent par des appels de méthodes asynchrones; donner le comportement d'un composant comme l'ensemble de services; utiliser une sémantique proche d'un langage de programmation; et traiter des abstractions de code utilisateur.