Adding Spatial Information to Software Component Model - The Localization Effect

Ali Hassan

N° d'ordre : 2012telb0240

Sous le sceau de l'Université européenne de Bretagne

Télécom Bretagne

En habilitation conjointe avec l'Université de Rennes 1

Ecole Doctorale – MATISSE

Adding Spatial Information to Software Component Model - The Localization Effect

---

Thèse de Doctorat

Mention : Informatique

Présentée par **Ali Hassan**

Département : Informatique

Laboratoire : IRISA

Directeur de thèse : Antoine Beugnard

Soutenue le 24 septembre 2012

Jury :

M. Christophe Dony, Professeur, Université de Montpellier II (Rapporteur)
M. Lionel Seinturier, Professeur, Université de Lille 1 (Rapporteur)
M. Antoine Beugnard, Professeur, Télécom-Bretagne (Directeur de thèse)
M. Olivier Barais, maître de conférence, Université de Rennes 1 (Examinateur)
M. Serge Garlatti, Professeur, Télécom-Bretagne (Examinateur)
M. Vivien Quéma, Professeur, INP/ENSIMAG + fonction et lieu d'exercice (Examinateur)

Acknowledgements

First I would like to thank France for giving such crystal example. France is a living example that we can be rich yet compassionate, hard-workers yet social, modern yet deep-rooted, and finally, successful yet modest.

Dr. Antoine Beugnard is an exceptional person, researcher, and team leader. I was lucky to have the opportunity to work under his supervision for three-very-rich years. He made exceptional effort to explain many things to me. He succeeded in that most of the time. One failure was 'diplomacy', where I think I have no chance at all. Thank you Dr. Beugnard.

My deep thanks are to Dr. Annie Gravey. She welcomed me in the department and she made all necessary decisions for me to have fair and professional opportunity. Thank you Dr. Gravey.

I deeply appreciate the effort the committee members made to evaluate this dissertation. Their comments, suggestions, and discussion are valuable and important.

I want also to thank my dear friends Ausama and Nawar for their sincere friendship.

After all of that I am surprised that majority of people do not see that we live in a close-to-ideal world.

ali - 2012

Abstract

Highly distributed environments (HDEs) are deployment environments that include powerful and robust machines in addition to resource-constrained and mobile devices such as laptops, personal digital assistants (or PDAs), smartphones, GPS devices, sensors, etc. Developing software for HDEs is fundamentally different from the software development for central systems and stable distributed systems.

This argument is discussed deeply and in-details throughout this dissertation. HDE applications are challenged by two problems: unreliable networks, and heterogeneity of hardware and software. Both challenges need careful handling, where the system must continue functioning and delivering the expected QoS.

This dissertation is a direct response to the mentioned challenges of HDEs. The contribution of this dissertation is the cloud component model and its related formal language and tools. This is the general title. However, and to make this contribution clear, we prefer to present it in the following detailed form:

1. We propose a paradigm shift from distribution transparency to localization acknowledgment being the first class concern.

2. To achieve the above mentioned objective, we propose a novel component model called cloud component (CC).

3. In this dissertation we propose a new approach to assemble CCs using systematic methodology that maintains the properties of CC model.

4. Cloud component development process and cloud component based systems development process.

5. Location modeling and advanced localization for HDEs are the pivotal key in our contribution.

6. Formal language to model single CC, CC assembly, CC development process, and CC based systems.

7. We finally present our fully-developed supporting tools:

   the cloud component management system CCMS, and the Registry utility.

Résumé

Les Environnements Hautement Distribués (HDEs) sont des environnements de déploiement de logiciels qui incluent des machines très diverses comme de gros serveurs mais aussi des appareils mobiles avec des ressources limitées comme les ordinateurs portables, les assistants numériques personnels (ou PDA), les téléphones intelligents, les appareils GPS, des capteurs, etc. Le développement de logiciels pour les HDEs est fondamentalement différent du développement des systèmes centralisés ou des systèmes distribués fermés. Cet argument est discuté en détail tout au long de cette thèse. Les HDEs posent deux problèmes principaux : des réseaux non fiables et l'hétérogénéité des matériels et logiciels. Ces deux défis nécessitent un traitement minutieux pour permettre aux systèmes de continuer à fonctionner en fournissant la qualité de service (QoS) attendue aux utilisateurs.

Cette thèse est une réponse directe aux défis mentionnés des HDEs. La contribution de cette thèse est le modèle de composant Cloud Component (CC). Nous présentons cette contribution en proposant 7 axes : (1) Un changement de paradigme pour passer de la transparence de la distribution à la reconnaissance de la localisation comme une préoccupation de première classe. (2) Un modèle de composant nommé Cloud Component (CC) pour réaliser ce changement avec (3) une nouvelle approche pour assembler ces composants – pas de connexions distantes. (4) Un processus de développement des Cloud Components où (5) la modélisation et la prise en compte de la localisation, et donc de l'hétérogénéité, sont la clé de notre contribution. (6) Un langage formel pour décrire les composants, vérifier leurs assemblages et décrire leur processus de développement. Enfin, nous présentons (7) les outils développés : le Cloud Component Management System (CCMS) qui déploie les variantes logicielles adaptées aux cibles matérielles en s'appuyant sur des descriptions ontologiques des logiciels et de leur cible, l'annuaire (Registry) et le vérificateur d'assemblage.

Pour répondre aux défis posés par HDEs et maintenir la qualité de service prévue aux utilisateurs, nous défendons la thèse d'un « changement de paradigme » dans la façon dont le logiciel est conçu et mis en œuvre. Pour appliquer ce changement, notre contribution présente plusieurs aspects ; chacun forme une contribution partielle qui ne peut être isolée des autres et dont l'union forme un tout cohérent. Notre contribution couvre le processus complet de développement logiciel pour HDEs : des spécifications au déploiement et à la gestion de l'exécution.

Contents

List of Figures

List of Tables

Chapter 1

Résumé en Français

1.1 Introduction

Les codes de réponse rapide (QR code comme dans la figure 1.1) ont connu une grande popularité ces derniers temps. Grace à l'utilisation de lecteur de codes QR, il est possible d'accéder à un des informations pour les gammes de produits commerciaux ou pour des projets scientifiques. Cependant, quand un utilisateur de smartphone installe un logiciel lecteur de code QR sur son téléphone, il y a une probabilité que ce logiciel ne fonctionne pas correctement. Tous les lecteurs de codes QR disponibles exigent maintenant une caméra autofocus, alors que de nombreux smartphones ne sont équipés actuellement que de focale fixe. Comme résultat, le logiciel ne sera pas en mesure de scanner le code QR. Nous pensons que ce défaut n'est ni isolé, ni superficiel. Il est profond et fondamental. En outre, il s'étend à l'ensemble du spectre du développement de logiciels pour environnements hautement distribués - HDEs.

1.1.1 Défis des Environnements Hautement Distribués

L'émergence des appareils mobiles tels que les ordinateurs portables, ordinateurs portables, assistants numériques personnels (PDA) et les téléphones intelligents, ainsi que l'avènement de diverses solutions de réseaux sans fil rendent des calculs possibles n'importe où. Il est maintenant possible de réaliser des applications à la fois simples et complexes pour être déployées sur un ordinateur et un téléphone intelligent en même temps. Ces applications comprennent des applications multimédia, des cartes et des applications GPS, les applications de réservation des voyages, et d'autres encore. De tels scénarios présentent plusieurs défis techniques: la compréhension profonde et suffisante des configurations logicielles existantes ou à venir; la mobilité du matériel; l'évolutivité à de grandes quantités de données et au nombre de dispositifs; et enfin l'hétérogénéité du logiciel s'exécutant sur chaque dispositif. En outre, le logiciel doit souvent s'exécuter sur des petits appareils, caractérisés par des ressources limitées telles que la taille de l'écran, la puissance limitée, la faible bande passante du réseau, un CPU lent, une mémoire limitée, et une connectivité faible [7]. Nous appelons ces environnements des environnements hautement distribués (HDEs) comme dans la figure 1.2. Les HDEs contiennent toujours de puissantes et robustes machines, mais ils sont aussi composés d'appareils mobiles comme les ordinateurs portables, les assistants numériques personnels (ou PDA), les smart-téléphones, les GPS, des capteurs, etc. Le développement de logiciels pour HDEs est fondamentalement différent du développement pour les systèmes centraux et distribués stables [14] et [3] (voir section 2.5). Cet argument est discuté en profondeur et dans les détails tout au long de cette thèse.

Des chercheurs en génie logiciel et les praticiens ont combattu avec succès la

Figure 1.1: QR code pour l'URL de la page anglaise mobile Wikipedia principale. (Un article de Wikipédia).

complexité croissante du développement des logiciels pour les systèmes centraux et systèmes distribués stables en employant les principes de la transparence de la distribution. Dans la transparence de la distribution, une couche *middleware* est prévue pour manipuler et à cacher toutes les communications à distance (un appel distant apparaît comme un appel local). En outre, la transparence de la distribution masque de nombreuses distinctions entre les périphériques tels que l'architecture du processeur et des systèmes d'exploitation en utilisant des couches logicielles telles que la machine virtuelle Java par exemple.

Les applications HDE sont distribués à grande échelle, mobiles et déployées sur un large éventail de matériel, et donc très dépendantes de l'environnement de déploiement sous-jacent. Malheureusement, dans le réseau les échecs de connectivité ne sont pas rares: les appareils mobiles font face à de fréquentes et imprévisibles pertes de connectivité [15]. En conséquence, l'hypothèse d'avoir une connectivité stable n'est plus valide.

Plus grave, l'hétérogénéité des dispositifs est masquée pour les HDEs. La raison en est simple: les différences entre les appareils sont beaucoup plus fondamentales que dans les environnements distribués stables. Les différences entre

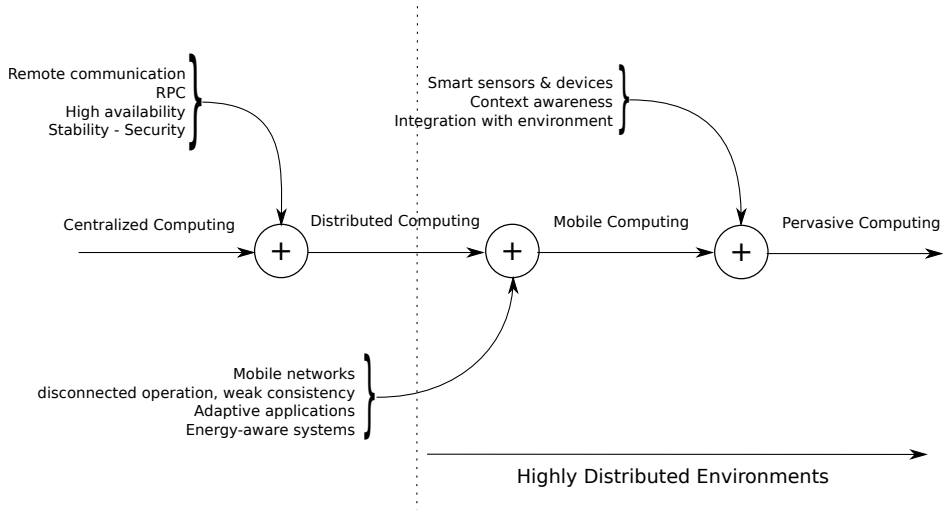


Figure 1.2: Environnements hautement distribués (HDEs) [1].

les smartphones, les tablettes et les ordinateurs portables sont claires. Toute tentative visant à masquer ces différences conduit soit à des répercussions négatives sur la qualité de service soit à l'échec de l'application. L'argument est encore correct entre les smartphones seuls.

Pour ces deux raisons, les applications HDE sont remises en cause par deux problèmes: les réseaux non fiables et l'hétérogénéité des matériels et logiciels. Les deux défis nécessitent une attention minutieuse pour que le système continue à fonctionner et garantir la QoS attendue. Fonctionnement en mode déconnecté provoque un logiciel s'exécutant sur chaque appareil temporairement ses activités indépendamment de d'autres périphériques réseau. Cela présente un grand défi pour les systèmes logiciels qui sont fortement tributaires de la connectivité réseau, parce que chaque sous-système local est généralement tributaire de la disponibilité de ressources non-locales. Le manque d'accès à une ressource distante peut mettre un terme à un sous-système particulier ou même rendre le

système entier inutilisable [7]. L'autre défi est pas moins sévère. Si l'application est déployée sur un hôte spécifique sans une évaluation adéquate de la compatibilité matériel / logiciel, il y a une forte probabilité que l'application fournisse des résultats médiocres.

Les approches actuelles de développement de logiciels partagent un objectif commun: faire croire à la répartition transparente à la fois au programmeur d'application et aux utilisateurs. Toutefois, en cachant la distribution, ces approches ne tiennent pas compte des aspects liée à des déconnexions et des erreurs liées. En général, les applications distribuées sont conçues de la même manière qu'une application centralisée [16]. réussir un développement de logiciels pour les HDEs doit traiter les défis suivants [16]:

- Les ressources limitées: de nombreux équipements tels que les PDA et les smartphones ont des ressources qui sont limitées, tandis que les technologies actuelles et les modèles implicitement s'attendent à des ressources que l'on trouvent habituellement dans un bureau moyen.
- La connectivité réseau : les modèles actuels de développement de logiciels sont compatibles avec les environnements avec une connectivité constante, une bande passante élevée, et une faible latence pour l'acheminement des appels distants. Les HDEs sont loin de satisfaire de telles hypothèses. Par conséquent, des références à des appels à distance sont souvent impossibles.
- La complexité: les modèles actuels de développement de logiciels sont basés sur une architecture client-serveur où le nombre et la localisation des clients et les serveurs sont entièrement gérés. Dans les HDEs, la mobilité des équipements et de la nature sporadique des connexions entre les machines produit des changements complexes de la topologie du réseau.

> Les systèmes distribués considérés par les technologies actuelles ne tiennent pas compte de telles structures.

Nous concluons cette section par deux déclarations faites par deux chercheurs dans le domaine. Le premier, Malek et al. [3] a remarqué:

> La transparence (cacher la distribution, à savoir, emplacement, et l'interaction des objets distribués) est considérée comme un droit fondamental de l'ingénierie des systèmes logiciels distribués, cependant, ce même concept, la transparence de distribution, a été démontré souffrir de lacunes majeures lorsqu'elles sont appliquées largement dans HDEs.

Deuxièmement, l'argument avancé par Guerraoui [14].

> La transparence à la distribution est impossible à réaliser dans la pratique. Précisément à cause de cette impossibilité, il est dangereux de donner l'illusion de la transparence.

1.1.2 Problématique

À partir d'une spécification de l'application C avec les environnements de déploiement attendus L. Procédez comme suit:

1. Mettre en place un système S qui est conforme à C et s'exécute sur L.
2. Déployer et exécuter S sur L', où L' est une variation de L.

1.1.2.1 Remarque Une

L dans la définition ci-dessus est une collection d'environnements hautement distribués.

Si L n'est pas un environnement distribué (c'est à dire qu'il s'agit d'un système centralisé) alors notre méthode proposée n'est pas applicable [1].

Si L est un environnement stable distribués alors la méthode proposée n'est pas applicable [2].

Par la 'collection', nous dire qu'il est possible pour L d'inclure plusieurs scénarios de déploiement.

1.1.2.2 Remarque Deux

L' pourrait être exactement L. Ce n'est généralement pas le cas. Le délai entre la définition de L et la définition L' pourrait être de mois voire d'années. Mais, les détails techniques évoluant rapidement, nous nous attendons à un changement rapide.

1.2 Etat de l'Art

Comme mentionné dans la section d'introduction, le problème HDE est décrit dans la littérature au cours des dernières années. Nous ne sommes ni le premier à discuter de ce problème, ni ne sommes le premier à proposer des solutions pour construire des applications pour ces environnements. Dans cette section, nous allons discuter de deux projets qui ont proposé des solutions pour le développement de logiciels HDE.

1. En fait, elle est toujours applicable, cependant, le mérite principal du modèle n'est pas utilisé efficacement.
2. même commentaire que ci-dessus.

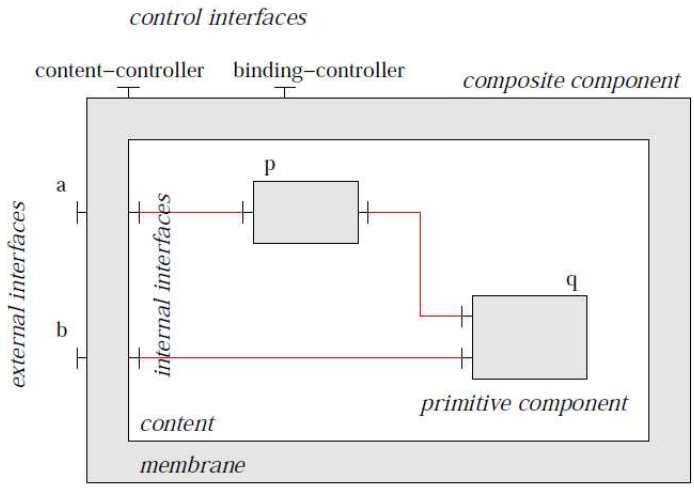


Figure 1.3: Un composant Fractale normal (de [2]).

1.2.1 L'approche Cubik

Didier Hoareau *et al.* a parlé des défis de HDEs [2, 16, 17]. Toutefois, leur solution a une portée différente de le nôtre. Tout d'abord, ils élargissent le modèle de composant déjà existant Fractal. Deuxièmement, ils ne modélisent et ne gérent pas la déconnexion et la connexion réseau.

L'approche Cubik utilise le concept *'proxy'*, et ajoute ce concept au modèle de composant Fractal. Un composant normal Fractale est présenté dans la figure 1.3. Dans cette figure, nous voyons un 'composant composite', que nous appellerons A, avec deux sous-composants internes (p et q) à l'intérieur. C'est une conception normale de Fractal.

Dans la figure 1.4 nous voyons le même composant conçu en utilisant l'approche Cubik. Dans cette figure, le composant A devient trois composants qui sont déployées sur trois dispositifs différents: A_1 est déployé sur m_1, A_2 est déployé sur

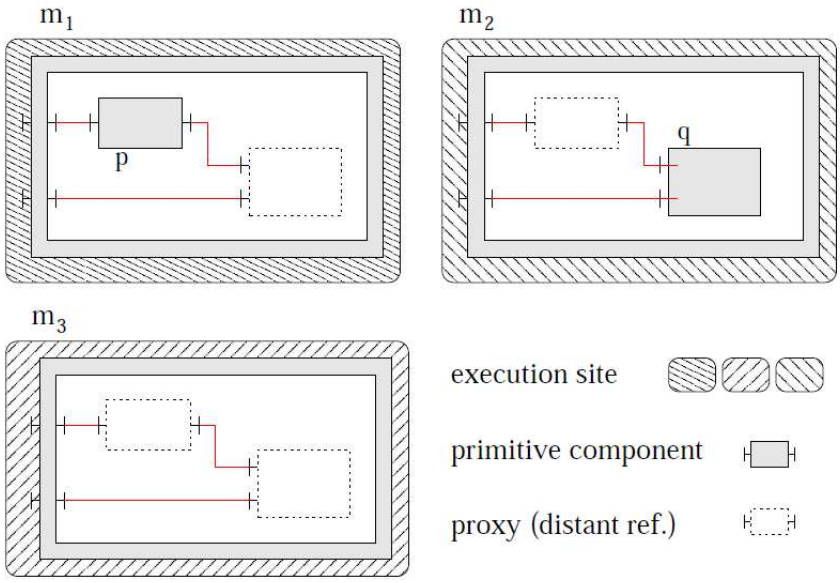


Figure 1.4: L'approche Cubik (de [2]).

m_2, et A_3 est déployé plus de m_3. Les trois composants: A_1, A_2, et A_3 ont les mêmes propriétés fonctionnelles (c'est-à-dire qu'ils offrent les mêmes fonctions et exigent les mêmes fonctions les uns des autres). Cependant, ils ont une conception interne différente. A_1 a un proxy de q et un composant réel p. A_2 a un proxy de p et un composant réel q. A_3 n'a pas de composants réels et deux mandataires de p et q.

1.2.2 L'approche de Sam Malek

Cette équipe propose un cadre et des outils pour soutenir une ingénierie logicielle complète du cycle de vie pour le développement d'applications HDE [3, 18, 19]. Voir la figure 1.5.

Ce cadre comprend les outils suivants: XTEAM, Desi, Prism MW. En outre,

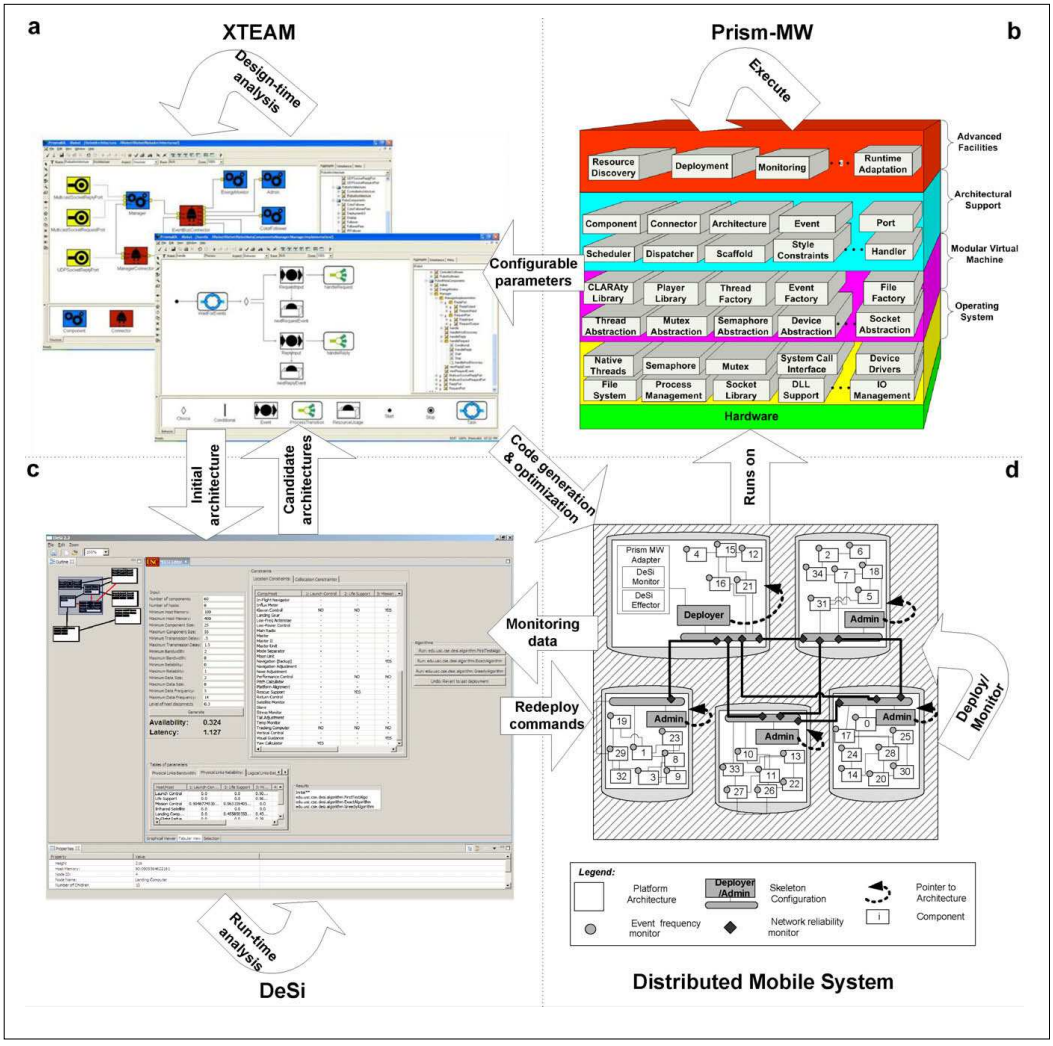


Figure 1.5: Le système proposé dans le [3].

ce cadre dépend fortement de l'ingénierie dirigée par les modèles dans la modélisation et la génération de code. Cela implique plusieurs modèles, des métamodèles, et des transformations de modèle. De plus, ce cadre adopte une analyse très complexe de la mobilité qui inclut: la mobilité du logiciel, la mobilité logique, la mobilité des composants et la mobilité du matériel.

1.3 Contribution

Cette thèse est une réponse directe aux défis mentionnés des HDEs. La contribution de cette thèse est le modèle de composant "cloud component (CC)". Toutefois, nous détaillons les contributions ci-après:

1.3.1 Première contribution

Nous proposons un changement de paradigme de la *transparence de distribution* à la *reconnaissance de la localisation* comme étant une préoccupation première importance. En d'autres termes, nous ne cachons plus la localisation (pour l'abstraire), au contraire, nous reconnaissons tous les aspects liés à l'emplacement, y compris la spécification des dispositifs, les différentes caractéristiques des réseaux qu'ils utilisent, les fonctions de sécurité, et toutes les caractéristiques liées à la mise en place de l'environnement. Nous discutons les HDEs et ce changement de paradigme dans la section 6.

1.3.2 Deuxième contribution

Pour atteindre l'objectif mentionné ci-dessus, nous proposons dans la section 7.1 un modèle de composant intitulé *cloud component* (CC). Ce modèle

comprend l'environnement de déploiement prévu dans sa définition, c'est-à-dire nous élevons l'importance de l'environnement de déploiement pour être égale à la fonctionnalité du composant. L'autre caractéristique importante de ce nouveau modèle est qu'il est *fondamentalement distribué*. Un simple CC est généralement réparti sur de nombreux hôtes distants, la spécification de ces hôtes sont considérée et fondamentalement reconnue au cours du processus de développement des CC, et tous les aspects liés à la communication, la coordination, et la qualité de services sont déplacées à l'interieur de la frontière de la CC.

1.3.3 Troisième contribution

Un composant logiciel peut être considéré comme l'unité d'assemblage [3]. Cela est vrai pour tous les modèles de composants, y compris le modèle *cloud component*. Dans cette thèse, nous proposons une nouvelle approche pour assembler des CC en utilisant une méthodologie systématique qui maintient la propriétés du modèle CC. Nous proposons une démarche pour construire de grands systèmes utilisant des CC en tant que blocs de construction. En outre, nous présentons une technique pour vérifier automatiquement la validité de cette assemblage. Assemblage de composants et de vérification de l'assemblage sont présentés dans le chapitre 7.2 .

1.3.4 Quatrième contribution

Nous proposons un processus de développement des CC. Dans cette contribution, deux facteurs ont été considérés comme pivot. Le premier facteur est la

3. Dans cette thèse, nous préférons utiliser le mot *assemblage* plutôt que *composition* puisque la sortie de cette opération (assemblage) n'est pas un composant logiciel.

pertinence de notre processus de développement logiciel qui doit être un processus conforme aux processus bien communément acceptés. Le deuxième facteur est la compatibilité entre ce processus de développement et les applications HDE. Notre processus de développement est discuté dans la section 7.3.

1.3.5 Cinquième contribution

L'utilisation des lieux et de la localisation pour les HDEs sont la clé de notre contribution. Pour y parvenir, nous proposons une ontologie de modélisation basée sur l'environnement de déploiement. Cette ontologie sert de base à la vérification de la compatibilité logiciels/hardware de notre approche. Voir la figure 1.6. Ces sujets sont abordés dans la section 8.

1.3.6 Sixième contribution

Nous proposons une formalisation pour la modélisation des CC, d'un assemblage de CC, du processus de développement des CC et des systèmes à base de CC. Ce langage formel est présenté dans la section 7. Le modèle de *cloud component* et son assemblage sont présentés informellement avec une notation graphique et formellement avec une notation mathématique. La notation informelle permet d'accélérer la compréhension des concepts généraux alors que le notation formelle ouvre la porte à un large éventail de la travaux théorique sur des sujets tels que l'inférence de type de composant, les sous-types, etc, et fournit un langage précis pour décrire les détails. En outre, les approches formelles permettent au concepteur de produire des représentations lisibles par la machine où des outils automatisés peuvent vérifier les propriétés spécifiques au moment de la conception, qui à son tour, augmente le niveau de confiance

dans la justesse de la conception.

1.3.7 Septième contribution

Dans la section 9 nous présentons nos outils de soutien: le CCMS, un système de gestion des *cloud components*, et l'utilitaire *Registre*. Ces outils facilitent l'installation, le déploiement, les vérifications de compatibilité, et la gestion de l'exécution. Ces outils, ainsi que le vérificateur d'assemblage et le vérificateur de logiciel/hardware sont indispensables et ils font du développement en utilisant le modèle CC un processus facile et puissant.

1.3.8 Remarque

La contribution de cette thèse a plusieurs faces, mais ces faces sont en cohérence. Chacune de ces faces forme une contribution partielle, toutefois, chaque contribution partielle ne veut rien dire si on l'isole de la proposition globale. En outre, le mérite de la proposition globale ne peut être saisi par la lecture d'un apport partiel. Le mérite de la proposition n'est évident que si toutes les parties de ce travail sont organisés ensemble.

1.4 Outils, implémentation et validation

Pour soutenir notre proposition théorique nous avons pleinement implanté ce qui suit:

1. Le vérificateur d'assemblage de *cloud components*. Cet outil est utilisé pendant la conception. Il vérifie si le système à base de CC respecte le

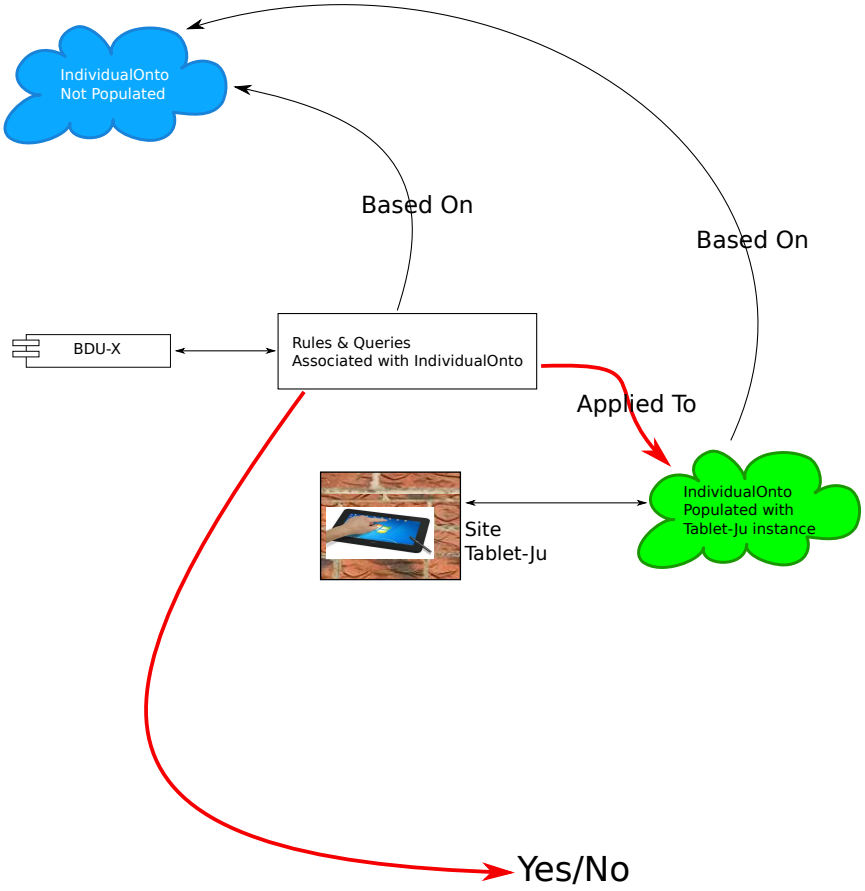


Figure 1.6: Logiciel/hardware vérificateur de compatibilité.

modèle d'assemblage CC - en particulier les cardinalités sur les interfaces.

2. Le vérificateur basé sur une ontologie logiciel / hardware. Cet outil est appelé avant le déploiement de toute partie de l'application basée sur les CC. Cet outil vérifie si le logiciel est compatible avec le dispositif déploiement cible. Voir la figure 1.6.

3. Le Système de gestion des *cloud components* (CCMS) et l'utilitaire Registre. C'est le cœur du déploiement et de la gestion de l'environnement d'exécution. La tâche principale de CCMS est le déploiement automatique des applications basées sur des CC.

4. L'application multimédia. Pour valider notre proposition, nous avons complètement réalisée cette application à base de CC. En utilisant ce cas d'étude, nous avons montré comment une application basée sur des CC offre la QoS promise dans tous les cas et à des points de livraison divers. En outre, nous avons montré comment déployer automatiquement et gérer des applications basées sur CC en utilisant le CCMS et le Registre.

1.5 Conclusion

Jusqu'à présent, le développement de logiciels pour HDEs est ad-hoc. Chaque développeur de l'application tente de mettre en œuvre des techniques et des contrôles pour répondre aux spécifications sur ces environnement. Cependant, il n'y a pas de processus systématique, ni de modèle personnalisé qui peut être utilisé systématiquement pour produire des applications de haute qualité pour HDEs.

Dans ce travail, nous proposons un modèle de composant logiciel, le modèle de *cloud component*, pour combler cette lacune entre le besoins de développement

des logiciels actuels et les techniques d'ingénierie et les méthodes de développement logiciels disponibles. Le modèle de *cloud component* est basé sur un changement de paradigme de la transparence de la distribution à la localisation comme préoccupation de première classe. En d'autres termes, nous n'avons plus à cacher les emplacements, au contraire, nous reconnaissons tous les aspects liés à la localisation y compris la spécification des dispositifs, les caractéristiques des réseaux qu'ils utilisent, les spécifications de réseau différents, les caractéristiques de sécurité, et toutes les propriétés associées à l'environnement de déploiement. En outre, nous proposons une théorie d'assemblage afin de bâtir une base pour les CC. Une notation formelle est proposée pour les CC, les assemblages de CC, leur processus de développement. Cette notation formelle ouvre la porte à un large éventail de sujets théoriques, y compris l'inférence de type de composant, sous-types, etc. Cette approche formelle permet au concepteur de produire des représentations lisibles par la machine où les outils automatisés peuvent vérifier les propriétés spécifiques au moment de la conception, qui à leur tour, augmente le niveau de confiance dans la correction de la conception.

Une des principales valeurs du modèle CC est la prise en compte de la compatibilité entre le logiciel et matériel dans les HDEs. Il s'agit d'un défi majeur en sachant l'hétérogénéité de ces milieux. Notre modèle est le premier à utiliser une modélisation ontologique du matériel et des exigences des logiciels afin de vérifier leur comptabilité. Ce vérificateur, ainsi que le processus de développement proposé des CC sont les fondements de notre affirmation selon laquelle textit le modèle CC garantit la QoS attendue au point d'utilisation de l'utilisateur final. Nous avons soutenu ce modèle avec des outils pleinement mis en œuvre: vérificateur d'assemblage de CC, l'ontologie logiciel / matériel, le système de gestion des *cloud components* (CCMS) et du Registre. Enfin, nous avons pleinement

mis en œuvre un système CC avec une application multimédia, afin de valider notre proposition.

La contribution de cette thèse a plusieurs faces, mais ces faces sont cohérentes. Chacune de ces faces forme une contribution partielle, toutefois, chaque contribution ne veut rien dire si on l'isole de la proposition globale. Le mérite de la proposition globale ne peut être saisi par la lecture d'un apport partiel. Le mérite de la proposition n'est évident que si toutes les parties de ce travail sont étudiées ensemble.

Il pourrait être considéré comme une des limites de ce travail que nous n'avons pas de vérification dynamique dans notre modèle. Par exemple, nous ne décrivons pas la dynamique (temporelle) caractéristiques d'un rôle de CC, comme résultat, c'est la responsabilité du concepteur pour assurer le bon comportement dynamique des rôles lors de l'exécution. D'autre part, d'éviter de telles méthodes formelles est destiné à maintenir le modèle facile à utiliser par les ingénieurs logiciels moyenne.

Une autre limitation est la condition de l'existence d'une puissante machine qui exécute le CCMS et l'utilité du Registre. En outre, nous supposons un lien entre cette machine et tous les dispositifs de déploiement. Alors que la dernière condition pourrait être assouplie, nous n'avons pas étudié l'effet de la non-existence absolue d'une telle machine par rapport au modèle CC. Est-il encore utilisable? Et comment y parvenir? La majorité des applications ne nécessitent pas une telle condition extrême, cependant, il est encore intéressant d'étudier le potentiel du modèle CC dans une situation grave telle.

Part I

INTRODUCTION

Chapter 2

Introduction & Motivation

2.1 Quick Response Code - QR

Quick response code (QR code as in figure 2.1) has enjoyed great popularity recently. Using QR code reader, it is possible to access a large volume of information that ranges from commercial products to scientific posters. However, when a smartphone user installs QR code reader software on his/her phone, there is a probability that this software will not operate properly. All QR code readers available now require an auto-focus camera, while many smartphones currently are equipped with fixed-focus camera. As result, the software will not be able to scan the QR code. We think that this fault is neither isolated nor shallow. It is deep and fundamental. Moreover, it spans to the whole spectrum of software development for highly distributed environments - HDEs. This simple example is used to emphasize the fact that software development for HDEs is *different* from developing applications for stable distributed environments.

Figure 2.1: QR code for the URL of the English Wikipedia Mobile main page, `http://en.m.wikipedia.org`. (From Wikipedia).

2.2 Highly Distributed Environments Challenges

The emergence of mobile devices such as portable notebook computers, handheld personal digital assistants (PDAs), and smart phones, and the advent of various wireless networking solutions make computation possible anywhere. It is possible now to expect both simple and complex applications to be deployed over a cluster computer and a smart-phone at the same time. These applications include multimedia applications, maps and GPS applications, trip reservation applications, and more. Such scenarios present several technical challenges: deep and sufficient understanding of existing or prospective software configurations; mobility of hardware; scalability to large amounts of data and numbers of devices; and heterogeneity of the software executing on each device and across devices. Furthermore, software often must execute on 'small' devices, characterized by highly constrained resources such as small display size, limited power, low network bandwidth, slow CPU speed, limited memory, and unreliable connectivity [7]. We refer to such environments as highly distributed environments (HDEs) as an figure 2.2. HDEs still include powerful and robust machines but they are rather composed of resource-constrained and mobile devices such as

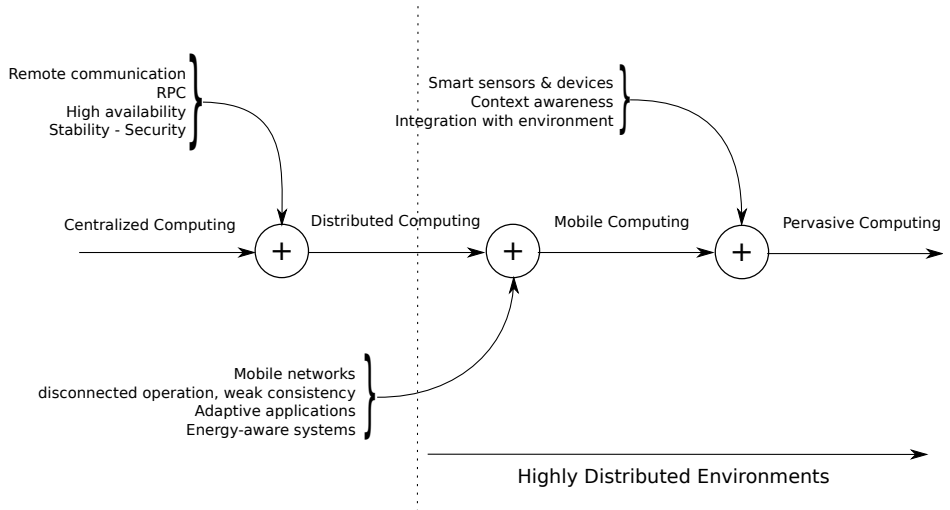


Figure 2.2: Highly distributed environments (HDEs) [1].

laptops, personal digital assistants (or PDAs), smart-phones, GPS devices, sensors, etc. Developing software for HDEs is fundamentally different from the software development for central systems and stable distributed systems [14] and [3] (section 2.5). This argument is discussed deeply and in-details throughout this dissertation.

Software engineering researchers and practitioners have successfully dealt with the increasing complexity of software development for central systems and stable distributed systems by employing the principles of distribution transparency. In distribution transparency a middleware layer is expected to handle and hide all remote communications (remote call appears as local call). In addition, distribution transparency masks many distinctions between devices such as processor architecture and operating systems by utilizing other software layers such as Java virtual machine as an example.

HDE applications are highly distributed, mobile, and deployed of a wide

range of hardware, and therefore highly dependent on the underlying deployment environment. Unfortunately, network connectivity failures are not rare: mobile devices face frequent and unpredictable connectivity losses [15]. As result, the assumption of having stable connectivity is no more valid.

More seriously, the heterogeneity of devices is unmaskable for HDEs. The reason is simple: the differences between devices are far more fundamental than in stable distributed environments. The differences between smartphones, tablet PCs, and laptops are clear. Any attempt to mask these differences will lead to either negative impact on the QoS or to the failure of the application. If we want to be more precise, the same argument is correct among smartphones alone.

For these two reasons, HDE applications are challenged by two problems: unreliable networks, and heterogeneity of hardware and software. Both challenges need careful handling, where the system must continue functioning and delivering the expected QoS. Disconnected operation forces systems executing on each individual device to temporarily operate independently from other network hosts. This presents a major challenge for the software systems that are highly dependent on network connectivity, because each local subsystem is usually dependent on the availability of non-local resources. Lack of access to a remote resource can halt a particular subsystem or even make the entire system unusable [7]. The other challenge is not less sever. If the application is deployed on a specific host without proper assessment of the software/hardware compatibility, there is a high probability that the application will perform poorly.

Current software development approaches share a common goal: making aspects related to the distribution transparent to both the application programmer and the users. However, hiding distribution, these approaches do not incorpo-

rate aspects related to disconnections and related errors. In general, distributed applications are designed to same way as a centralized application [16]. Successful software development for HDEs must sufficiently handle the following challenges [16]:

- Limited resources: many equipments such as PDAs and smartphones has resources that are limited while the current technologies and models implicitly expect resources that usually found in an average desktop.

- Network connectivity: current software development models are compatible with environments with constant connectivity, high bandwidth, and low latency for the routing of remote calls. HDEs are far from satisfying such assumptions. Therefore, references to remote calls are often invalidated.

- Complexity: current software development models are based on a client-server architecture where the number and location of clients and servers are fully managed. In HDEs, the mobility of equipment and the sporadic nature of connections between machines result in the creation of network topology that is changing and complex. Distributed systems considered by current technologies [1] does not consider such structures.

2.3 Software Engineering

The goal of this section is to define software engineering, describe some typical software life cycle phases, and artifacts used and produced in them. We think this is important for two reasons. First reason is related to this dissertation. This dissertation involves several topics such as software components,

1. Please read section 5.1 for a list and discussion of current models and technologies.

formal languages, ontology design, etc. However, this is a software engineering dissertation. Second reason is related to software engineering itself. This domain is relatively new domain. In spit of that, software engineering community and computer science community have formulated a set of core knowledge in this domain. Core knowledge include: theory, tools, programming languages, processes, etc. Moreover, this core knowledge is well documented through institutions and societies such as the Institute of Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM). This core software engineering knowledge is what ranked-universities include in the computer science curriculum. It is very important to draw a line between this software engineering core knowledge, and other software-engineering related attempts, activities, processes, languages, tools, etc. For example, Java is taught in all computer science departments, while Orwell is not. There is a fundamentally important value in being 'close' to the core software engineering knowledge. It is a criteria of judging any contribution in this domain. If this contribution is close to the core knowledge software engineering, it is positive point. If not, it is negative point.

We will proceed in this section to provide a classical, widely accepted, definition of software engineering along with a brief definition of the phases of software life cycle. The most commonly accepted definition of software engineering is the one given in the IEEE Standard Glossary for Software Engineering [20], where software engineering is defined as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software." Based on [20, 21] software life cycle phases are:

- Analysis phase determines what has to be done in a software system. After determining what kind of software is needed to be developed, the requirements phase is the first and the most important step. In this phase, the requirements for a software product are defined and documented. This is usually done in collaboration with end-users and domain experts. A complete specifications is the result of this phase.
- Design phase defines detailed designs for application domain, architecture, software components, interfaces, and data. All the design should be verified and validated to satisfy requirements. The more formal designs are defined, the less potential errors will be in the subsequent phases.
- Implementation phase creates a software product starting from the design documentation and models. This phase also debugs and documents the software product. In this phase programming languages (C, Java, etc) are used to encode specified designs, and testing techniques are utilized to find and correct any potential bugs. Besides eliminating software bugs, it is also important to be able to check whether implementations are fully valid with respect to the models.
- Integration phase (installation phase or deployment phase) The period of time in the software life cycle during which a software product is integrated into its operational environment and tested in this environment to ensure that it performs as required.
- Operation and maintenance phase. The software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements.
- Retirement phase (optional).

Through out this dissertation we will use the above defined software life cycle [2] as a standard. We will consider it a privilege to use it as it is unless there is a genuine need for modification and customization.

2.4 Software Components

Software engineering researchers and practitioners have successfully dealt with the increasing complexity of distributed systems by employing the principles of software architecture. In software architecture software systems are modeled using major components (loci [3] of computation), communication (loci of component interaction), and their configurations (also referred to as topologies) [7, 22, 23].

We think that using software components is fundamental to counter the increasing complexity of distributed systems [7,16,24]. Examples of industry component models are EJB [25], CORBA Component Model [26], and OSGi [27]. Fractal [28] and SOFA [12, 29, 30] are software component models proposed by academia. Unfortunately, this success in stable distributed systems is not possible in highly distributed environments HDEs. These component models were not designed to face the challenges HDEs raise. For example, EJB and SOFA have no support for highly restricted devices, and Fractal does not consider disconnected operations at all [2, 16, 17]. In other words, all component models in academia and industry successfully faced challenges in distributed environments, such as the complexity that results from the application size and distribution,

2. Software development process, software life cycle, and software development cycle are used interchangeably in this dissertation.

3. loci is the plural of Latin locus for place or location. In our context it is used in the semantic of 'collection'.

however, these component models face *the exact same limitations when it comes to highly distributed environments HDEs.*

For the rest of this dissertation, when we say 'the limitations of current approaches' we mean both 'current component-based approaches' and 'current approaches that are not based on components'. An example of the first one is SOFA. Service oriented architectures (SOA) [31–34] and remote procedure call (RPC) [35] are examples for the second one.

Software components have two important and strong features [13, 24]:

- Software component reuse. This point is concerned by reducing the cost of software development and minimizing time to market. These considerations are always important as software engineering is a highly competitive domain.
- Building large systems from smaller components. This point is concerned by the feasibility of software development. Without dividing large systems into orthogonal sub-systems, the development is rather "impossible".

In this dissertation we concentrate on the second feature. Software component reuse is out of the scope of this work. We think that the development of dependable software for HDEs that delivers the expected QoS at the user endpoint is currently a priority. This is correct even if the development process is long and costly. This work is not about reusing third party components, nor reducing time of development by implementing any automatic code generation, neither partial nor complete. This work is an attempt to reach the same success software development for distributed environments reached in terms of dependability, QoS, and systematic development, but now for HDEs.

2.5 Back to HDEs Challenges

As Malek et al. [3] have noticed "transparency (i.e. hiding distribution, location, and interaction of distributed objects) is considered a fundamental concept of engineering distributed software systems, as it allows for the management of complexity associated with the development of such systems". This is usually achieved through the utilization of a middleware layer that has as a main function (among others) to make remote calls appear as local calls. That is correct for stable distributed systems, however, this same concept, distribution transparency, has been shown to suffer from major shortcomings when applied extensively in HDEs [3].

Similar argument has been proposed by Guerraoui [14]. In this paper, a clear distinction has been made between programming for centralized systems and programming for distributed systems. This argument is much stronger than our argument since we accepted distribution transparency in stable networks, while Guerraoui does not.

> *"Distribution transparency is impossible to achieve in practice. Precisely because of that impossibility, it is dangerous to provide the illusion of transparency."*

The author continues: "One might argue that the generation of static stubs and skeletons together with optimized serialization techniques might lead to very good performance over a fast transmission network and indeed make a remote invocation look like a local one. This might indeed be true in a LAN under the very strong assumption that no process, machine, or communication failure

occurs." The solution from his point of view is "Distribution Awareness".

That leaves us in the following situation: there is excessive and increasing need to build complex mobile and pervasive systems for entertainment and professional uses. And at the same time, the fundamental engineering techniques available are inherited from stable distributed environments, and suffer from several drawbacks and weaknesses when utilized in these new environments (discussed in section 5.1). The only available answer currently is applying ad-hoc techniques to overcome these drawbacks and weaknesses.

2.6 Problem Statement

Having application specification C along with expected deployment environments L. Do the following:

1. Develop a system S that complies with C and L.
2. Deploy and run S over L', where L' is a variation of L.

We call (1) above: major-phase-one (or development major phase), and (2) above: major-phase-two (or deployment major phase).

2.6.1 Remark One

L in the definition above is a collection of highly distributed environments. If L is not a distributed environment (i.e. it is a centralized system) then our proposed method is not applicable [4]. If L is a stable distributed environment then our proposed method is not applicable [5]. By collection we mean it is possible

4. In fact it is still applicable, however, the main merit of the model is not effectively used.
5. Same as above comment.

for L to include several deployment scenarios.

2.6.2 Remark Two

L' could be exactly L. However, it is usually not. The time between the definition of L and the definition L' could be months or even years. And since the definitions of both are mostly technology details, then we expect fast change.

2.7 Example - The Multimedia Application

In this dissertation, we will use the Multimedia application as a case study to explain and validate several portions of this work. The purpose of Multimedia application is to be a single application to store, search, process, and play all multimedia files like pictures/images, music, and video. This application is expected to be deployed over a highly distributed platform such as the deployment environment in figure 2.3. Video streaming is one service of MULTIMEDIA application, and it is a very popular service for most PC, laptop, and smartphone users.

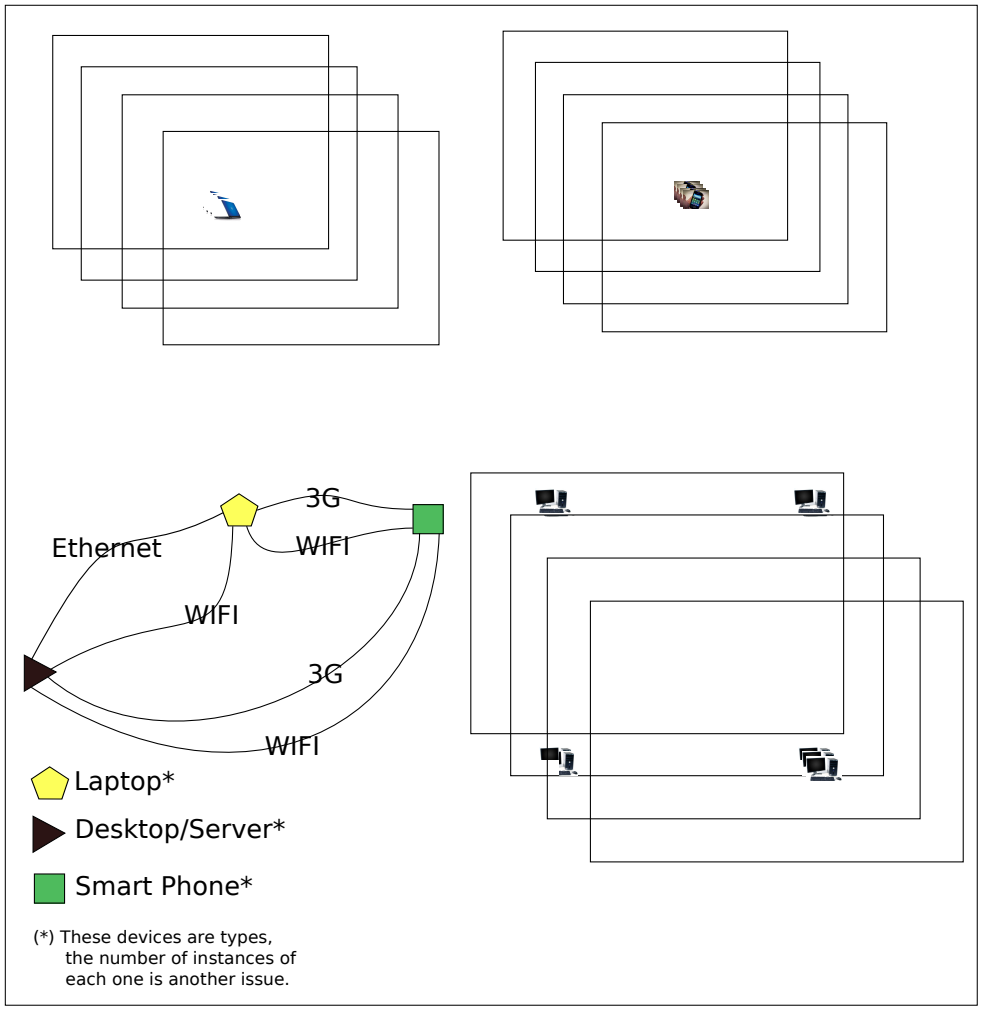


Figure 2.3: Highly distributed environment over which the Multimedia application will be deployed at runtime.

Chapter 3

Contribution

This dissertation is a direct response to the mentioned challenges of HDEs. The contribution of this dissertation is the cloud component model and its related formal language and tools. This is the general title. However, and to make this contribution clear, we prefer to present it in the following detailed form:

3.1 First Contribution

We propose a paradigm shift from *distribution transparency* to *localization acknowledgment* being the first class concern. In other words, we no more hide or abstract location, on the contrary, we acknowledge all aspects related to location including the specification of devices, the networking paradigms they use, the different network specifications available, security features, and all related characteristics of the deployment environment. We discuss HDEs and paradigm shift needed in section 6.

3.2 Second Contribution

To achieve the above mentioned objective, we propose in section 7.1 a novel component model called *cloud component* (CC). This model includes the expected deployment environment in its definition, i.e. we raise the importance of deployment environment to be equal to the functionality required from the component. The other important feature of this novel model is that it is *fundamentally distributed*. A single CC is usually distributed over many distant hosts, the specification of these hosts are considered and fundamentally acknowledged during the development process of this CC, and all aspects related to communication, coordination, and quality of service are migrated to be internal to the border of the CC.

3.3 Third Contribution

A software component can be thought of as unit of assembly [1]. This is true for all component models including cloud component model. In this dissertation we propose a new approach to assemble CCs using systematic methodology that maintains the properties of CC model. CC assembly is a tool to build large systems using CCs as building blocks. Moreover, we present a technique to automatically check the validity of this assembly. Cloud component assembly and checking are presented in chapter 7.2.

1. In this dissertation we prefer to use the word *assembly* rather than *composition* since the output of this operation (assembly) is not a software component.

3.4 Fourth Contribution

Cloud component development process and cloud component based systems development process. In this contribution two factors have been considered pivotal. The first factor is the relevance of our proposed software development process to the well-accepted software development process. The second factor is the compatibility between this development process and HDE applications. Our novel development process is discussed in section 7.3.

3.5 Fifth Contribution

Location modeling and advanced localization for HDEs are the pivotal key in our contribution. To achieve that we propose ontology based deployment environment modeling, ontology based cloud component requirements modeling, and ontology based software/hardware compatibility checker. These topics are discussed in section 8.

3.6 Sixth Contribution

Formal language to model single CC, CC assembly, CC development process, and CC based systems. This formal language is introduced in section 7. The cloud component model and CC assembly are presented informally and formally with a mathematical notation. The informal notation allows for faster comprehension of the general concepts. While the formal notation opens the door for a wide range of theoretical topics including component type inference, subtypes, etc, and provides a precise language to describe details. In addition,

formal methods allow the designer to produce machine readable designs where automated tools can verify specific properties at design time, which in turn, increases the level of confidence in the correctness of design.

3.7 Seventh Contribution

In section 9 we present our fully-developed supporting tools: the cloud component management system CCMS, and the Registry utility. These tools make installation, deployment, compatibility checks, and runtime management fully automatic. These tools, along with the assembly checker and the software/hardware checker are indispensable and they practically make the development using CC model an easy and powerful development.

3.8 Remark

The contribution in this thesis has several faces, but still, these faces are cohesive. Each of these faces form a partial contribution, however, this partial contribution does not mean anything if isolated from the overall proposal. Moreover, the merit of the overall proposal can not be grasped by reading one partial contribution. The merit of the proposal is evident only if all parts of this work are cohesively organized.

3.9 Back to the Quick Response Code

The challenge described in section 2.1 can be naturally handled using our proposal: the cloud component model, as in figure 3.1. For simplification, we

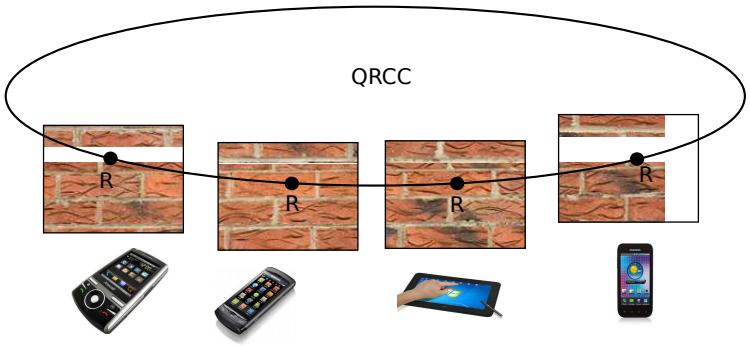


Figure 3.1: QR application as a cloud-component-based system.

show this application as a single cloud component `QRCC`. The role `R` is instantiated on several devices. The role `R` is responsible for the user interface, scanning the QR image, and contacting the QR-database (it is part of the `QRCC` that do not appear in the figure) to provide the user with the information. Before the instantiation (deployment) of `R`, the CCMS (the tool responsible of the deployment of all CC-based systems) will check the specifications of the deployment device. In section 2.1 we mentioned the camera problem. Now, the CCMS will check if the camera is auto-focus before deployment. If yes, the deployment of the `R` instance proceeds. If no, the CCMS checks if there is implementation variant of `R` for this case. If yes, the right implementation variant is deployed, if no, the deployment stops with an explanation message. In all cases the QoS of the application is respected at the user endpoint.

Part II

STATE OF ART

In this part we will attempt to discuss related research and industry work. The following considerations are adopted while organizing this part.

First we do not claim this is extensive list of related work, rather, it is the best of our knowledge. More important, it is high-quality and up-to-date collection of research and technology. Second, this is not a 'survey' part where we list and discuss work in specific field. Such survey is out of the scope of this dissertation. This part is a list and discussion of research tracks and technologies *under the scope of our proposed theory.* That is why we have a 'discussion' section whenever there is a need to compare and/or comment about related work with respect to our proposal.

At any point where we do not provide sufficient presentation about a related topic, such as Enterprise JavaBeans, we invite the reader to check the provided references for comprehensive information. Also, whenever the reader finds a comparison with our proposal, such as CC formal modeling language, we invite him/her to check part III for detailed presentation of such feature. This approach might not be very common, however, we think it serves our hope to provide readable and organized manuscript.

Chapter 4

Ontologies & Logic

4.1 DL, OWL, and Protégé

4.1.1 DL and OWL

Description logics (DLs) [36–38] are a family of knowledge representation languages that can be used to represent the knowledge of an application domain in a structured and formally well-understood way. The suitability of DLs as ontology languages has been highlighted by their role as the foundation for several web ontology languages, including "Web Ontology Language: OWL" [39–41]. OWL has a syntax based on RDF Schema [42], but the basis for its design is the expressive *DL SHIQ* [43], and the developers have tried to find a good compromise between expressiveness and the complexity of reasoning. Although reasoning in SHIQ is decidable, it has a rather high worst-case complexity (exponential time). Nevertheless, highly optimized SHIQ reasoners such as FaCT++ [44], Racer [45] and Pellet [46] behave quite well in practice [10].

We would like to point out briefly some of the features of *DL SHIQ* that make this version DL expressive enough to be used as an ontology language. First, *DL SHIQ* provides number restrictions. Second, *DL SHIQ* allows the formulation of complex terminological axioms like "humans have human parents." Third, *DL SHIQ* also allows for inverse roles, transitive roles, and subroles. It has been argued that these features play a central role when building ontologies using OWL [47].

To show how decidability is affected by the expressive power of the DL variant, it is shown that SHIQ extended by symbolic number restrictions, i.e. replacing the explicit numbers n in number restrictions by variables α that stand for arbitrary nonnegative integers, has an undecidable satisfiability and subsumption problems [37,48–50]. This trade-off has led to a set of requirements that may seem incompatible: efficient reasoning support and convenience of expression for a language as powerful as a combination of RDF Schema with a full logic. Indeed, these requirements have prompted Web Ontology Working Group to define OWL as three different sub-languages, each of which is geared towards fulfilling different aspects of these incompatible full set of requirements:

- OWL Full: The entire language is called OWL Full, and uses all the OWL languages primitives. As a disadvantage, the language has become so powerful as to be undecidable, dashing any hope of guarantees on complete and efficient reasoning.
- OWL DL: In order to regain computational efficiency, OWL DL (short for: Description Logic) is a sub-language of OWL Full which restricts the way in which the constructors from OWL and RDF can be used. The advantage of this is that it permits efficient reasoning support. The disadvantage is that we loose full compatibility with RDF.

– OWL Lite: An even further restriction limits OWL DL to a subset of the language constructors. For example, OWL Lite excludes enumerated classes, disjointness statements and arbitrary cardinality (among others). The advantage of this is a language that is both easier to grasp (for users) and easier to implement (for tool builders). The disadvantage is of course a restricted expressiveness.

OWL uses the XML syntax of RDF. Unfortunately, and after a fairly short use, it will become clear that RDF/XML does not provide a very readable syntax [37].

4.1.2 Protégé

Protégé [1] [51] is an open source ontology editor. The Protégé platform supports modeling ontologies via the OWL editor [2]. In Protégé, ontologies can be exported into a variety of formats including RDF(S), OWL, and XML Schema. Protégé is based on Java. Moreover, it is extensible and provides an open environment that supports plug-ins integration. That makes it a flexible base for rapid prototyping and application development.

4.1.3 Discussion

In our work, and throughout this dissertation we used F-logic (section 8.2), OntoStudio, and OntoBroker (section 8.2.2). It is outside the scope of this dissertation to have a theoretical comparison between these two groups:

1. F-logic, OntoStudio, and OntoBroker. We will call this the first group.
2. Description logic, Web Ontology Language - OWL, and Protégé. We will

1. http://protege.stanford.edu/
2. Protégé also supports the Protégé-Frames.

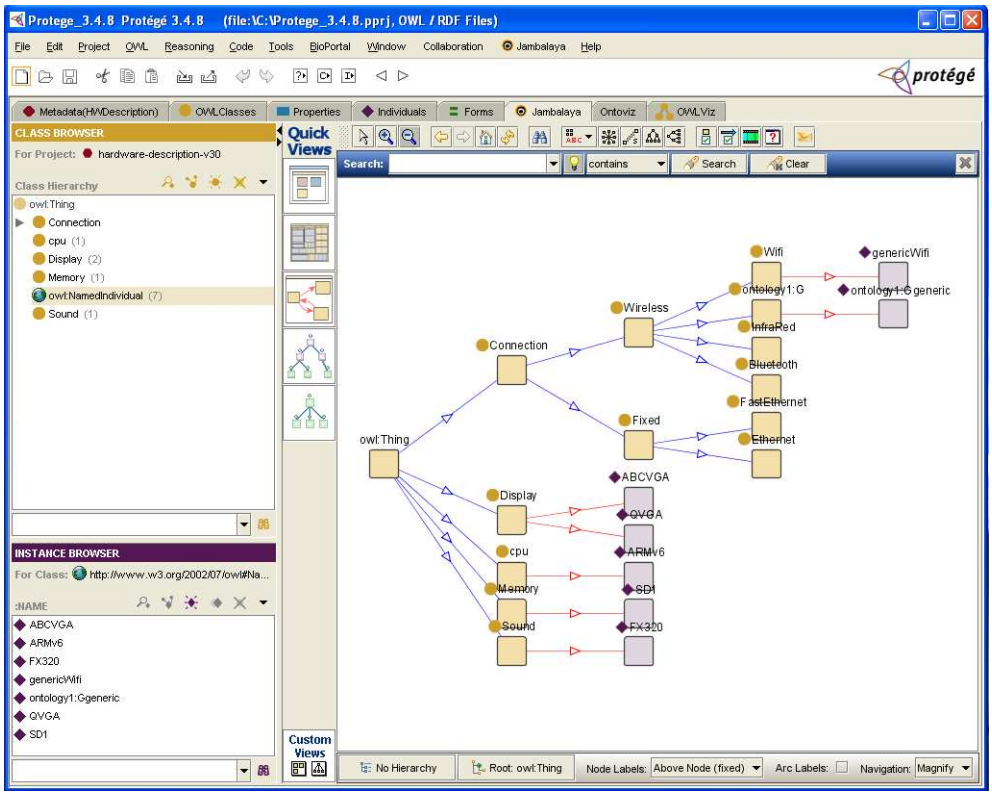


Figure 4.1: Protégé with Jambalaya visualization.

call this the second group.

However, we will mention our experience in using both of these two groups.

1. We used the second group for six months. It was potentially the tools and logic to build our ontology related modelling and checker.
2. Some of the points that the development team of Protégé thinks as positive and powerful points, we think they negatively affected our experience with this tool (and the underlying languages and syntax).
   (a) First: this tool, Protégé, is an open source and open environment. That means there are different groups of development. At some point, using a specific plug-in will require (depend) the existence of another plug-in or component. It happened several times that we could not satisfy all the dependencies, and we ended up unable to use the function we need. The lack of single point installation is, sometimes, a real barrier.
   (b) Second: Protégé has a large community. This is used to advocate that we could find help when we need. This is not as simple as people predict. In fact, we failed to find sufficient documentation about specific topics. Even if this documentation exists, it might be located on the 'web' where we do not have enough information how to reach it. When we arrived to the query point, we found this group (the second group) to become excessively distributed, with no sufficient documentation.
3. As mentioned above, the RDF syntax leads owl files to be unreadable. This might become a real limitations, especially if the developer wants to modify the ontology directly using a text editor.

4. Both groups have good graphical visualization for ontologies. Two integrated visualization functions in OntoStudio: Graph View and Ontology Visualizer. While Protégé uses OWLViz and Jambalaya.
5. As far as the underlying logic (formal language) we found both F-logic and description logic to be sufficient for our use. Both have enough documentation and advanced reasoners. However, we found the syntax of F-logic to be more convenient and readable.
6. Ontoprise successfully documented their products. Within few manuals and tutorials, the developer finds all necessary information locally on his/her hard disk (or as a printed book). Moreover, there is no third party components, nor third party documentation.

We found that the single point installation of Ontoprise products (OntoStudio and OntoBroker) to be the wining factor for the first group.

4.2 Ontology Support for Software Engineering

4.2.1 Review of Literature

While different methodologies consider different phases for software life cycle, we use the phases of software life cycle as defined in [20,21] (also discussed in section 2.3). These phases are: analysis phase, design phase, implementation phase, integration phase (deployment phase), maintenance phase, and retirement phase.

Ontologies have been considered as a solution to improve the state of the art in the area of requirement engineering (analysis phase). Breitman and Leite argue that ontologies should be sub-products of the requirement engineering phase [52]. An example is the DOGMA ontology engineering framework [53]. The following

list provides an adequate source of information regarding the incorporation of ontology in the analysis phase: [54–62].

The integration of ontologies into software design phase and activities can be seen through the following:

- Model Driven Engineering (MDE) and Ontologies [61, 63–66].
- Inconsistencies discovery at design time using model reasoning. The main finding of this track is that reasoning over UML class diagrams is EXPTIME-hard [67, 68].
- Enhancing model transformations through ontology support [69–71].

Ontology support to implementation phase and integration (deployment) phase are rather minimal. Examples on ontologies support for implementation through MDE can be found in [63, 72, 73]. The goal of MDE is to allow for automatically generating as much implementation code as possible. Unfortunately, the current state of the art indicates that many implementation details should still be done manually [10]. The most important contribution of ontologies to software integration is semantic web services. For example [74, 75].

4.2.2 Discussion

By reading the above mentioned list of literature, we find that the contribution of ontology to the current state of art software development life cycle is still in its initial stages. In other words, the software development is not, even minimally, dependent on ontologies currently. The second conclusion is that we find that there is an increasing interest in acknowledging non-traditional aspects in software development such as unified vocabulary and parameter semantics. We think that ontology is a good candidate in fulfilling many such related needs.

To be specific, ontology allows more machine-to-machine 'understanding', which leads to better automated tasks.

The most important conclusion in this section is that there is no ontology support [3] for the following tracks in software development process:

- Deployment environment modelling [4]. This is fundamental for software design and implementation phases in our proposed model.
- Software component requirements modelling. This is fundamental in implementation phase.
- Software/hardware checker. This is fundamental in deployment phase.

We think that our proposed ontology-based-localization is a contribution to the current state of art ontology-based-applications (please read section 8.5).

4.3 Automatic Question Answering & Automatic Ontology Building

4.3.1 Automatic Ontology Building

Automatic ontology building is divided into two fields: first is *machine learning* witch is related to building the ontology itself (before population). Second is *information extraction* which is concerned by populating an existing ontology automatically using available 'textual' documents. For detailed presentation of traditional (non-automatic) ontology building please read 8.3.1, and for detailed presentation of traditional (non-automatic) ontology population please read 8.3.2.

3. Based on our best knowledge.
4. Context awareness is supported by ontologies, however it is not deployment environment modelling. It will be discussed in section 5.2.

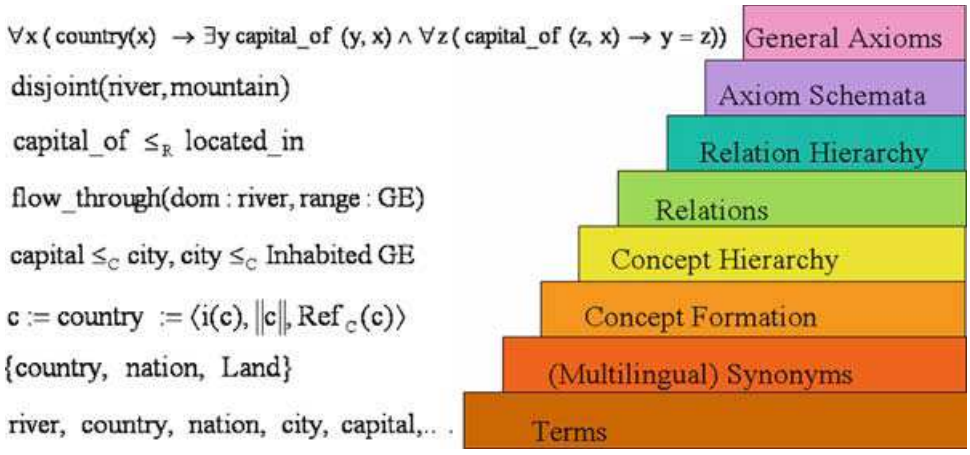


Figure 4.2: Ontology learning (copy from [4]).

Most ontology design patterns may be filled by manual work, but the use of machine learning mechanisms as a tool for suggesting ontological constructs is an increasingly important means.

The various tasks relevant in ontology learning have been previously organized in a layer diagram. This ontology learning layer was introduced in [4] and is shown in figure 4.2. It clearly focuses on learning the TBox [5] part of an ontology. The following literature list provides rich and up-to-date information on this research track: [76–92]. For a set of realized tools for automatic ontology learning please refer to: [55, 93–98].

Once the ontology is built, it is ready for population. Automatic population of an ontology has a bi-directional relationship with *Information Extraction (IE)*. IE can extract ontological information from documents. Conversely, ontologies can be exploited to interpret the textual document content for IE purposes. Among related literature we can list: [99–104].

5. TBox statements describe a set of concepts, concept hierarchy, and properties for these concepts. TBox and ABox constitute a full ontology (populated one).

4.3.2 Question Answering - QA

Using ontologies means performing a query over the ontology. Queries are written in SPARQL-like languages [6]. These languages are technical. Only programmers and experts can use such formal interfaces. Most users would prefer natural language QA. This research field attempts to propose methods for mapping natural-language questions into structured SPARQL queries (or similar). For example (from [105]):

"A big US city with two airports, one named after a World War II hero, one named after a World War II battle field."

This natural-language question can be expressed as in listing 4.1.

Listing 4.1: SPARQL query (from [105]).

```
Select ?c Where
{
?c hasType City .
?a1 hasType Airport . ?a2 hasType Airport .
?a1 locatedIn ?c . ?a2 locatedIn ?c .
?a1 namedAfter ?p . ?p hasType WarHero .
?a2 namedAfter ?b . ?b hasType BattleField .
}
```

State-of-art tools include YAGO-QA [105] and Watson [106] from IBM. For related techniques and algorithms, please read: [107–109].

6. SPARQL is a knowledge base query language. It is similar to the F-logic based query language from Ontoprise. We use the latter throughout this dissertation (please read section 8.3.3).

4.3.3 Discussion

Automatic ontology building and population, and natural language question answering are challenging and exciting research fields at the intersection of machine learning, data and text mining, natural language processing and knowledge representation. Fully automatic techniques for these tasks are not feasible currently and possibly will not be feasible in the future.

In our work we did all of these tasks manually. However, we felt an urging need for the above mentioned automation. Especially in ontology query. It is very difficult barrier to need an overall knowledge of an ontology to be able to write a query. Unfortunately, this is the current state of art, and this is what we did in our work as described in this dissertation. Please read chapter 8.

Chapter 5

Software Engineering & Highly Distributed Environments

5.1 Component Models and Connectors

This section is organized as follows: we start with a fast discussion of the insufficiency of current software component models to handle the challenges of HDEs in 5.1.1. This discussion will not include any specific component model nor specific feature of a specific model. After that we present a list of component models in section 5.1.2. In section 5.1.3 we discuss component border, Sofa model, and connectors. Fast comments about software/hardware compatibility and ease of use are in sections 5.1.4 and 5.1.5. We conclude this section by an attempt to position our model, CC model, among the other five well-known component models in 5.1.6.

5.1.1 Current Component Models Limitations

After analyzing several component technologies, we found that they follow a common paradigm. These component models rely on strong assumptions, and they emulate local call on top of distributed networks, and finally they consider any deviation from their implicit or explicit assumption as exceptions. All of these points are considered limitations with regards to HDEs as we explain in the following:

Rely on strong assumption A common way to distribute a component-based application consists of installing each component instance on a host; the distribution then refers to the fact that a component can make distant invocations to the services implemented by another component [2, 110]. This type of architecture usually relies on rather strong assumptions [2]:

1. The stability of the execution platforms (the component server is highly available - usually with backup recovery system).
2. All hosts have sufficient resources which include processing power, memory, and power supply.
3. The connectivity is reliable and has good characteristics (low latency, enough bandwidth, stable, no disconnections, etc.).

In general, an application designed using this architecture can not be installed and executed on deployment environments with hosts that are potentially volatile and limited in resources, especially when disconnected network operation and weak consistency of the characteristics of the connectivity are possible or frequent, which is the case in HDEs [2, 17].

Emulation The distributed component models mentioned above share a com-

mon goal: making aspects related to the distribution transparent to both the application programmer and the users. They hide distribution by making remote call appears to the caller as local call, but to some ad-hoc and limited exceptions (see next point). However, by hiding distribution, these mechanisms do not incorporate aspects related to disconnections, mobility, and all other complexities mentioned in the previous section [31]. In general, distributed applications are designed using the same techniques as a centralized application [16, 111].

Exceptions Most common component technologies were not originally designed for HDE. Therefore, they consider any deviation from the strong assumptions mentioned above such as inaccessibility of a machine or the unavailability of certain resources as exceptions. The treatment of the various changes that may occur within the network is usually done by adding code to adapt to these new events. This code will increase the complexity of applications [16, 111] with specific and ad-hoc extensions and poor methodological guidelines.

Typical HDE applications are highly distributed, decentralized, and mobile. Therefore, they are *highly dependent on the underlying network* [7, 112, 113]. We believe that the successful paradigm *'distribution transparency'* in stable distributed networks and current component models is no more dependable in highly distributed environments HDE. There is fundamental need to move from *hiding* the details of the underlying network and devices into *acknowledging* all of these aspects and details. It is possible to achieve that by introducing the concept *'location'*. We call this a *paradigm shift from 'distribution transparency' to 'localization acknowledgment'*. Please read section 6.

5.1.2 Survey of Component Models

We would like to provide an overview of current component models and their main characteristics. It is not possible to provide a complete list, however, the following list gives an overview of a wide spectrum of software components models:

- Fractal [28].
- Robocop (Robust Open Component Based Software Architecture for Configurable Devices Project) [114].
- Pecos (PErvasive COmponent Systems) [115].
- BIP (Behaviour, Interaction, Priority) [116].
- OpenCOM [117].
- ProCom (PROGRESS Component Model) [118].
- CCM (CORBA Component Model) [26].
- EJB (Enterprise JavaBeans) [25].
- Koala [119].
- MS COM (Microsoft Component Object Model) [120].
- OSGi (Open Services Gateway Initiative) [27].
- BlueArX [121].
- COMDES II (COMponent-based design of software for Distributed Embedded Systems, version II) [122].
- Rubus [123].
- SOFA (Software Appliances) [30].
- AUTOSAR (AUTomotive Open System ARchitecture) [124].
- KobrA (KOmponentenBasieRte Anwendungsentwicklung [1]) [125].

1. In German.

For comparison we chose five component models: SOFA, CORBA Component Model (CCM), Fractal, Enterprise JavaBeans (EJB), and OSGi. We think that this sample of models is wide spread [2]. Moreover, it honestly and accurately represents the current state-of-art and state-of-practice software component models in both academia and industry. The information provided in this section, 5.1.2, is merely technical details and it is collected from the references of models and from related component survey documents. The author of this dissertation did not add his opinion nor position in this section. It is important to mention that domain specific software component models (such as COMDES II) are out of the scope of this comparison. Cloud component model is not domain specific, and can not be compared with such models.

5.1.2.1 SOFA

SOFA [5,30] is a component model developed at Charles University in Prague. SOFA uses a hierarchical component model with connectors, which are also first class entities like components. The component model is defined using the meta-model presented in figure 5.1. This approach has many advantages such as the support of MDD, the possibility of automated generation of meta-data repositories with a standard interface, a standard format for data exchange among repositories, support for automated generation of model editors, etc. As the particular technology for defining the meta-model and generating a meta-data repository, SOFA team has been using EMF. A brief description of main entities of the meta-model is as follows:

The NamedEntity and VersionedEntity classes are reused multiple times in the meta-model. All other classes featuring a name inherit from NamedEntity. The

2. based on our best knowledge.

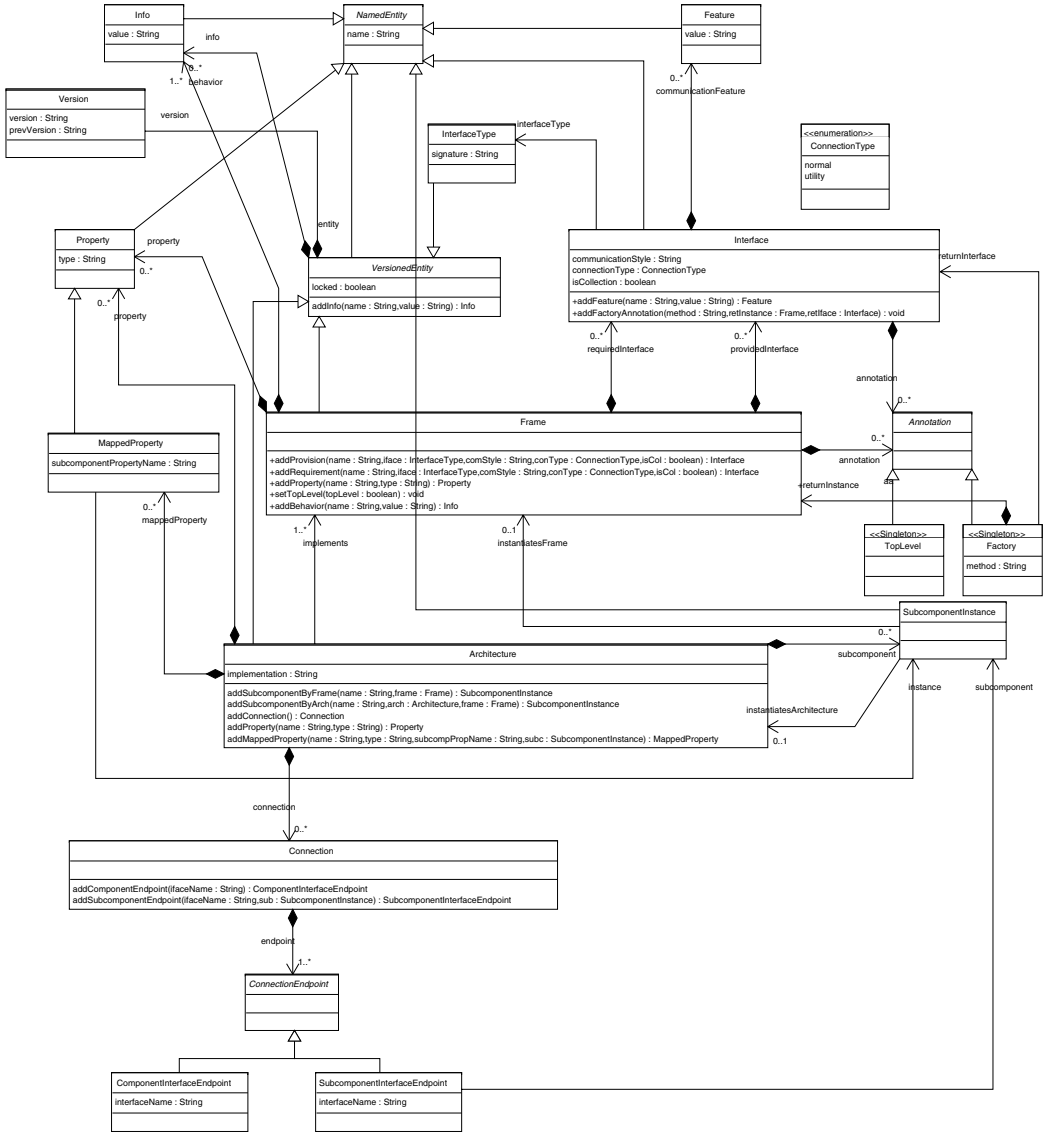


Figure 5.1: SOFA 2.0 meta-model. From [5].

VersionedEntity class further extends NamedEntity by adding a version. A black-box view of a component is defined by the Frame class (it inherits from the VersionedEntity). The provided, resp, required, interfaces of a frame are modeled by the provideInterface, resp. requiredInterface, association with the Interface class, which is further associated with the InterfaceType class defining the real type of the interface. Also, Frame is associated with the Property class, which defines the namevalue properties used to parameterize components (these values are specified at the deployment time). A gray-box view of a component is defined by the Architecture class. The component's architecture implements at least one frame captured by the association between the Frame and Architecture classes (the option of multiple frames for an architecture allow for taking different views on the component behavior) and contains subcomponents and connections among them. If the architecture is empty, then the component is primitive and is directly implemented. Architectures can also add other properties (again captured via the association with the Property class), and/or can expose subcomponents' properties as their own (captured by the association with the MappedProperty class). Connections among subcomponents are realized via connectors. At the meta-model level, connectors are just links among components' interfaces and they are captured by the Connection and Endpoint classes. The communication style of a connector and its non-functional features, which it has to provide at runtime (like secure connection, etc.) are defined by the Feature class, which is associated with Interface.

5.1.2.2 CORBA Component Model (CCM)

CCM [26] is based on Corba object model and it was introduced as a basic model of the OMGs component specification. The specifications of CCM defines the following

- An abstract model.
- A programming model.
- A packaging model.
- A deployment model.
- An execution model.
- A metamodel.

CORBA components communicate with outside world through ports. CCM uses a separate language for the component specification: Interface Definition Language (IDL). CCM provides a Component Implementation Framework which relies on Component Implementation Definition Language and describes how functional and nonfunctional part of a component should interact with each other. In addition, CCM uses XML descriptors for specifying information about packaging and deployment. Furthermore, CCM has an assembly descriptor which contains metadata about how two or more components can be composed together.

5.1.2.3 Fractal

Fractal [28,126] (figure 5.2) is a component model developed by France Telecom R&D and INRIA. It intends to cover the whole development lifecycle (design, implementation, deployment and maintenance/management) of complex software systems. The main features of this model are:

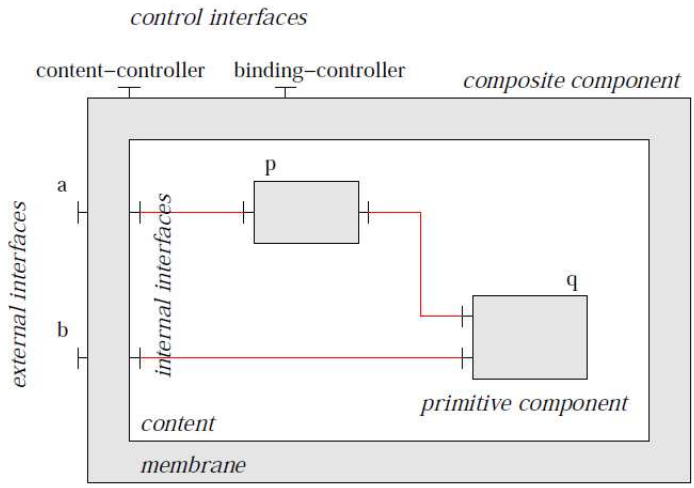


Figure 5.2: A normal Fractal component (from [2]).

- Composite components (components that contain sub-components), in order to have a uniform view of applications at various levels of abstraction.
- Shared components (sub-components of multiple enclosing composite components), in order to model resources and resource sharing while maintaining component encapsulation.
- Introspection capabilities, in order to monitor and control the execution of a running system.
- Re-configuration capabilities, in order to deploy and dynamically configure a system.

It includes several features, such as nesting, sharing of components and reflexivity in that sense that a component may respectively be created from other components, be shared between components and can expose its internals to other components. The main purpose of Fractal is to provide an extensible, open and general component model that can be tuned to fit a large variety of applications

and domains. Fractal includes different instantiations and implementations: a C-implementation called Think, which targets especially the embedded systems and a reference implementation, called Julia and written in Java.

5.1.2.4 Enterprise JavaBeans (EJB)

EJB [25], developed by Sun MicroSystems envisions the construction of object oriented and distributed business applications. Enterprise JavaBeans (EJB) is currently available in version 3.0. This standard provides a reasonable component-model and a rich set of services and facilities, e.g. for persistence management and transaction control. Additionally, the isolated treatment of different aspects – e.g. application logic and security – allows separation of concerns for development, execution, and administration. It provides a set of services, such as transactions, persistence, concurrency, interoperability. EJB differs three different types of components (The EntityBeans the SessionBean and the MessageDrivenBeans). Each of these beans is deployed in an EJB Container which is in charge of their management at runtime (start, stop, passivation or activation) and EFPs (such as security, reliability, performance). EJB is heavily related to the Java programming language.

5.1.2.5 Open Services Gateway Initiative (OSGi)

OSGi [27] is a consortium of numerous industrial partners working together to define a service-oriented framework with an open specifications for the delivery of multiple services over wide area networks to local networks and devices. The OSGi framework forms the core of the OSGi Service Platform Specifications. It provides a general-purpose, secure, and managed Java framework that supports

the deployment of extensible and downloadable applications known as bundles. Contrary to most component definitions, OSGi emphasis the distinction between a unit of composition and a unit of deployment in calling a component respectively service or bundle. It offers also, at contrary to most component models, a flexible architecture of systems that can dynamically evolve during execution time. This implies that in the system, any components can be added, removed or modified at run-time. In relying on Java, OSGi is platform independent. There exists several additions of OSGi that provides additional characteristics.

5.1.3 Encapsulation and Extra Functional Properties

We think that the most attractive point in components in general is simply *encapsulation*[3]. That is correct for components in electronics (IC chips), mechanics (engine, speed box, braking system, etc), and also in software components. Lau et al. noted: "encapsulation has the potential to counter complexity" page 720, [24]. "We believe that the ideal (software component) model should have the key characteristics of encapsulation and compositionality" page 720, [24].

Traditionally software component models encapsulate *functionalities*. Attempts to encapsulate extra functional properties (EFPs) such as security or reliability are currently under research and have limited success. To support this remark we will rely on the survey [13].

As it appears from table 4 in [13] that all industry-successful component models, such as EJB, OSGi, MS COM and CCM, encapsulate functionality and have no support for extra functional properties. None of them define any interface

3. This idea is discussed also in section 6.1.1.

with a QoS contract level as defined in [127]. Only component models from academia, such as Sofa, try to include more inside the component border and exhibit interfaces with QoS values.

In the following, we will try to inspect the SOFA attempt. "SOFA 2.0 uses a hierarchical component model" [12,30]. Obviously, allowing nesting in the component model will lead to huge encapsulation. However, since our concern now is EFPs, let us see what is the effect of component nesting on EFPs. Basically, a hierarchical component is composed of other components. Now let us assume that the level of nesting is limited to one, and suppose component A has inside components B_1, B_2, B_3, and B_4, and they are composed (assembled) together to achieve the required functionality. To predict the EFPs of A we need to know the EFPs of its inside components, and apply the/some EFPs composition theory. But, this theory does not exist [13] page 17. What they (SOFA developers) do is, for small cases, and for directly composable EFPs such as memory usage, they apply EBP and LQN [12] page 395. Now, what about EFPs such as security? Is it composable? No [13] page 11. What if nesting is much more deeper, and what if some of the internal components are third party with very few information about internals (black box)? Clearly we will get hard situation. "Clearly, the composition of EFPs still belongs to the research challenges" [13] page 17. As result, SOFA wishful thinking of predicting EFPs of a component by composing EFPs of its internal components lakes the "theory of EFPs composition." Until this theory is developed and proven successful in industrial usage, the support of EFPs in SOFA is an open area for research.

The other encapsulation-related claim of SOFA is "support of multiple communication styles by using connectors" [30,128]. In other words, communication

is encapsulated inside sofa component by two techniques: first one is connectors, which allows to use multiple communication mechanisms. Second is again hierarchical model, so a large component will include the connector(s) and the components that wants to communicate. This way, communication is encapsulated. But, this method relies on the assumption that: "we can separate functional aspects from communication aspects[4]." As result we place the first inside the component and the second inside the connector. This is completely false. There is no evidence of such possibility, on the contrary, this is considered bad design approach that might lead to "degradation of the system performance" [13] page 10. Such separation is attractive because it makes implementation easier, but there is ethical need to always mention that there might be (probably) a significant cost. Again, without the well development of the theory of separation of concerns, the SOFA communication encapsulation that is based on connectors and hierarchical components is a wishful research thinking.

5.1.4 Software/Hardware Compatibility

Based on our best knowledge all mentioned five component models do not address the compatibility between software and hardware, neither directly, nor indirectly. Some of them, mainly Sofa, EJB and OSGi, even restrict their deployment environment by devices that runs Java Virtual Machine. The availability of JVM is limitation that our model does not suffer.

Of course, the design and implementation teams can make necessary arrangements for the application they are developing to be compatible with the expected deployment devices using any component model. However, this is exterior to the

4. If we do not use connectors we avoid this separation of concerns problem, but this way we are not using SOFA.

model itself. In other words, this is not a feature in the model that the design and implementation teams facilitate. Currently, all software development for HDEs is using such ad-hoc approaches. Our proposed component model fills this gap by a genuine integration of the expected deployment environment L into the definition of the cloud component itself.

Technology	**Page of first occurrence**	**Comments**
Meta-models	389	
LTL	389	Linear Temporal Logic
Promela	389	
EBP	391	Extended Behavior Protocols
EBP2PR translator	389, 408	
Spin model checker	389	
Carleton LQN solver	389	
MDD, EMF	390	
Formal Performance Models	395	
LQN	395	Layered Queuing Networks
SPN	395	Stochastic Petri Nets
FSP	396	Finite State Process
CSP-like notation	396	Communicating and Sequential Processes
Series of transformations	407	"Transformations are *mostly* done manually with *some* parts automated"
Tool chain	408	
JPF	412	Java path finder

Table 5.1: Technologies used to implement inventory application using SOFA component model [12], pages 388-417. The team who designed and implemented the application in [12] is the same team who proposed SOFA.

5.1.5 Ease of Use

In this section we will make a brief attempt to investigate the ease of use of CC model and tools with comparison to the other component models. All of these models are proposed to be used by software engineers. As result, software

engineers need to be able to learn and master these models in reasonable time. The wide usage of EJB, CCM, and OSGi in industry is a clear evidence of its relatively close relation to commonly accepted software engineering skills as described in section 2.3. In [12], page 390:
"Usage of SOFA 2.0 and the verification and prediction tools and approaches is quite easy; an average computer science student takes approximately five days to learn about SOFA 2.0 itself and another five days to write behavior specification of simple components." In the same document, where they implement an inventory system, they mention using technologies listed in table 5.1. In this table we did not list technologies that are usually taught in computer science curriculum as part of the undergraduate program, such as NFA, JAVA, introduction to complexity theory, and parallel computing theory. We use [129] as an average reference of such a program. As result, any "average computer science graduate" needs to learn these technologies from scratch in order to "use SOFA and the verification and prediction tools and approaches". We are sure that the five days estimate is anything but realistic.

On the contrary, the average computer science graduate was in mind when designing CC model and related tools. We think with normal training that lasts for several weeks (up to few months), a software engineer can start using CC model and tools effectively.

5.1.6 General Comparison

Table 5.2 presents an attempt to position our model among current component models, from both industry and academia. In this section, we will make a comparison between these six component models.

	Cloud Component Model	EJB	Fractal	CORBA Component Model	OSGi	SOFA 2.0
Modelling	Formal Language for software architecture. Ontology for deployment environment modeling	N/A	ADL-like language (Fractal ADL, Fractal IDL). Annotations. (Fractlet).	N/A	N/A	Meta-model based specification language
Implementation of applications	Language Independent	Java	Java (Julia, Aokell) C/C++ (Think) .Net lang. (FracNet)	Language Independent	Java	Java
Deployment	At run-time	At run-time	At run-time	At run-time	At compilation and at run-time	At run-time
Interface distinctive features	Local access only	N/A	Component interface, Control interface	Facet and receptacle, Event sink and source	Dynamic interface	Utility interface. Possibility to annotate interface and control evolution
Exogenous Binding	No	No	Yes	No	No	Yes
Vertical Binding	No	No	Delegation	No	No	Delegation

Table 5.2: Comparison between CC model and other well-known component models. All non CC information are from tables 1, 2, and 3 in [13].

– All industry models in this table, namely EJB [25], CORBA Component Model [26], and OSGi [27], have no modelling support of any kind. This is negative in the context of medium and large scale applications where design is a key for software quality. Fractal [28], from academia, depends on special languages to support modelling, while SOFA [12, 29, 30], academia also, uses existing modeling approach. CC model uses formal language to model the architecture and the development process, assembly of CCs. Moreover, CC model uses ontologies (formal-language based) to model deployment environment and the requirements of each CC in the system. Finally these ontologies are used at deployment time to check software/hardware compatibility.

– Only CC model and CORBA Component Model give the designer the choice of appropriate programming language. In these two models, more

flexibility also can be achieve through a mix of programming language. The other four models are Java dependent, except Fractal witch gives the option of several programming languages. However, Fractal forces the implementation programming language to be chosen and fixed. If we use Julia (an implementation of Fractal environment) then we have no more the choice of using C language for parts of the application [5] .

– All component models allow run-time deployment, redeployment, and binding between components. EJB, CORBA Component Model, OSGi, and CC model do not use connectors (Exogenous Binding) and do not support component nesting (Vertical Binding). On the other hand, Fractal and Sofa both support nesting, and Sofa alone uses connectors.

– We think CC model falls in a new category that does not sacrifice the powerful advantages of industry successful component models, and at the same time adds academia-related-theoretical support. First, all industry successful component models do no use connectors (Exogenous Binding), and do not support component nesting (Vertical Binding), same as CC model. We think these two features are important because they result easy to use component models that are attractive for real-life application development. Moreover, These two features have negative impact on the deployment and performance of resulting applications (please read sections 5.3.1 and 5.1.3). Second, all industry successful component models lack the support of modeling languages. While this point might be marginal in small systems, it becomes vital for medium-scale and large-scale designs. CC model attempts to handle this point effectively, without affecting the ease of use of the model for average software engineers. Third, and finally,

5. This argument is based on our best knowledge. Checking [126] we found Julia and Cecilia, however, we found no way to mix these two implementations.

distinctive feature of our model that it allows local access and prohibits remote access to any interface (role). This feature is one of the core contributions of this new model.

Component	Connector	Paradigms
Local	Local	Curent/embedded systems
Local	Distributed	Curent component approach
Distributed	Local	Cloud Component approach
Distributed	Distributed	Most general

Table 5.3: Access point location feature for components and connectors.

If we classify components and connectors as software artifacts with access points that can either be distributed or local, we have four possibilities (please see table 5.3). Our proposal is clearly a trade-off, avoiding the general case and changing the perspective over components.

The difference between components and connectors is largely accepted but this relies on the assumption that: "we can separate functional aspects from communication aspects". As result we place the first inside the component and the second inside the connector. There is no evidence of such possibility, on the contrary, this is considered bad design approach that might lead to "degradation of the system performance" [13] page 10. Our proposal encapsulates all functionalities, including communication, and avoid this separation of concerns problem.

5.2 Context Awareness

In the literature several definitions of the term context can be found [1,130,131]. In this section we will use the following definition from [130]:

"Context is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."

When dealing with context information it is always a challenge to describe contextual facts and interrelationships in a precise and traceable manner. For this reason ontologies seem to be well suited to store the knowledge concerning context [131]. Context Ontology Language (CoOL) is an example of modeling context information using ontologies [131]. Another example is CONON [6].

It is possible to imagine a relation between this 'context' and our 'location'. One common aspect is that both of them are part of HDEs. Another common point is that they are both modelled to improve the QoS of HDE applications. The third, and most clear point, is that the term 'location' itself is used heavily in context modelling, as in figure 5.3.

It is inviting here to mention the major difference between location, as we use it, and location, as in context-aware software. When we use the term location we mean a deployment device, i.e. a computing device. When they use location they mean geographical location. In our work, `Galaxy Mini` is a location. In their work `Garden` is a location.

5.3 HDE Solutions

As mentioned in the introduction & motivation section, HDE problem is described in literature during the last years. We are neither the first to discuss this problem, nor we are the first to propose solutions to build applications for

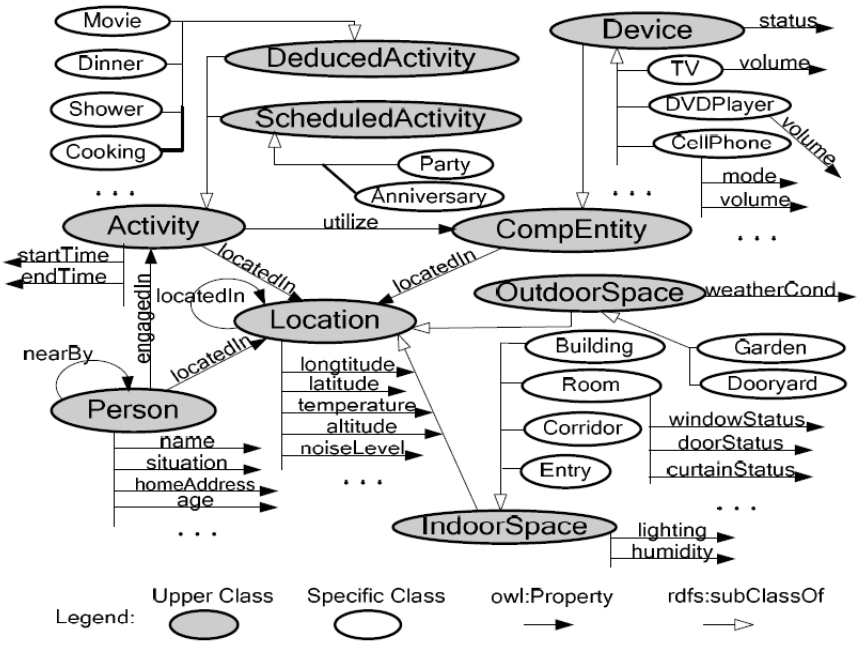


Figure 5.3: Location modelling in CONON. From [6].

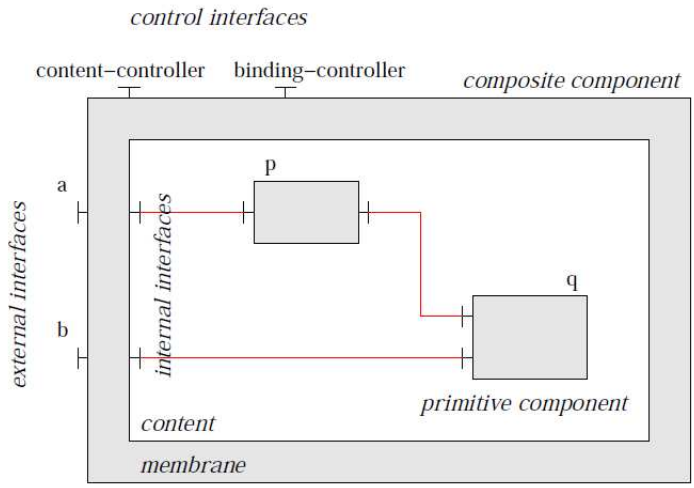


Figure 5.4: A normal Fractal component (from [2]).

these environments. In this section we will discuss two project who proposed solutions for HDE software development.

5.3.1 Cubik Approach

Didier Hoareau *et al.* dealt with the challenges of HDEs [2,16,17]. However, their solution has different scope from ours. First, they expand the already existing component model Fractal. Second, they only model and handle the disconnection of network connection among all characteristics of HDE. In the following we discuss that in details.

This research track uses the concept 'proxy' and adds this concept to the Fractal component model. A normal Fractal component is presented in figure 5.4. In this figure we see one 'composite component', let us call it A, with two internal

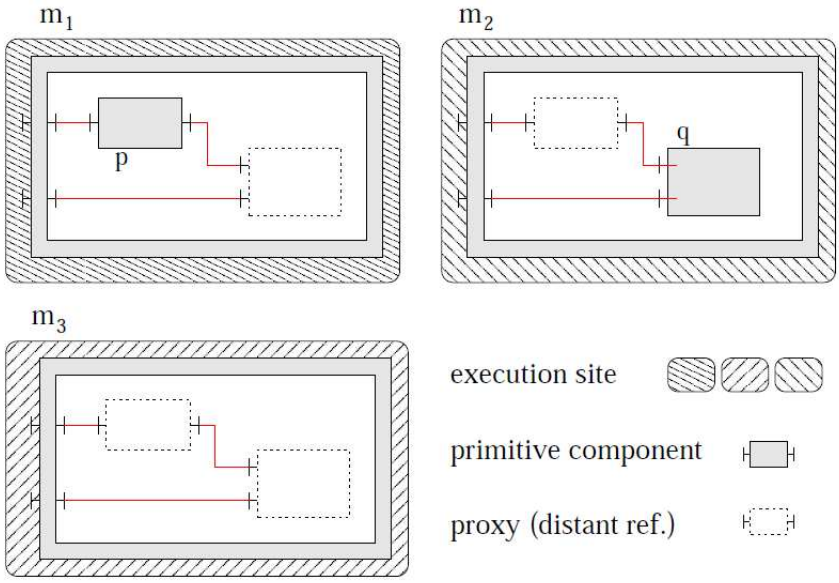


Figure 5.5: The Cubik approach (from [2]).

sub-components (p and q) inside it. This is a normal Fractal design. In figure 5.5 we see the same component designed using the Cubik approach. In this figure, component A become three composite components that are deployed over three different devices: A_1 is deployed over m_1, A_2 is deployed over m_2, and A_3 is deployed over m_3. The three components: A_1, A_2, and A_3 have the same functional properties (i.e. they offer the same functions and require the same functions as each other). However, they have different internal design. A_1 has one proxy of q and one real component p. A_2 has one proxy of p and one real component q. A_3 has no real components and two proxies of p and q.

The approach has two ideas:

– This approach handles restricted resources on devices. For example, if m_3 is a smartphone, it is possible that A_3 can be deployed over it even if p and q are excessively-resource-consuming.

– If the connection between m_1 and m_3 is disconnected (temporarily), the service of A_3 does not stop completely. Rather, it 'degrades'. By degrades, the developers of Cubik mean: A_3 stops the services borrowed from p (through the proxy), and continues the services borrowed from q through the proxy.

5.3.1.1 Discussion

The Cubik approach handles the same problem this dissertation is devoted to, however with many differences on the handling way.

– Degradation of service is not a solution. Yes it is better than shutting of the whole system, but their could be better attempts such as [132–135].
– Software/hardware compatibility check:

  [16] proposes the language 'Constrained Deployment Language - CDL' for modelling the software needs, and the hardware specifications. After that, they use third party 'Constraint Satisfaction Problem - CSP' tool to generate the optimal deployment plan.

  In the following we will comment on this approach:

  The CDL language proposed is an XML language. And as we can see from figure 8.1 it is an informal language. As result, this language allows the definitions of terms only, with little or no specification of the meaning of the term. That contrasts the logical languages that have rigorously formalized logical theories. With formal languages, the amount of meaning specified and the degree of formality increases (thus reducing ambiguity); there is also increasing support for automated reasoning.

  Another problem is the translation from CDL to CSP. In [16] it is men-

tioned that this transformation is automatic. However, we could not found that automatic transformation. And even if this automatic translation exists, it will suffer the consequences of the informal nature of the CDL language itself.

The main problem with this approach is the issue of composite components. Fractal is composite component model. The problem of setting up (modelling) the needs if a composite component is fundamentally different from the problem of modelling the needs of primitive component:

> *"The definition of resource constraints on a composite component poses some difficulties"* [16] page 104.

In fact these are not 'some difficulties', rather it is the absence of composition theory of extra functional properties (discussed in section 5.1.3). This is a direct negative consequence of nested models.

– The 'nesting' effects:

  As a consequence of nesting component model, Cubic suffer the following two problems: First: only one instance of each component can be instantiated (two or more instances are prohibited), page 118 of [16]. Second: can not check the software/hardware compatibility for composite components, page 104 of [16].

As result, we think that Cubik model is an attempt to handle HDE challenges. However, we do not think this solution to be sufficient for practical use, due to the above discussed limitations.

5.3.2 Prism

We will discuss the Prism approach in two tracks:

5.3.2.1 The Marija Mikic-Rakic Track

Marija Mikic-Rakic provides a sophisticated response to one challenge proposed by HDEs, which is the discontinuity of services where the system needs to continue functioning in the near absence of the network [7, 113]. This work proposes a redeployment solution as part of a middleware called Prism-MW.

This work is purely mathematical modelling work. The problem of finding an optimal deployment architecture is converted to an optimization problem. Mathematically: maximize the system availability function; subject to the following constraints:

1. The required memory for each component.
2. The frequency of interaction between any pair of components.
3. Each host's available memory.
4. The reliability of the link between any pair of hosts.

Since items three and four can change at runtime due to mobility and/or complete absence of one or more devices, the initial optimal deployment can be non-optimal after a short or long period of time. As result, the system need to be re-formulated, re-solved, and possibly, components need to be redeployed as in figure 5.6.

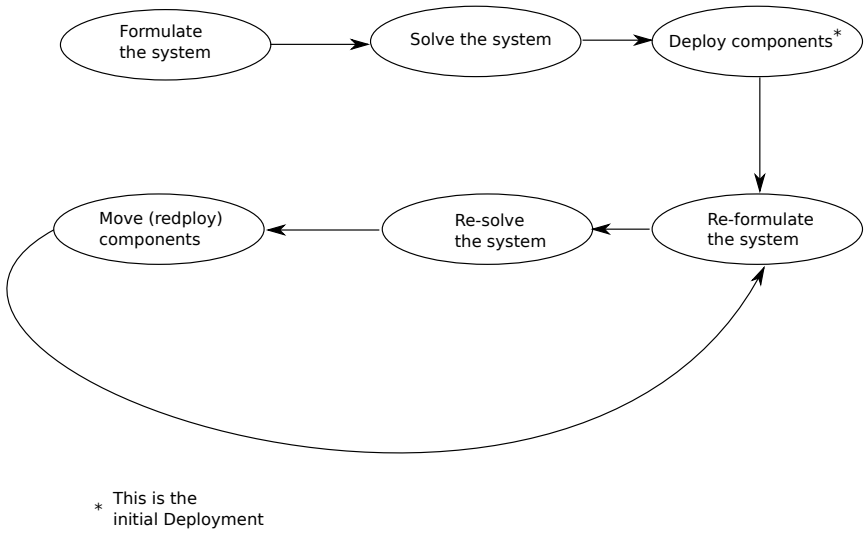


Figure 5.6: The optimal deployment cycle in [7].

5.3.2.2 Discussion of Marija Mikic-Rakic Track

This work did not study the concept 'software component' at all. The word software component is repeated in [7] over and over, but we could not find any trace of what exactly the author of [7] meant. By further reading about the 'Prism-MW' (especially reading [3]) we found that they used C and Java code only. They handled a compiled piece of C code as a software component. The problem with this approach is that it contradicts the widely-accepted definition of software components. A C code has no clear border and clear access points. This simply means that this work ignores the encapsulation value of a software component completely. This has a negative impact on the software development process (please read section 6.1 where we discuss the importance of encapsulation).

Another limitation of this work is that it handles the system availability

only (more precisely, it maximizes the system availability). This work does not handle other extra functional properties. For example, this model can not handle details related to video playback quality, or the camera specification on an smartphone and its compatibility with certain application.

Finally, the validation part in this work emphasized on the problem of maximizing the system availability through redeployment, where the authors provide impressive numerical results. However, they ignored software engineering issues such as: ease of use, post-deployment management and runtime support, systematic development, etc.

5.3.2.3 The Sam Malek Track

Finally, the work of Sam Malek *et. al.* This team proposes a framework and tools to support the complete software engineering life-cycle for the development of HDE applications [3, 18, 19]. Please see figure 5.7.

This framework includes the following tools: XTEAM, DeSi, Prism-MW. Moreover, this framework highly depends on model driven engineering in modelling and code generation. This involves several models, meta-models, and model transformation. Also, this framework adopts a very complex analysis of mobility that includes: software mobility, logical mobility, component mobility, hardware mobility.

5.3.2.4 Discussion of Sam Malek Track

While their tools help overcome the challenges posed by HDEs, these challenges become natural details in our novel cloud component model, where they

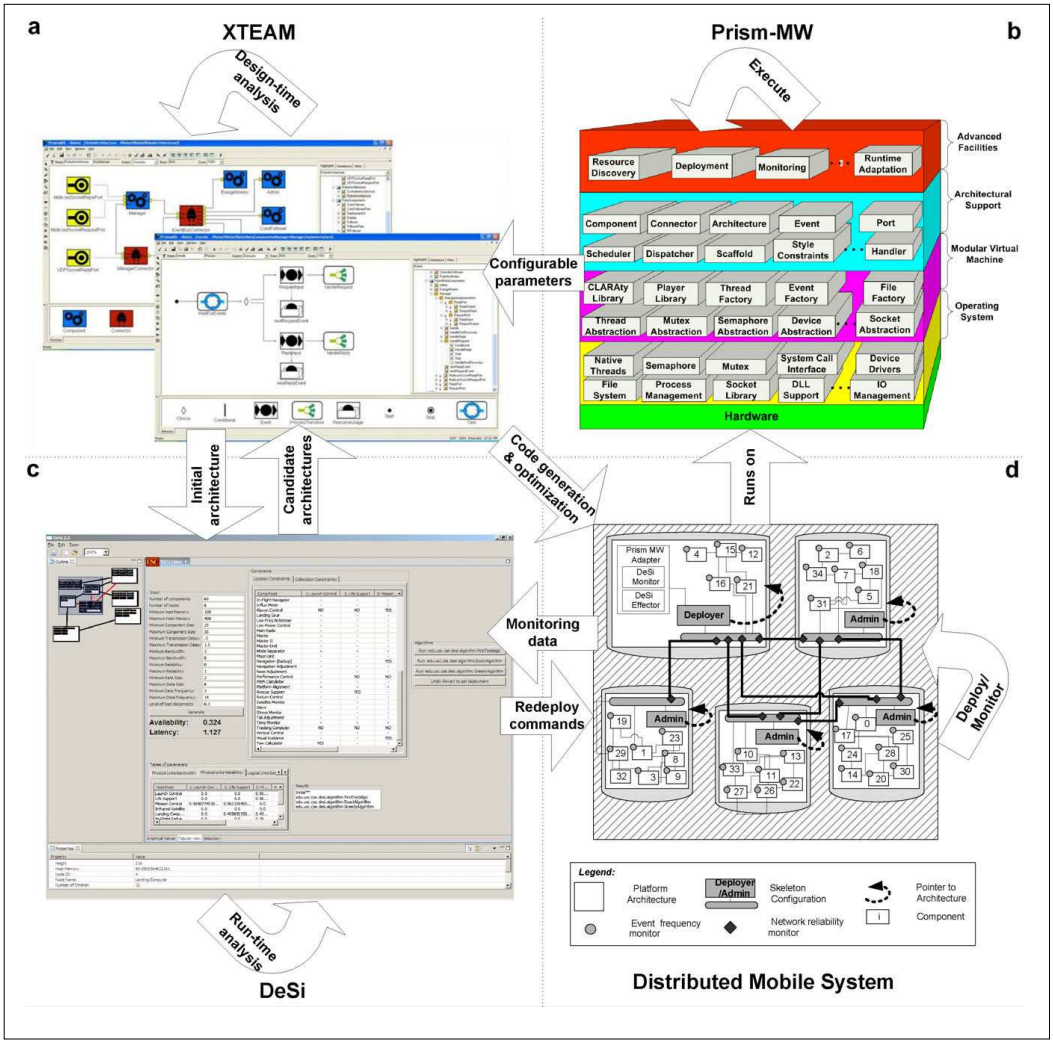


Figure 5.7: The framework proposed in [3].

can be handled systematically. A major disadvantage in the work described in [3] is that they try to mix large number of objectives and technologies. Is it necessary to support "mobile components" to be able to develop HDE software with the required QoS? We do not see any evidence of that. Invoking Software mobility, logical mobility, and component mobility in [3] do not seem to contribute fundamentally to the main objective of handling problems related to physical mobility and HDEs. In our work, the only defined mobility is the physical mobility, which is the mobility of devices. Another major difference with [3] is the deployment optimization. We do not think this should be automated. We think it is sufficient to build the software to be compatible with all devices that are part of the expected deployment environment, and at deployment time, we check the compatibility between the device and the software to be deployed over it. This checker provides yes or no only.

To summarize, this work has two important positive points. First: it tries to handle the complete software development life cycle for HDEs. This is more comprehensive than tackling one aspect of this development process. Second, it has a deep and comprehensive study of the challenges that appear in software development for HDEs. However, we think that this framework itself is challenging. We do not have any evidence that software development using this framework is easier than software development without this framework. To explain the last sentence we would like to use the criteria discussed in section 2.3. If we compare figure 5.7 with the process described in section 2.3, we will find a very far relationship. They are not even close to each other. Even the text in [3] is not helpful in providing any link with the widely accepted software development approach. It is correct that 'software development for HDEs is different from software development for stand-alone applications and even normal

distributed applications'. More over, this dissertation is based on this true fact. However, we think, 'optimal' software development process for HDEs should still be a familiar process, as much as possible.

5.3.3 The Olympus Approach

The Olympus approach [136] is based on the following ideas:

1. The developer should be unaware of the deployment environment details.
2. The development process should be rapid.
3. It is possible to predict all the needs of future development needs.

Based on these points, Ranganathan *et. al.* built the olympus framework. "The developer can not be expected to know how various tasks are to be performed in different environments. This places a bottleneck on the rapid development of applications for these environments" [136]. This is a direct contradiction with our approach. Further more, rapid design is not important for us at all, QoS is important.

Also, "Thus we came to the conclusion that it is necessary to provide developers with a higher level of abstraction while programming these environments. So that they need not to be aware of the specific resources available, context, policies, and user preferences while developing their programs" [136]. In our dissertation we argue this is impossible without huge price. We do not follow this approach, rather we insist on localization acknowledgment.

Now regarding the work of Consel and Lancia [8], it is based on Olympus, as result it is in the opposite direction of our work. "This paper proposes to push the Olympus approach further by integrating the ontological modeling of a pervasive computing environment into a programming language, namely

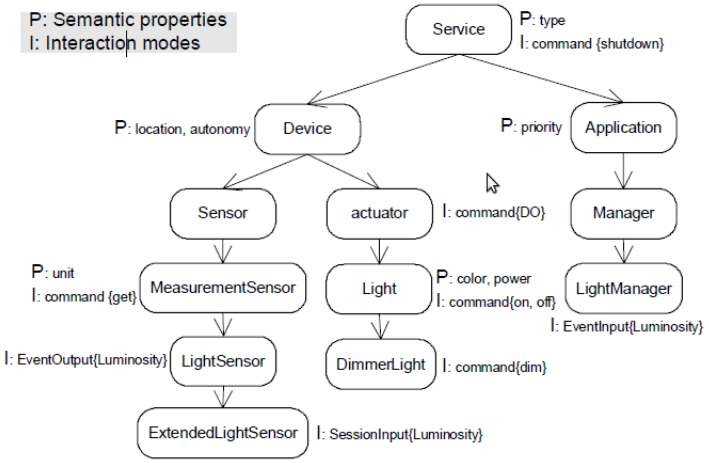


Figure 5.8: Ontology Usage in [8].

Java" [8]. Figure 5.8 is extracted from [8]. In this figure we see an ontological hierarchy defining abstract services. We can see from this figure and paper that they use ontology in different context, mainly they use it to define services, while we use ontology to model the deployment environment, the requirements of BDUs, and for software/hardware compatibility checks.

The main difference between our approach and this approach is that they want the developer to do high abstract programming without any knowledge nor responsibility regarding the execution platform, algorithms, code, etc. This approach targets rapid development and assumes that there is a middleware (namely Olympus) that is capable of fulfilling the gap. We do not accept that. If the developer needs some functionality, it is his responsibility this functionality will work as expected in the target deployment environment. There is no other entity that will do the magic for the developer.

5.4 The 'Medium' Approach

This dissertation is part of the project: Components for Mobile and Adaptive Architecture - CAMA [6]. In the following we will show the related work in this project. Eric Cariou in 2002 proposed the *medium* construct [137]. Medium is defined as: abstraction of communication. Cariou tried to solve the following problem: in a distributed application there is a mix of communication related aspects and functionality related aspects at design time. During implementation, the traceability of the communication aspects from design to implementation are not clear. He thinks this is a major problem in distributed applications. He proposed the medium construct, and a UML [7] based software development process that allows:

- UML-based design of medium-based distributed applications. Cariou emphasizes that this is a platform independent stage.
- Several steps that transforms this PIM into a Platform Specific Model-PSM.

After that, Kabore showed how an ordered sequence of model transformations can be used to describe and automate the above mentioned process [138]. The work of Kabore is dependent on model driven architecture (MDA) from object management group (OMG).

6. CAMA is part of the computer science department in Telecom Bretagen.
7. This work is heavily dependent on UML terminology, views, and graphical notation.

Part III

CONTRIBUTION

In this part we present, in details, our proposed novel software engineering paradigm. In organizing this part, we used several sections to cover several aspects of this proposal. It is important to notice that when our contribution is based on other work, we devote a section (or several sections) to discuss such work. This is not a mix between contribution and state of art, rather, we wanted the presentation context to be smooth with a solid foundation. An example is section 8 where we present our novel deployment environment modeling and related checker. In this section we clearly devote several sections for ontology and ontology related languages and techniques. We think these sections (particularly in this position in the manuscript) are essential to clarify our ontology-based proposal.

Chapter 6

Paradigm Shift

To respond to the challenges discussed in the previous section, and to maintain expected software quality which also mentioned in the previous section, we think we need to pass a "paradigm shift" from the way software is designed and implemented currently to our new vision that this dissertation is devoted to. This is a paradigm shift from *distribution transparency* to *localization acknowledgment* being the first class concern.

The contribution in this thesis has several faces, but still, these faces are cohesive. Each of these faces form a partial contribution, however, this partial contribution does not mean anything if isolated from the overall proposal. Moreover, the merit of the overall proposal can not be grasped by reading one partial contribution. The merit of the proposal is evident only if all parts of this work are cohesively organized.

6.1 Software Component Border: A New Vision

6.1.1 Software Components - Complexity Management

We believe that the attractive point in components in general is simply *encapsulation*. That is correct for components in electronics (IC chips), mechanics (engine, speed box, braking system, etc), and also in software components. Lau et al. noted: "encapsulation has the potential to counter complexity" page 720, [24]. "We believe that the ideal (software component) model should have the key characteristics of encapsulation and compositionality" page 720, [24].

Why encapsulation counters complexity? We think this is directly related to the thinking paradigm of human. Human was creative enough to discover that he/she can tackle problems that, by far, exceed his/her comprehension. This is *the divide and conquer algorithm*. Divide and conquer is a very powerful approach in handling complex and difficult problems. Moreover, this approach was successfully utilized for hundreds of years, and proven to be effective and attractive. In software engineering, architectural decomposition is a direct utilization of divide and conquer algorithm. There are two requirements in order to utilize the divide and conquer approach over a certain problem. First, the problem itself needs to be breakable into sub-problems. This could be applied to sub-problems also, until we reach a simple-to-comprehend, and a simple-to-solve sub-problems. Second, the solutions for sub-problems need to be composable to generate the/a solution for the original complex problem.

Starting from these two requirements, we can see the direct relationship between divide and conquer from one side and software components from the other

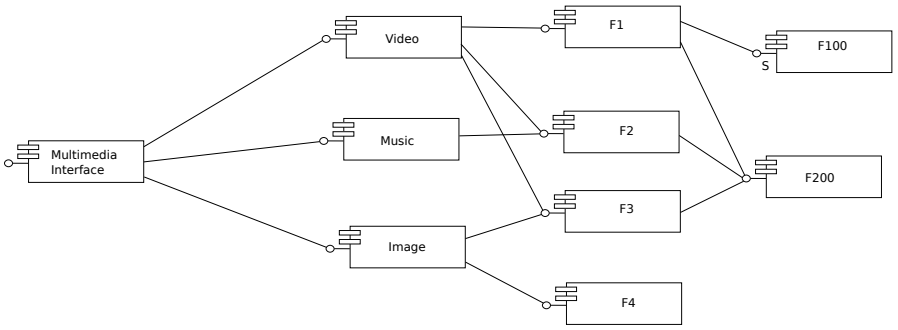


Figure 6.1: Component assembly as realization of the divide and conquer approach.

side. The first requirement is analogous to the decomposition (for example the functional decomposition) that is carried out at the design phase of software development. The second requirement is directly analogous to the component assembly. The key in this analogy is encapsulation. Encapsulation allows the programmer to look at a small sub-problem alone, without any care about the whole system. Moreover, encapsulation makes the assembly of the partial solutions (implemented components) possible through their interfaces. Otherwise we would involve in code merging with close to impossible compatibility process. This is the power of software components. This idea is presented in figure 6.1. From that figure we can argue that it is much easier to comprehend and implement the component `F200` than implementing the whole `Multimedia` application. This is the main idea in divide and conquer approach. From the same figure, we can see also that the assembly of those components collectively achieve the functionality expected from the multimedia application. The last point is the second requirement in the divide and conquer approach.

To conclude this section, encapsulation, realized through software components, is a powerful tool to be used in software engineering for large systems.

6.1.2 Component Border

From the previous section, we found that encapsulation is important. Encapsulation itself involves two fundamental elements: first one is the border, and second one is the interface(s). That is true for all encapsulation techniques including software components. As presented in section 5.1, majority of software component models share a common mentality which is encapsulating functional properties inside the component border [13].

If we assume that the system in figure 6.1 is a stand alone application that runs locally on a single machine, we see the perfect sense of this mentality. In this context, a complex application is decomposed into several sub-problems that are easier to comprehend and implement. Finally, the assembly of components through their interfaces is responsible of delivering the expected functionality of the initial multimedia system.

This perfect sense will not stay the same in figure 6.2. Let us assume that the same application is implemented in both figures, but in 6.2 the application is distributed (with some components not shown in the figure). In this context we have two problems to worry about: first is the remote communication path between `F1` and `F100`, and second is the large difference in resources available in sites A, B, and C.

Would the initial argument about encapsulation and assembly of solutions to sub-problems still holds? *Our answer is: definitely it will not hold.* The compatibility between component `F1` and sites A, B, and C is a major problem that was not taken into account in traditional component models. As result, we can not depend on the fact that we have a solution for this sub-problem

that will run as expected on all sites. On the other hand, the assembly between component `F1` and component `F100` in figure 6.1 is far more dependable and predictable than the same assembly in figure 6.2. The result of this point is that we can not assume that the solutions for sub-problems are assembled to form the solution for the original problem. As result, the benefit from encapsulation, and consequently software components, is not possible in the case of HDEs in figure 6.2.

Cloud component model solves these two problems, and restores the encapsulation, and consequently software components, to its original powerful objectives, even for HDEs. In figure 6.3 cloud component `F100` if functionally equivalent to the component `F100` in figure 6.2, with the interface providing the same functions. CC `F100` provides its services `S` to the other CCs that need it (in this case CC `F1`) locally on sites A, B, and C. Migrating remote communication to be internal to the component border has a fundamental effect on the encapsulation power of the software component. In this case, the component, i.e. the CC, is aware of the communication paradigms available, along with their characteristics. This will allow the CC to effectively match the underlying infrastructure, and maintains the promised functionality at the delivery point.

This is pure authority (jurisdiction) shift. In all available component models, the remote communication is external to the component border, as result, it is a place of ambiguity responsibility. Any point from source to destination, along with the whole communication path could be blamed for the errors that might occur. In CC this is not the case. There is no source, neither destination. The interface `S` is local at the point of usage, and its the responsibility of the CC to provide the promised services with the promised QoS.

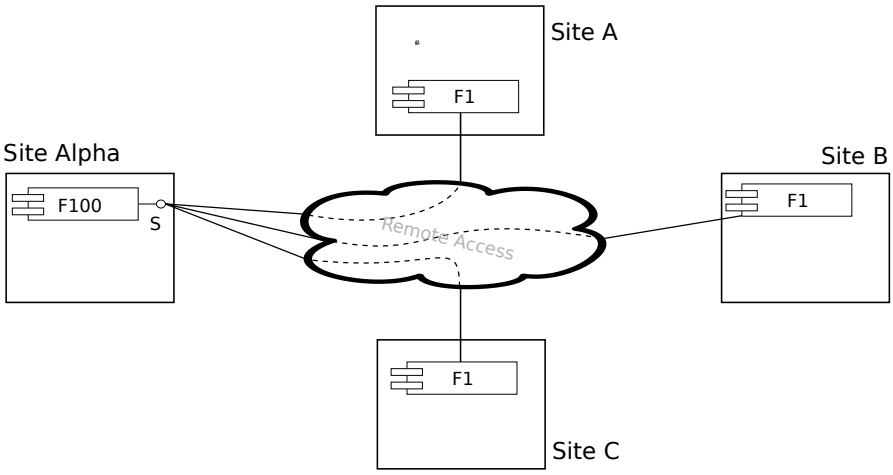


Figure 6.2: A distributed application using current component model.

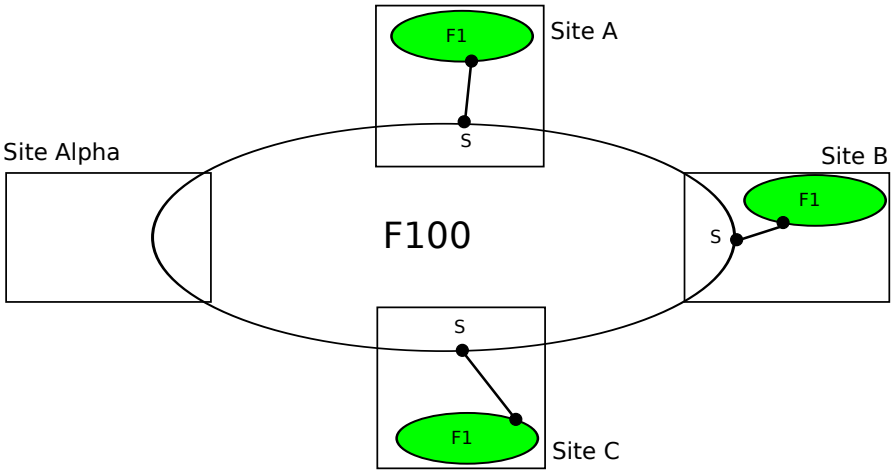


Figure 6.3: The same distributed application of figure 6.2 after applying the proposed paradigm shift.

This new definition of software component border is a novel contribution in our proposal, and it is fundamental to the overall proposal. The solution for the other problem is discussed in the next section.

6.2 Hardware/Software Compatibility: A New Vision

As discussed earlier, in order for the divide and conquer approach to work, sub-problems need to be easy to comprehend, and easy to solve. In software engineering terms, sub-problems should be easy to implement and the implementation can run correctly to provide the promised functionality with the expected QoS. This is straight forward in stable distributed environments. In these environments, devices are usually desktops with unlimited power supply, stable connection, adequate processing power, etc. This is not the case in HDEs, as in figures 6.2 and 6.3. Our proposal is based on raising the level of importance of location (the host device) to be of highest priority starting from the very beginning of the software development process. We call this *localization acknowledgment.* Moreover, this is not a recommendation. We made this factor an indispensable part of the formal definition of the CC, please refer to section 7.1. If we look at figure 6.2 we will notice the need of compatibility between component `F1` and sites A, B, and C. There is no component model currently that handles this problem at the component model level (please read section 5.1.4). As result, software engineer has two options, either to ignore this problem completely, or to use ad-hoc techniques to partially acknowledge this need, hoping the software will work. For example, suppose there is a certain component that needs 10 MBytes dedicated RAM. Also, this component is available to download over the Internet. It is possible for the programmer of such com-

ponent to design and implement techniques to 'read' the specifications of the device before deployment (in this case the available RAM), and then make the deployment decision. We think this is ad-hoc because it does not follow any theoretical model, moreover it is extremely limited [1].

CC model supports localization acknowledgment through the following constructs:

- The expected deployment environment is fundamental and essential part of the CC definition, at the formal level, and from the very beginning of the software development process.
- An implementation variant that is compatible with each device in the expected deployment environment.
- Ontology based hardware/software checker. This tool is invoked during the deployment of any CC, and is responsible of choosing the right implementation variant for the device concerned (the device on which the CC is being deployed).

This localization acknowledgment forms the heart contribution of our novel model: the cloud component model.

6.3 Global View

Bad communication is taken care of by the new definition of CC border. Compatibility of CC with device in HDEs is taken care of by the ontology based checker. As result, we restored the divide and conquer conditions to be valid for HDEs. As result, we have a new component model that can do for

1. Can this component query if the camera in the smartphone is fixed-focus, full-focus, or Auto-focus?

HDEs exactly what EJB, OSGi, and CCM did for both centralized and stable distributed environments.

Chapter 7

Cloud Component Model, Assembly, and Development Process

7.1 Cloud Component Model

Current software component models can not cope with the challenges posed by HDEs [139]. Based on that, we chose to propose a new component model that fills this gap. We chose the name *cloud component* - *CC* since our component model encompasses physical borders and hence hides the technologies, implementation variants, and architecture choices used to conform to the physical topology of the underneath infrastructure. Also, cloud [1] is related to elasticity, i.e. developing a software where both the number of users and the available resources can increase or decrease. We think the CC model offers several con-

1. In the context of Cloud Computing mainstream.

structs to support such elasticity. As example, cardinality (will be discussed later) is a way to introduce (and bound) elasticity starting from design, passing through the development process, and ending in deployment and running.

Our approach in presenting cloud component model is based on two different notations: the informal notation and the mathematical/formal notation. The informal notation is easier to understand and is highly dependent on figures, while the formal notation is more compact, more precise, and less ambiguous. The formal notation allows us to communicate precise details easier, and allows us to easily present statements and proofs.

Table 7.1: The set of symbols used to construct the formal notation.

Concept	**Symbol**	**Comments**
Cloud component	Ω	ΩA is read CC A
Roles	Λ	Type
	Γ	Instance
Cardinality	K	
Localized at	$\downarrow$	
Role Multiplicity	μ	
CC Multiplicity	M	
Location - Type	T	
Location - Host	H	Specific device
CC assembly	σ	A list of connections between CCs
Connect Operator	$\otimes$	A binding between

Continued on next page

Table 7.1 – continued from previous page

Concept	Symbol	Comments
		two roles
Package	Θ	
BDU	Υ	General
	Ξ	An implementation of a Role
	Φ	A BDU that has no existence on the CC border
Set of	$\overline{symbol}$	$\overline{\Omega}$ is set of CCs
Define	$\equiv$	
Design step	$\rightsquigarrow\rightsquigarrow$	This step involves design choices
Dependency	$\longrightarrow$	A dependency between BDUs or packages
Multiple dependency	$\overset{*}{\rightarrow}$	BDU level only
Package assembly	ω	
BDU assembly	ρ	

7.1.1 The definition of cloud component

Definitions 1 to 5 collectively form the definition of cloud component.

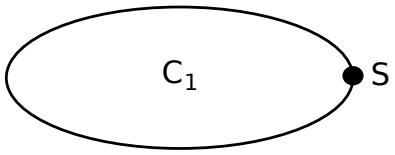


Figure 7.1: CC style with a single interface S.

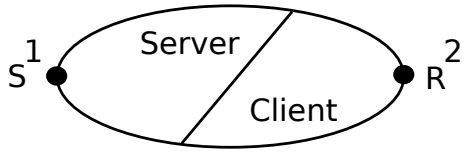


Figure 7.2: CC with two roles, cardinality, and location.

7.1.1.1 Definition 1: Roles

Let C_1 be a cloud component with single interface S as illustrated in figure 7.1. S is defined through an Interface Definition Language - IDL. We assume S defines the signature of provided and required functions of C_1. The contract of this interface could be more sophisticated, but we restrict its definition for the sake of simplicity.

A cloud component can have several interfaces : P, Q, R, etc. We call these interfaces 'roles' because their identification (set of functions they gather) is guided by the way the component can be used through this interface. The cloud

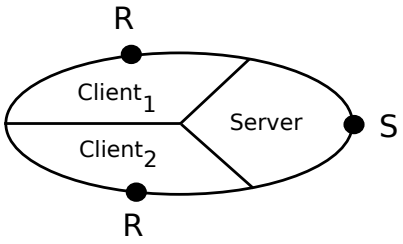


Figure 7.3: Right: CC *com* with two roles and three hosts.

component in figure 7.2 has two roles: S and R.

When an interface (role) is used as a user (human) interaction point only, the above mentioned IDL becomes kind of back-end, and the user-interface is the front end. This special case conforms to the general definition of any CC role.

7.1.1.2 Definition 2: Cardinality

Each cloud component can have several roles. In addition, the role is allowed to have several instances, i.e., several carbon copies of the same IDL. The total number of instances of a role in a running version of the component is called: the cardinality of the role. In figure 7.2 the role S has cardinality one and the role R has cardinality two. Combined with location property (explained later), this approach will encapsulate the communication and all its details and semantics inside the component border.

7.1.1.3 Definition 3: Connection

Once the component border is defined, the connection rules can be defined. In order to suppress ambiguity of 1-to-many or many-to-many connections identified in [110] we allow a role to connect to only one role of another cloud component in a one-to-one connection.

This rule applies at the instance level, when cloud components are actually implemented. In order to allow a 1-to-many or many-to-many connectivity, we use 'role cardinality'.

7.1.1.4 Definition 4: Multiplicity

Cardinality is a number $k \in N$. We can allow more complex structure by not specifying k (at some point of the design). Instead, we put constraints on k called multiplicity. For example a role R can have multiplicity [1..5], [1..$*$], or simply $*$.
At this level of definition, we are not bounded by decidability features but only consider constraints definition.

7.1.1.5 Definition 5: Location

Each role is assigned a location to run on. The location in the most basic form is a computing host/device. In figure 7.2 a cloud component has two different roles, S and R. Role S has one instance that is located at the host $Server$. The role R has two instances that are located at the host $Client$. Figure 7.3 presents a cloud component that has two different roles, S and R. Role S has one instance that is located at the host $Server$. The role R has two instances one of them is located at the host $Client_1$ and the other is located at host $Client_2$.

One should not mix our definition of location with the *geographic location*. Our model does not define or recognize geographic location, rather, we acknowledge location as a computing/electronic device that might be mobile or not. It is fundamental to assert that location is integral part to the CC definition, in other words, without location specification the cloud component definition will not be complete. Finally, and at design and implementation stages, the collection of all locations are called *'the expected deployment environment.'*

7.1.2 Formal definition of cloud component

A single cloud component is defined using the following four-tuple:

1. A finite set of roles $\overline{\Lambda}$.
2. A finite set of multiplicities for these roles $\overline{\mu}$.
3. A set of possible deployment environments $\overline{L}$. Each L is either a finite set of hosts $\overline{H}$, or a finite set of host types $\overline{T}$.
4. A function Z that maps roles to location types or hosts.

$$\Omega \equiv (\overline{\Lambda},\ \overline{\mu},\ \overline{L},\ Z)$$

The following formally defines the cloud component *com* in figure 7.3:

$\Omega com \equiv (\overline{\Lambda},\ \overline{\mu},\ \overline{L},\ Z)$ where:

$\overline{\Lambda} = \{\Lambda S,\ \Lambda R\}$

$\overline{\mu} = \{(\Lambda S,\ 1),\ (\Lambda R,\ 2)\}$

$\overline{L} = \{\{TServer,\ TClient_1,\ TClient_2\}\}$

$Z:\ \ \Lambda S \downarrow TServer,\ \ \Lambda R \downarrow TClient_1,\ \ \Lambda R \downarrow TClient_2$

The formal definition is read as follows: the CC *com* is defined using its four-tuple. The set of roles contains two roles: role type S and role type R. Role type S has multiplicity 1, and role type R has multiplicity 2 [2]. The set of expected deployment environments has only one set, which contains three host types: host type $Server$, host type $Client_1$, and host type $Client_2$. Finally Z can be read as: The role S is localized at host of type $Server$. Role R has

2. Generally the multiplicity of a role is defined as a range $a..b$. If only one integer is used then $a = b$.

two instances, one is localized at host of type $Client_1$ and the other is localized at a host of type $Client_2$. Symbols used to construct the formal notation are summarized in table 7.1.

7.1.3 Formal definition of cloud component based system

Generally, a software component can be thought of as 'unit of composition'. This is true for all component models including cloud component model. In CC model, roles are the only access points of the component. A role can serve as a connection port where component C_1 connects to other component C_2 as in figure 7.4. We choose to assembly components in specific architecture to achieve our desired system specifications. *As result we have σ the set of assembly rules that includes the dependency rules between CCs, and all role connections.* Cloud component assembly will be discussed in detail in section 7.2.

A system built using cloud components consists of:

1. A finite set of cloud components $\overline{\Omega}$.
2. A finite set of multiplicities for these cloud components $\overline{M}$.
3. A set of assembly rules σ.
4. A set of possible deployment environments $\overline{L}$.

As result, the system type is fully defined using the "four-tuple" notation:

$$S \equiv (\overline{\Omega},\ \overline{M},\ \sigma,\ \overline{L})$$

Finally we define the system instance $\hat{S}$. Let S be a CC based system that is defined as above. $\hat{S}$ is an instance of that system and is defined using the following five-tuple:

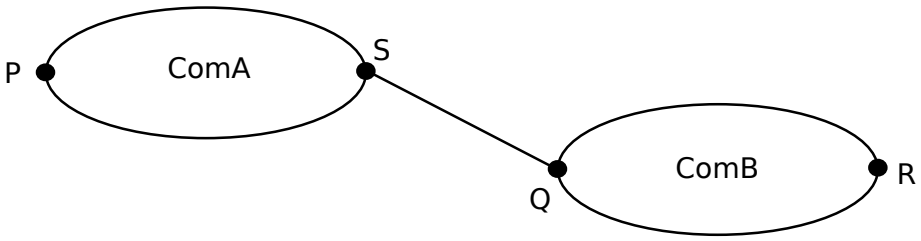


Figure 7.4: Two CCs are composed using roles `S` and `Q`.

1. The system type S that we want to instantiate.
2. The function τ that takes a cloud component as a parameter and returns the number of instances of it.
3. The function K that takes a role as a parameter and returns its cardinality, i.e. number of instances.
4. The deployment environment L which is a finite set of hosts $\overline{H}$.
5. The function Z that maps each instance in $\overline{\Gamma}$[3] to L. In other words, the function Z maps role instances to devices.

$$\hat{S} \equiv (S, \tau, K, L, Z)$$

7.2 Cloud Component Assembly

Generally, a software component can be thought of as 'unit of composition (i.e. assembly)'. This is true for all component models including cloud component model. CC assembly is a tool to build large systems using CCs as building blocks [140].

3. Γ is defined in table 7.1.

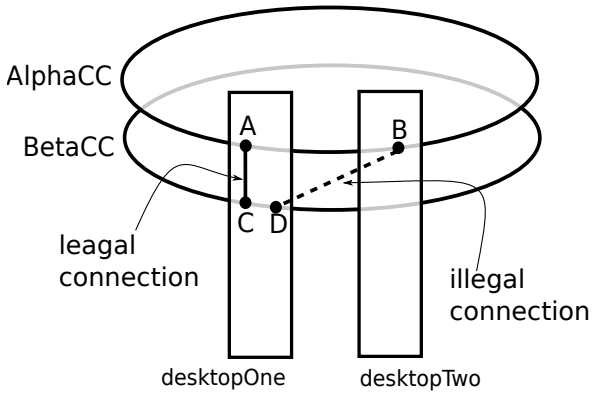


Figure 7.5: Two CCs `AlphaCC` has two role instances `A` and `B`, and `BetaCC` has two role instances `C` and `D`. `A`, `C`, and `D` are hosted by `desktopOne`, while `B` is hosted by `desktopTwo`. Therefore, the connection between `A` and `C` is legal, whereas the connection between `B` and `D` is not permitted.

In CC model, roles are the only access points of the component. A role can serve as a connection port where component `ComA` connects to other component `ComB` as in figure 7.4. The word 'connect' in this context has the following semantic: ΛS connects to ΛQ means that ΛS provides the same functions ΛQ requires and ΛS requires the same functions ΛQ provides.

7.2.1 Assembly Constraints

In this section we will present the detailed semantics of the assembly of two cloud components. Any design need to respect these constraints in order to comply with the CC model [139].

7.2.1.1 First constraint - one-to-one

An instance of role `S` can connect to one instance only of any other role at any time instance. We raised the importance of this constraint from being a recom-

mended design choice to be a fundamental model constraint for several reasons. One of these reasons is to remove ambiguities in the connections. Another and important reason is to control the design precisely, and to be able using this control to ensure the delivery of the expected quality of service. As an example, let us take the role `S` in figure 7.15 from the banking example. And suppose `S` is hosted by some regular desktop. If `S` is expected to have 10 connections, i.e. 10 clients that want to use the video service, is completely different from `S` is expected to have 10^6 connections at the same time. The difference exists in the design, implementation, and the deployment host (probably a normal desktop will not be able to serve 10^6 connections). This difference should be recognized from the very early stages in the design, and this is done in CC model by setting the multiplicity (or cardinality) constraints over roles.

7.2.1.2 Second constraint - local connections only

Two instances of two roles can connect to each other only if both of them are instantiated at the same host as in figure 7.5. If they are instantiated at different hosts they simply can not connect to each other. This is a direct result of the paradigm shift discussed in chapter 6. It is fundamental in our model to migrate all remote communications to be internal to the border of the CC itself. This migration means that these remote communications are designed and implemented using the special software development process [4] of the CC model, and more important, passed all checks necessary to ensure the quality of service expected.

4. We propose a novel software development process to build CCs and CC based systems. The description of this process is in section 7.3.

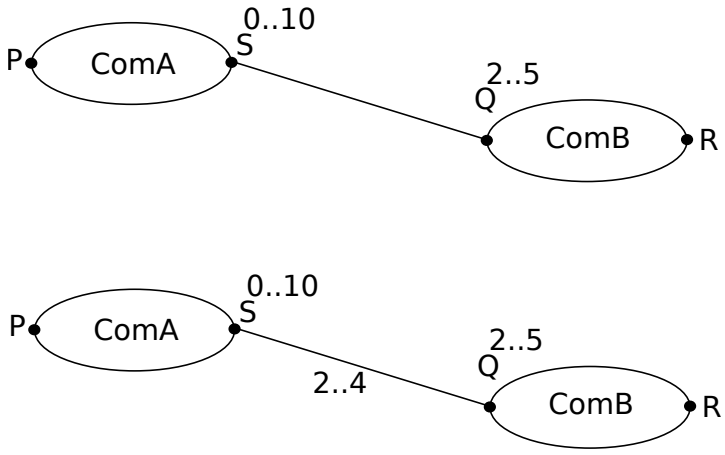


Figure 7.6: The importance of the 'connection multiplicity'. Up: No information. Bottom: The multiplicity of the connection is defined: [2..4].

7.2.1.3 Third constraint - Connection multiplicity

When there is a connection between two roles, that does not mean that all instances of these two roles should connect to each other. Figure 7.6 (up) is an update of figure 7.4 by adding multiplicities to roles `S` and `Q`. To understand the connection in this figure we need to see the uncertainty that exist at this phase of design. During runtime, there might be one instance of `S` and five instance of `Q`, or nine instances of `S` and two instance of `Q`. So how many connections we have at runtime between `S` and `Q`? To answer this question we need to remember that the final responsibility of the design is held by the designer himself, we only provide an advanced model and accompanied tools and checkers. To facilitate the assembly design we add the *connection multiplicity*, which is a range `[min..max]`, where `min` is the minimum number of connections that must exist at runtime, and `max` is the maximum number of connections that might exist at runtime, as in figure 7.6 (bottom). Usually these numbers reflect

the need of either of the roles, or both. For example if I have a role `W` that connects an ATM machine (CC `ATM`) to the bank system (role `S` of CC `Agent`), I can expect `W` to need only one connection at runtime, i.e. [1..1]. On the other hand I expect `S` to allow zero or more connections at runtime, i.e. [0..$*$]. Please see figure 7.15.

7.2.2 Formal definition of cloud component assembly

CC assembly is based on the connection operator $\otimes$, which is a binary operator that takes two CC roles and returns true if the designer explicitly listed those two roles to be connected (this is done in σ as described later), otherwise it returns false. The set of assembly rules is called σ and is defined using the following context free grammar:

$E \rightarrow \{\ I\ \}$

$I \rightarrow IJ,\ \mid\ IJ\ \mid\ \epsilon$

$J \rightarrow (\Omega var.\Lambda var \otimes \Omega var.\Lambda var,\ \ int,\ \ int)$

Where *var* and *int* are terminals such that: *var* represents any string of characters and *int* represents a positive integer. This grammar will recognize the following syntax:

$\sigma = \{(\Omega name.\Lambda name \otimes \Omega name.\Lambda name, min, max),$

$(\Omega name.\Lambda name \otimes \Omega name.\Lambda name, min, max),$

$..., (\Omega name.\Lambda name \otimes \Omega name.\Lambda name, min, max)\}.$

The following shows the assembly in figure 7.6 (bottom) using formal notation:

$\sigma = \{(\Omega ComA.\Lambda S \otimes \Omega ComB.\Lambda Q,\ \ 2,\ \ 4)\}$

In general we can write:

$\sigma = \{(\Omega ComA.\Lambda S \otimes \Omega ComB.\Lambda Q,\ \ m,\ \ n)\}$

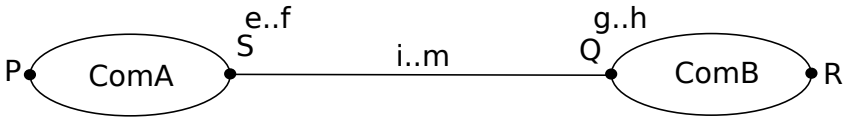


Figure 7.7: CC assembly normal form A. Ranges are always consistent (i.e. $min \leqslant max$).

This connection has the following semantics: at least m instance of S connect to m instance of Q, and at most n instance of S connect to n instance of Q. *This is correct for one and only one instance of each cloud component.*

7.2.3 Remark

The connection operator $\otimes$ is symmetric:

$\Omega ComA.\Lambda S \otimes \Omega ComB.\Lambda Q \Longleftrightarrow \Omega ComB.\Lambda Q \otimes \Omega ComA.\Lambda S$

The above statement is read as follows: role S is connected to role Q if and only if role Q is connected to role S.

7.2.4 Assembly checking algorithm

7.2.4.1 CC assembly normal form A

Figure 7.7 presents the general case of assembly, which is defined as normal form A assuming there is a single instance of any and all CCs. The connection here has the following semantics (as mentioned in the definition): at least `i` instance of `S(Q)` connect to `i` instance of `Q(S)`, and at most `m` instance of `S(Q)` connect to `m` instance of `Q(S)`. This is correct for one and only one instance of each cloud component `ComA` and `ComB`[5].

5. This is redundant since there is one and only one instance of each and all CC in normal form A.

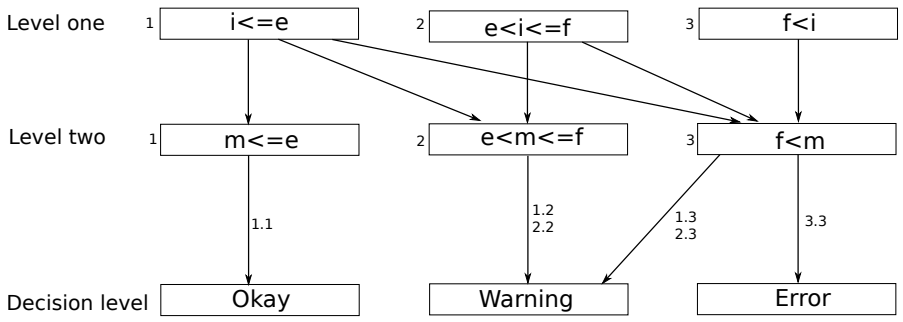


Figure 7.8: The relation between the two ranges $[e..f]$ and $[i..m]$ in figure 7.7. We start with level one, and depending on the value of `i` we move to level two where we inspect the value of `m`. The label(s) on the arrows leading to the decision level indicate the decisions made on the upper two levels.

The two ranges $[e..f]$ and $[g..h]$ are not related in any way since cloud components may have been designed independently. On the other hand, the two ranges $[e..f]$ and $[i..m]$ are related as in figure 7.7. Cases presented in figure 7.8 can be reduced to the following four cases:

1. $i \leq e \ \ \& \ \ m \leq e \Rightarrow$ Valid
2. $i \leq e \ \ \& \ \ e \leq m \Rightarrow$ Warning
3. $e < i \leq f \Rightarrow$ Warning
4. $f < i \Rightarrow$ Error

The same argument holds for the two ranges $[g..h]$ and $[i..m]$ in figure 7.7. Depending on the numbers, we have three cases:

1. Valid: in this case we do not have a chance of connection problems at runtime if the instantiation of roles respected the design.
2. Warning: in this case the designer need to be careful because even if the instantiation respected the minimum requirements, we might face invalid situations. For example, if $e < i \leq f$, and at runtime we have only `e`

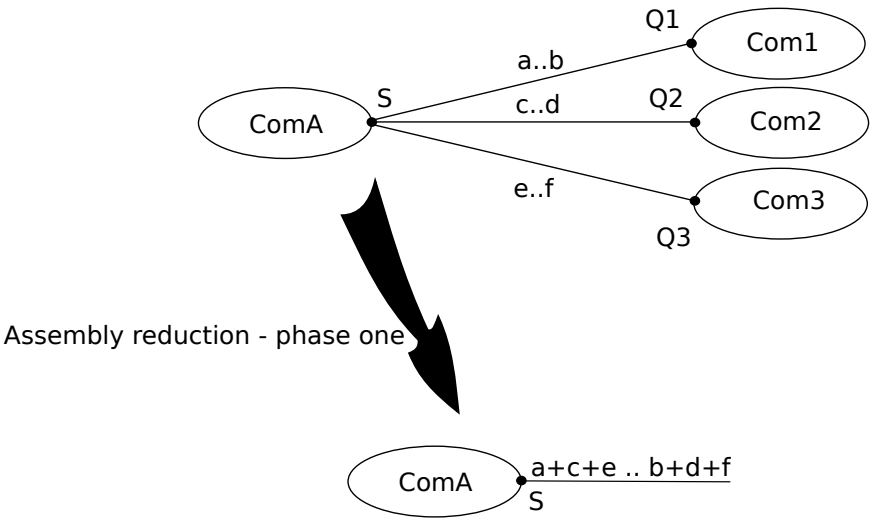


Figure 7.9: Up: CC assembly normal form B. Multiple connections - role `S` is connected to three roles `Q1`, `Q2`, and `Q3`. Bottom: Role `S` after assembly reduction - phase one.

instances of `S` (legal situation), and we need `i` connections to `S`. This situation will produce runtime error. As result, before asking for `i` connections to `S`, the application must instantiate at least `i` instances of `S` (possible because $i \leq f$).

3. Error: here we do not have enough instances of the role to satisfy the minimum connections need.

7.2.4.2 Assembly reduction - phase one

A role (specifically, role type) is not limited to be connected to only one other role, rather, this number is unlimited. At runtime, this role is expected to have several instances, where each instance is connected to one other role. This is assembly normal form B and presented in figure 7.9 (up). To check this assembly we need to get it back to normal form A in figure 7.7. We call this conversion

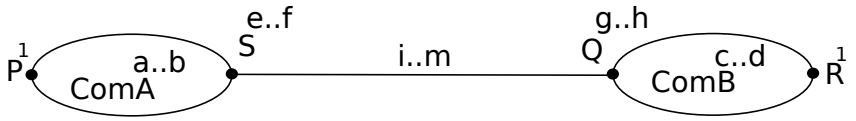


Figure 7.10: CC assembly normal form C. Other CCs can connect to Q, S, etc. Omitted for space.

from the form normal form B to normal form A: assembly reduction - phase one. For role `S` in figure 7.9 this is accomplished as in figure 7.9 (bottom). Formally: Let $\sigma = \{(S \otimes Q_1, a_1, b_1), (S \otimes Q_2, a_2, b_2), \cdots, (S \otimes Q_n, a_n, b_n)\}$. After assembly reduction - phase one, we get: $\sigma = \{(S, Q, a, b)\}$ such that: $a = \sum_{i=1}^{n} a_i$, $b = \sum_{i=1}^{n} b_i$, and Q is virtual role for checking only. This is for role S only and must be done for all other roles that have connections to more than one role.

7.2.4.3 CC assembly normal form C

Figure 7.10 presents CC assembly normal form C. The connection here has the following semantics: at least `i` instance of `S(Q)` connect to `i` instance of `Q(S)`, and at most `m` instance of `S(Q)` connect to `m` instance of `Q(S)`. This is correct for one and only one instance of each cloud component. Moreover, CCs `ComA` and `ComB` have multiplicities $[a..b]$ and $[c..d]$ respectively.

Because of the multiplicities of the CCs, we are unable to use the checking procedure used for normal form A directly on normal form C. To be able to check this assembly, we will follow several assembly reductions starting from this general model. Namely, we will apply assembly reduction phase two and then assembly reduction phase three.

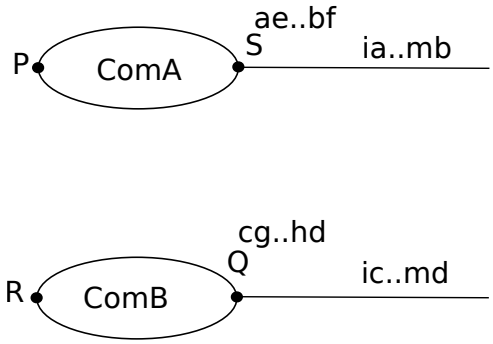


Figure 7.11: Connection multiplicity.

7.2.4.4 Assembly reduction - phase two

Assembly reduction phase two reduces the multiplicity of the CC to be incorporated (inserted) into the multiplicities of its roles. Formally, let:

$\Omega ComA \equiv (\{\Lambda P, \Lambda S\}, \{(\Lambda P, 1), (\Lambda S, e, f)\}, \overline{L}, Z)$

and

$\Omega ComB \equiv (\{\Lambda Q, \Lambda R\}, \{(\Lambda Q, g, h), (\Lambda R, 1)\}, \overline{L}, Z).$

$\sigma = \{(\Omega ComA.\Lambda S \otimes \Omega ComB.\Lambda Q, i, m)\}.$

Now let:

$S \equiv (\{\Omega ComA, \Omega ComB\}, \{(\Omega ComA, a, b), (\Omega ComB, c, d)\}, \sigma, \overline{L}).$

Assembly reduction phase two produces the new multiplicities for all roles:

$\overline{\mu_{ComA}} = \{(\Lambda P, a, b), (\Lambda S, ae, bf)\}$

and

$\overline{\mu_{ComB}} = \{(\Lambda Q, cg, hd), (\Lambda R, c, d)\}.$

7.2.4.5 Assembly reduction - phase three

Assembly reduction phase three is trickier. The multiplicity of the connection $[i..m]$ is affected by both CC's multiplicities, namely $[a..b]$ and $[c..d]$. The objective of this phase is to end up with the connection multiplicity $[x..y]$.

In figure 7.11 We see the effect of the multiplicity of each CC over the connection multiplicity. In this figure we have two different ranges (multiplicities).

To end up with the connection multiplicity $[x..y]$ we use the approach presented in figure 7.12. In this figure all possible cases of the two ranges in figure 7.11 are presented with arrows to show the final x and y. From this figure we can conclude this simple rule: 'for x we choose the max of the mins, and for y we choose the max of the maxs'. Formally: $x = max\{ia,\ ic\}$, and $y = max\{mb,\ md\}$. By the end of this phase we will get back to normal form A that can be checked directly as in figure 7.13.

7.2.4.6 Inclusive algorithm

To convert this assembly theory into an effective tool, we arranged the above mentioned phases into one inclusive algorithm as in figure 7.14. This algorithm accepts CC-based systems in normal form C as an input. And it generates a list of notifications based on figure 7.8. This algorithm starts with integrity checks: `check_1 .. check_5.` These checks ensure that the input files are a formal CC-based system, in normal form C, and with valid parameters [6].

The algorithm in figure 7.14 is fully implemented using C programming lan-

6. For example: the same role name can not exist twice. Also, a range $a..b$ means $a \leqslant b$. If $a > b$ the algorithm aborts.

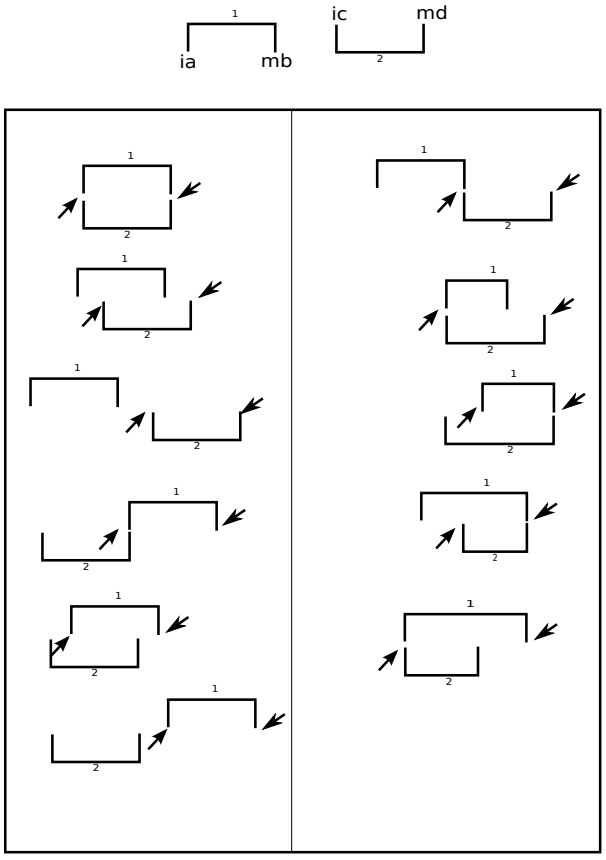


Figure 7.12: Listing of all cases of possible connection multiplicities in assembly reduction - phase two

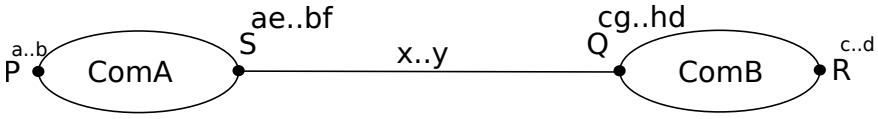


Figure 7.13: The result after reduction phase two and three on figure 7.10- CC multiplicities are completely removed.

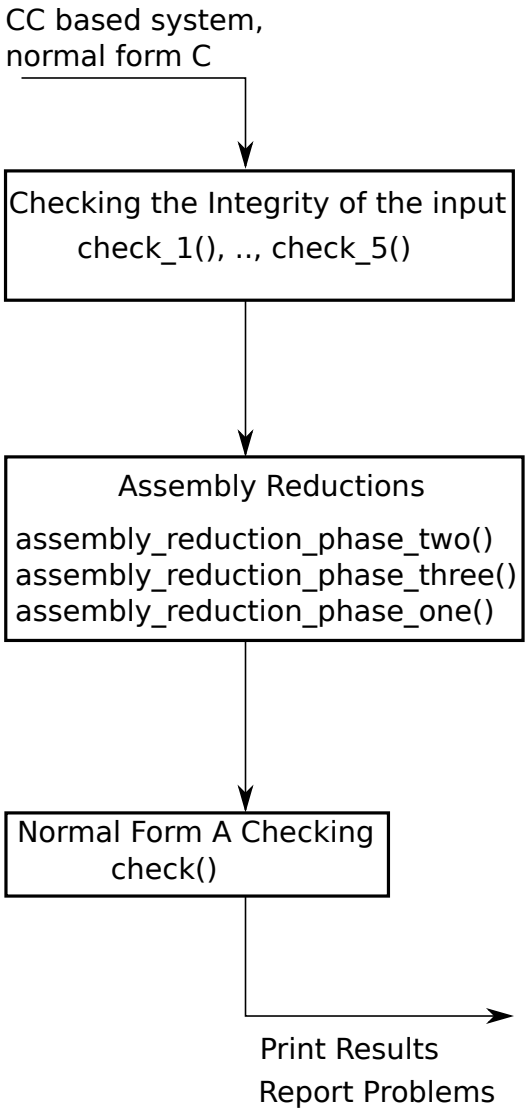


Figure 7.14: Inclusive checking algorithm. The integrity checks, namely, check1() through check5(), ensure that the input is not corrupted with respect to normal form C.

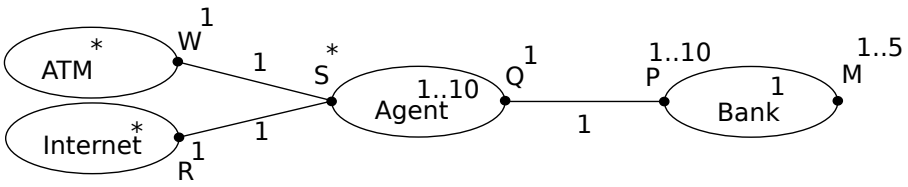


Figure 7.15: The banking system in normal form C - Enterprise Edition.

guage along with Lex and YACC utilities. Also, it is used to check the banking system example in the following section.

7.2.5 Example - Banking System

In this section we present a simple banking system to explain the algorithm proposed in section 7.2. The banking example is presented in figure 7.15. The $*$ symbol can be reduced to `[0..MAXINT]` for computations.

In this example the system is built using four CCs. The `Bank` CC is responsible for all database systems, security, transactions, and accounts. It is basically the backbone of the system. The `Agent` CC is the filter that any access to `Bank` will pass through. In other words, nobody can directly access `Bank` CC. `ATM` CC is installed over all ATM machines to allow customers to access their accounts, and perform bank transactions. Similarly, `Internet` CC is installed on the customers devices to allow them to access their accounts using Internet banking.

We encoded this example using the formal language presented previously, and used the automatic assembly checker to check the design. The assembly checker generated the output presented in figure 7.16. The checker reports expected warnings and no errors. The two warnings in this figure are related to

```
Warning!!!!
Problem type 1.2
Conncetion to P.
This connection is not safe because it is dependent on the max kardinality.
Warning!!!!
Problem type 1.2
Conncetion to Q.
This connection is not safe because it is dependent on the max kardinality.

The design has potential problems. Please read messages.
```

Figure 7.16: The output generated by the assembly checker (partial output) for the banking system - Enterprise Edition.

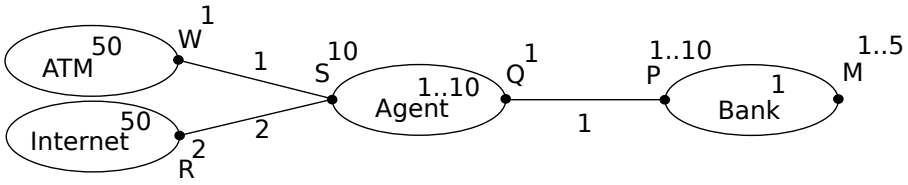


Figure 7.17: The banking system in normal form C. Limited Edition.

```
Warning!!!!
Problem type 1.2
Conncetion to P.
This connection is not safe because it is dependent on the max kardinality.
Warning!!!!
Problem type 1.2
Conncetion to Q.
This connection is not safe because it is dependent on the max kardinality.
Error!!!! Conncetion to S.
This connection is illegal because it is exceeds the max kardinality.

The design has major errors. Please read messages.
```

Figure 7.18: The output generated by the assembly checker (partial output) for the banking system - Limited Edition

the connection $Q \otimes P$. If we look at this connection we see that the initial deployment could be one instance of ΛP, one instance of $\Omega Agent$, and one instance of ΛQ. And an assembly between ΓQ and ΓP. This is valid. Now if the deployment plan has an order of another instantiation of $\Omega Agent$ without second instance of ΛP, a crash might occur. CCMS (the tool responsible of the deployment of all CC-based systems) will not complain before the crash because it is legitimate to instantiate up to 10 instances of $\Omega Agent$. For more discussion about these details please refer to section 9.

We modified the banking system slightly in figure 7.17. For this system, the checker produces the output in figure 7.18. The error reported in this figure is due to the fact that the number of instances of `ATM` and `Internet` CCs, and consequently, `W` and `R` roles, exceeds the multiplicity of the role `S`.

7.2.6 The Deployment Conjecture

7.2.6.1 The Conjecture Statement

If a cloud-component-based design (in general, the CC system is in normal form C) passes the assembly checker (figure 7.14) without errors, then we guarantee there exists a deployment plan (section 9.1.7) that can deploy the whole system without any runtime error. On the contrary, if the checker in section 7.2.4 generates error(s), then there does not exist a deployment plan for such a system.

7.2.6.2 Comments

1. Warnings do not affect the deployment conjecture. Warnings simply means that the design has a potential problem at runtime if the deployment plan does not carefully order instantiation of CCs and roles.
2. We think that the formal proof of this conjecture is out of the scope of this dissertation. The reason is that we wanted to have a certain level of balance between software engineering work and formal methods work.
3. As an explanatory argument we would like to state the following:
   The system in figure 7.17 generates an error if passed to the assembly checker. We can see clearly that there is no possibility to correctly deploy this system. The initial deployment requires a deployment of 50 instances of ΩATM and 50 instances of $\Omega Internet$. This immediately results 150 connection to ΛS. However, we have at most 100 instances of ΛS at runtime. Since in CC assembly model we have one-to-one connections, this means that we have 50 roles will be unable to connect. This is a direct violation of the model in figure 7.17.

7.3 Cloud component Development Process

While different methodologies consider different phases for software life cycle, we use the phases of software life cycle as defined in [20, 21] to build our novel (customized) cloud component development process. CC is a building block in CC based systems. In this section, we concentrate on the development process of a single CC. We propose the following six-stages, iterative, and incremental software development process:

7.3.1 Stage One - Specifications

At this point we have:

1. The CC specification which includes the required functionality and the required QoS.
2. The expected deployment environment(s).

The CC specification is written in English as default. If the application requires special specification language, it could be used. On the other hand, The expected deployment environment has a formal notation.

$$\overline{L} = \{L_1,\ L_2 \ldots, L_n\}$$

$$L_i = \{TLocationName_1, TLocationName_2, \ldots, TLocationName_n\}$$

This means that, in general, the expected deployment environment is not unique. In other words, we can expect several scenarios of deploying the CC. Each expected deployment environment L_i is a set of location types. Location type is used at design time, while host name is used at deployment time. As result, we will have the following notation at deployment time:

$$L_i = \{HDeviceName_1, HDeviceName_2, \ldots, HDeviceName_n\}$$

We used the prefix `T` and the prefix `H` to denote type and device ID respectively. The difference is not fundamental, however it is important to clarify the distinction for the designer. Any name the has the prefix `T` (or the prefix `H`) is defined as an instance in an ontology file (.obl file). Please read section 8.5.1.

CC model moves one step forward with regard to location, and defines the function Z. The function Z maps CC roles to location types.

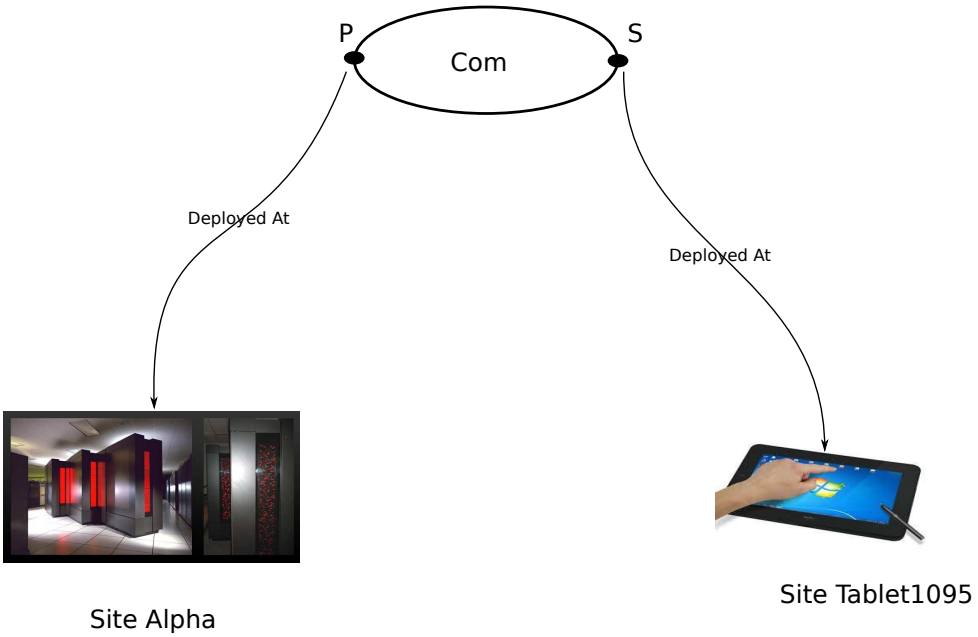


Figure 7.19: Graphical view of the following formal localization:
$Z: \ \Lambda P \downarrow TAlpha, \ \ \Lambda S \downarrow TTablet1095$.

$$Z: \ \Lambda Role_i \downarrow TLocationName_k, \ \ \Lambda Role_j \downarrow TLocationName_l, \ \ ...$$

It is very important here to mention that we define networks in our model by defining endpoints. In other words, in the ontology definition of a device or a type of devices, we precisely define its networking capabilities, as in figure 7.19. In this figure we have the following deployment function:

$Z: \ \Lambda P \downarrow TAlpha, \ \ \Lambda S \downarrow TTablet1095$

Where both $Alpha$ and $Tablet1095$ are precisely defined in an ontology file. Their ontology definition includes precise definition of their expected runtime networking capabilities. Now, if there is a kind of communication between roles ΛP and ΛS, then the designer can design and implement the software for these two parts based on the expected runtime networking capabilities of these two

sites. This is sufficient, as long as the expected characteristics are respected at runtime.

To summarize the relation between a CC and the expected deployment environments, the CC could be imagined to be deployed over several different deployment environments. Moreover, different parts of the CC (roles, internal BDUs [7]) can be deployed over different parts of a single deployment environment.

This stage is to ensure the completeness and the correctness of the two items listed at the beginning of this stage. This includes all documents in English, all related formal files, and all related ontology files [8].

7.3.2 Stage Two - Localization Choice

As we can see from stage one above, a single cloud component needs to be designed and implemented to be compatible with large variation of deployment environments. These deployment environments differ from each other significantly. In figure 7.20 the cloud component ΩCom need to be deployed over the deployment environment L. To be able to proceed, we need to eliminate all variations. In other words, we should not have more than one target device for any role or internal BDU. In figure 7.21 we have role ΛP to be deployed over target of type $TDesktop355$. Also, role ΛS has to be deployed over target of type $TDesktop355$. And we have location of type $TAlpha$ available for the designer to be used for internal BDUs.

It is clear that we could have chosen different option such as role ΛS has to be deployed over target of type $TTablet1095$. But this is not important, since

7. Defined in section 7.3.4 and discussed also in chapter 9

8. For better comprehension with respect to these files it is important to read part III completely.

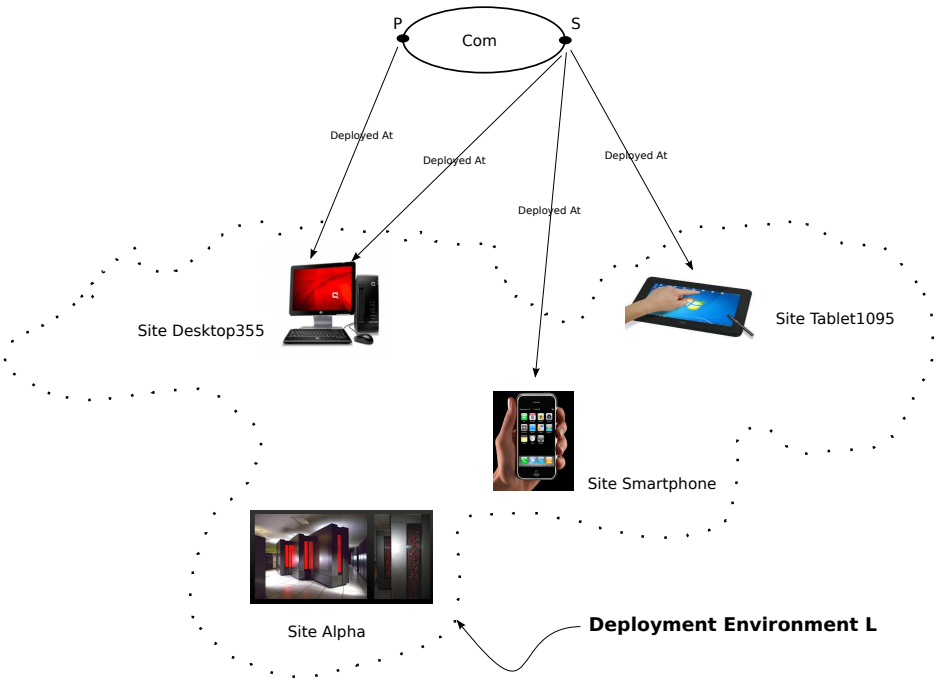


Figure 7.20: The cloud component ΩCom along with its expected deployment environment L. Role ΛS has to be deployed over several different types of devices. Location type $TAlpha$ is part of the deployment environment where internal BDUs are expected to be deployed over it.

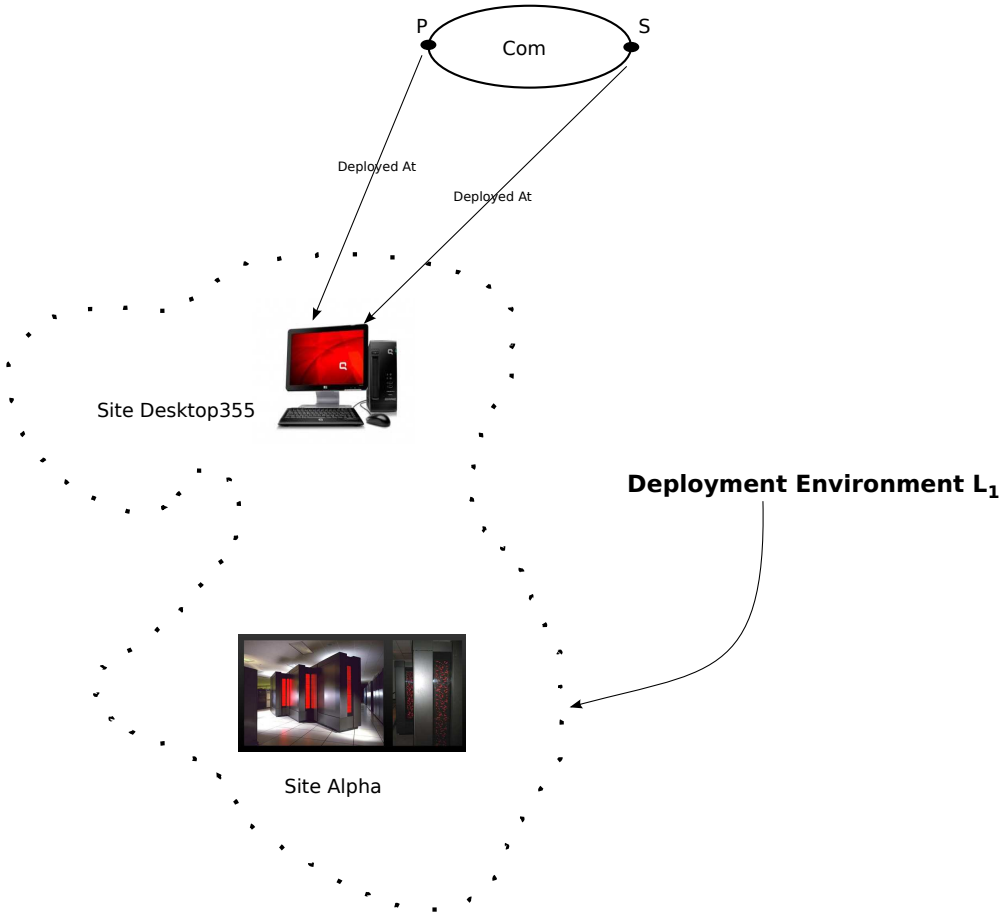


Figure 7.21: In this figure, the localization of border BDUs is fixed, but the localization of internal BDUs is free.

this is an iterative process. As result, we will be back to this stage, stage two, over and over until we had enumerated all possible options.

7.3.3 Stage Three - Package View

In this stage we use the divide and conquer approach for the second time after we used it for the first time in decomposing the original system into assembly of cloud components. For more information about divide and conquer approach please read section 6.1.1. In this stage the cloud component is decomposed into a finite set of packages. Packages generated are dependent on each other. A package is purely conceptual (meaning that is exists only at development time), this is the contrary to BDUs which exist at runtime (BDUs are discussed in stage four). The decomposition in this stage is usually a functional decomposition as in the architecture phase in usual software development process. All diagrams and formal descriptions that result in this stage can be called the 'package level' or the 'package view' of the cloud component development process.

The merit of this stage is evident when the functionality of the cloud component is complex and can not be comprehended and tackled as a single entity, as in figure 7.22. Normally, this is the case. A cloud component is not simple software that could be translated into a simple code. Rather, it is usually a complex structure. This phase is important to tackle this complexity using divide and conquer approach. In this stage the creativity of the designer controls all decisions he/she makes. The cloud component model does not force any guidelines in this regard.

Unlike [141] where they use 'package' as a basic deployment unit in free and open source distributions, we use 'package' as a purely conceptual design object

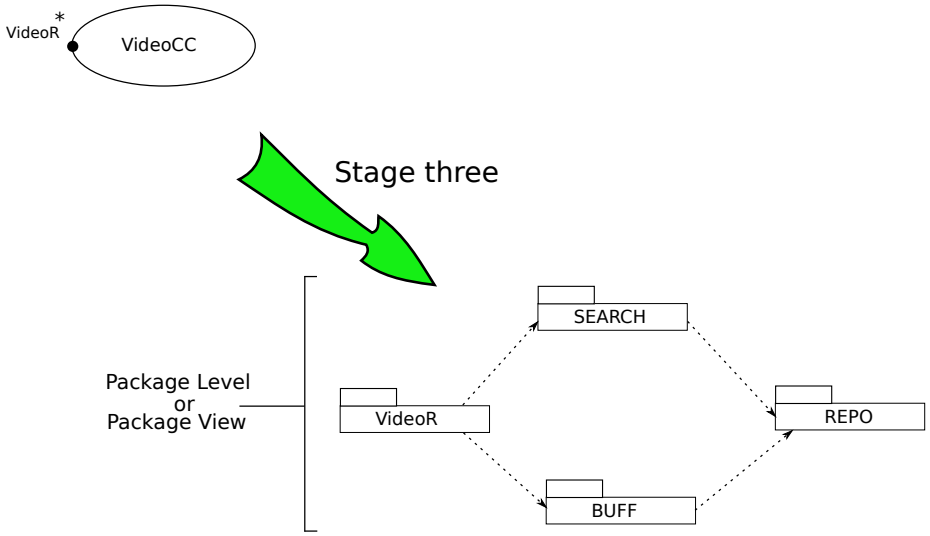


Figure 7.22: Stage three of the CC development process.

(a package does not exist at runtime). This definition of package (our definition) is widely accepted and used in software engineering communities. We illustrate this design process in the next section.

7.3.4 Stage Four - BDU View

Each package is realized through finite set of *Basic Deployment Units (BDUs)*. BDUs generated are dependent on each other. Basic Deployment Unit is a tangible (physical) artifact that is deployable and executable. The BDU is the smallest architectural unit possible in a CC design. It is not nested (also CC is not nested). It is local with respect to the deployment device (in other words a BDU can not be distributed). All diagrams and formal descriptions that result in this stage can be called the 'BDU level' or the 'BDU view' of the cloud component development process.

Each BDU is required to have an ontology based description of its deployment

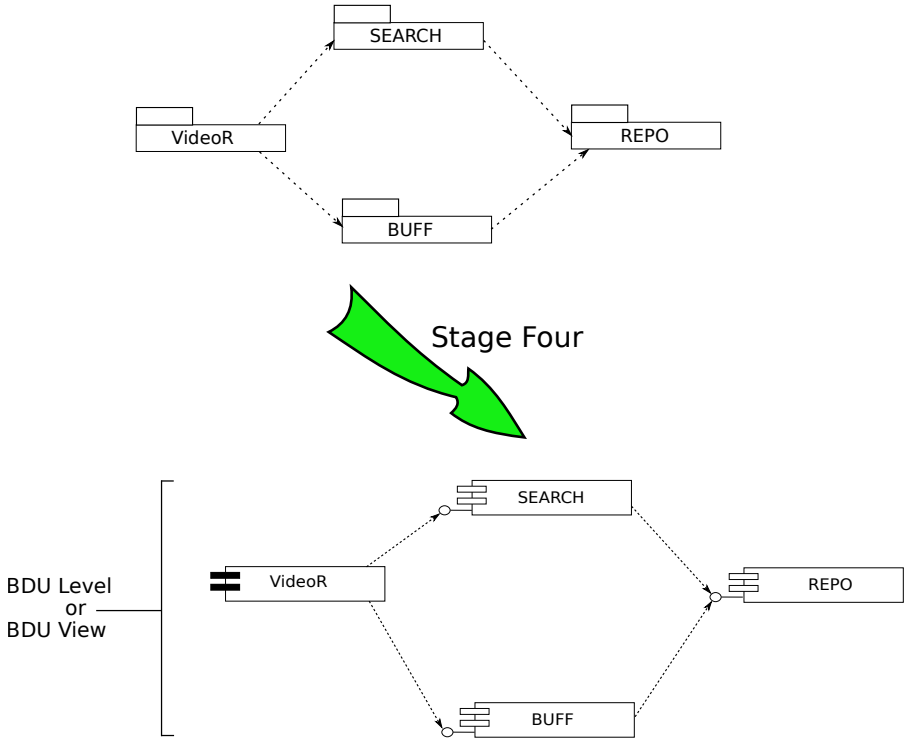


Figure 7.23: Stage four of the CC development process. A BDU with black-filled left side represents a role BDU (ie. on the CC border).

requirements. This ontology description will be used before deployment by the software/hardware checker. In figure 7.21 we have role ΛP to be deployed at site $TDesktop355$. By the end of stage four we will have a BDU (at least one) which is an implementation of this role ΛP. This BDU is compatible with site $TDesktop355$. However, the ontology that describes site $TDesktop355$ is not the ontology that describes the deployment requirements of ΛP. Again, the deployment requirements of each BDU in an otology form is the responsibility of the software engineer who implemented the BDU.

7.3.5 Stage Five - BDU Localization

Each BDU generated in stage four is mapped to an expected host of the expected deployment environment that was specified in stage two in the CC development process. This step is called: *localization*. By the end of this step, we should be able to have a cloud component that is compatible with the deployment environment that was specified in stage two in the CC development process.

7.3.6 Stage Six - Iteration

We choose the next expected deployment environment in stage two, and repeat stages three to five. We continue this loop until we have an implementation variant for each expected deployment environment described in the definition of the CC (stage one).

7.3.7 Formal Notation for the CC Software Development Process:

The six stages presented above, along with the figures are human readable. We propose a formal notation to describe these steps. This formal notation is accurate, compact, and most important: *machine readable*. This formal notation will be read by tools (such as cloud component management system CCMS) in order to perform required operations over the cloud component system. In this section we will provide the formal notation that describes this development process by presenting the formal description of each stage. Precisely, we will formally describe figures: 7.22, 7.23, 7.24.

Figure 7.22:

$\Omega VideoCC \rightsquigarrow\rightsquigarrow \overline{\Theta}_{Vid} = \{\Theta Search,\ \Theta VideoR,\ \Theta Repo,\ \Theta Buff\}$

$\omega(\overline{\Theta}_{Vid}) = \{\Theta VideoR \longrightarrow \Theta Search, \Theta VideoR \longrightarrow \Theta BUFF,$

$\Theta Search \longrightarrow \Theta REPO,\ \Theta BUFF \longrightarrow \Theta REPO,\ \}$

Figure 7.23:

$\Theta Search \rightsquigarrow\rightsquigarrow \Upsilon Search = \Phi Search$

$\Theta REPO \rightsquigarrow\rightsquigarrow \Upsilon REPO = \Phi REPO$

$\Theta BUFF \rightsquigarrow\rightsquigarrow \Upsilon BUFF = \Phi BUFF$

$\Theta VideoR \rightsquigarrow\rightsquigarrow \Upsilon VideoR = \Xi VideoR$

$\overline{\Upsilon}_{Vid} = \{\Upsilon VideoR,\ \Upsilon Search,\ \Upsilon REPO,\ \Upsilon BUFF\}$

$\overline{\Upsilon}_{Vid} = \{\Xi VideoR,\ \Phi Search,\ \Phi REPO,\ \Phi BUFF\}$

$\rho(\overline{\Upsilon}_{Vid}) = \{\Xi VideoR \longrightarrow \overline{\Upsilon}\{\Phi Search,\ \Phi BUFF\},$

$\overline{\Upsilon}\{\Phi Search,\ \Phi BUFF\} \longrightarrow \Phi REPO\}$

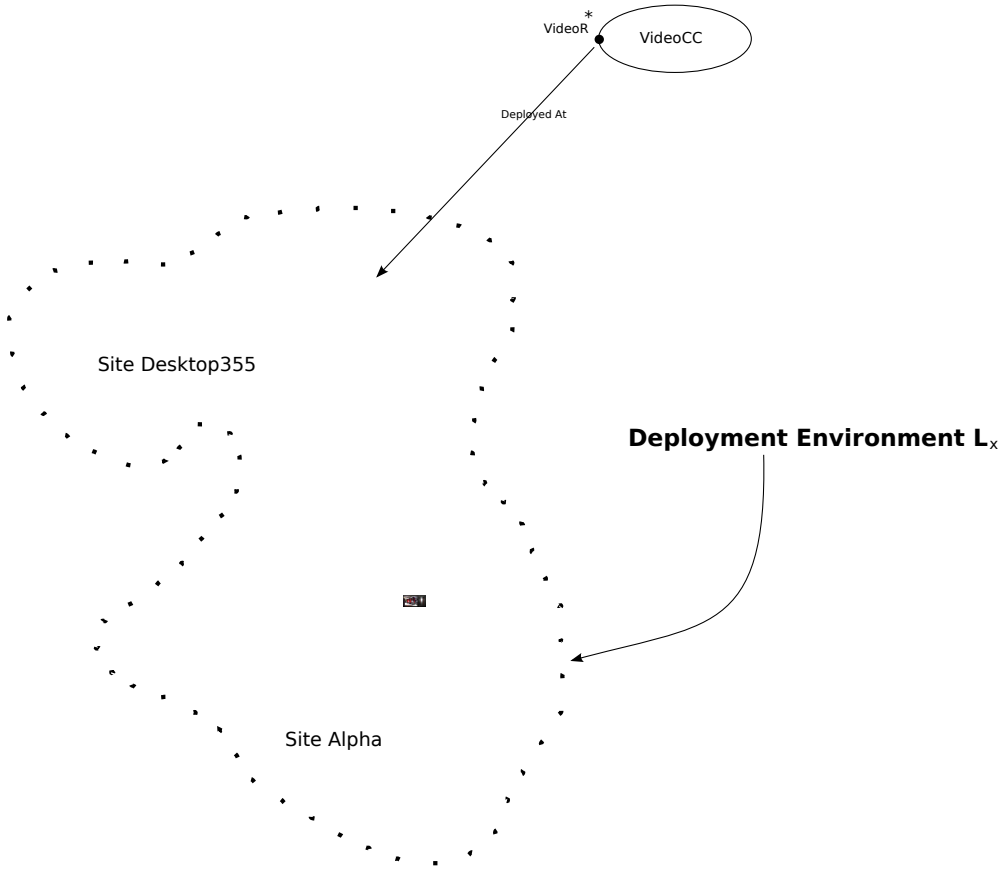


Figure 7.24: One localization option of CC $\Omega VideoCC$.

Figure 7.24:

$\overline{\Upsilon}_{Vid} \downarrow \overline{T}L_x$

Localization:

$\Xi VideoR \downarrow TDesktop355$

$\Phi Search \downarrow TAlpha$

$\Phi BUFF \downarrow TAlpha$

$\Phi REPO \downarrow TAlpha$

Or:

$\overline{\Upsilon}coms_1 = \{\Phi REPO,\ \Phi Search,\ \Phi BUFF\}$

$\overline{\Upsilon}coms_1 \downarrow TAlpha$

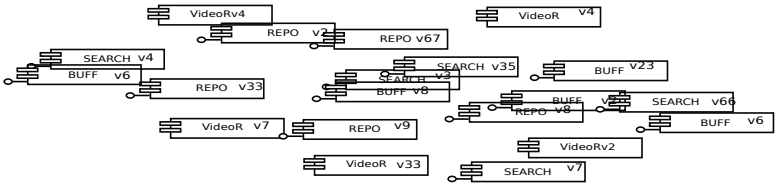
$\Xi VideoR \downarrow TDesktop355$

7.3.8 Software Complexity Management

This kind of software development results a complexity that grows fast enough to be unmanageable, even with such simple application, please see figure 7.25 (up). Moreover, it is possible to have several instances of each CC at run-time. As result, the complexity of deployment and run-time management becomes prohibitive. This is where the potential of our novel approach becomes evident. The border of a cloud component encapsulates all implementation details as in figure 7.25 (middle) and enables to consider all variants as a single unit of management. For large systems, the power of CC model becomes essential, where the system requires the assembly of many cloud components as in figure 7.25 (bottom). The deployment and runtime management of such systems over thousands of heterogeneous mobile devices, where at the same time we guarantee software/hardware compatibility and QoS, is the merit of our proposal.

This model allows us to provide tools that ease the following activities:

- The automatic deployment of cloud component based systems using CCMS. Please read chapter 9.
- Automatic software/hardware compatibility check at deployment time. This check allows CCMS to choose the right implementation variant for each deployment device. Please read chapter 8.



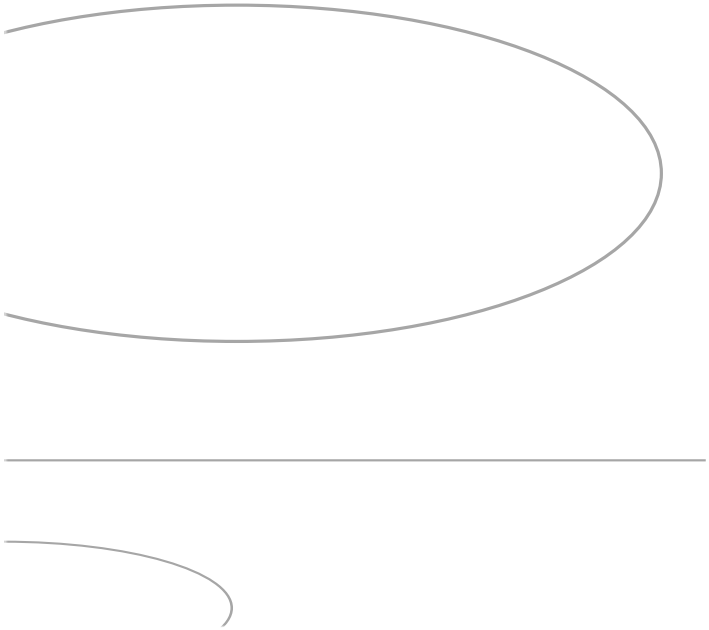


Figure 7.25: The *encapsulation power* of CCs. The result of software development process can be unmanageable (up). On the other hand, CC approach with the CC border makes this management handy and natural (middle). In this figure we wanted to emphasize the general case where a single CC is part of a CC-based system (bottom).

– Run-time system management using CCMS.

7.4 Case Study - $\Omega VideoCC$ Implementation

The purpose of MULTIMEDIA application [9] is to be a single application to store, search, process, and play all multimedia files like pictures/images, music, and video. This application is expected to be deployed over a highly distributed platform. Video streaming is one service of MULTIMEDIA application, and it is a very popular service for most PC, laptop, and smartphone users.

This service has its potential because the user does not have to wait for a full download. He or she can start watching the video immediately after they click the related button. This promise is possible, however, we need to be very precise when handling details. The size of a video file is affected by many factors including (but not limited to) resolution and frame rate. Table 7.2 presents several common streaming videos and their corresponding required transfer rate. It is very important to maintain data transfer above the listed minimum transfer rate, otherwise the user will suffer undesired pauses during playback, which reduces the QoS expected. Another important factor is to notice that the numbers presented in this table are empirical. Numbers such as ISP theoretical bandwidth or expected data rates are completely irrelevant.

Tests in this section are performed over a deployment environment with the following hosts [10]:

– A normal server. This server uses fixed connection (Fast Ethernet).

– set_1: a set of laptops with Wi-Fi connections (and possible 3G connec-

9. Also mentioned in section 9.4
10. Same environment used in section 9.4.

Resolution	**Frame rate** (frame/sec)	**Video size for one minute** (MByte)	**Min transfer rate** (KByte/sec)
un-named 480x270	30	4.25	72.57
360p	30	6.62	113.00
480p	30	8.60	146.80
un-named 320x200	25	1.01	17.24
480p	15	1.53	26.13
480p (audio)	15	1.75	29.87
720p	15	2.78	47.47

Table 7.2: Empirical characteristics of common available videos for streaming. The size may differ due to embedded audio, differing frame sizes and aspect ratios, and inter-frame compression. The audio of entry six has a higher quality than the audio of entry five.

tions).

– set_2: a set of smart-phones with android operating system and 3G and Wi-Fi connections (we choose which connection to use based on the test wanted).

7.4.1 Using CC to build Multimedia Application

To build any application using CCs, the starting point always is to define *the expected deployment environment.* The expected deployment environment for our application consists of a server, which can be any powerful desktop, a normal desktop, a normal laptop, and a smartphone. We can think of a deployment environment which include any number of these devices, all together. Or simply, one server and one laptop. We can not have a deployment environment with one laptop and one smartphone only, a server must exists [11].

11. Formally defined in section 7.4.4 - part A.

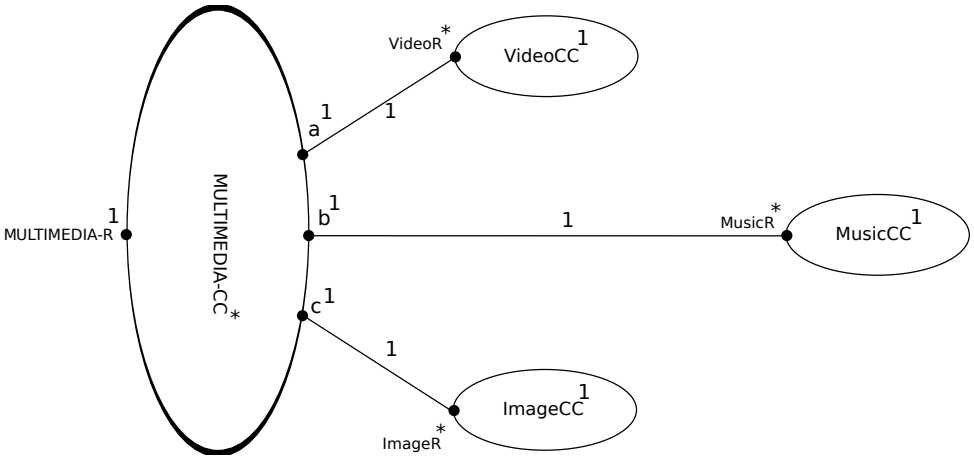


Figure 7.26: MULTIMEDIA application. Cloud component view.

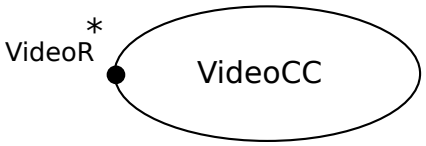


Figure 7.27: VideoCC with its single role VideoR.

In addition, all servers and desktops have Fast Ethernet while laptops and smartphones connect using Wi-Fi and 3G.

As an architecture choice the following cloud components will be used, as explained in figure 7.26:

1. Image cloud component - ImageCC
2. Music cloud component - MusicCC
3. Video cloud component - VideoCC
4. General CC that is used to access the functionality of the whole application: MULTIMEDIACC

ImageCC, MusicCC, and VideoCC are responsible for the functionalities related

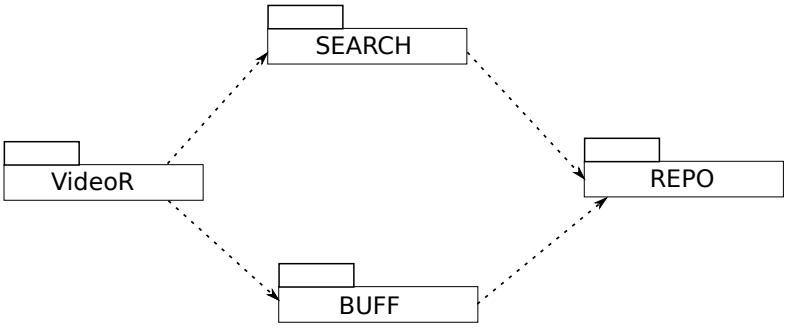


Figure 7.28: The package level of VideoCC.

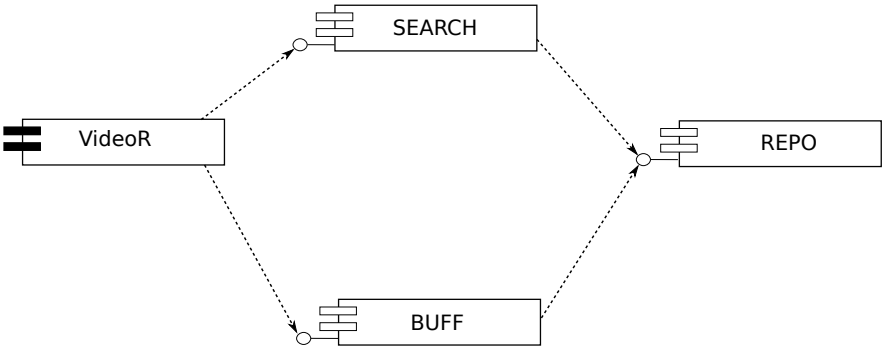


Figure 7.29: The BDU level of VideoCC for deployment environment with desktops and laptops. A BDU with black-filled left side represents a role BDU (ie. on the CC border).

to images, music, and videos respectively [12]. We will present the development of $VideoCC$ only [13], please see figure 7.27. The other CCs are developed similarly.

7.4.2 VideoCC Development for Desktops and Laptops

We start with a simple design and implementation where the user is using a laptop and/or a desktop. Package decomposition of $VideoCC$ is presented graphically in figure 7.28. In this figure we can see that this cloud component can be accessed through its only role $VideoR$. There is a package called $REPO$ which is responsible of archiving all videos. Another two packages are for searching and buffering [14].

BDU level of $VideoCC$ is presented graphically in figure 7.29. Here we can see a direct implementation of each package as a single BDU. Also, $REPO$ is a simple and localized (not distributed) archive [15].

Timing results are presented in figure 7.30. From this figure we see that the throughput is 646.0 KBytes/second and 454.7 KBytes/second for Wi-Fi and 3G respectively. Using table 7.2 we can see that this throughput is safe for streaming all videos presented in that table.

7.4.3 VideoCC Development for Smartphones

We move on to allow the role to be deployed on a smartphone. We port the same previous design, the only difference is that we need to change the programming language from C to Java for $VideoR$ only. Unfortunately, we faced

12. Formally defined in section 7.4.4 - part B.
13. Formally defined in section 7.4.4 - part C.
14. Formally defined in section 7.4.4 - part D.
15. Formally defined in section 7.4.4 - part E.

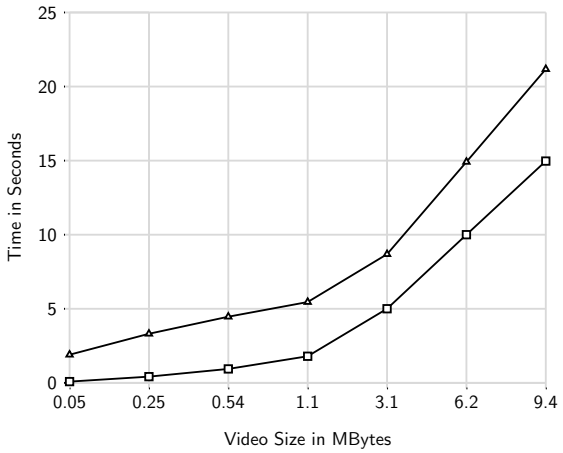


Figure 7.30: Experimental results for streaming several videos where the role is deployed on a laptop. The curve with square nodes is for the laptop with Wi-Fi connection, while the curve with triangle nodes is for the laptop with 3G connection.

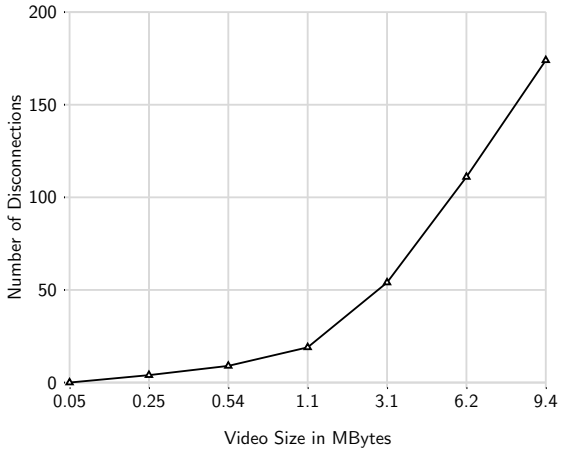


Figure 7.31: Experimental results show the total number of disconnected operation during video streaming when the role is deployed over a smartphone with 3G connection. Ideally, this number should be zero.

an unexpected problem during runtime. The software operates as expected if the smartphone is connected via Wi-Fi, and crashes if it is connected via 3G. This is clear disconnected operation fault as in figure 7.31. We decided that we need to modify the design to allow resume after crash. In other words, if the connection between the smartphone and the video provider is broken, the system re-establishes the connection peacefully, from the last saved state. Timing results are presented in figure 7.32. From this figure we see that the throughput is 568.2 KBytes/second and 15.21 KBytes/second for Wi-Fi and 3G respectively. Using table 7.2 we can see that the throughput of Wi-Fi on smartphone is safe for streaming all videos presented in that table. On the contrary, the throughput of 3G on smartphone is not sufficient for streaming any video presented in that table. To solve this problem, we moved one step ahead and changed the design to allow the user to connect to multiple video-source at the same time [16]. We call this: *multi-channel video streaming*. The graphical representation of this new architecture is presented in figure 7.33.

Timing results are presented in figure 7.34. From this figure we see that the throughput is 19.57 KBytes/second. Using this throughput, we succeeded to stream one video from table 7.2 on the smartphone that is connected using 3G without any pauses. This result was not possible using the previous designs and implementations.

16. Formally defined in section 7.4.4 - part F.

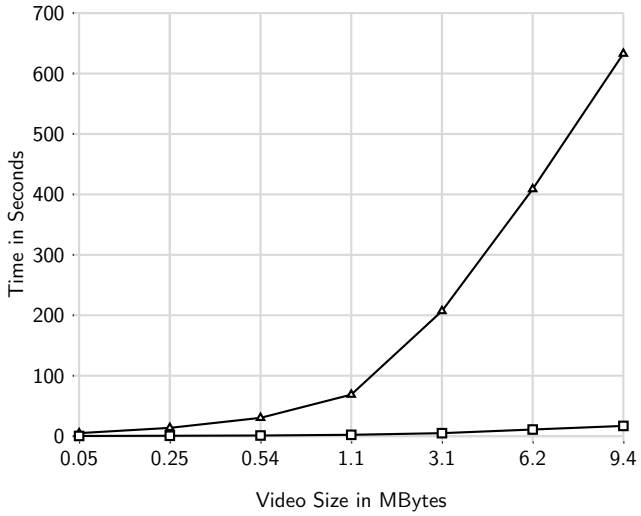


Figure 7.32: Experimental results for streaming several videos where the role is deployed on a smartphone. The algorithm has the *resume-support* feature. The curve with square nodes is for the smartphone with Wi-Fi connection, while the curve with triangle nodes is for the smartphone with 3G connection.

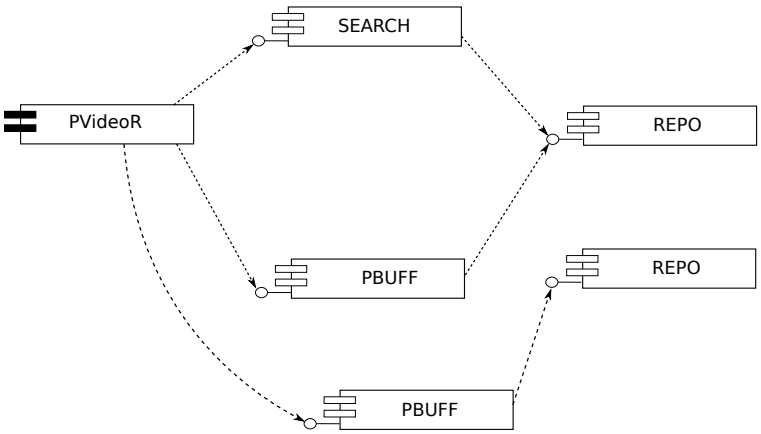


Figure 7.33: The BDU level of VideoCC. In this figure, the multi-channel video streaming architecture is presented. This architecture will allow parallel streaming of a single video. A BDU with black-filled left side represents a role BDU.

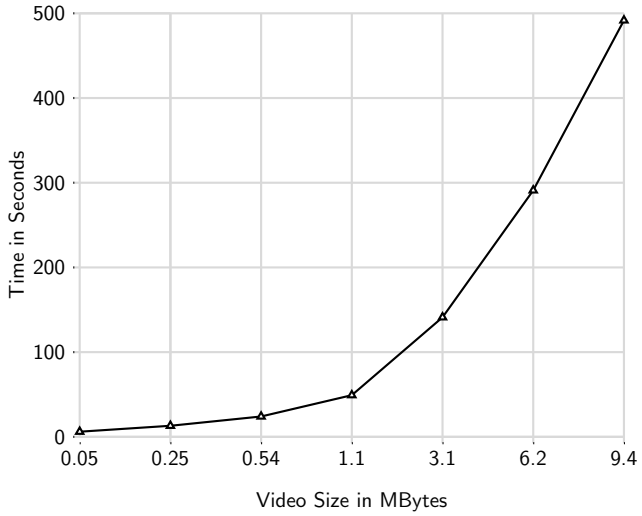


Figure 7.34: Experimental results for streaming several videos where the role is deployed on a smartphone with a 3G connection and utilizing the multi-channel video streaming technique.

7.4.4 Formal Language Description of Multimedia Devlopment Process

In this section we provide the formal description of the development process described in section 7.4.1. Again, this formal description is important for many reasons, one of them is that it is machine readable, and allows CCMS [17] to execute the deployment plan.

We will present this formal description divided into parts, in correspondence to section 7.4.1.

Part A:

$L_1 = \{Tserver, Tdesktop\}$

17. Chapter 9.

$L_2 = \{Tserver,\ Tlaptop\}$

$L_3 = \{Tserver,\ Tsmartphone\}$

$\overline{L} = \{L_1,\ L_2,\ L_3\}$

Part B:

The formal definition of the MULTIMEDIA application is presented as follows:

$MULTIMEDIA \equiv (\overline{\Omega},\ \overline{M}\ \sigma,\ \overline{L})$

where:

– Set of roles:

$\overline{\Omega} \equiv \{\Omega ImageCC,\ \Omega MusicCC,\ \Omega VideoCC,\ \Omega MULTIMEDIACC\}$

– Multiplicities:

$\overline{M} = \{(\Omega MULTIMEDIACC, *), (\Omega VideoCC, 1),$

$(\Omega MusicCC, 1), (\Omega ImageCC, 1)\}$

– Assembly:

$\sigma = \{(\Omega VideoCC.\Lambda VideoR \otimes \Omega MULTIMEDIACC.\Lambda a, 1, 1),$

$(\Omega MusicCC.\Lambda MusicR \otimes \Omega MULTIMEDIACC.\Lambda b, 1, 1),$

$(\Omega ImageCC.\Lambda ImageR \otimes \Omega MULTIMEDIACC.\Lambda c, 1, 1)\}$

Part C:

$\Omega VideoCC \equiv (\overline{\Lambda}_{VideoCC},\ \overline{\mu}_{VideoCC},\ \overline{L},\ Z_{VideoCC})$

$\overline{\Lambda}_{VideoCC} = \{\Lambda VideoR\}$

$\overline{\mu}_{VideoCC} = \{(\Lambda VideoR, *)\}$

Part D:

$\Omega VideoCC \rightsquigarrow\rightsquigarrow \overline{\Theta}\{\Theta Search,\ \Theta VideoR,$

$\Theta Repo,\ \ \Theta Buffer\}$

Package dependencies $\omega(\overline{\Theta})$ are:

$\Theta VideoR \longrightarrow \Theta Search$ and

$\Theta VideoR \longrightarrow \Theta BUFF$ and

$\Theta Search \longrightarrow \Theta REPO$ and

$\Theta BUFF \longrightarrow \Theta REPO$.

Part E:

$\Theta Search \rightsquigarrow\rightsquigarrow \Phi Search$ and

$\Theta REPO \rightsquigarrow\rightsquigarrow \Phi REPO$ and

$\Theta BUFF \rightsquigarrow\rightsquigarrow \Phi BUFF$ and

$\Theta VideoR \rightsquigarrow\rightsquigarrow \Xi VideoR$.

BDU dependencies $\rho(\overline{\Upsilon})$ are:

$\Xi VideoR \longrightarrow \overline{\Phi}\{\Phi Search, \Phi BUFF\}$ and

$\Phi Search \longrightarrow \Phi REPO$ and

$\Phi BUFF \longrightarrow \Phi REPO$.

Localization for L_1 are

$\Xi VideoR \downarrow Hdesktop$,

$\Phi Search \downarrow Hserver$,

$\Phi REPO \downarrow Hserver$,

$\Phi BUFF \downarrow Hserver$

Localization for L_2 are

$\Xi VideoR \downarrow Hlaptop$,

$\Phi Search \downarrow Hserver$,

$\Phi REPO \downarrow Hserver$,

$\Phi BUFF \downarrow Hserver$

Part F:

$\Theta Search \rightsquigarrow\rightsquigarrow \Phi Search$

$\Theta REPO \rightsquigarrow\rightsquigarrow \Phi REPO$

$\Theta BUFF \rightsquigarrow\rightsquigarrow \Phi PBUFF$

$\Theta VideoR \rightsquigarrow\rightsquigarrow \Xi PVideoR$

BDU dependencies $\rho(\overline{\Upsilon})$ for multi-channel video streaming architecture:

$\Xi PVideoR \longrightarrow \Phi Search$

$\Xi PVideoR \xrightarrow{*} [\Phi PBUFF]^*$

$\Phi Search \longrightarrow \Phi REPO$

$\Phi PBUFF \longrightarrow \Phi REPO$

One localization scenario for L_3 and multi-channel video streaming architecture:

$\Xi PVideoR \downarrow Hsmartphone$

$\Phi Search \downarrow Hserver_1$

$\Phi REPO \downarrow Hserver_1$

$\Phi PBUFF \downarrow Hserver_1$

$\Phi PBUFF \downarrow Hserver_2$

$\Phi REPO \downarrow Hserver_2$

7.4.5 QoS Support

Our claim that CC model guarantee QoS at the user endpoint is based on the following two points:

1. The CC software development process. This development process is customized to consider different expected deployment environments as a fundamental concern. As we have seen for the development of $\Omega VideoCC$, we have four implementation variants for the different deployment scenarios. These implementation variants do not offer any changed functionality, rather, they target maintaining the level of end-user-QoS above the threshold mentioned in table 7.2.
2. Software/Hardware checker (please read section 8.5.1). At deployment time, the cloud component management system - CCMS (Please read chapter 9) will invoke an ontology-based-checker to identify the target device (i.e. the device on which CCMS will deploy the BDU at). After that, CCMS will run an ontology query to find the implementation variant (among the installed list of variants) that is compatible with that device. This compatibility check includes the device characteristics, along with the networking available, etc.

7.4.6 $\Omega VideoCC$ Complexity Management

Back to section 7.3.8, we will see how CC model reduces complexity related to software deployment and management. We can still do more optimization to

$\Omega VideoCC$ implementation. However, and if we stop here, we have four different implementation variants, some with completely different designs. Moreover, each implementation variant is composed of several basic deployment units - BDUs, and these BDUs are deployed over different machines at runtime. This kind of software development has a complexity that grows fast enough to be unmanageable, even with such simple application, please see figure 7.25 (up). Moreover, $\Lambda VideoR$ is expected to have thousands of instances at runtime, where these instances are deployed over a large set of heterogeneous devices. As result, the complexity of deployment and run-time management becomes prohibitive. This is where the potential of our novel approach becomes evident. The border of $\Omega VideoCC$ encapsulates all implementation details (BDUs and formal description) as in figure 7.25 (middle) and enables to consider all variants as a single unit of management. All operations after this point (i.e. deployment, compatibility check, runtime management) are automatic and they are the responsibility of CCMS

Chapter 8

Location & Localization

The heart of the cloud component model and of the proposal of this dissertation in general is the location and the localization concepts. If we can abstract all deployment devices with an average host, and all network operations with send/receive operations, and still, this abstraction does not negatively affect neither the software development nor the runtime QoS, our proposal will lose its merit. That is exactly why this dissertation is related to the challenges of the HDEs and not related to the challenges of stable distributed environments.

8.1 Introduction to Ontology

8.1.1 Ontology Definition

The word 'ontology' is used with different semantics in different communities [10]. In this dissertation, we are restricting our attention to the ontology in Computer Science, where we refer to an ontology as a special kind of com-

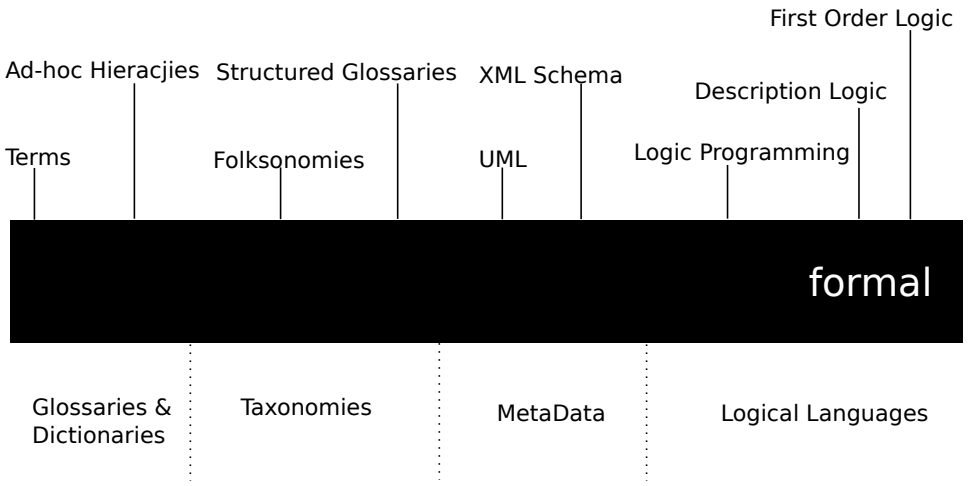


Figure 8.1: Different languages according to [9]. Typically, logical languages are eligible for the formal, explicit specification, and, thus, ontologies (From [10]).

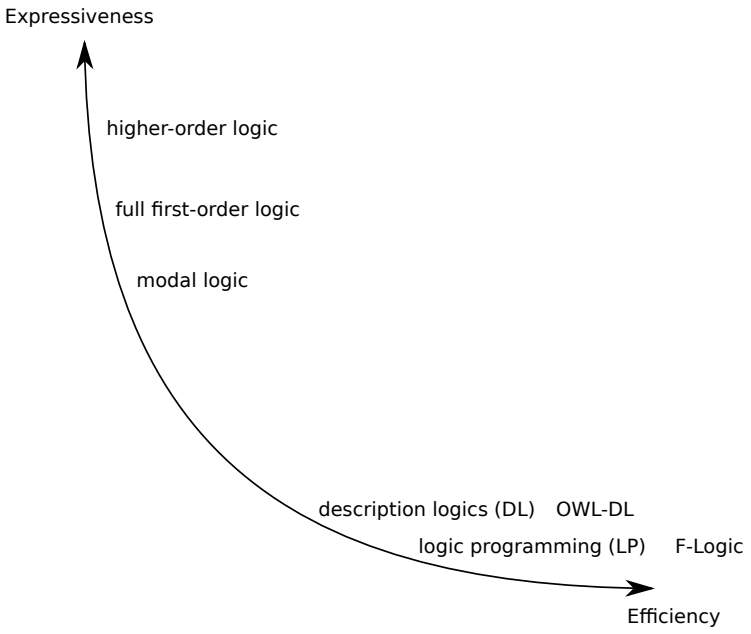


Figure 8.2: The trade-off between expressiveness and efficiency among logical languages [10].

putational artifact [10]. In 1993, Gruber provided the following definition of ontology: "explicit specification of a conceptualization" [142]. This definition is widespread in widely cited [10]. Other definitions of an ontology have appeared literature, examples of these definitions could be found in [54, 143]. In 1997, Borst provided the following definition of ontology: "formal specification of a shared conceptualization" [144]. This definition explicitly asserts that conceptualization should be expressed in a formal (machine readable) format. In 1998, Studer et al. [145] merged these two definitions stating that: "An ontology is a formal, explicit specification of a shared conceptualization."

As mentioned above, an ontology is a computational artifact. To build and use this artifact, it is necessary to use a *'language'*. This language is independent from the domain of witch the ontology is expected to model. Figure 8.1 lists several languages and language categories in a *continuum shape* [9, 10]. As we move along the continuum to the right, we can notice the following [10]:

- The amount of meaning specified by the language increases.
- The degree of formality increases.
- The ambiguity decreases.
- The support for automated reasoning increases.

It is difficult to draw a strict line in figure 8.1 of where the criterion of formal starts on this continuum. In practice, the rightmost category (i.e. logical languages) is usually considered as formal. Moreover, inside logical languages category, there is a trade-off between *expressiveness* and *efficiency* as shown in figure 8.2. On the one end, we find higher-order logic, full first-order logic, or modal logic. They are very expressive, but do often not allow for sound and complete reasoning and if they do, reasoning sometimes remains untractable [10].

At the other end, we find subsets of first-order logic. These subsets feature decidable and more efficient reasoners [10]. At this end we find the following:

- First, languages from the family of description logics (DL). OWL-DL is an example of a member in the family of description logics. All members of DL languages are strict subsets of first-order logic.
- Second, the major paradigm that comes from logic programming (LP) [146]. F-Logic is one highly recognized representor of LP. Though logic programming often uses a syntax comparable to first-order logics, it assumes a different interpretation of formulae.

In this dissertation, F-Logic is used for all ontology related modeling.

8.1.2 The Semantic of Semantic

Ontologies are about semantics (i.e. meaning or understanding). More precisely, communicating semantics. For the ontology to achieve its objective, it needs to fulfill the fundamental requirement which is facilitating the communication: human-to-human [1] , human-to-machine, or machine-to-machine [10]. This communication can be approximated using the Semantic Triangle [2] proposed by Ogden and Richard in 1923 [11]. Please see figure 8.3. The idea in this figure is further explained in figure 8.4. When the sender wants to communicate the concept 'Thing', he/she uses a word (or a sign). When the receiver receives this 'sign', it (the sign) invokes a concept on his/her mind. The receiver uses this concept to identify the individual the sign was intended to refer to. Hopefully the receiver succeeds to point out to the same 'thing' the sender meant. Un-

1. Please read section 8.1.3 where the author of the dissertation presents his position regarding this point.
2. Ferdinand de Saussure (1857 - 1913) is one of the linguists who laid the foundations of semiotics. He proposed the concept of the sign/signifier/signified/referent. This concept forms the core of the field.

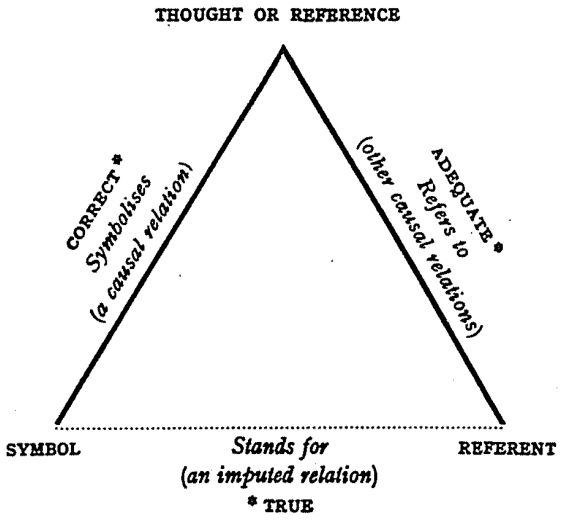


Figure 8.3: Ogden Semantic Triangle. Figure is copied from [11]. This idea was first proposed by Ferdinand de Saussure, a linguist who is considered one of the fathers of semiotics.

fortunately, the sign can erroneously invoke the wrong concept and finally lead to different 'thing' than intended to. Unavoidably, different agents will arrive at different conclusions about the semantics and the intention of the same message (sign) [10]. Here comes the role of ontologies. When agents commit to a common ontology they can limit the conclusions associated with the communications of specific sign, and ideally, removing ambiguity completely. "Thereby, not only the act of reference becomes clearer, but also the connection between sign and concept changes from a weakly defined relationship of 'invokes' into a logically precise meaning of 'denotes' [3]. Likewise, the meaning of a concept is now determined by a precise logical theory." [10].

3. Please read section 8.1.3 where the author of the dissertation presents his position regarding this point.

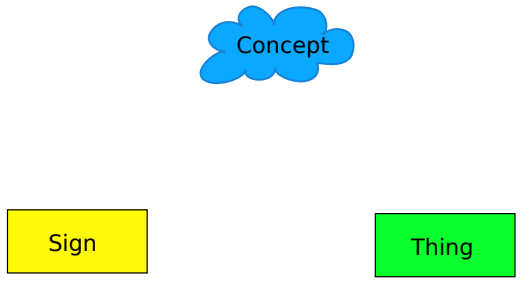


Figure 8.4: Approximation of the communication: human-to-human, human-to-machine, or machine-to-machine (from [10]). Based on Ogden Semantic Triangle in figure 8.3. The instable bended arrow represents the overall communication context.

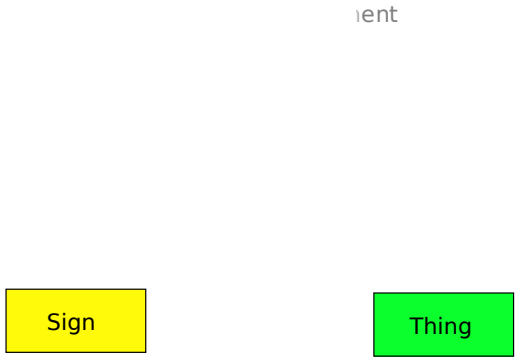


Figure 8.5: The incorporation of ontologies in the communication approximated in figure 8.4 (from [10]).

8.1.3 Discussion

The ideas presented in section 8.1.2 are found in almost all ontology related literature, such as [147–150]. The author of this dissertation has several comments on these ideas.

First we need to distinguish between computational artifacts from one side and human intellectual power from the other side. Mixing these two concepts is not useful, moreover, there need to be proper illustration to the reader about which context is implicitly meant. We strongly agree on the importance of ontology as a computational artifact. Moreover, we strongly agree on the importance of increasing the ability of computational devices to process semantics. For these purposes, figures 8.3, 8.4, and 8.5 in additions to the ideas in section 8.1.2 are highly useful.

On the other hand, we strongly disagree that figure 8.3 is a model of human thinking or communication. It is a useful model to build computational artifacts, however, we should be very clear for scientific and ethical reasons, that we do not advocate simplifications of human intellectual power just to provide scientific modeling. In the following we will present two attributes of the inter-human communication that disagree with the simplified model presented in section 8.1.2.

First, there are concepts (terms) that are cultural dependent or even personal dependent where it is impossible to build a shared ontology for, such as ‘freedom’. Yet, these concepts, including ‘freedom’, are shared and used among people everyday. Human uses his ‘magical power’ to live with such ambiguity. More surprisingly, people from different cultures, where they have completely different

semantics of such terms, they communicate these terms comfortably.

Second, in the modern intellectual trend, led by Abu Tammam [4] and Abo Nuwas [5], "It is fundamental that the name (term) is not equal to the named object (or abstraction), the relationship between the two is sign or symbol. It is neither accordance nor identity. In other words, it is a probability relationship, not a certainty relationship" [151]. Moreover, "The creative text (or speech or poem) in this modern trend is a vast horizon of semantics or 'realities'. It is not an idea or collection of ideas that we can clearly see and understand, one by one" [6] This trend arrived to the conclusion: "Eloquence does not exists without getting far from the clear and precise semantics of the used words" [151]. This intellectual trend opposed the 'traditional trend' that believes in the 'certainty relationship' between the 'word used' and the 'semantic intended'.

In this dissertation, we use the term 'semantic' in a restricted context of computational sciences. If we mean another context, such as human linguistics, it will be appropriately and explicitly mentioned.

4. His name is Habib Ibn Aws. He was born in 788 AD in Syria and died in 845 AD in Iraq. Many studies about his work, along with references to his own work can be found in [151,152].

5. His name is al-Hasan ben Hani. He was born in 756 AD and died in 810 AD in Iraq. He was one of the greatest of classical Arabic poets and intellectual figures. Many studies about his work, along with references to his work can be found in [151,153,154].

6. Adunis or Adonis. His name is Ali Ahmad Said Isbar. He is a Syrian intellectual figure. He has written more than twenty books. One of these books was 'Le fixe et le mouvant' [151], a four volumes book that had a special attention.

8.2 F-Logic

8.2.1 F-Logic Definition

F-logic a frame-based logic language that is used for representing ontologies and building intelligent applications on top. A number of implementations of F-logic exist, both commercial and open source academic systems. OntoBroker [7], a commercial implementation of F-logic, will be used throughout this dissertation as the main formal language for building ontologies.

F-logic based ontology starts with class (concept) hierarchies, then defines relations and attributes of the concepts, defines the relationships among classes and objects (instances) using rules, and finally populates the concepts with concrete instances.

Listing 8.1: F-logic constructs.

```
01- /* concept hierarchies */
02- man::person.
03- /* relations */
04- person[hasFather{0:1} *=> man].
05- person[hasSon *=> man].
06- /* rules */
07- ?X[hasSon -> ?Y] :- ?Y:man[hasFather -> ?X].
08- /* population */
09- Jack:man.
10- Michel:man.
11- Michel:man[hasFather -> Jack].
```

7. Registered Trade Mark.

```
12- /* query */
13- ?- X:man[hasSon -> Michel].
```

In listing 8.1 we can find all constructs used in F-logic. Line `02` shows the concept hierarchy. Lines `04` and `05` show how to add a relation to a concept. Rules are explained in line `07`. Ontology populations is shown in lines `09` to `11`. A query is presented in line `13`. In section 8.3 we will explain how to practically and effectively use these constructs to build an ontology. For further information on the F-logic please read [10, 155, 156].

8.2.2 Implementations of F-Logic

There are several major implementations of F-logic, including FLORID [157], OntoBroker [158], and FLORA-2 [159]. Each implementation introduces a number of extensions to F-logic as well as restrictions to make their particular implementation methods more effective. Ontoprise, the company produced OntoBroker, started using the term 'Objectlogic' instead of F-logic since 2010 to acknowledge these modifications. FLORID and FLORA-2 were developed in the academia. Their main goal is to provide free platforms for experimenting with innovative features in the design of an F-logic based rule language. OntoBroker is a commercial system. Its main emphasis is on efficiency and integration with external tools and systems. This dissertation takes a view of F-logic as an ontology modeling language as well as a language for building applications that use these ontologies. The ability to cover both sides of the engineering process, ontologies and applications, is a particularly strong aspect of F-logic [10].

FLORID implements F-logic using a dedicated bottom-up deductive engine,

which handles objects directly through an object manager. In that sense, it is similar to object-oriented databases. In contrast, FLORA-2 and OntoBroker use relational engines, which do not support objects directly. Instead, both systems translate F-logic formulas into statements that use predicates (relations) instead of F-logic molecules, and then execute them using relational deductive engines. The target engine of FLORA-2 is XSB. XSB is a Prolog-like inference engine with numerous enhancements, which make XSB into a more declarative and logically complete system than the usual Prolog implementations. The inference mode of XSB is top-down with a number of bottom-up- like extensions. OntoBroker uses its own relational deductive engine. Its main inference mode is bottom-up, but it includes several enhancements inspired by top-down inference, such as dynamic filtering [160] and Magic Sets [161].

8.3 Ontology Life Cycle

Many books and papers are devoted for the construction of an ontology including [162,163]. In this section we will use OntoBroker and F-logic to show how to build an ontology. This is fundamental to understand the software/hardware compatibility check that our dissertation is based on. The ontology development process consists of three major phases as explained in figure 8.6.

8.3.1 Ontology Design & Creation

The starting point of the ontology life cycle is designing the ontology. In this stage, the designer usually has a global understanding of the domain he/she wants to model using ontology. Details are accumulated gradually, however,

the designer need to be able answer questions such as if a specific concept is in or out of the scope of the domain he/she is creating the ontology for. After having global idea of the domain being modeled, the designer can go a step further and list all keywords (terms) in that domain. These concepts can be thought of as keywords that are used in that domain. It is the responsibility of the designer to list all important keywords, otherwise, the ontology will lack the power of acknowledging the semantics of some entities in that domain. For the device ontology, the list of terms in figure 8.7 represents a subset of the overall concepts in this domain.

After choosing the list of terms, complete or incomplete, we need to understand the relationships between these terms (concepts). Here is the power of ontologies, and at the same time, the most difficult task in ontology design. The term 'Windows' in figure 8.7 is related to the term 'OperatingSystem', and the term 'ThreeG' is related to the term 'Connectivity'. These relations collectively creates the semantics in the ontology. The more rich and precise these relations are, the more meaningful semantics the ontology provides. Ontology languages such as F-Logic and Web Ontology Language (OWL) provide several constructs to express these semantics. These constructs include: concepts, concept hierarchy, relations, attributes, and instances. The designer has the freedom to utilize these constructs to encode data and semantics of the objective domain of knowledge. This is a creative operation.

As we can see from figure 8.8, the concept `LCD` is a subconcept of the concept `Screen`. Also, the concept `Touch` is a subconcept of the concept `Screen`. Using this concept hierarchy we are encoding semantics. If any object is an instance of the concept `Touch` that means this object is a display screen which

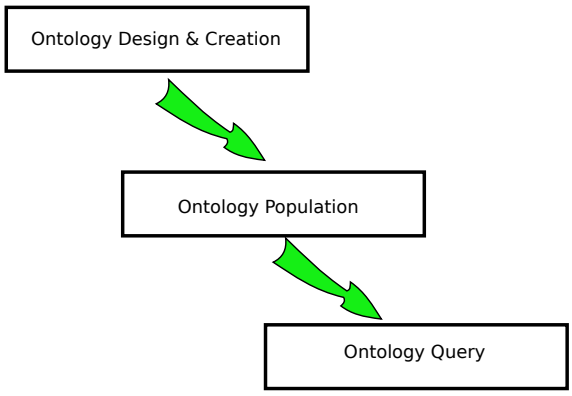


Figure 8.6: The ontology development process.

Connectivity	WideScreen	ThreeG
HardwareElement	HD	Wireless
Architecture	Integrated	Server
ResolutionH	Webcam	Host
Linux	HardwareElement	SWYPE
OperatingSystem	ReadWrite	Desktop
Windows	CD-DVD-ROM	Laptop
AudioFormat	Dual-Layer	SmartPhone
Resolution	Storage	Cellphone
SoftwareElements	Screen	Touch
Android	VideoMemory	LCD
VideoFormat	Memory	Apple
Symbian	VideoCard	
ResolutionW	Display	

Figure 8.7: A list all keywords (terms) in domain being modelled using ontology. Only partial list is shown in the figure.

has touching capabilities (i.e. we can interact with it by direct touch). We still do not have any information about the technology of this object. If this object is an instance of the concept `LCD` in addition, then it is an LCD touch screen (otherwise it could be CRT touch screen). Until now we used two techniques to encode semantics: first is the concept name, and second is the concept hierarchy.

F-logics provides us with another tool to encode semantics: Relations. Each relation has domain and range. Domain and range are concepts. Relations bind concepts, or in other words, clarify the relationship between these con-

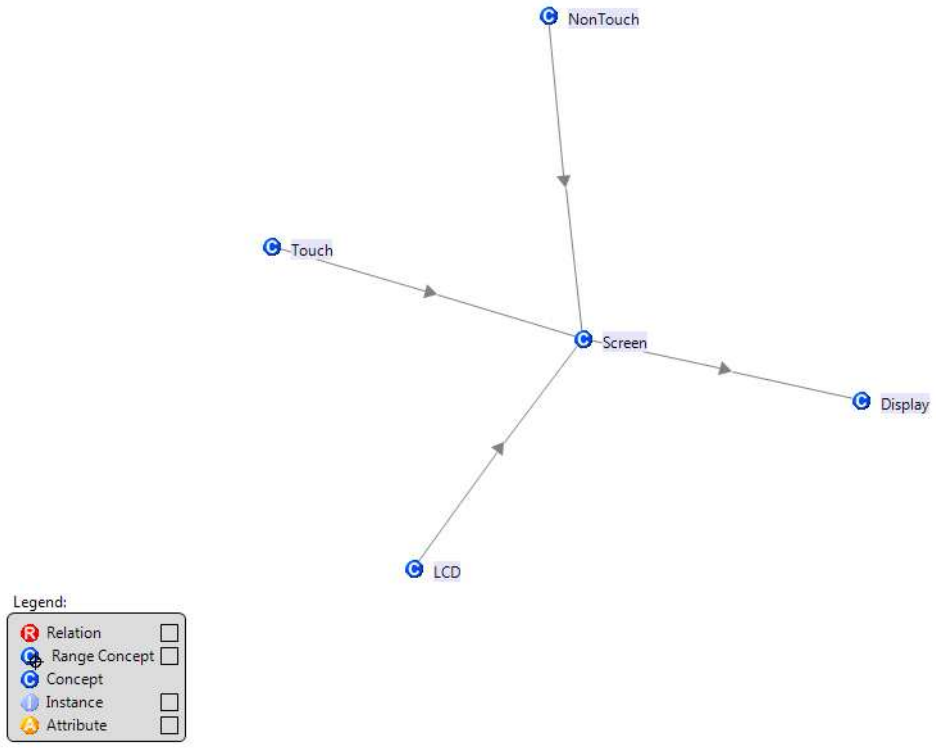


Figure 8.8: The sub-concept construct is a tool to build the concept hierarchy.

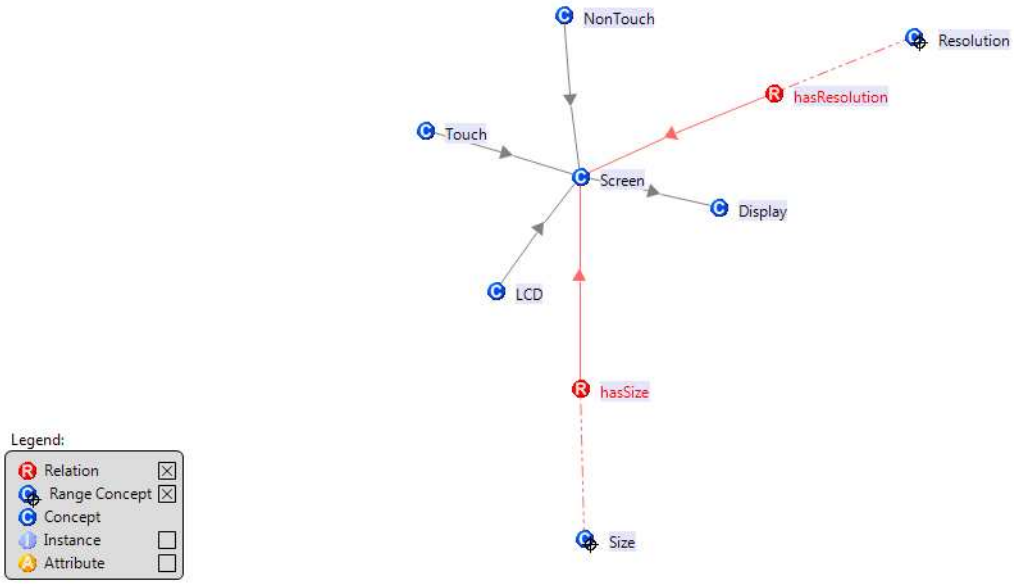


Figure 8.9: The 'relation' construct relates two concepts in a meaningful way.

cepts. In figure 8.9 we can see the relation `hasResolution` relates the concept `Resolution` to the concept `Screen`. In F-logic this is written as:

```
Screen[hasResolution {0:*} *=> Resolution].
```

The semantics in this relation are encoded in two different places: first by attaching the range concept to the domain concept, and second, in the relation name. We have to choose meaningful names for relations that reflect the semantics we want to encode. The name of the relation clarifies the kind of relationship between the domain concept and the range concept. To make this more clear, we go back to the second step in the design where the designer lists all keywords (terms) in that domain he/she wants to model, as in figure 8.7. After the relation above, the two terms `Screen` and `Resolution` are no more

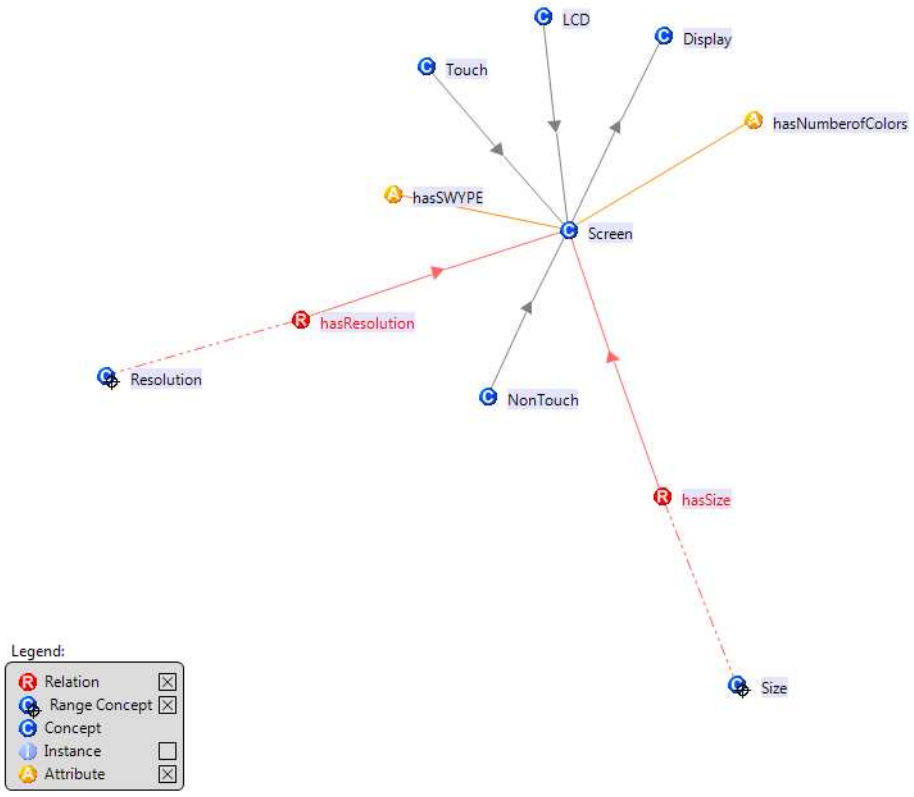


Figure 8.10: The 'attribute' construct.

isolated. Because of this relation, we know that each screen has a resolution, and the reasoner is able to analyze this in addition.

The attribute construct allows us to model semantics, and at the same time, save data. As we can see in figure 8.10, we did not model the keyword 'SWYPE' as a concept. Rather, we modeled it as an attribute, where the semantics is encoded in the attribute name, and the attribute data can be filled with true or false. The construct attribute is similar to the construct relation. The only difference is that attribute has a range of basic types such as integer or string, while relation has a range of type concept. Back to figure 8.10, we can see that we

have another attribute of the concept `Screen`, which is `hasNumberofColors`. This attribute has a range of integer. In F-logic, the above two attributes are written as follows:

```
Screen[hasSWYPE {0:*} *=> _boolean].
Screen[hasNumberofColors {0:*} *=> _int].
```

Finally, it is important to point out to the following two facts. First: the group of relations and attributes are called properties. Second: ontology development is an iterative process. During later phases, such as population or query phases, it is possible to discover errors, or important features that are missing. This will require going back to the design phase and make the necessary modifications.

8.3.2 Ontology Population

In this phase we add instances to the ontology we designed in the previous phase. Each instance is an object of a specific concept, i.e. this instance is 'instance of' this concept. Also, an instance can be 'instance of' two or more concepts. In figure 8.11, `GalaxyMini` is an instance of the concept `SmartPhone`. And as it appears in that figure, there are several relations and two attributes attached to the concept `SmartPhone`. In order for the operation of adding `GalaxyMini` to the ontology to be complete, we need 'to fill all of these blanks'. All of the relations and attributes have one side bound to `GalaxyMini`, and the other side is bound to `null`. We need to bind the other side of each relation to some instance, and the other side of each attribute to some value. In figure

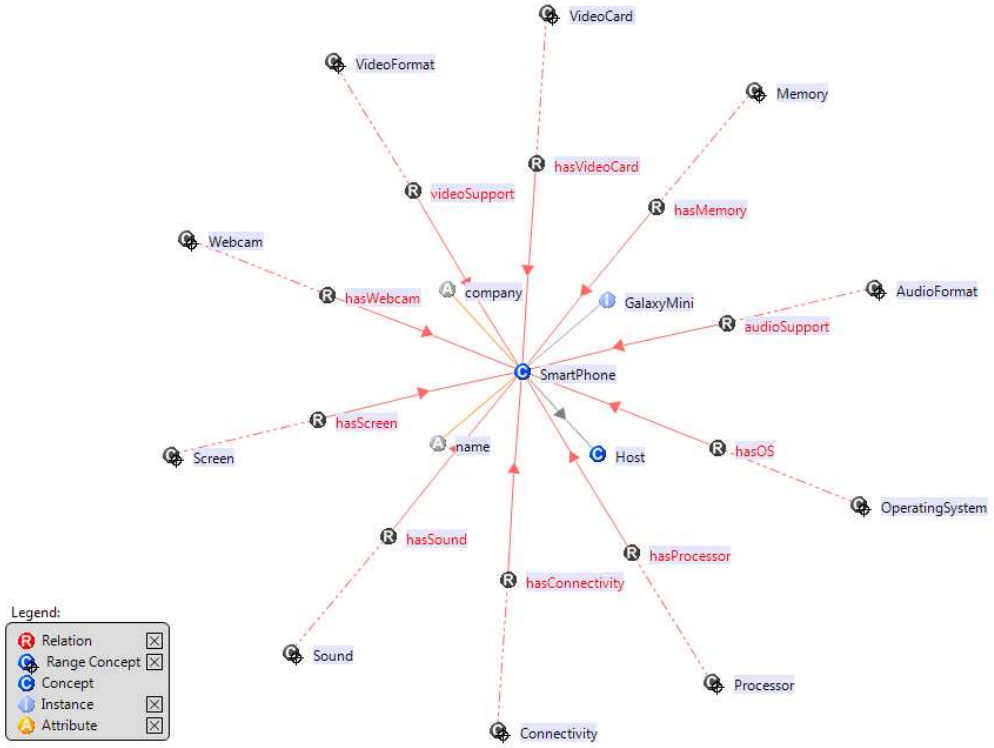


Figure 8.11: Ontology population.

8.12 we can see how this stage is done for `GalaxyMini`.

8.3.3 Ontology Query

This is the phase where the ontology-based application can benefit from this advance tool: the ontology. The application needs certain information to make some decision, it formulates its requirement as an ontology query, and then it passes this query to the ontology reasoner, where the reasoner applies its algorithms over the populated ontology. The reasoner finally passes the query results back to the application.

Ontobroker from Ontoprise provides an ontology query language which is

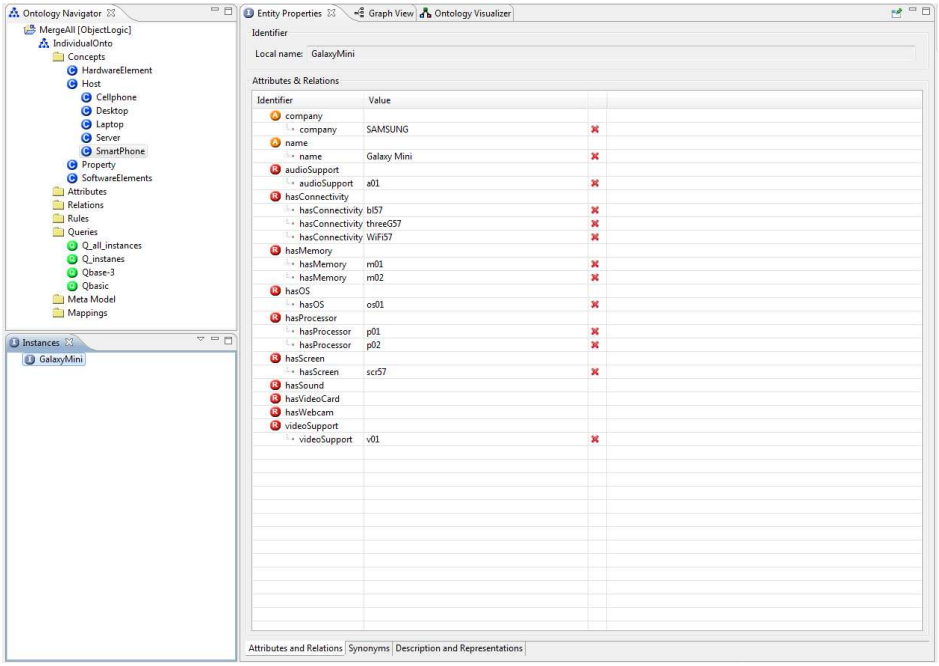


Figure 8.12: A snapshot of OntoBroker screen shows how to populate an ontology.

based on F-logic. As an example, suppose the application needs to handle the following request:

"Give me all hosts that has High Definition playback."

The F-logic query will be as follows in listing 8.2.

Listing 8.2: An ontology query using F-logic language.

```
?-
?Host1:Host
and
?host1[hasScreen->?Screen1]
and
?Screen1:Screen[hasResolution->?Resolution1]
and
?Resolution1[resolutionUnit-> ?Resolution1_resolutionUnit]
and
?Resolution1[hasResolutionW->?Resolution1_hasResolutionW]
and
?Resolution1[hasResolutionH->?Resolution1_hasResolutionH]
and
_greaterThan(?Resolution1_hasResolutionW,1280)
and
_greaterThan(?Resolution1_hasResolutionH,720)
and
_unify(?Resolution1_resolutionUnit,"px").
```

For further information on the F-logic and F-logic query language please read [10, 155, 156].

8.4 Design Considerations

8.4.1 Concept or Property

When modelling a domain, we might face the decision on whether we model a specific distinction as a property (relation or attribute) or as a set of concepts [162]. Apart from our ontology, if we want to model the domain of wine using ontology, we might face the following choice: do we model white wine, red wine, and rose wine as three different concepts or we model them as one concept, wine, with a color attribute which can be assigned the value of color? Both options are correct, however, they reflect a design choice that might lead to a good or bad design at the overall scope. This is the case for all modeling and programming techniques because they provide the designer with several and different constructs that might be used interchangeably. Back to our main question, do we use a concept or a property in situations where both of them look valid? To answer this question for the general case we need to know that concepts have higher modeling power than properties. In other words, the difference between two objects who belong to two different concepts is more important than the difference between two objects who belong to the same concept but have two different property values. This is the key. If the importance of the distinction we want to model is marginal, we can use properties. On the other hand, if this distinction is important in the domain we are modelling, then we should encode the semantic of this distinction in the concept hierarchy, with expressive concept names [162]. The decision about the importance of the distinction we are modeling is the responsibility of the designer. He/she should have an overall view of the domain, and should be able to answer questions of this kind. Back

to the wine example, it is clear that the issue of red wine and white wine is not simply a color issue. That is because they are paired with different food, has different properties, etc. As result, they should be modeled as two different concepts in the concept hierarchy.

8.4.2 Concept or Instance

Deciding whether a particular distinction in the domain is modeled as a concept in the ontology or an instance in that ontology depends on the granularity of the design. This granularity is decided by the designer who is aware of the potential applications, and also, can expect the level of details queries will include. In any ontology, instances are the most specific detail that can be modeled. In other words, instances are the finest granularity possible. There is always more details to model, however, the designer can choose the finest level of details he/she believes will be sufficient for all the expected applications.

In figure 8.13 we model all computing devices in a concept hierarchy. From that figure we can see that we have two levels of concepts, the `Host` level, and the `SmartPhone, Laptop, ..., Server` level. We could have only one level: `Host`, and model all devices as instances of it. Or, we could have more levels, for example, the concept `SmartPhone` could have more classification based on operating system, price, brand, etc. In figure 8.14 we can see that `GalaxyMini` device is modeled as instance of `SmartPhone` concept. This is a design choice. Back to the discussion above, we could have Galaxy as a concept, which is subconcept of Samsung, and then have Galaxy Mini as instance of the concept Galaxy. But we chose to ignore these semantic details in the concept hierarchy because we do not expect any valuable applications that will depend

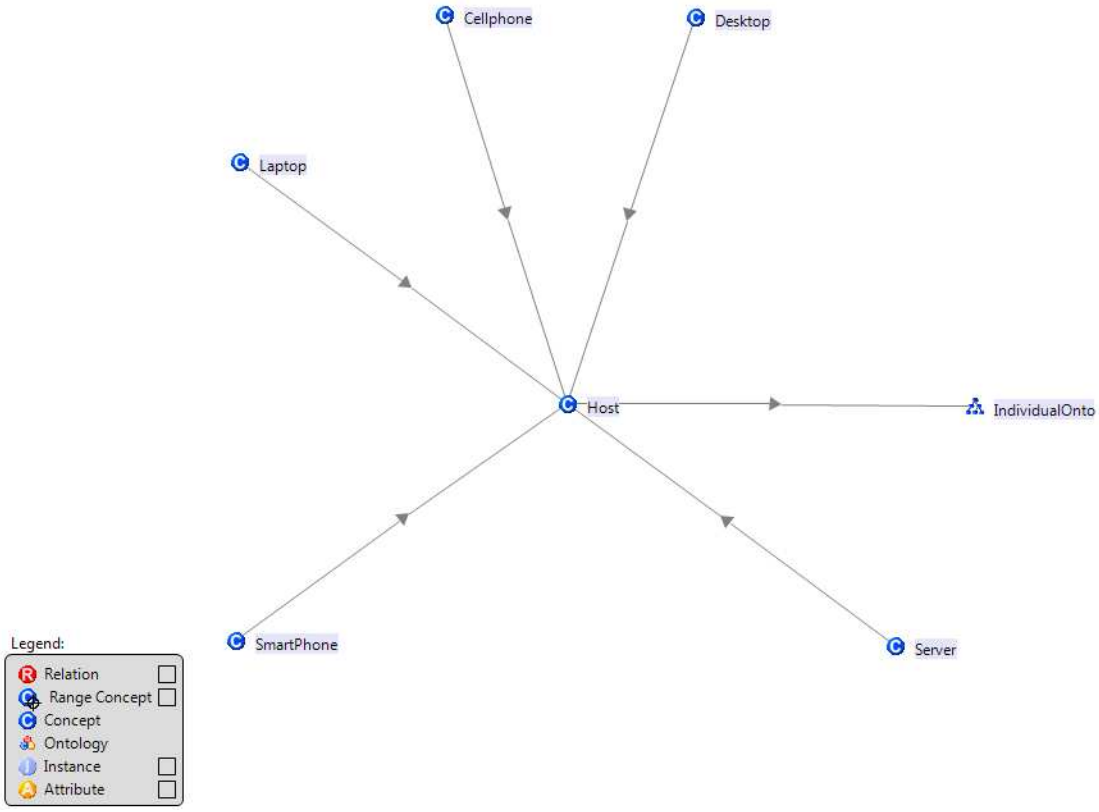


Figure 8.13: The concept-instance choice.

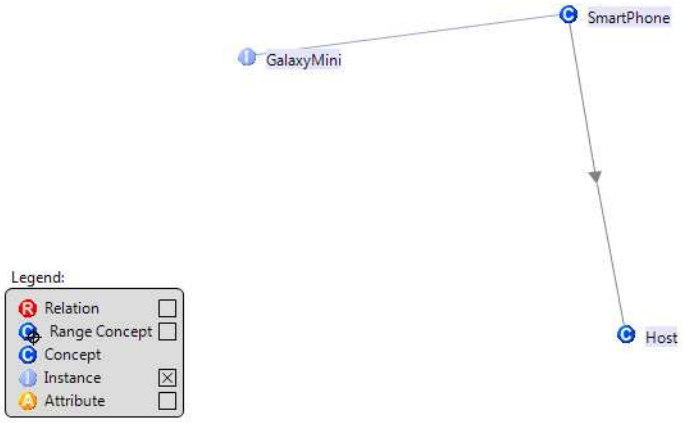


Figure 8.14: The concept-instance choice - again.

on these details significantly.

8.5 Contribution

8.5.1 Software/Hardware Compatibility Checker

The heart of our model is the ability to 'match' software and hardware. Match here means 'compatibility'. At design time, we explained in section 7.3 how to build compatible implementation variant for different deployment devices. This is major-phase-one in problem statement defined in section 2.6. In figure 8.15 we show how to ensure this compatibility at deployment time, i.e. major-phase-two. It is very important in figure 8.15 to notice the following:

- First, and for easiness, we will call the blue ontology: IndividualOnto-A and the green ontology: IndividualOnto-B.
- IndividualOnto-B is an extended version of IndividualOnto-A. This extension is population only. No other modification, addition, or deletion is allowed.
- The software requirement modeling is done during major-phase-one. Using IndividualOnto-A, designer/programmer of a BDU (`BDU-X` in figure 8.15) list all requirements of the BDU as F-logic rules and queries. During this, there is no information regarding IndividualOnto-B.
- If rules and queries generated (manually by the designer/programmer) in the above step, when applied over IndividualOnto-B, return `Tablet-Ju`, then we can deploy `BDU-X` over `Tablet-Ju`. Otherwise, we can not proceed with this deployment over this device.

To capture the contribution in this checker, it is fundamental to notice:

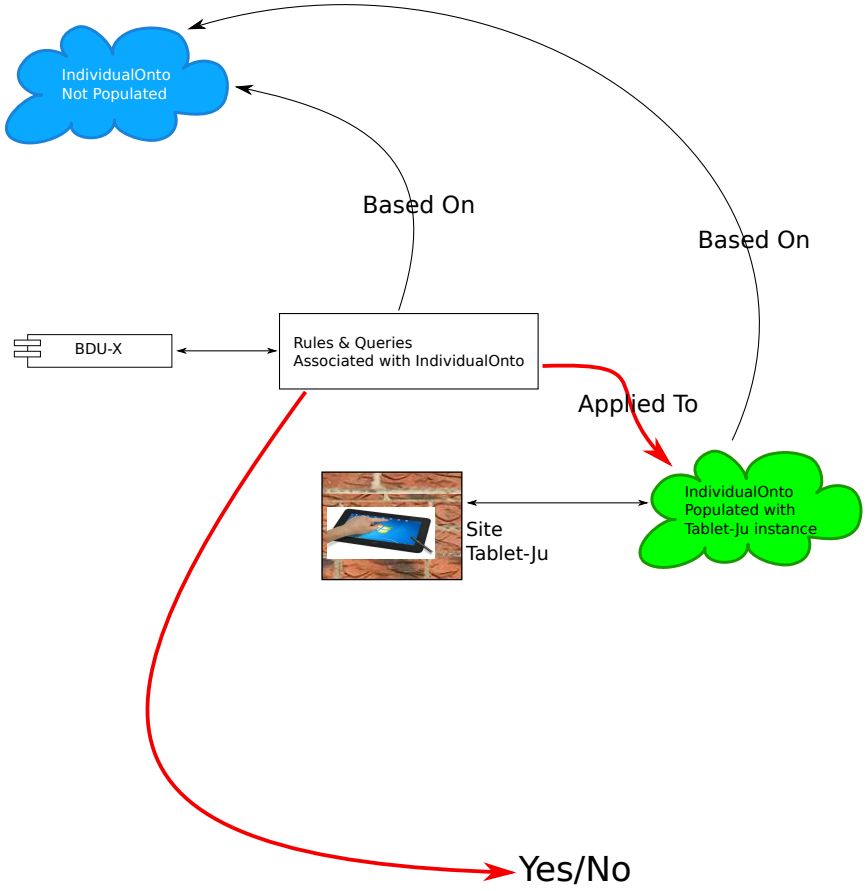


Figure 8.15: Software/hardware compatibility checker.

- The time difference: it could be more than months, and may be years that separate the implementation of IndividualOnto-A, IndividualOnto-B, and rules and queries in figure 8.15.
- The personnel difference: different teams implement IndividualOnto-A, IndividualOnto-B, and rules and queries in figure 8.15. These teams might belong to different companies in different countries.
- The required automation: at deployment, there need to be automatic (machine based) BDU/device checker.

Based on figures 8.4 and 8.5 we can see that ontology provides the required knowledge sharing approach to facilitate unification of knowledge (terms and semantics) between all of the mentioned personnel, over the above mentioned long period of time, and in machine comprehensible way (for the automatic checker).

8.5.2 IndividualOnto

In this work, we provide the base ontology: IndividualOnto. That means we finished phase one of the ontology life cycle presented in section 8.3. IndividualOnto is designed to serve the purpose presented in this thesis. In figure 8.16 we see a partial visualization of the concept hierarchy. Moreover, these concepts are related by 'relations', however, these relations do not appear in this figure. In figure 8.17 we see the concept hierarchy with the concept `HardwareElement` being the root. We want to emphasis that this ontology (IndividualOnto) is constant. Neither the concept hierarchy, nor the relation/attribute parts can be modified. If they are modified, the concepts, along with their semantics, are no more shared by all parties in figure 8.15. We think that IndividualOnto is

sufficient to model all deployment environment details including all current networking technologies[8]. In listing 8.3 we show how to populate IndividualOnto by adding the full description of the smartphone 'Galaxy Mini' from Samsung (we did not show the full description in the listing). Assuming that the development team is designing and implementing `BDU-X` in figure 8.15. Moreover, this BDU needs a high definition display in order to perform properly. The implementation team model this requirement as in listing 8.2. More complex requirements of BDUs can be modeled using a combination of rules and queries. This is decided by the implementation team.

If this query (in listing 8.2) is passed to IndividualOnto populated by 'Galaxy Mini' (listing 8.3), then the result of the query will be empty set (the important point is that `GalaxyMini` instance of concept `Host` will not be a member of the returned set of instances). This is because `GalaxyMini` has a screen of `320x240` as shown in its modeling, and this is not HD display. As result, the deployment of `BDU-X` can not proceed on `GalaxyMini`.

Listing 8.3: `GalaxyMini` Instance in `IndividualOnto`.

```
GalaxyMini : SmartPhone .
GalaxyMini [ audioSupport -> a01 ] .
GalaxyMini [ company -> "SAMSUNG" ] .
GalaxyMini [ hasConnectivity -> WiFi57 ] .
GalaxyMini [ hasConnectivity -> bl57 ] .
GalaxyMini [ hasConnectivity -> threeG57 ] .
GalaxyMini [ hasMemory -> m01 ] .
```

8. Many readers will think that this statement is strong. Usually computer scientists prefer constructs to be modifiable which is contrary to our statement that IndividualOnto is constant. It is out of the scope of this work to study the possibility of modifying IndividualOnto. It is easy to add a new concept to IndividualOnto, however, does the whole model stays consistent? What about backwards compatibility?

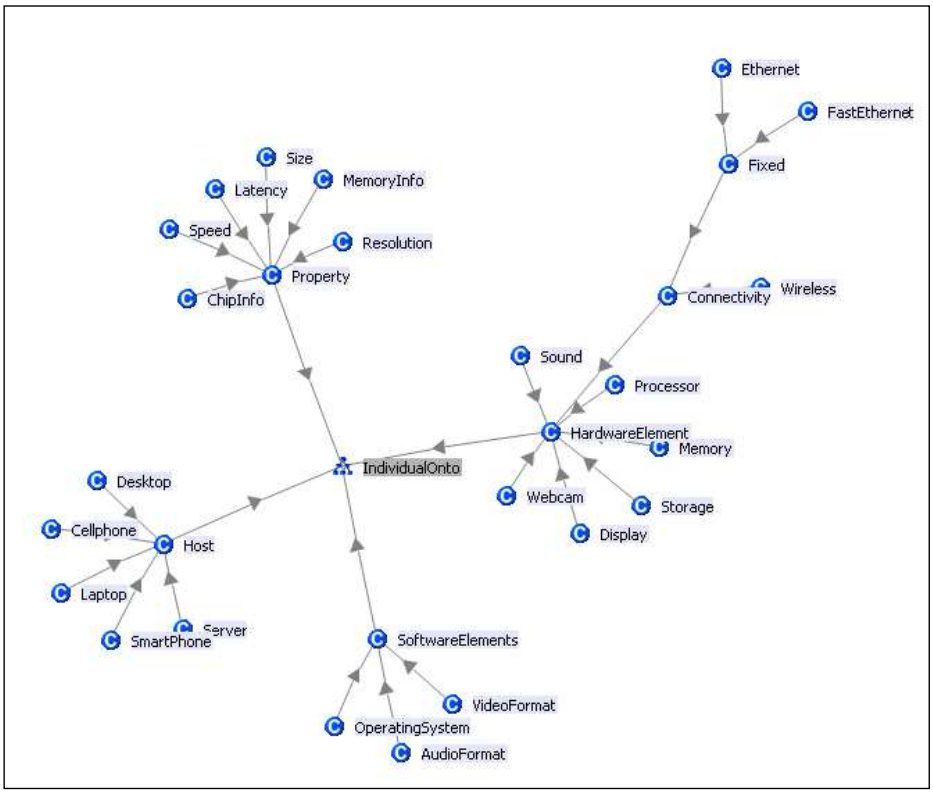


Figure 8.16: A partial visualization of the concept hierarchy of `IndividualOnto`.

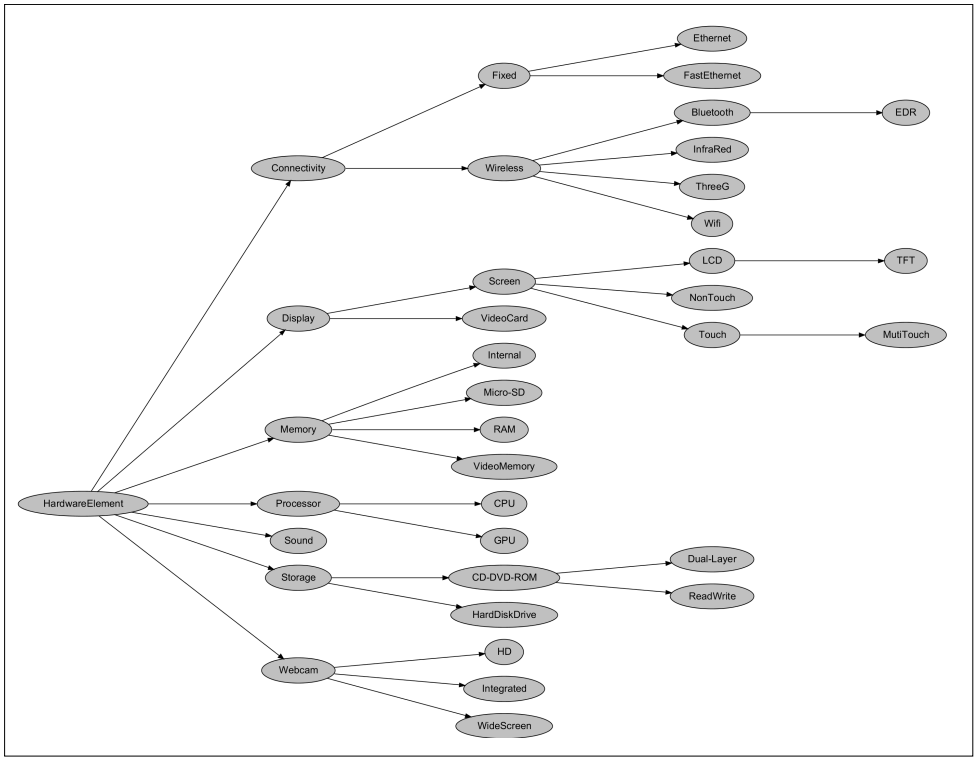


Figure 8.17: The concept hierarchy of `IndividualOnto` with the concept `HardwareElement` being the root.

```
GalaxyMini[hasMemory->m02].
GalaxyMini[hasOS->os01].
GalaxyMini[hasProcessor->p01].
GalaxyMini[hasProcessor->p02].
GalaxyMini[hasScreen->scr57].
GalaxyMini[name->"Galaxy  Mini"].
GalaxyMini[videoSupport->v01].
a01:AudioFormat.
WiFi57:Wifi.
a01[supportsFormat->"AAC"].
a01[supportsFormat->"AAC+"].HDQuery
a01[supportsFormat->"DNSe"].
a01[supportsFormat->"MP3"].
a01[supportsFormat->"eAAC+"].
WiFi57[hasStandard->"IEEE802.11"].
WiFi57[hasVersion->"b"].
WiFi57[hasVersion->"g"].
WiFi57[hasVersion->"n"].
bl57:EDR.
bl57[hasVersion->"2.1"].
threeG57:ThreeG.
threeG57[hasMaxSpeed->s57].
threeG57[hasStandard->"HSDPA"].
s57:Speed.
s57[speed->7.2].
s57[speedUnit->"Mbps"].
```

```
scr57:MutiTouch.
scr57:TFT.
scr57[hasNumberofColors->16000000].
scr57[hasResolution->QVGA].
scr57[hasSWYPE].
scr57[hasSize->normal_size].
scr57[resolutionUnit->"px"].
normal_size:Size.
normal_size[size->3.14].
normal_size[sizeUnit->"inche"].
QVGA:Resolution.
QVGA[hasResolutionH->320].
QVGA[hasResolutionW->240].
QVGA[resolutionUnit->"px"].
Resolution[hasResolutionH {0:*} *=> _int].
Resolution[hasResolutionW {0:*} *=> _int].
Resolution[hasTotalResolution {0:*} *=> _int].
Resolution[resolutionUnit {0:*} *=> _string].
```

8.5.3 Used Technology

To build IndividualOnto we used OntoStudio[9] version 3.0.2 from Ontoprise. For the query engine, we used OntoBroker[10] Enterprise Edition version 6.0.3 from Ontoprise. Population of IndividualOnto is done using OntoStudio. The query in listing 8.2 is written using an editor and stored appropriately in the

9. Registered trade mark.
10. Registered trade mark.

formal description of BDU-X. At deployment time, cloud component management system (CCMS, please read chapter 9) extracts the query from the formal description, and incorporates it into the Java code designed to access populated IndividualOnto through OntoBroker.

OntoBroker has a web service interface to submit queries and commands. The OntoBroker web service provides four operations [11]:

- "query" - used to run F-Logic or SPARQL queries.
- "command" - used to send commands to the OntoBroker server.
- "queryBatch" - special operation to send multiple queries in a single message to OntoBroker.
- "executePreparedQuery" - used to run a prepared query.

8.5.4 Existing Device Ontology

The Foundation for Intelligent Physical Agents (FIPA) is an IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies [164].

FIPA proposed a device ontology in 2001 and this ontology was revised and updated until version E in December 2002 [165]. FIPA ontology is very flat and the concept hierarchy is poor. As result, most knowledge are saved as data (text and numerical values) without semantics.
As an example connection-description is the only existing concept to model communication/networking. Moreover, connection-description has no relations. Using this ontology, Bluetooth, Wifi, and high speed fixed Ethernet are modelled as instances of this concept. This is extremely limited

11. The manual provided by Ontoprise with OntoBroker version 6.0.3 Enterprise Edition.

because the reasoner is unable to answer simple queries such as: which of these technologies is wireless, let alone more complex and semantic-based queries.

It is expected that other proposals appear while preparing this manuscript. This is natural because ontology is gaining wide acceptance in computer science community. Moreover, it is a close to perfect tool for such modelling as discussed in previous sections.

Chapter 9

Tools - Cloud Component Management System

In this chapter we will present two important and indispensable tools: the cloud component management system CCMS, and the registry. The benefits of these tools clearly appear during the post-implementation phases of the development process of CC based systems. This includes: installation, deployment, several types of checking, CC binding, running, and post-running management.

As mentioned earlier, this research project aims at proposing a solution that supports the complete software engineering life-cycle for the development of HDE applications. This development process starts with the specifications of the applications, and ends when the full application runs and delivers the expected functionality along with the expected QoS. Our proposed development process of a single CC is is presented in section 7.3. This process results a set of *Basic Deployment Units (BDUs)* along with the formal mapping of these BDUs to the original CC, as in figure 9.1. These BDUs collaboratively, achieve the

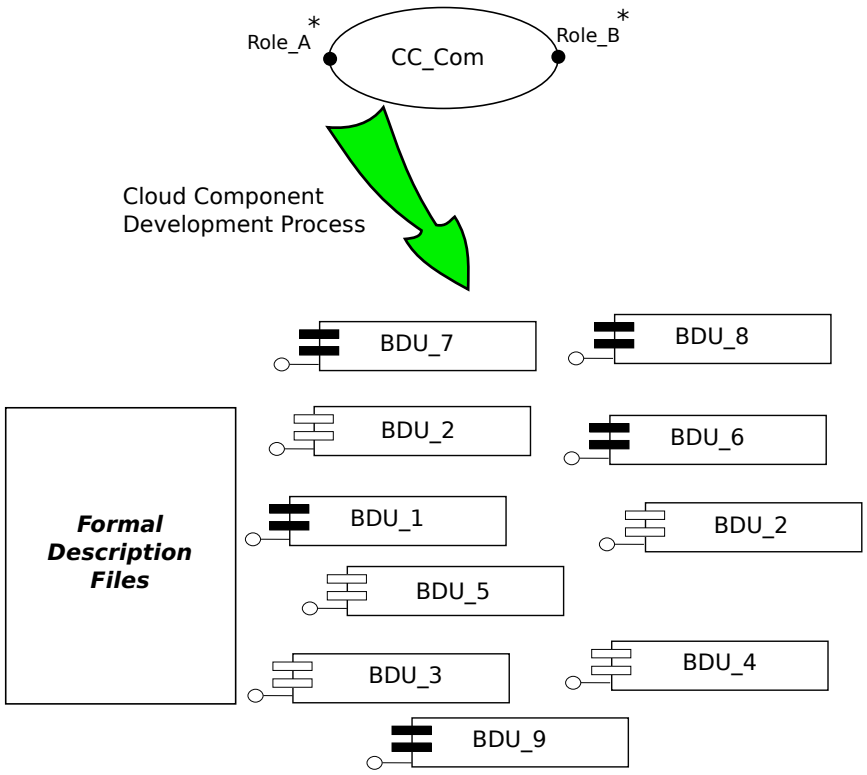


Figure 9.1: The output of the CC development process is a set of BDUs along with necessary formal description files.

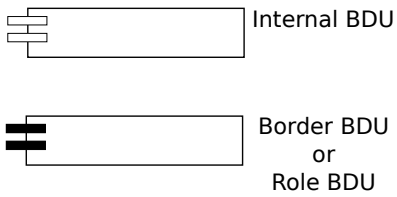


Figure 9.2: BDU types.

functionality of the CC.

9.1 Required Definitions

The following are brief definitions. More details will be provided in the following sections. Please notice that there might be an unclear detail inside a definition, or simply a new vocabulary that is not previously defined. This detail and vocabulary will be defined and clarified later. Otherwise, each definition will span several pages if we include all related details inside the definition. Again these are not exhaustive, precise, and formal definitions. Rather, they are short and expressive to help the reader proceed easily with general idea about these terms.

9.1.1 Basic Deployment Unit - BDU

Basic Deployment Unit is a tangible (physical) artifact that is deployable and executable. The BDU is the smallest architectural unit possible in a CC design. It is not nested (also CC is not nested). It is local with respect to the deployment device (in other words a BDU can not be distributed). There are two types of basic deployment units. The first type is realization of a role, and the second type is the realization of an internal functionality. In the formal

language we proposed, we use the symbol Ξ to prefix the first type and the symbol Φ to prefix the second type. In the graphical form, the first type will have solid (black filled) left side while the second type has none-filled left side as in figure 9.2. For information about the relationship between BDUs and the ontology-based-checker please read section 8.5.

9.1.2 Registry Utility

Is a utility that saves the state of a CC based system, including all of its installed CCs, deployed CCs, deployed BDUs, etc. This utility can be accessed directly by an administrator (interactive mode) or remotely through the network (network mode).

9.1.3 Incremental Deployment

Incremental deployment tackles the deployment operation of a CC based system gradually. In other words we do not deploy a CC based system in one operation, rather, this deployment involves several (many) operations (stages) that are usually separated in time. The time interval between two deployment stages might be milliseconds, minutes, days, or even more.

9.1.4 Installation of a cloud component

CC at the implementation level is a set of BDUs along with a set of formal description files. To install a single CC, it is required to place (copy) these BDUs and files to the directory CAB, and to register this CC to be installed.

9.1.5 Deployment of a cloud component

A complex operation that is done by CCMS. Includes reading formal definition of the CC, moving appropriate archives from CAB to their appropriate destination devices, extracting archives, checking dependencies, ..., and finally registering the new state of the CC [1]. It is not necessary to perform these operations together (transaction semantics).

9.1.6 Deployment of a BDU

Could be part of CC deployment, or standalone operation. Deployment of a single BDU includes moving appropriate archive from CAB to the appropriate destination device, extracting this archive, checking dependencies, binding to the required BDUs, and finally, adding this BDU to the deployed list and update related states in the registry.

9.1.7 CC Deployment Plan

The deployment plan is a series of BDU deployment operations, dependency checking, assembly operations, registry query, and registry updates. The deployment plan is written manually by the system designer and carried out automatically by the CCMS. The deployment plan existence is determined by the deployment conjecture (please read section 7.2.6).

1. Please note that the intention of this definition is to give general idea and to make the reading of the following sections easier.

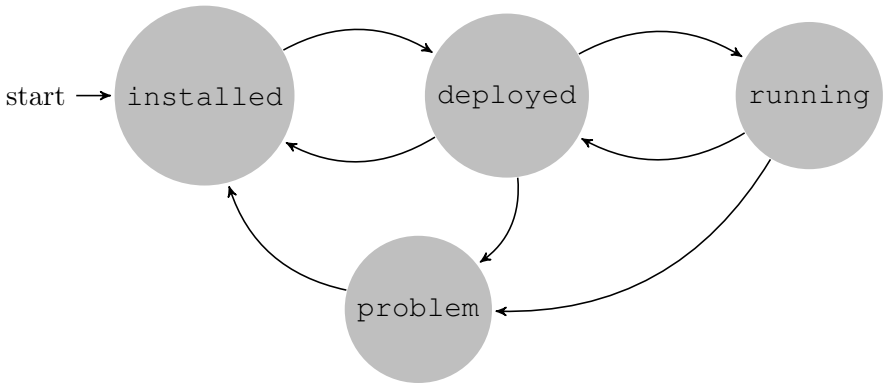


Figure 9.3: The four different states at which a CC or a BDU can exist at runtime. It can move from one state to the other by CCMS operation or administrator manual intervention. It is very important to mention that this state is time dependent.

9.1.8 Cloud Component Management System - CCMS

Is a tool that is responsible of installation, deployment, assembly, and post-running management of a CC based system. This whole section, i.e. section 9, is devoted for this tool.

9.2 Registry Utility

Registry is a large and complex data structure that saves the dynamic state of a CC based systems. This state is dynamic because it changes by time, and is complex because it includes the states of each and every BDU of the system. Please see figure 9.3.

To make more accurate estimation of the size of the registry database, we need to imagine a cloud component based system as in figure 9.4 at run time. From that figure we can see that we have only one instance of $\Omega VideoCC$ at

any time. However, how many instances of $\Omega MultimediaCC$ exists at some point of time? The answer could be 0, 100, 1000, or more than 10 millions. In the last case, these instances are deployed over millions of devices. Also, these instances require the instantiation of roles $\Lambda VideoR$, $\Lambda MusicR$, and $\Lambda ImageR$ over specific devices. How many BDUs are involved in this scenario? This depends on the design and implementation, however it is a large number in any case. All of these details must be kept accurately in the registry. Otherwise, important decisions during deployment can not be decidable. The size of the registry database in some of our tests was more than 12 GBytes.

In our tests, we run the registry utility on a powerful server that has 12 processors and 24 GBytes of RAM. Registry keeps the data structure in RAM, however, and for many reasons, it is capable of saving this information to storage. Accessing registry is done using one of two methods. First: the interactive mode. Usually this is used by administrators to register newly installed CCs, or to fix errors. Second: the network mode. This mode is the normal way that CCMS performs registry access. That is natural since registry runs on a dedicated server, as result, any access need to be network access. Registry utility provides two sets of API functions:

- Modify Set.
- Query Set.

Listing 9.1: Registry utility modify set.

```
Available Options:
0 - delete deployed BDU.
1 - delete deployed CC.
2 - delete installed CC.
3 - edit deployed BDU.
```

```
4 - edit installed CC.
5 - edit deployed CC.
6 - add deployed BDU.
7 - add deployed CC.
8 - add installed CC.
9 - list all deployed CC.
10- list all installed CC.
11- exit.
```

The modify set is presented in listing 9.1. The query set is similar, but more detailed, read only, and provides access to each and every piece of date in the database. Listing 9.2 gives an overview of our implementation choice on implementing the registry utility database. The operations in listing 9.1 work on the data structure presented in listing 9.2. For example the operation `edit deployed CC` allows the user (or CCMS) to modify the name of any wanted deployed CC. If we need to make more changes we can choose `edit deployed BDU` or `delete deployed BDU`, etc.

Listing 9.2: The base foundation of the data structure of the registry utility database.

```
#define STRING_SIZE 100
enum status_options
{
        UNKNOWN,
        INSTALLED,
        DEPLOYED,
        RUNNING,
```

```
        PROBLEM
};
enum BDU_type
{
        Unknown,
        Phi,
        Xi
};
struct BDU
{
        char bdu_name[STRING_SIZE];
        enum BDU_type bdu_type;
        char bdu_version [STRING_SIZE];
        enum status_options bdu_status;
        char bdu_location[STRING_SIZE];
        char bdu_remarks[10*STRING_SIZE];
};
struct cloud_component
{
        char CC_name [STRING_SIZE];
        int BDU_number;
        struct BDU* bdu_list;
};
struct simple_list
{
        char CC_name [STRING_SIZE];
```

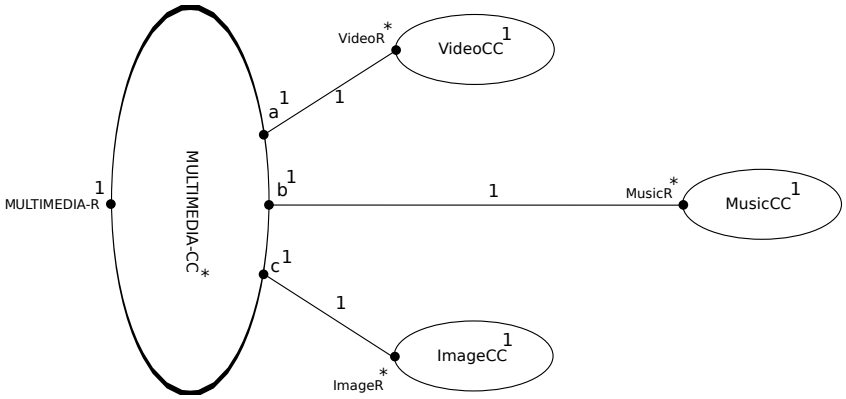


Figure 9.4: The Multimedia cloud component-based system.

```
        char CC_remarks[10*STRING_SIZE];
};
struct installed_list_s
{
        int number_of_installed_CC;
        struct simple_list* CC_list;
};
struct deployed_list_s
{
        int number_of_deployed_CC;
        struct cloud_component* CC_list;
};
```

9.3 Deployment of a CC based system - Deployment Plan - CCMS

For effectively understanding the deployment problem of cloud component based systems, we need to consider the following factors:

- CC is a distributed component in nature.
- Each CC can have zero or more instances at runtime (depending on the multiplicity of the CC itself).
- Each role of a CC can have zero or more instances at runtime (depending on the multiplicity of the role itself).
- The deployment environment of a CC based systems ranges from several devices to hundreds or more of heterogeneous machines and networks.
- The existence of multiple implementation variants to support the above mentioned heterogeneity.

That directly leads to the conclusion that deploying CC based systems over such deployment environments is not a trivial task. Fortunately we succeeded to automate this difficult task 100% using the important tool: the cloud component management system - CCMS. It is important not to confuse the deployment plan with the deployment as a phase in the life-cycle of any software. The deployment plan is part of the development of the CC based system. Moreover, it is an artifact (not an operation). Finally it is written manually by the development team. On the other hand, the deployment of the CC based system is not an artifact, rather, it is an operation. This operation is usually manual for most software up till now (with some exceptions). This operation is fully automated in our proposal. The real power of this contribution is that we do not propose an

automatic deployment of a specific application, rather, we propose the automatic deployment of a class of applications, which is the CC-based-applications. Any application that is written using CC model can be automatically deployed with zero price. All required tools are already provided [2].

CCMS performs what we call *'incremental deployment'* to tackle this problem gradually by executing the *deployment plan* provided by the design/implementation teams as part of the formal description of the cloud component based system. Before deployment the first step for any CC-based-system is to run a certain utility called *ccPack*, which creates archive files (tar files and apk files) that contains the required BDUs. The ccPack utility depends on the formal description of the system to pack the correct BDUs together, along with the appropriate meta-data. These archives are the installation artifacts. Archives are placed inside a special directory called CAB. Up till now we did not start the deployment phase. We can call these operations pre-deployment. CCMS can access CAB to do the required operations.

To complete the installation [3] we need to register the cloud components and all of the installed BDUs. The administrator can perform this step manually through appropriate access to the registry utility using interactive mode. The installation action is summarized as follows:

- Pack developed BDUs using ccPack utility. This step generates tar and apk archives.
- Put archives in CAB directory.
- Register installed CCs and their BDUs using registry utility.

2. Mainly CCMS, the Registry utility, and the ontology-based checker.
3. Please recall that installation precedes deployment. They are completely different operations. The first is manual (semi-automated via ccPack utility). The later is 100% automatic.

In section 7.2 we proposed a CC assembly model along with an automatic assembly checker. If a CC based design passes this checker without errors, then we guarantee there exists a deployment plan that can deploy the whole system without any runtime error. On the contrary, if the checker in section 7.2.4 generates error(s), then there does not exist a deployment plan for such a system.

The deployment plan is a series of BDU deployment operations (section 9.1), dependency checking, assembly operations, registry query, and registry updates. The deployment plan is carried out automatically by the CCMS. CCMS, while performing the deployment plan, faces the deployment of a role (more precisely, its BDU), of which there is a binding (assembly) to another role such as Λb and $\Lambda MusicR$ in figure 9.4. These two roles need to connect at runtime. This is the responsibility of CCMS. Based on the constraint discussed in section 7.2.1.2, this connection can not be remote, it is local with respect to the deployment device. We facilitate this constraint by realizing this connection using *shared memory*. CCMS allocates a local block of RAM. This block of memory serves as a common channels between the two BDUs. They can perform function request, pass parameters, and receive results.

9.3.1 Remark

CCMS itself is a HDE application. The deployment of CCMS is manual and is not related to the deployment of CC-based-systems that is described in this section.

9.4 The Deployment of Multimedia system

The purpose of MULTIMEDIA application [4] is to be a single application to store, search, process, and play all multimedia files like pictures/images, music, and video. This application is expected to be deployed over a highly distributed platform, please see figure 9.4.

Passing the design in figure 9.4 to the assembly checker described in section 7.2.4 will result six warnings and no errors. These warnings are related to the multiplicity $*$ of roles $\Lambda VideoR$, $\Lambda MusicR$, and $\Lambda ImageR$ and the multiplicity $*$ of the CC $\Omega MultimediaCC$. Since $*$ means zero or more, then there is a need to ensure the existence of the role instance before an attempt to bind to it. This should be reflected in the deployment plan. However, since the assembly checker did not generate any errors, there exists a deployment plan for this system (this is explained in the previous section).

The deployment of a single instance of $\Omega MultimediaCC$ over an end-user device contains the following operations:

- Checking the states of $\Omega VideoCC$, $\Omega MusicCC$, and $\Omega ImageCC$ to ensure they are all running.
- Deploy an instance of $\Lambda VideoR$, $\Lambda MusicR$, and $\Lambda ImageR$ on that device, and register these instances.
- Deploy $\Omega MultimediaCC$ on that device and register it as deployed (not running).
- Deploy an instance of Λa, Λb, Λc, and $\Lambda MultimediaR$ on that device, and register these instances.

4. Also mentioned in section 7.4

- Bind $\Omega MultimediaCC$ to the rest of CCs through binding role instances as in figure 9.4. This binding is done by the CCMS which creates three shared memory channels. Each channel is used by one of the three bindings: $(\Gamma a, \Gamma VideoR)$, $(\Gamma b, \Gamma MusicR)$, and $(\Gamma c, \Gamma ImageR)$.
- Register this instance of $\Omega MultimediaCC$ as running.

The above steps reflect the deployment plan of the MULTIMEDIA application [5]. Of course, this is partial snapshot of the complete deployment plan.

For validation purposes, we conducted actual automatic deployment of the multimedia system using the full implementation of:

- The cloud component management system - CCMS.
- The registry utility.
- The multimedia application.

The deployment experiments are performed over a deployment environment with the following hosts:

- A powerful server (12 processors and 24 GBytes RAM) to run the registry utility. This server uses fixed connection (Fast Ethernet).
- A normal server to run CCMS tool. This server also uses fixed connection (Fast Ethernet).
- set_1: a set of laptops with Wi-Fi connections.
- set_2: a set of smart-phones with android operating system and 3G connections.

Listing 9.3: A snapshot of the output log of the registry before the deployment of the first MultimediaCC instance. This is partial snapshot.

```
number_of_deployed_CC: 3
```

5. Please note that there is no cyclic dependency in this deployment plan. Cyclic dependency should always be avoided.

```
>>>>>>>>>>>>>>>>>> order = 0 >>>>>>>>>>>>>>
CC name: ImageCC
RUNNING
Number of BDUs: 1
**********
Start BDU number 0:
BDU name: Image_BUFF
Phi
version: v50.02
RUNNING
location: Virtual Machine
remarks: Testing Deployment
**********
End BDU number 0:
>>>>>>>>>>>>>>>>>> order = 1 >>>>>>>>>>>>>>
CC name: MusicCC
RUNNING
Number of BDUs: 1
**********
Start BDU number 0:
BDU name: Music_BUFF
Phi
version: v12.00
RUNNING
location: Virtual Machine
remarks: Testing Deployment
```

```
**********
End BDU number 0:
>>>>>>>>>>>>>>>>>> order = 2 <<<<<<<<<<<<<<<<
CC name: VideoCC
RUNNING
Number of BDUs: 1
**********
Start BDU number 0:
BDU name: Video_BUFF
Phi
version: v118.09
RUNNING
location: Virtual Machine
remarks: Testing Deployment
**********
End BDU number 0#
```

Listing 9.4: A snapshot of the output log of the registry after the deployment of the first MultimediaCC instance. This is partial snapshot.

```
number_of_deployed_CC: 4
>>>>>>>>>>>>>>>>>> order = 0 <<<<<<<<<<<<<<<<
CC name: ImageCC
RUNNING
Number of BDUs: 2
**********
Start BDU number 0:
```

```
BDU name: Image_BUFF
Phi
version: v50.02
RUNNING
location: Virtual Machine
remarks: Testing Deployment
**********
End BDU number 0:
**********
Start BDU number 1:
BDU name: ImageR
Xi
version: vXXX
RUNNING
location: Laptop
remarks: Automatic testing ..........
**********
End BDU number 1:
>>>>>>>>>>>>>>>>>> order = 1 >>>>>>>>>>>>>>
CC name: MusicCC
RUNNING
Number of BDUs: 2
**********
Start BDU number 0:

(omitted)
```

```
>>>>>>>>>>>>>>>>>>> order = 3 >>>>>>>>>>>>>>>
CC name: MultimediaCC
RUNNING
Number of BDUs: 4
**********
Start BDU number 0:
BDU name: a
Xi
version: vXXX
RUNNING
location: Laptop
remarks: Automatic testing ..............................
**********
End BDU number 0:
**********
Start BDU number 1:
BDU name: b
Xi
version: vXXX
RUNNING
location: Laptop
remarks: Automatic testing ..............................
**********
End BDU number 1:
**********
```

```
Start BDU number 2:
BDU name: c
Xi
version: vXXX
RUNNING
location: Laptop
remarks: Automatic testing ..............................
**********
End BDU number 2:
**********
Start BDU number 3:
BDU name: MultimediaR
Xi
version: vXXX
RUNNING
location: Laptop
remarks: Automatic testing ..............................
**********
End BDU number 3
```

In listings 9.3 and 9.4, a snapshot of the registry is presented before and after the deployment of the first instance of $\Omega MultimediaCC$. In figure 9.5 we show the timing results when no concurrent requests are made. The total $\tau(MultimediaCC)$ (please review 7.1.3) is equally distributed between set_1 and set_2 for all cases. While in figure 9.6, set_1 and set_2 made their request concurrently. In this experiment also, the total $\tau(MultimediaCC)$ is equally

distributed between set_1 and set_2 for all cases.

The QoS of the Multimedia application itself over the different devices is discussed in section 7.4.5. The most important result from these experiments was that there were no runtime errors, neither at deployment stage, nor at running stages. This validates our proposal as follows:

- Following CC model will allow us to build systems that can *scale up* to many users, in this case 1000 user, or more, easily.
- The deployment conjecture is validated in this case study.
- These experiments *practically show the automatic deployment* of a CC-based-system over a HDE. Again, this automatic deployment is achieved without errors and without incompatibilities between software and hardware.

The main reason for these positive results is that all related details were taken care of during design and implementation with the support of the CC model, formal notation and checkers, CCMS/Registry tools, and ontology modeling and checking support (discussed in section 8.5). That is a direct consequence of *Localization Acknowledgment.*

Figures 9.5 and 9.6 show that concurrent deployment requests has no effect on the deployment time. A laptop with Wi-Fi connection needs between 340 milliseconds and 520 milliseconds to deploy a Multimedia instance, while a smartphone with 3G needs between 8.8 seconds and 9.5 seconds for the same deployment. We think this is the effect of the network latency. It is worth noting also that the registry size in these experiments was 19 GBytes. For larger systems, and/or many-same-size systems the registry size will grow linearly. At the same time, we think the registry is efficient enough for all query and update opera-

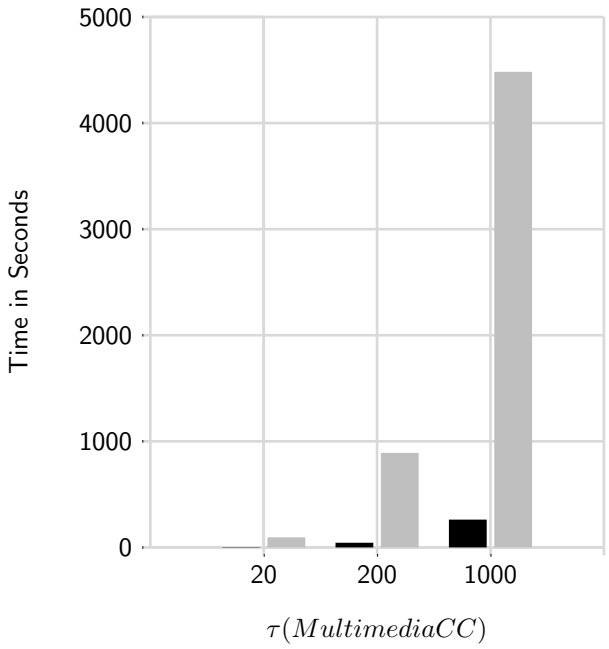


Figure 9.5: Total deployment time for the complete multimedia system for different values of $\tau(MultimediaCC)$ (Explained in 7.1.3). Deployment requests are sequential. The black column represents time required for set_1 (laptops, WiFi) to finish the experiment. The gray column represents time required for set_2 (smart-phones, 3G) to finish the experiment.

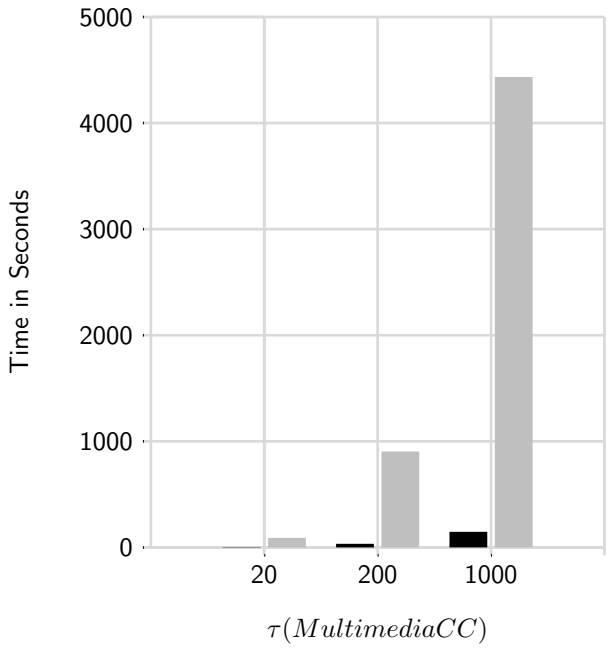


Figure 9.6: Total deployment time for the complete multimedia system for different values of $\tau(MultimediaCC)$ (Explained in 7.1.3). Deployment requests are concurrent. The black column represents time required for set_1 (laptops, WiFi) to finish the experiment. The gray column represents time required for set_2 (smart-phones, 3G) to finish the experiment.

tions. If needed, many optimizations could be applied to this utility to reduce the size needed, and/or to speedup query/update operations.

9.5 Chapter Comments

The deployment of a HDE application is not a simple task as discussed in section 9.3. Using the theory proposed in this dissertation, along with CCMS tool and Registry utility, we think this task became affordable (technical wise) for software developers immediately. The real merit of the tools proposed in this chapter is evident when we see the application domain of these tools. It is not the MULTIMEDIA application only, but rather, any CC-based application a programmer (or a software development company) develops in the future. This chapter is not intended to provide a detailed description of the usage of the tools. Rather, it is designed to introduce these tools, the operation they accomplish, and the automation they provide. Our hope is that the computer science community and the software development community will find this approach useful.

Part IV

CONCLUSION

Chapter 10

Conclusion

Highly distributed environments are a newcomer to the computing world. HDEs are deployment environments that include normal stable devices and networks such as servers, workstations, and fixed Ethernet. However, HDEs include in addition laptops, smartphones, sensor networks, along with WIFI, 3G, and radio frequency (RF) connections. This set of heterogeneous hardware and the collection of reliable and unreliable networks constitute the newcomer: HDEs.

These environments have created several challenges to software development such as: disconnected operations, unreliable networks, low resources in user devices, small display, etc. And yet, software is still expected to be reliable and show the expected QoS.

There is no doubt that current software development process, along with current software development skills in general can not meet these objectives sufficiently. Until now, software development for HDEs is ad-hoc. Each application developer tries to implement some techniques and checks for his/her application to meet the specifications over these environment. However, there is no standard process,

nor customized model that can be used systematically to produce high quality applications for HDEs.

In this work we propose a software component model, the cloud component model, to fill this gap between current software development needs, and the available software engineering theory and methods. Cloud component model is based on a paradigm shift from distribution transparency to localization acknowledgment being the first class concern. In other words, we no more hide or abstract location, on the contrary, we acknowledge all aspects related to location including the specification of devices, the networking paradigms they use, the different network specifications available, security features, and all related characteristics of the deployment environment. Also, we propose a theory to assemble CCs in order to build a CC-based applications. Formal notation is proposed for CC, CC assembly, CC development process, and CC based systems. This formal notation opens the door for a wide range of theoretical topics including component type inference, subtypes, etc, and provides a precise language to describe details. In addition, formal methods allow the designer to produce machine readable designs where automated tools can verify specific properties at design time, which in turn, increases the level of confidence in the correctness of design.

One of the main values of the CC model is its support for a full compatibility between software and hardware in HDEs. This is a major challenge knowing the heterogeneity of these environments. Our model uses the first proposed ontology based hardware modeling and software requirement modeling and the ontology based software/hardware checker. This checker, along with the proposed CC development process are the foundations of our claim that *CC model guaran-*

tee the expected QoS at the user end point. We supported this model with fully implemented tools including: CC assembly checker, CC ontology based software/hardware checker, cloud component management system (CCMS) and registry utility. Finally, we fully implemented a CC-based system, the multimedia application, to validate our proposal.

The contribution in this dissertation has several faces, but still, these faces are cohesive. Each of these faces form a partial contribution, however, this partial contribution does not mean anything if isolated from the overall proposal. Moreover, the merit of the overall proposal can not be grasped by reading one partial contribution. The merit of the proposal is evident only if all parts of this work are cohesively organized. The limitations of this model and the future work are discussed in the following sections.

10.1 Limitation of the Proposed Approach

It is an old dream of computer science to verify the correctness of software systems by fully automatic means. Despite of well-known fundamental limits uncovered by the theory of computability, there has been great progress in the study of automatic methods for semantic analysis of software in the last decades. In particular it has been studied how to apply model checking to software. Model checking verifies temporal-logic specifications by exhaustive state-space exploration. It is an automated technique promising to ensure (limited) correctness of software and to find certain classes of bugs automatically. The work of Javier Esparza, Wan Fokkink, Marta Kwiatkowska, and Markus Muller-Olm can be reviewed for more information over this topic.

It could be considered as a limitations in this work that we do not have such dynamic checking in our model. For example, we do not describe the dynamic (temporal) characteristics of a CC role, as result, it is the responsibility of the designer to ensure the correct dynamic behavior of roles at runtime. On the other hand, avoiding such formal methods is intended to keep the model easy to use by average software engineers.

Another limitation is the requirement of the existence of a powerful machine that runs the CCMS and the Registry utility. Moreover, we assume a connection between this machine and all deployment devices. While the last condition could be relaxed, we did not study the effect of the absolute non-existence of such a machine over the CC model. Is it still usable? And how to achieve that? Majority of applications does not require such extreme condition, however, it is still interesting to study the potential of the CC model in such severe situation.

10.2 Future Work

10.2.1 Automatic acquisition of device ontology

As described in figure 8.15 section 8.5.1, the `IndividualOnto` is populated by `Tablet-Ju` instance. This population process is done manually. A human reads the specifications in the device, and populate the ontology accordingly. Using 'information extraction' discussed in section 4.3.1, it is possible to achieve this automatically. This will allow for fast construction of an up-to-date comprehensive knowledge base for almost all devices available. This knowledge base is automatically updated upon the arrival of a new device.

10.2.2 Easy modelling of software requirements

Back to figure figure 8.15. When the designer/programmer needs to model the runtime requirements of `BDU-X` he/she needs to model it as an ontology query, or a mix of rules and queries. This should be done using a formal language such as F-logic. A simple High Definition (HD) display requirement in F-logic is presented in listing listing 8.2. It is inviting to automate this part. The automation could use `IndividualOnto` and the software requirement in English language. The output of the automated tool will be the equivalent F-logic rules and queries. In section 4.3.2 we discussed such approach: 'question answering'.

Adding this feature to our model will make it extremely effective and easy to use. Moreover, if we succeeded in the two features (i.e. this section and sections 10.2.1), we will succeed in hiding ontology to a great extend. Ontology will be completely a back-end support in our model, while the front-end is a simple and easy to use interface. Our model is easy to use, and adding these two features will make the difference between our model and the normal software development process negligible (please read section 2.3).

10.2.3 Study the effect of CC-model on software component reuse

Through out this dissertation we emphasized on the encapsulation attribute of the software component in general. Also, we were interested by the encapsulation power of the CC model. That was one of the major contributions of this work.

However, there is a another value of software components that many literature acknowledge, which is 'software reuse'. We think that software components in general succeeded in its first attribute which is encapsulation, and based on that, allowed building large applications using the assembly of 'parts' which are software components. On the other hand, software components as a tool to facilitate software reuse were less successful. We think there are several reasons behind that. One of these reasons could be that current component models describe their functionality on the source side not on the destination side. It could be fruitful to study the effect on our proposed paradigm shift over software component reuse. Moreover, the effect on the cloud component model on the component contract in general.

10.2.4 Formal theory for CC state & BDU state

In figure 9.3 section 9.1.7 we discussed the four states that a CC could be in at any any given time. Also, these are the four states that any BDU could be in at any any given time. All of these states are saved in the registry utility at runtime. Since a CC is composed of BDUs, there is a direct effect of the state of a BDU over the state of the CC which this BDU is a compound of. Currently, we leave this issue to the designer of each individual application. In other words, inside the deployment plan, we have explicit instructions to check the state of a BDU (or a CC) before certain actions such as deployment of a new role.

We think it could be useful, for theoretical reasons at least, to further study this relationship. A formal theory could describe this relationship. This theory could be useful for better runtime management of CC based systems. It also could allow us to better understand the reliability of such systems.

BIBLIOGRAPHY

Bibliography

[1] T. Strang and C. Linnhoff-Popien, "A context modelling survey," in *Proceedings of First International Workshop on Advanced Context Modelling, Reasoning And Management*, 2004. xi, 4, 23, 74

[2] D. Hoareau and Y. Maheo, "Distribution of a hierarchical component in a non-connected environment," in *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, EUROMICRO '05, (Washington, DC, USA), pp. 143–151, IEEE Computer Society, 2005. xi, xii, 8, 9, 28, 58, 65, 77, 78

[3] S. Malek, G. Edwards, Y. Brun, H. Tajalli, J. Garcia, I. Krka, N. Medvidovic, M. Mikic-Rakic, and S. G. S., "An architecture-driven software mobility framework," *Journal of Systems and Software*, vol. 83, pp. 972–989, June 2010. xi, xii, 2, 6, 9, 10, 23, 30, 82, 83, 84, 85

[4] P. Cimiano, *Ontology Learning and Population from Text: Algorithms, Evaluation and Applications*. Springer, 2006. xii, 53

[5] T. Bureš, M. Děcký, P. Hnětynka, J. Kofroň, P. Parízek, F. Plášil, T. Poch, O. Šerý, and P. Tůma, "The common component modeling example," ch. CoCoME in SOFA, pp. 388–417, Berlin, Heidelberg: Springer-Verlag, 2008. xii, 61, 62

[6] X. H. Wang, D. Q. Zhang, T. Gu, and H. K. Pung, "Ontology based context modeling and reasoning using owl," in *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, PERCOMW '04, (Washington, DC, USA), pp. 18–, IEEE Computer Society, 2004. xii, 75, 76

[7] M. Mikic-Rakic, *Software architectural support for disconnected operation in distributed environments*. PhD thesis, University of Southern California, Los Angeles, CA, USA, December 2004. xii, 2, 5, 22, 24, 28, 59, 81, 82

[8] C. Consel, W. Jouve, J. Lancia, and N. Palix, "Ontology-directed generation of frameworks for pervasive service development," *Pervasive Computing and Communications Workshops, IEEE International Conference on*, vol. 0, pp. 501–508, 2007. xii, 86, 87

[9] M. Uschold and M. Gruninger, "Ontologies and semantics for seamless connectivity," *SIGMOD Rec.*, vol. 33, pp. 58–64, Dec. 2004. xvi, 156, 157

[10] S. Staab and R. Studer, *Handbook on Ontologies*. Springer Berlin Heidelberg, 2009. xvi, xvii, 45, 51, 155, 156, 157, 158, 159, 160, 161, 165, 175

[11] C. K. Ogden and I. A. Richards, *The Meaning of Meaning: A Study of the Influence of Language upon Thought and of the Science of Symbolism*. Routledge & Kegan Paul, 1923. xvi, 158, 159

[12] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, eds., *The Common Component Modeling Example: Comparing Software Component Models*. No. 5153 in Lecture Notes in Computer Science, Berlin, Heidelberg: Springer-Verlag, 2008. xix, 28, 68, 70, 71, 72

[13] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. Chaudron, "A classification framework for software component models," *IEEE Transactions on Software Engineering*, vol. 37, pp. 593–615, 2011. xix, 29, 67, 68, 69, 72, 74, 96

[14] R. Guerraoui and M. E. Fayad, "Oo distributed programming is not distributed oo programming," *Commun. ACM*, vol. 42, pp. 101–104, Apr. 1999. 2, 6, 23, 30

[15] Y. Zhang, V. Paxson, and S. Shenker, "The Stationarity of Internet Path Properties: Routing, Loss, and Throughput," tech. rep., AT&T Center for Internet Research at ICSI, 2000. 3, 24

[16] D. Hoareau, *Composants ubiquitaires pour reseaux dynamiques*. PhD Dissertation, Universite de Bretagne Sud, Decembre 2007. 5, 8, 25, 28, 59, 77, 79, 80

[17] D. Hoareau and Y. Mahéo, "Middleware support for the deployment of ubiquitous software components," *Personal and Ubiquitous Computing*, vol. 12, pp. 167–178, January 2008. 8, 28, 58, 77

[18] G. Edwards and N. Medvidovic, "A methodology and framework for creating domain-specific development infrastructures," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pp. 168 –177, sept. 2008. 9, 83

[19] G. Edwards, S. Malek, and N. Medvidovic, "Scenario-driven dynamic analysis of distributed architectures," in *Proceedings of the 10th international conference on Fundamental approaches to software engineering*, FASE'07, (Berlin, Heidelberg), pp. 125–139, Springer-Verlag, 2007. 9, 83

[20] "IEEE Standard Glossary of Software Engineering Terminology." Standards Coordinating Committee of the Computer Society of the IEEE. `http://ieeexplore.ieee.org/servlet/opac?punumber=2238`, last visit April 28^{th}, 2012. 26, 50, 127

[21] S. R. Schach, *Object-Oriented and Classical Software Engineering*. McGraw-Hill, 2006. 26, 50, 127

[22] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, pp. 40–52, Oct. 1992. 28

[23] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, vol. 123. Prentice Hall, 1996. 28

[24] K.-K. Lau and Z. Wang, "Software component models," *IEEE Transaction on Software Engineering*, vol. 33, pp. 709–724, October 2007. 28, 29, 67, 94

[25] E. E. Group, "JSR 220: Enterprise JavaBeansTM,Version 3.0 EJB Core Contracts and Requirements Version 3.0, Final Release," May 2006. 28, 60, 66, 72

[26] "OMG CORBA Component Model v4.0." `http://www.omg.org/spec/CCM/4.0/`, last visit February 6^{th}, 2012. 28, 60, 64, 72

[27] OSGi Alliance, "OSGi Service Plaform Core Specification, V4.1," 2007. 28, 60, 66, 72

[28] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java," *Software: Practice and Experience*, vol. 36, no. 11-12, pp. 1257–1284, 2006. 28, 60, 64, 72

[29] T. Bures, P. Hnetynka, and F. Plasil, "Sofa 2.0: Balancing advanced features in a hierarchical component model," in *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, (Washington, DC, USA), pp. 40–48, IEEE Computer Society, 2006. 28, 72

[30] "`http://sofa.ow2.org/`." last visit February 6^{th}, 2012. 28, 60, 61, 68, 72

[31] V. Hourdin, J.-Y. Tigli, S. Lavirotte, G. Rey, and M. Riveill, "Slca, composite services for ubiquitous computing," in *Proceedings of the International Conference on Mobile Technology, Applications, and Systems*, Mobility '08, (New York, NY, USA), pp. 11:1–11:8, ACM, 2008. 29, 59

[32] N. Bussiere, D. Cheung-Foo-Wo, V. Hourdin, S. Lavirotte, M. Riveill, and J.-Y. Tigli, "Optimized contextual discovery of web services for devices," 2007. 29

[33] A. Charfi, B. Schmeling, A. Heizenreder, and M. Mezini, "Reliable, secure, and transacted web service compositions with ao4bpel," in *Proceedings of the European Conference on Web Services*, ECOWS '06, (Washington, DC, USA), pp. 23–34, IEEE Computer Society, 2006. 29

[34] M. Vallee, F. Ramparany, and L. Vercouter, "Flexible composition of smart device services," 2005. 29

[35] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, pp. 39–59, Feb. 1984. 29

[36] F. Baader and U. Sattler, "An overview of tableau algorithms for description logics," *Studia Logica*, pp. 5–40, 2001. 45

[37] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003. 45, 46, 47

[38] D. Calvanese, G. D. Giacomo, M. Lenzerini, and D. Nardi, *Reasoning in expressive description logics.* In Alan Robinson and Andrei Voronkov, editors, Handbook of Automated Reasoning, chapter 23, pages 1581-1634. Elsevier, 2001. 45

[39] D. L. McGuinness and F. van Harmelen, "OWL Web Ontology Language Overview." `http://www.w3.org/TR/2003/WD-owl-features-20030331/`, last visit April 26^{th}, 2012. 45

[40] F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein, "OWL Web Ontology Language Reference." `http://www.w3.org/TR/2003/WD-owl-ref-20030331/`, last visit April 26^{th}, 2012. 45

[41] P. F. Patel-Schneider, P. Hayes, and I. Horrocks, "OWL Web Ontology Language Semantics and Abstract Syntax." `http://www.w3.org/TR/2003/WD-owl-semantics-20030331/`, last visit April 26^{th}, 2012. 45

[42] P. Hayes and B. McBride, "RDF Semantics." `http://www.w3.org/TR/rdf-mt/`, last visit April 26^{th}, 2012. 45

[43] I. Horrocks, U. Sattler, and S. Tobies, "Practical reasoning for very expressive description logics," *Journal of the Interest Group in Pure and Applied Logic*, vol. 8(3), pp. 239–264, 2000. 45

[44] D. Tsarkov and I. Horrocks, "FaCT++ description logic reasoner: system description," in *Proc. of the Int. Joint Conf. on Automated Reason-*

ing (IJCAR 2006), vol. 4130 of Lecture Notes in Artificial Intelligence, pp. 292–297, Springer, 2006. 45

[45] V. Haarslev and R. Möller, "Racer system description," in *Proceedings of the First International Joint Conference on Automated Reasoning*, IJCAR '01, (London, UK, UK), pp. 701–706, Springer-Verlag, 2001. 45

[46] E. Sirin and B. Parsia, "Pellet: An OWL DL Reasoner," in *In Proceedings of the 2004 Description Logic Workshop (DL 2004)*, 2004. 45

[47] I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen, "From SHIQ and RDF to OWL: The Making of a Web Ontology Language," *Journal of Web Semantics*, vol. 1(1), pp. 7–26, 2003. 46

[48] F. Baader and U. Sattler, "Description logics with symbolic number restrictions," in *In Proceedings of the 12th European Conference on Artificial Intelligence (ECAI 96)*, pp. 283–287, John Wiley & Sons, 1996. 46

[49] F. Baader and U. Sattler, "Number restrictions on complex roles in Description Logics: A preliminary report," in *In Proceedings of the 5th International Conference on the Principles of Knowledge Representation and Reasoning(KR 96)*, pp. 328–338, 1996. 46

[50] F. Baader and U. Sattler, "Expressive number restrictions in Description Logics," *Journal of Logic and Computation*, vol. 9(3), pp. 319–350, 1999. 46

[51] J. H. Gennari, M. A. Musen, R. W. Fergerson, W. E. Grosso, M. Crubézy, H. Eriksson, N. F. Noy, and S. W. Tu, "The evolution of protégé: an environment for knowledge-based systems development," *Int. J. Hum.-Comput. Stud.*, vol. 58, pp. 89–123, Jan. 2003. 47

[52] Breitman, Karen Koogan and Leite, Julio Cesar Sampaio do Prado, "Ontology as a requirements engineering product," in *Proceedings of the 11th IEEE International Conference on Requirements Engineering*, RE '03, (Washington, DC, USA), pp. 309–319, IEEE Computer Society, 2003. 50

[53] P. Spyns, Y. Tang, and R. Meersman, "An ontology engineering methodology for DOGMA," *Appl. Ontol.*, vol. 3, pp. 13–39, Jan. 2008. 50

[54] J. Hendler, "Agents and the semantic web," *IEEE Intelligent Systems*, vol. 16(2), pp. 30–37, Mar. 2001. 51, 157

[55] A. Maedche and S. Staab, "Ontology learning for the semantic web," *IEEE Intelligent Systems*, vol. 16, pp. 72–79, Mar. 2001. 51, 53

[56] T. Burger, "Putting business intelligence into documents.," in *In Proc. of the WSh. on Semantic Business Process and Product Lifecycle Management*, 2007. 51

[57] S. W. Lee and R. A. Gandhi, "Ontology-based active requirements engineering framework," in *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, APSEC '05, (Washington, DC, USA), pp. 481–490, IEEE Computer Society, 2005. 51

[58] D. Damian, "Stakeholders in global requirements engineering: Lessons learned from practice," *Software, IEEE*, vol. 24, pp. 21 –27, march-april 2007. 51

[59] K. Siorpaes and M. Hepp, "myontology: The marriage of ontology engineering and collective intelligence," in *In Proc. of the Wsh. on Bridging the Gep between Semantic Web and Web 2.0*, pp. 127–138, 2007. 51

[60] K. C. Desouza, Y. Awazu, and P. Baloh, "Managing knowledge in global software development efforts: Issues and practices," *IEEE Softw.*, vol. 23,

pp. 30–37, Sept. 2006. 51

[61] S. B. Marc, M. Ehrig, A. Koschmider, A. Oberweis, and R. Studer, "Semantic alignment of business processes," in *In Proc. of the 8th International Conference on Enterprise Info. Sys.*, pp. 191–196, 2006. 51

[62] F. Lautenbacher and B. Bauer, "A survey on workflow annotation & composition approaches.," in *In Proc. of the Wsh. on Semantic Business Process and Product Lifecycle Management - SBPM*, 2007. 51

[63] S. Cranefield, "Uml and the semantic web," in *In Proceedings of the Semantic Web Working Symposium*, pp. 113–130, 2001. 51

[64] D. Gaaevic, D. Djuric, V. Devedzic, and B. Selic, *Model Driven Architecture and Ontology Development*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. 51

[65] Y. Pan, G. Xie, L. Ma, Y. Yang, Z. Qiu, and J. Lee, "Model-driven ontology engineering," vol. 4244, pp. 57–78, 2006. 51

[66] R. Gronmo, M. Jaeger, and H. Hoff, "Transformations between uml and owl-s," in *In Proc. of the 1st European Conference of Model Driven Architecture – Foundations and Applications* (A. Hartman and D. Kreische, eds.), vol. 3748 of *Lecture Notes in Computer Science*, pp. 269–283, Springer Berlin / Heidelberg, 2005. 51

[67] D. Berardi, D. Calvanese, and G. De Giacomo, "Reasoning on UML class diagrams," *Artif. Intell.*, vol. 168, pp. 70–118, Oct. 2005. 51

[68] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering*, FOSE '07, (Washington, DC, USA), pp. 37–54, IEEE Computer Society, 2007. 51

[69] G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, and M. Wimmer, "Lifting metamodels to ontologies: a step to the semantic integration of modeling languages," in *Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems*, MoDELS'06, (Berlin, Heidelberg), pp. 528–542, Springer-Verlag, 2006. 51

[70] S. Roser and B. Bauer, "An approach to automatically generated model transformations using ontology engineering space," in *In Proceedings of the 2nd international WSh. on Semantic Web Enabled Software Eng.*, 2006. 51

[71] F. Jouault, J. Bézivin, and I. Kurtev, "Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering," in *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, (New York, NY, USA), pp. 249–254, ACM, 2006. 51

[72] A. Kalyanpur, D. J. Pastor, S. Battle, and J. Padget, "Automatic mapping of owl ontologies into java," in *Proceedings of Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE)* (G. R. F. Maurer, ed.), pp. 98–103, June 2004. 51

[73] M. Voelkel and Y. Sure, "Rdfreactor - from ontologies to programmatic data access," in *In Poster Proceedings of the 4th International Semantic Web Conference*, 2005. 51

[74] S. Dietze, A. Gugliotta, , and J. Domingue, "A semantic web services-based infrastructure for context-adaptive process support," in *Web Services, 2007. ICWS 2007. IEEE International Conference on*, pp. 537 –543, july 2007. 51

[75] D. Oberle, *The Semantic management of middleware*. Springer, 2006. 51

[76] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999. 53

[77] K. Frantzi, S. Ananiadou, and H. Mima, "Automatic recognition of multi-word terms:. the C-value/NC-value method," *International Journal on Digital Libraries*, vol. 3, pp. 115–130, 2000. 53

[78] K. Frantzi and S. Ananiadou, "The C-Value/NC-Value domain independent method for multi-word term extraction," *Journal of Natural Language Processing*, vol. 6(3), pp. 145–179, 1999. 53

[79] Z. S. Harris, *Mathematical Structures of Language*. Wiley, 1968. 53

[80] G. Grefenstette, "SEXTANT: exploring unexplored contexts for semantic extraction from syntactic analysis," in *Proceedings of the 30th annual meeting on Association for Computational Linguistics*, ACL '92, (Stroudsburg, PA, USA), pp. 324–326, Association for Computational Linguistics, 1992. 53

[81] D. Lin, "Automatic retrieval and clustering of similar words," in *Proceedings of the 17th international conference on Computational linguistics - Volume 2*, COLING '98, (Stroudsburg, PA, USA), pp. 768–774, Association for Computational Linguistics, 1998. 53

[82] J. R. Curran, "Ensemble methods for automatic thesaurus extraction," in *Proceedings of the ACL-02 conference on Empirical methods in natural language processing - Volume 10*, EMNLP '02, (Stroudsburg, PA, USA), pp. 222–229, Association for Computational Linguistics, 2002. 53

[83] M. Baroni and S. Bisi, "Using cooccurrence statistics and the web to discover synonyms in technical language," in *In Proceedings of the 4th In-*

ternational Conference on Language Resources and Evaluation (LREC), pp. 1725–1728, 2004. 53

[84] P. D. Turney, "Mining the Web for Synonyms: PMI-IR versus LSA on TOEFL," in *Proceedings of the 12th European Conference on Machine Learning*, EMCL '01, (London, UK, UK), pp. 491–502, Springer-Verlag, 2001. 53

[85] P. Resnik, "Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language," *Journal of Artificial Intelligence Research (JAIR)*, vol. 11, pp. 95–130, 1999. 53

[86] M. Strube and S. P. Ponzetto, "Wikirelate! computing semantic relatedness using wikipedia," in *proceedings of the 21st national conference on Artificial intelligence - Volume 2*, AAAI'06, pp. 1419–1424, AAAI Press, 2006. 53

[87] C. Fellbaum, *WordNet, an electronic lexical database*. MIT Press, 1998. 53

[88] D. Widdows, "Unsupervised methods for developing taxonomies by combining syntactic and statistical information," in *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, NAACL '03, (Stroudsburg, PA, USA), pp. 197–204, Association for Computational Linguistics, 2003. 53

[89] F. A. Lisi and F. Esposito, "Two orthogonal biases for choosing the intensions of emerging concepts in ontology refinement," in *Proceedings of the 2006 conference on ECAI 2006: 17th European Conference on Artificial*

Intelligence August 29 – September 1, 2006, Riva del Garda, Italy, (Amsterdam, The Netherlands, The Netherlands), pp. 765–766, IOS Press, 2006. 53

[90] G. Bisson, C. Ndellec, and L. Caamero, "Designing clustering methods for ontology building –the mo'k workbench," in *Proceedings of the Ontology Learning Workshop at ECAI*, pp. 13–19, 2000. 53

[91] P. Cimiano, A. Hotho, and S. Staab, "Learning concept hierarchies from text corpora using formal concept analysis," *J. Artif. Int. Res.*, vol. 24, pp. 305–339, Aug. 2005. 53

[92] D. Faure and C. Nedellec, "A corpus-based conceptual clustering method for verb frames and ontology acquisition," in *Proceedings of the LREC Workshop on Adapting lexical and corpus resources to sublanguages and applications*, pp. 5–12, 1998. 53

[93] P. Buitelaar, P. Cimiano, and B. Magnini, *Ontology Learning from Text: Methods, Applications and Evaluation*. IOS Press, 2005. 53, 235

[94] P. VELARDI, R. NAVIGLI, A. CUCHIARELLI, and F. NERI, "Evaluation of OntoLearn, a methodology for automatic population of domain ontologies," in *[93]*, pp. 92–106, 2005. 53

[95] P. Buitelaar, D. Olejnik, and M. Sintek, "A protege plug-in for ontology extraction from text based on linguistic analysis," in *In Proceedings of the 1st European Semantic Web Symposium (ESWS)*, pp. 31–44, 2004. 53

[96] N. Aussenac-Gilles, S. Despres, and S. Szulman, "The TERMINAE Method and Platform for Ontology Engineering from Texts," in *Proceedings of the 2008 conference on Ontology Learning and Population: Bridg-*

ing the Gap between Text and Knowledge, (Amsterdam, The Netherlands, The Netherlands), pp. 199–223, IOS Press, 2008. 53

[97] P. Cimiano and J. Volker, "Text2Onto - A Framework for Ontology Learning and Data-driven Change Discovery," in *Proceedings of the 10th International Conference on Applications of Natural Language to Information Systems (NLDB)*, pp. 227–238, 2005. 53

[98] A. Maedche and S. Staab, "Discovering conceptual relations from text.," in *In Proceedings of the 14th European Conference on Artificial Intelligence (ECAI)*, pp. 321–325, 2000. 53

[99] P. Buitelaar, M. Sintek, and M. Kiesel, "A multilingual/multimedia lexicon model for ontologies," in *Proceedings of the 3rd European conference on The Semantic Web: research and applications*, ESWC'06, (Berlin, Heidelberg), pp. 502–513, Springer-Verlag, 2006. 53

[100] S. Nirenburg and V. Raskin, *Ontological semantics*. MIT Press, 2004. 53

[101] K. Bontcheva, V. Tablan, D. Maynard, and H. Cunningham, "Evolving gate to meet new challenges in language engineering," *Nat. Lang. Eng.*, vol. 10, pp. 349–373, Sept. 2004. 53

[102] B. Popov, A. Kiryakov, D. Ognyanoff, D. Manov, and A. Kirilov, "Kim - a semantic platform for information extraction and retrieval," *Nat. Lang. Eng.*, vol. 10, pp. 375–392, Sept. 2004. 53

[103] D. Ferrucci and A. Lally, "Uima: an architectural approach to unstructured information processing in the corporate research environment," *Nat. Lang. Eng.*, vol. 10, pp. 327–348, Sept. 2004. 53

[104] T. Hamon, A. Nazarenko, T. Poibeau, S. Aubin, and J. Derivière, "A robust linguistic platform for efficient and domain specific web content

analysis," in *Large Scale Semantic Access to Content (Text, Image, Video, and Sound)*, RIAO '07, pp. 226–240, 2007. 53

[105] P. Adolphs, M. Theobald, U. Schafer, H. Uszkoreit, and G. Weikum, "Yago-qa: Answering questions by structured knowledge queries," in *Semantic Computing (ICSC), 2011 Fifth IEEE International Conference on*, pp. 158 –161, sept. 2011. 54

[106] D. A. Ferrucci, E. W. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. M. Prager, N. Schlaefer, and C. A. Welty, "Building watson: An overview of the deepqa project.," pp. 59–79, 2010. 54

[107] L. Hirschman and R. Gaizauskas, "Natural language question answering: the view from here," *Natural Language Engineering*, vol. 7, pp. 275–300, Dec. 2001. 54

[108] C. Kwok, O. Etzioni, and D. S. Weld, "Scaling question answering to the web," *ACM Trans. Inf. Syst.*, vol. 19, pp. 242–262, July 2001. 54

[109] Z. Zheng, "Answerbus question answering system," in *Proceedings of the second international conference on Human Language Technology Research*, HLT '02, (San Francisco, CA, USA), pp. 399–404, Morgan Kaufmann Publishers Inc., 2002. 54

[110] S. Matougui and A. Beugnard, "Two ways of implementing software connections among distributed components," in *OTM Conferences (2)*, pp. 997–1014, 2005. 58, 107

[111] C. Tibermacine, D. Hoareau, and R. Kadri, "Enforcing architecture and deployment constraints of distributed component-based software," *Fundamental Approaches to Software Engineering*, pp. 140–154, 2007. 59

[112] M. Mikic-Rakic and N. Medvidovic, "Software architectural support for disconnected operation in highly distributed environments," in *Component-Based Software Engineering* (I. Crnkovic, J. Stafford, H. Schmidt, and K. Wallnau, eds.), vol. 3054 of *Lecture Notes in Computer Science*, pp. 23–39, Springer Berlin / Heidelberg, 2004. 59

[113] M. Mikic-Rakic and N. Medvidovic, "Architecture-level support for software component deployment in resource constrained environments," in *Component Deployment*, pp. 31–50, 2002. 59, 81

[114] H. Maaskant, "A Robust Component Model for Consumer Electronic Products," in *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices* (P. Stok, ed.), vol. 3 of *Philips Research Book Series*, pp. 167–192, Springer Netherlands, 2005. 60

[115] T. Genbler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, P. Müller, and C. Stich, "Components for embedded software: the PECOS approach," in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '02, (New York, NY, USA), pp. 19–26, ACM, 2002. 60

[116] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in bip," in *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, SEFM '06, (Washington, DC, USA), pp. 3–12, IEEE Computer Society, 2006. 60

[117] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas, "An Efficient Component Model for the Construction of Adaptive Middleware," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, (London, UK, UK), pp. 160–

178, Springer-Verlag, 2001. 60

[118] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković, "A Component Model for Control-Intensive Distributed Embedded Systems," in *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, CBSE '08, (Berlin, Heidelberg), pp. 310–317, Springer-Verlag, 2008. 60

[119] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," *Computer*, vol. 33, pp. 78 –85, mar 2000. 60

[120] D. Box, *Essential COM*. Addison-Wesley, 1997. 60

[121] J. E. Kim, O. Rogalla, S. Kramer, and A. Hamann, "Extracting, specifying and predicting software system properties in component based real-time embedded software development," in *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pp. 28 –38, may 2009. 60

[122] X. Ke, K. Sierszecki, and C. Angelov, "COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems," in *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, (Washington, DC, USA), pp. 199–208, IEEE Computer Society, 2007. 60

[123] K. Hanninen, J. Maki-Turja, M. Nolin, M. Lindberg, J. Lundback, and K.-L. Lundback, "The Rubus component model for resource constrained real-time systems," in *Industrial Embedded Systems, 2008. SIES 2008. International Symposium on*, pp. 177 –183, june 2008. 60

[124] AUTOSAR Development Partnership, "AUTOSAR – Technical Overview V2.0.1,," June 2006. `www.autosar.org`, last visit April 28^{th}, 2012. 60

[125] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wust, , and J. Zettel, *Component Based Product Line Engineering with UML*. Addison-Wesley Longman Publishing Co., Inc., 2002. 60

[126] "The Fractal Project." `http://fractal.ow2.org/`, last visit June 18^{th}, 2012. 64, 73

[127] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, "Making components contract aware," *Computer*, vol. 32, pp. 38–45, July 1999. 68

[128] J. Feljan, L. Lednicki, J. Maras, A. Petricic, and I. Crnkovic, "Classification and survey of component models," tech. rep., December 2009. 68

[129] "Course Description - Computer Science Department. Oklahoma Stage University." `http://cs.okstate.edu/courses.html`, last visit May 4^{th}, 2012. 71

[130] A. K. Dey, "Understanding and using context," *Personal Ubiquitous Comput.*, vol. 5, pp. 4–7, Jan. 2001. 74

[131] T. Strang, C. Linnhoff-Popien, and K. Frank, "Cool: A context ontology language to enable contextual interoperability," in *Distributed Applications and Interoperable Systems* (J.-B. Stefani, I. Demeure, and D. Hagimont, eds.), vol. 2893 of *Lecture Notes in Computer Science*, pp. 236–247, Springer Berlin / Heidelberg, 2003. 74, 75

[132] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," *ACM Trans. Comput. Syst.*, vol. 10, pp. 3–25, Feb. 1992. 79

[133] G. H. Kuenning and G. J. Popek, "Automated hoarding for mobile computers," in *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, (New York, NY, USA), pp. 264–275, ACM, 1997. 79

[134] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek, "Rover: a toolkit for mobile information access," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, (New York, NY, USA), pp. 156–171, ACM, 1995. 79

[135] A. Fuggetta, G. Picco, and G. Vigna, "Understanding code mobility," *Software Engineering, IEEE Transactions on*, vol. 24, pp. 342 –361, may 1998. 79

[136] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas, "Olympus: A high-level programming model for pervasive computing environments," in *Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*, PERCOM '05, (Washington, DC, USA), pp. 7–16, IEEE Computer Society, 2005. 86

[137] E. Cariou, A. Beugnard, and J. M. Jézéquel, "An architecture and a process for implementing distributed collaborations," in *Proceedings of the Sixth International ENTERPRISE DISTRIBUTED OBJECT COMPUTING Conference (EDOC'02)*, EDOC '02, (Washington, DC, USA), pp. 132–, IEEE Computer Society, 2002. 88

[138] E. C. Kabore, *Contribution a l'automatisation d'un processus de construction d'abstractions de communication par transformations successives de modeles.* PhD Dissertation, Telecom Bretagne/Universite de Rennes 1, Decembre 2008. 88

[139] A. Beugnard and A. Hassan, "A Calculus for a New Component Model in Highly Distributed Environments," in *5th International Workshop on Harnessing Theories for Tool Support in Software*, pp. 106–124, 2011. `http://urn.nb.no/URN:NBN:no-29716`. 103, 112

[140] A. Beugnard and A. Hassan, "A New Component Model for Highly Distributed Environements," in *8th International Symposium on Formal Aspects of Component Software - FACS*, 2011. Downloadable at: `http://departements.telecom-bretagne.eu/info/publications`. 111

[141] P. Abate and R. Di Cosmo, "Predicting upgrade failures using dependency analysis," in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering Workshops*, ICDEW '11, (Washington, DC, USA), pp. 145–150, IEEE Computer Society, 2011. 133

[142] T. R. Gruber, "A Translation Approach to Portable Ontology Specifications," *Journal of Knowledge Acquisition*, vol. 5(2), pp. 199–220, June 1993. 157

[143] W. Swartout and A. Tate, "Guest editors' introduction: Ontologies," *IEEE Intelligent Systems*, vol. 14, pp. 18–19, 1999. 157

[144] W. N. Borst, *Construction of Engineering Ontologies*. PhD Dissertation, Institute for Telematica and Information Technology, University of Twente, Enschede, The Netherlands, 1997. 157

[145] R. Studer, V. R. Benjamins, and D. Fensel, "Knowledge engineering: principles and methods," *Data Knowl. Eng.*, vol. 25, pp. 161–197, Mar. 1998. 157

[146] S. K. Das, *Deductive Databases and Logic Programming.* Addison Wesley, 1992. 158

[147] J. C. Arpírez, O. Corcho, M. Fernández-López, and A. Gómez-Pérez, "Webode: a scalable workbench for ontological engineering," in *Proceedings of the 1st international conference on Knowledge capture*, K-CAP '01, pp. 6–13, ACM, 2001. 162

[148] C. Tempich, E. Simperl, M. Luczak, R. Studer, and H. S. Pinto, "Argumentation-based ontology engineering," *IEEE Intelligent Systems*, vol. 22, pp. 52–59, Nov. 2007. 162

[149] P. Haase, F. van Harmelen, Z. Huang, H. Stuckenschmidt, and Y. Sure, "A framework for handling inconsistency in changing ontologies," in *Proceedings of the 4th international conference on The Semantic Web*, ISWC'05, (Berlin, Heidelberg), pp. 353–367, Springer-Verlag, 2005. 162

[150] P. Haase, J. Volker, and Y. Sure, "Management of dynamic knowledge," *Journal of Knowledge Management*, vol. 9(5), pp. 97–107, 2005. 162

[151] A. A. S. Esber, *Le fixe et le mouvant.* PhD Dissertation, Universite Saint-Joseph de Beyrouth - Lebanon, 1975. 163

[152] J. Meisami and P. Starkey, *Encyclopedia of Arabic Literature.* Routledge, 1998. 163

[153] P. F. Kennedy, *The Wine Song in Classical Arabic Poetry: Abu Nuwas and the Literary Tradition.* Open University Press, 1997. 163

[154] P. F. Kennedy, *Abu Nuwas: A Genius of Poetry.* OneWorld Press, 2005. 163

[155] M. Kifer, G. Lausen, and J. Wu, "Logical foundations of object-oriented and frame-based languages," *Journal of the ACM*, vol. 42, pp. 741–843, July 1995. 165, 175

[156] G. Yang and M. Kifer, "Reasoning about anonymous resources and meta statements on the semantic web," *Journal on Data Semantics*, vol. LNCS 2800, pp. 69–97, 2003. 165, 175

[157] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst, "Managing semistructured data with FLORID: a deductive object-oriented perspective," *Information Systems*, vol. 23, pp. 589–613, Dec. 1998. 165

[158] S. Decker, M. Erdmann, D. Fensel, and R. Studer, "Ontobroker: Ontology based access to distributed and semi-structured information," in *Proceedings of the IFIP TC2/WG2.6 Eighth Working Conference on Database Semantics- Semantic Issues in Multimedia Systems*, DS-8, (Deventer, The Netherlands, The Netherlands), pp. 351–369, Kluwer, B.V., 1998. 165

[159] G. Yang, M. Kifer, and C. Zhao, "Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web," in *Proceedings of International Conference on Ontologies, Databases and Applications of Semantics (ODBASE)*, pp. 671–688, 2003. 165

[160] M. Kifer and E. L. Lozinskii, "A framework for an efficient implementation of deductive database systems," in *Proceedings of the 6th Advanced Database Symposium*, (Tokyo Japan), pp. 109–116, 1986. 166

[161] C. Beeri and R. Ramakrishnan, "On the power of magic," *The Journal of Logic Programming*, vol. 10, pp. 255–300, 1991. 166

[162] N. F. Noy and D. L. McGuinness, "Ontology development 101: A guide to creating your first ontology," tech. rep., Stanford Knowledge Systems

Laboratory and Stanford Medical Informatics, 2001. 166, 176

[163] A. M. Hoss, *Ontology-based methodology for error detection in software design.* PhD Dissertation, Louisiana State University, August 2006. 166

[164] "The Foundation for Intelligent Physical Agents," 12 2002. `http://www.fipa.org/`, last visit June 18^{th}, 2012. 187

[165] "FIPA Device Ontology Specification," 12 2002. `http://www.fipa.org/specs/fipa00091/`, last visit June 18^{th}, 2012. 187